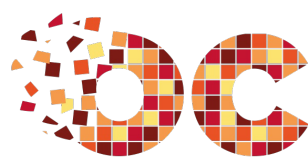


La cryptographie asymétrique : RSA

Par Simon Chabot (chabotsi)



OPENCLASSROOMS

www.openclassrooms.com

*Licence Creative Commons 7 2.0
Dernière mise à jour le 18/09/2010*

Sommaire

Sommaire	2
Partager	1
La cryptographie asymétrique : RSA	3
Partie 1 : La théorie	4
RSA ? Qu'est-ce donc ?	4
Les inventeurs du système RSA	4
Comment ça fonctionne ?	4
Comment Bob fait-il pour envoyer un message chiffré à Alice ?	5
Comment Alice déchiffre-t-elle le message reçu ?	5
Qu'est-ce que l'attaque du milieu ?	5
Comment crée-t-on nos clés ?	7
Création de la clé publique	7
Création de la clé privée	7
Comment trouver U ?	7
U = -713 n'est pas acceptable !	9
Ca y est !	9
Des clés sécurisées	9
Et le chiffage, comme ça se passe ?	10
Tout fonctionne avec des chiffres !	11
Quelle solution prendre ?	11
Un programme de calcul	11
Le chiffage proprement dit	12
Etape 1 : remplacement des caractères par leurs valeurs ASCII	12
Etape 2 : Premier calcul, la puissance	12
Etape 3 : Deuxième et dernier calcul : le modulo	13
On a fini	13
Effectuer le chiffage grâce à une seule et unique fonction	14
Le déchiffrage	14
L'utilisation de la clé privée	15
Etape 1 : Premier calcul, la puissance	15
Etape 2 : Le modulo	15
Etape 3 : Le remplacement	15
Le tout en une fonction	15
Partie 2 : La pratique	16
GMP - "Arithmetic without limitation"	16
L'intérêt de cette bibliothèque	16
Installation sous Windows	16
Préparation...	16
Compilation...	17
Installation sous GNU/Linux	17
Créer un projet avec notre nouvelle bibliothèque	18
Notre premier programme	18
Compiler notre code	18
Comment utiliser GMP ?	19
L'objet mpz_class	20
Les opérateurs usuels	20
Sommes, différences, produits, quotients.	20
Les divisions	21
Quelques fonctions mathématiques...	22
La fonction "modulo"	22
La fonction "puissance"	23
Une fonction idéale pour RSA !	23

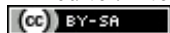


La cryptographie asymétrique : RSA

Par  [Simon Chabot \(chabotsi\)](#)

Mise à jour : 18/09/2010

Difficulté : Intermédiaire  Durée d'étude : 20 jours



Bonjour chers Zéros !

Il est sans doute probable que vous vous soyez demandés comment faisait-on pour chiffrer des données confidentielles (par exemple, les paiements par cartes bleues, ou bien les paiements en lignes, etc..). Et bien je vais tenter d'y répondre en vous présentant le fonctionnement du système RSA. Et par la suite la création d'un programme en C++ qui permet de chiffrer des messages grâce à ce système tant utilisé.

Partie 1 : La théorie

Dans cette partie, nous allons voir comment il est possible de chiffrer (et de déchiffrer) un message grâce au système RSA.

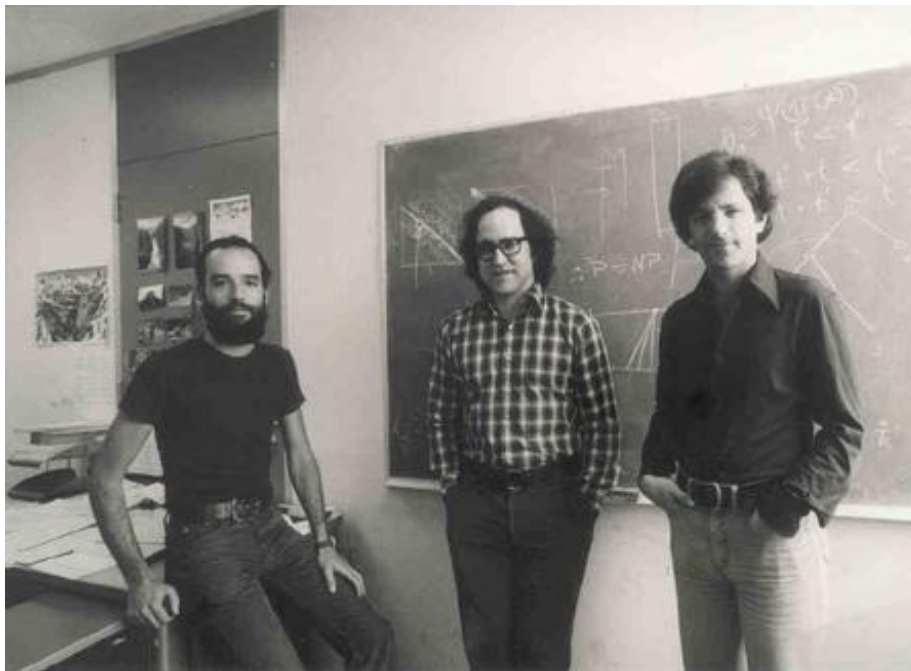
📁 RSA ? Qu'est-ce donc ?

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...

Pour commencer, voyons d'abord par qui il a été inventé. Nous finirons par étudier son fonctionnement général.

Les inventeurs du système RSA

Le système de chiffrement RSA a été inventé par trois mathématiciens : Ron Rivest, Adi Shamir et Len Adleman, en 1977 (On retrouve le sigle RSA dans les noms des inventeurs).



Voici une photo des trois mathématiciens en question

Ce système de chiffrement est beaucoup utilisé dans le e-commerce (paiement en ligne, etc...), mais aussi dans les cartes bleues, dans les logiciels (exemple : [OpenSSH](#)), etc...

Comment ça fonctionne ?



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 boîte aux lettres (où n'importe qui peut mettre - chiffrer - des messages) et seul le propriétaire de la clé de la boîte (clé privée) peut lire - déchiffrer - les messages)

Le système de chiffrement RSA est un système de chiffrement dit *asymétrique*.

Pour la suite du tutoriel, on va prendre deux personnages : Alice et Bob. (Ce sont les personnages que l'on utilise très souvent quand on parle de cryptographie, alors je ne vais pas changer 🤪)

J'ai donc l'honneur de vous présenter nos chers compagnons, en chair et en os, Alice :



et Bob :



Alice et Bob possèdent tout deux une clé publique et une clé privée.

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

Nous avons vu précédemment que 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.



Où réside la force du système RSA ?

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.



Mais, ce genre de système, ce n'est pas infallible, si ?

Eh non, malheureusement, ce n'est pas infallible...

Il existe plusieurs attaques contre le système RSA. Notamment les attaques de Wiener, Hastad, par chronométrage, ou encore l'attaque du milieu. Pour ma part, je vais détailler l'attaque du milieu, car c'est une attaque "type" des systèmes asymétriques (Pour plus d'explications sur les autres types d'attaques cliquez [ici](#))

Qu'est-ce que l'attaque du milieu ?

Afin d'expliquer le plus clairement possible ce qu'on appelle "attaque du milieu", je vais faire intervenir un troisième personnage,

"l'espion" :



Le rôle de l'espion est de se faire passer pour Bob aux yeux d'Alice, et de se faire passer pour Alice aux yeux de Bob. L'espion va se placer "au milieu" d'Alice et Bob.

Il va récupérer dans l'annuaire la clé publique d'Alice. Puis va s'arranger pour que Bob prenne sa clé publique au lieu de celle d'Alice (sans que Bob ne s'en aperçoive, évidemment). Bob va donc chiffrer le message avec la clé publique de l'espion. Lors de l'envoi du message chiffré de Bob à Alice, l'espion va l'intercepter, et grâce à sa propre clé privée le déchiffrer et le lire. Une fois qu'il aura le message en clair, il peut éventuellement le re-chiffrer avec la clé publique d'Alice et lui envoyer le message afin de passer inaperçu...



L'espion se place entre Bob et Alice afin d'intercepter le message

Voilà, normalement vous devriez avoir compris comment on pouvait, en théorie, chiffrer et déchiffrer des messages avec un système asymétrique.

Dans le prochain chapitre, on va voir comment créer les clés privée, et publique ! (Là, il y aura un peu de mathématiques 😊)

Comment crée-t-on nos clés ?

Dans cette partie nous allons apprendre à créer les clés privée et publique qui nous serviront plus tard à chiffrer et déchiffrer les messages.

Il faut quelques notions en mathématiques, mais je vais faire de mon mieux pour être le plus explicite possible.

Création de la clé publique

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



Un nombre premier est un nombre supérieur ou égal à 2 qui possède deux diviseurs positifs distincts et uniquement deux (1 et lui-même). On peut prendre pour exemple 2, 3, 7, 53, 89... Il existe une infinité de nombres premiers

Soit P et Q , ces deux nombres premiers .

(Je vais prendre des valeurs à titre d'exemple pour P et Q , mais n'hésitez pas à les changer...)

Je prends $P = 53$ et $Q = 97$

Ensuite, on va prendre un autre nombre, N , tel que : $N = P \times Q$

Dans mon cas, j'ai $N = 53 \times 97 = 5141$

On pose $M = (P - 1) \times (Q - 1)$.



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

J'obtiens pour ma part, $M = (53 - 1) \times (97 - 1) = 4992$

Pour créer notre clé publique, il ne nous reste plus qu'à choisir un nombre C , **qui soit premier avec M** .

(Il en existe, là aussi, une infinité.) Je choisis $C = 7$.



Deux nombres sont premiers entre eux si et seulement s'ils n'ont aucun facteur commun (à l'exception de 1 et -1). Par exemple, 16 et 12 ne sont pas premiers entre eux, car ils ont au moins un facteur commun (ici, on a 2 et 4 en commun ($12 = 6 * 2 = 3 * 4$ et $16 = 8 * 2 = 4 * 4$))

Pour savoir facilement si deux sont premiers entre eux, il suffit que calculer le **PGCD** de ces deux nombres. S'il est égal à 1, alors ces deux nombres sont premiers entre eux. (J'ai bien $\text{PGCD}(4992, 7) = 1$)...

Au final, notre clé publique est composée de (N, C) . Dans mon cas, j'ai $(N = 5141, C = 7)$ comme clé publique.

Création de la clé privée

Pour créer notre clé privée, on va calculer un nombre U , qui va nous permettre "d'inverser la fonction de chiffrement" très facilement (On verra dans quelques chapitres comment faire). 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 $a \times u + b \times v = 1$ (u et v étant des nombres entiers). Or, on a dit peu avant que C et M devaient être premiers entre eux. Il existe donc deux nombres entiers u et v qui répondent à l'équation.

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

Comment trouver U ?

Pour trouver U , il faut utiliser un algorithme spécial. Le but de ce chapitre n'est pas de vous montrer comment fonctionne cet algorithme, mais plutôt ce que l'on peut faire avec. On va donc utiliser un petit programme qui permet de trouver la valeur de U cherchée.

Le programme est disponible pour GNU/Linux (*cf un peu pour loin pour l'installation*), et [ici](#) pour Windows.



Pour ceux qui seraient intéressés par cet algorithme, des informations sont disponibles [ici](#).

- Sous GNU/Linux, nous allons devoir compiler ce programme. Pour ce faire il faut auparavant avoir installé la bibliothèque **GMP**. (C'est la bibliothèque que l'on utilisera dans toute la partie II du tutoriel. Il faudra donc installer la bibliothèque GMP à un moment où un autre 😊 (Pour Windows, j'expliquerais comment faire dans le chapitre I de la partie II))
Ouvrez une console et copiez-y ce code, ça fera tout pour vous 😊

Code : Console - Installation du programme : GNU/Linux

```
cd ~
mkdir RSA
cd RSA
wget http://ftp.sunet.se/pub/gnu/gmp/gmp-4.2.2.tar.gz
tar xzf gmp-4.2.2.tar.gz
cd gmp-4.2.2
./configure --enable-cxx
make
make check
sudo make install
cd ..
rm gmp-4.2.2 gmp-4.2.2.tar.gz
cd /usr/lib/
sudo ln -s /usr/local/lib/libgmpxx.so.4 libgmpxx.so.4
cd ~/RSA
wget http://tuxweb79.free.fr/Bezout_linux.zip
unzip Bezout_linux.zip
cd Bezout
make
chmod +x bezout
```

Ceci aura pour effet de compiler, puis d'attribuer les droits d'exécution au programme (en ayant installé la bibliothèque GMP avant...);

- Sous windows, vous extrayez l'archive .zip. Le programme est déjà compilé, il vous suffit de le lancer depuis la ligne de commande en faisant : démarrer > exécuter > "cmd" puis :

Code : Console - Installation du programme : Windows

```
cd chemin/vers/le/dossier/du/programme/
```

Comment fonctionne le programme ?

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

Code : Console - Programme Bezout

```
./bezout C M
```



Sous Windows, au lieu de taper "./bezout", il faut taper "Bezout.exe".

Dans mon cas, j'ai :

Code : Console - Programme Bezout

```
./bezout 7 4992
PGCD = 1
```


$$u = -713$$

$$v = 1$$

Ainsi, je connais U.

U = -713 n'est pas acceptable !



Comment ça ? c'est pas bon ?! 😬

Bah.. pas tout à fait. L'algorithme du système RSA a besoin d'un U tel que $2 < U < M$, or là, on voit clairement que U est inférieur à 2 (dans mon exemple, mais ça arrive souvent 😬)

Comment faire pour y remédier ?

En appliquant la formule suivante (avec U_0 la solution trouvée précédemment) : $2 < (U_0 - K \times M) < M$. On peut trouver d'autres solutions à l'équation précédente correspondant à nos besoins. Il suffit de faire varier l'entier K pour obtenir une valeur dans l'intervalle $]2; M[$. On pose U une valeur de $(U_0 - K \times M)$ tel que $2 < (U_0 - K \times M) < M$.

Trouver directement un U valable grâce au programme précédent

Pour trouver directement un U qui convient, il suffit de lancer le programme Bezout, avec l'option "-rsa". Voici ce que j'obtiens dans mon cas :

Code : Console - Programme Bezout

```
./bezout -rsa 7 4992
4279
```

Je retiens U = 4279.


Ca y est !

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

Des clés sécurisées

Je ne sais pas si vous avez remarqué, mais on construit nos clés seulement à partir de P et Q.

De plus, tout le monde peut connaître N (qui est le produit de P et Q), car N fait partie de la clé publique. Il suffirait donc à

n'importe qui (notre espion , par exemple) de décomposer N en un produit de deux facteurs premiers (P et Q) pour

ensuite recalculer M, et grâce à M avoir la possibilité de trouver U... (Ce qui permettrait à l'espion de déchiffrer les messages)



La décomposition de n'importe quel nombre en facteurs premiers est unique (à l'ordre près). Ainsi, si on décompose N, on retrouve forcément les valeurs de P et Q initiales, et pas d'autres !



Eh ! tu me fais peur là !? C'est pas si sécurisé que ça ton truc ! 😬😬

En effet, ce n'est pas très sécurisé si P et Q sont de petits nombres premiers. En réalité, P et Q sont des nombres composés d'une bonne centaine de chiffres, ce qui fait qu'il est très difficile de décomposer N pour retrouver P et Q. Même des machines sur-

puissantes reliées entre elles ne pourraient pas casser N en un temps raisonnable...

Mais, pour nous, on va prendre des petits nombres premiers... Ça va permettre de simplifier grandement les calculs 😊

Quelques exemples de vrais facteurs

Voici un exemple de factorisation :

$N = 31074182404900437213507500358885679300373460228427275457201619488232064405$
 $18081504556346829671723286782437916272838033415471073108501919548529007337$
 $724822783525742386454014691736602477652346609$

qui est décomposable en :

$P = 16347336458092538484431338838650908598417836700330$
 $92312181110852389333100104508151212118167511579$

$Q = 1900871281664822113126851573935413975471896789968$
 $515493666638539088027103802104498957191261465571$

(Il a fallu faire travailler 80 processeurs Opteron de 2.2GHz pendant 5 mois. pour trouver P et Q 😊)

Voilà, maintenant que nous savons créer nos clés, nous allons voir dans le chapitre suivant comment chiffrer un message.

Et le chiffrement, comme ça se passe ?

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...

Tout fonctionne avec des chiffres !

Vous vous doutez sûrement que l'on va utiliser des chiffres pour chiffrer nos messages. Il va donc falloir trouver un moyen d'associer à chaque caractère un nombre. J'ai deux solutions à vous proposer :

1. Remplacer chaque caractère par sa position dans l'alphabet ;
2. Remplacer chaque caractère par sa valeur dans la table *ASCII*

Quelle solution prendre ?

Chacune des deux solutions a son avantage et son inconvénient. L'avantage de la solution 1) est qu'elle nous permet de gérer de très petits nombres (de 0 à 25), ce qui serait pratique pour les calculs. Son inconvénient découle directement de son avantage, on ne peut gérer que 26 caractères (donc pas de majuscules, pas d'espaces, pas de chiffres, pas d'accents...) Bref, vous avez compris ce n'est pas la meilleure solution pour nous, qui voulons chiffrer n'importe quel caractère.

Nous allons donc choisir la deuxième solution qui nous permet de gérer énormément de caractères, avec des chiffres, des espaces, des accents, etc...



Je vous conseille de mettre ce lien <http://www.asciitable.com/> en favori, car nous allons devoir nous y référer assez souvent.

Bien, passons à la suite 🤖

Un programme de calcul

Pour chiffrer nos messages, nous allons principalement utiliser le "modulo" et les "puissances". Ce genre d'opérations n'est pas facile à faire de tête (*à moins d'être un vrai génie* 😊). De plus, nous allons nous servir (malgré nos efforts) de nombres relativement grands. Toutes les calculatrices ne sont pas capables d'effectuer les opérations "modulo" et "puissance" avec de grands nombres. C'est pourquoi, je vous conseille d'installer un petit programme qui va faire tout juste ce que l'on veut. Pas plus, pas moins.

Il est disponible sous GNU/Linux et sous Windows.

- L'installation sous **Linux** se fait de la manière suivante : vous ouvrez une console, et vous y copiez les instructions suivantes :

Si vous n'avez pas installé GMP, il faudra le faire 🤖 ; copiez ce code, il l'installera pour vous :

Code : Console



```
cd ~
mkdir RSA
cd RSA
wget http://ftp.sunet.se/pub/gnu/gmp/gmp-4.2.2.tar.gz
tar xzf gmp-4.2.2.tar.gz
cd gmp-4.2.2
./configure --enable-cxx
make
make check
sudo make install
cd ..
rm gmp-4.2.2 gmp-4.2.2.tar.gz
cd /usr/lib/
sudo ln -s /usr/local/lib/libgmpxx.so.4 libgmpxx.so.4
cd ~/RSA
```

Pour installer le programme de calcul, copiez ces instructions :

Code : Console

```
cd ~
mkdir RSA
cd RSA
wget http://tuxweb79.free.fr/Calculs_RSA_linux.zip
unzip Calculs_RSA_linux.zip
cd Calculs_RSA
make
chmod +x calculs_RSA
```

- Pour **Windows**, téléchargez le programme de calcul [ici](#). Puis extrayez l'archive. Pour lancer le programme ouvrez l'invite de commandes (Démarrer -> exécuter -> "cmd"), puis :

Code : Console

```
cd chemin/vers/le/dossier/du/programme/
```

Voilà, c'est installé ; on va maintenant s'en servir 😊

Le chiffrement proprement dit

On va reprendre nos deux camarades, Alice et Bob. La situation est toujours la même, Bob veut envoyer un message chiffré à Alice. Il va donc récupérer sa clé publique. Dans l'annuaire, il voit :

Annuaire RSA

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

Il prend la clé publique d'Alice : (N = 5141, C = 7). (J'ai volontairement repris la même clé que dans le chapitre précédent afin que vous sachiez les valeurs de P Q et tout ce qui s'ensuit)

Etape 1 : remplacement des caractères par leurs valeurs ASCII

Bob veut envoyer le message "Bonjour !" à Alice. On a vu précédemment, qu'on devait travailler avec des nombres. On se réfère à la table [ASCII](#), et on effectue le remplacement.

$B \Leftrightarrow 66o \Leftrightarrow 111n \Leftrightarrow 110j \Leftrightarrow 106o \Leftrightarrow 111u \Leftrightarrow 117r \Leftrightarrow 114(\text{espace}) \Leftrightarrow 32! \Leftrightarrow 33$

Etape 2 : Premier calcul, la puissance

Ensuite, on va élever chaque sous-message (nombre) à la **puissance C** (7, dans notre cas)
On a alors :

$B \Leftrightarrow 66 \Rightarrow 66^7 o \Leftrightarrow 111 \Rightarrow 111^7 n \Leftrightarrow 110 \Rightarrow 110^7 j \Leftrightarrow 106 \Rightarrow 106^7 o \Leftrightarrow 111 \Rightarrow 111^7 u \Leftrightarrow 117 \Rightarrow 117^7 r \Leftrightarrow 114 \Rightarrow 114^7 (\text{espace}) \Leftrightarrow 32 \Rightarrow 32^7 ! \Leftrightarrow 33 \Rightarrow 33^7$

Vous pouvez à l'aide du programme de calcul, calculez les valeurs précédentes (66^7 , etc..) en tapant simplement

Code : Console



```
./calculs_RSA 66 7
```

dans la console et appuyez sur "entrée" pour voir le résultat.

Sous Windows, tapez simplement "calculs_RSA"

Etape 3 : Deuxième et dernier calcul : le modulo



Le modulo, c'est quoi ? 🤔

Le modulo, c'est le reste de la division euclidienne d'un nombre entier par un autre.

Un exemple de modulo :

"16 modulo 3" $\Leftrightarrow 16 \bmod(3) = 1$

Et oui, car dans 16, on peut placer 5 fois 3 (ce qui donne 15 😊) et il reste $16 - 15 = 1$.

Ainsi on note $16 \bmod(3) = 1$.

Nous avons effectué notre premier calcul, nous allons maintenant faire le deuxième calcul, à partir des résultats que l'on a obtenu précédemment.

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



Il est important d'utiliser le programme de calcul à partir de ce moment, sans quoi vous ne pourrez pas trouver la valeur finale...

On obtient (grâce au programme) :

$B \Leftrightarrow 66 \Rightarrow 66^7 \Rightarrow (66^7) \bmod(5141) = 286 \Rightarrow 111 \Rightarrow 111^7 \Rightarrow (111^7) \bmod(5141) = 1858 \Rightarrow 110 \Rightarrow 110^7 \Rightarrow (110^7) \bmod(5141) = 2127 \Rightarrow 106 \Rightarrow 106^7 \Rightarrow (106^7) \bmod(5141) = 280 \Rightarrow 111 \Rightarrow 111^7 \Rightarrow (111^7) \bmod(5141) = 1858 \Rightarrow 117 \Rightarrow 117^7 \Rightarrow (117^7) \bmod(5141) = 1774 \Rightarrow 114 \Rightarrow 114^7 \Rightarrow (114^7) \bmod(5141) = 727(\text{espace}) \Rightarrow 32 \Rightarrow 32^7 \Rightarrow (32^7) \bmod(5141) = 3675 \Rightarrow 33 \Rightarrow 33^7 \Rightarrow (33^7) \bmod(5141) = 244$

Pour calculer ces valeurs avec le programme de calcul il suffit de taper et de valider par "entrée" pour voir le résultat :

Code : Console

```
./calculs_RSA 66 7 5141
```

On a fini 😊

Voilà, plus de calculs à faire. Ainsi avec la clé publique d'Alice ($N = 5141$, $C = 7$), le message "Bonjour !" devient "386 1858 2127 2809 1858 1774 737 3675 244"

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 😊)

Effectuer le chiffrement grâce à une seule et unique fonction

On peut facilement ramener les deux opérations précédentes en une seule et unique fonction.

La voici : $f(x) = x^C \bmod(N)$. Dans cette fonction "x" représente la valeur ASCII du caractère à chiffrer.

Cette fonction élève x à la puissance C et calcule le reste de la division euclidienne de x^C par N.

Ainsi on a :

$$B \Leftrightarrow 66 \Rightarrow f(66) = 386 \dots \Leftrightarrow 33 \Rightarrow f(33) = 244$$



On utilisera cette fonction dans le programme que l'on réalisera dans la partie pratique du tutoriel.

Voilà, vous savez maintenant comment on chiffre des messages.

Vous pouvez vous entraîner à chiffrer d'autres messages, en créer d'autres clés. Cela peut être intéressant.

Dans le chapitre suivant, nous allons voir comment peut-on faire pour déchiffrer le message.

Le déchiffrage

Alice vient de recevoir un message de Bob, le voici : "386 737 970 204 1858". Notre mission, trouver sa signification 😊

L'utilisation de la clé privée

Nous allons nous mettre dans la peau d'Alice pour ce chapitre.

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

Donc on a reçu un message de Bob, et on veut le lire. *Le message est "386 737 970 204 1858".*



(J'ai volontairement pris un autre message que celui chiffré dans le chapitre précédent... (ça aurait été trop simple 😊))

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

Etape 1 : Premier calcul, la puissance

A l'instar du chiffrement, on va élever chaque sous-message (nombre) à une puissance. Cette puissance sera U

On a alors :

$$386 \Rightarrow 386^{4279} \quad 737 \Rightarrow 737^{4279} \quad 970 \Rightarrow 970^{4279} \quad 204 \Rightarrow 204^{4279} \quad 1858 \Rightarrow 1858^{4279}$$

Etape 2 : Le modulo

Dans cette étape, on va calculer le modulo des résultats obtenus précédemment par N (Nous avons N = 5141)

Ce qui nous donne :

$$386 \Rightarrow 386^{4279} \Rightarrow (386^{4279})_{\text{mod}(5141)} = 66737 \Rightarrow 737^{4279} \Rightarrow (737^{4279})_{\text{mod}(5141)} = 114970 \Rightarrow 970^{4279} \Rightarrow (970^{4279})_{\text{mod}(5141)} = 97204 \Rightarrow 204^{4279} \Rightarrow (204^{4279})_{\text{mod}(5141)} = 1181858 \Rightarrow 1858^{4279} \Rightarrow (1858^{4279})_{\text{mod}(5141)} = 111$$

Etape 3 : Le remplacement

Les résultats que nous venons d'obtenir sont en réalité la valeur ASCII du caractère original. On va donc se référer à la table ASCII (je vous l'avais dit, on s'en sert beaucoup 😊), et effectuer les remplacements nécessaires.

$$386 \Rightarrow 386^{4279} \Rightarrow (386^{4279})_{\text{mod}(5141)} = 66 \Leftrightarrow B \quad 737 \Rightarrow 737^{4279} \Rightarrow (737^{4279})_{\text{mod}(5141)} = 114 \Leftrightarrow r \quad 970 \Rightarrow 970^{4279} \Rightarrow (970^{4279})_{\text{mod}(5141)} = 97 \Leftrightarrow a \quad 204 \Rightarrow 204^{4279} \Rightarrow (204^{4279})_{\text{mod}(5141)} = 118 \Leftrightarrow v \quad 1858 \Rightarrow 1858^{4279} \Rightarrow (1858^{4279})_{\text{mod}(5141)} = 111 \Leftrightarrow o$$

Voilà, nous venons de reconstituer le message envoyé par Bob : "Bravo"

Le tout en une fonction

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

Cette fonction est : $f(x) = x^U \text{mod}(N)$ (Où U et N les valeurs de la clé privée, et x le sous-message).

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



Vous noterez que cette fonction est quasiment identique à celle qui nous sert à chiffrer.

Voilà, vous êtes maintenant capables de chiffrer et de déchiffrer des messages si vous connaissez les clés. Sympathique non ?

Partie 2 : La pratique

C'est ici que nous allons utiliser ce que nous avons vu dans la partie précédente.

Que les choses soient claires pour tout le monde : le but de cette partie n'est pas de créer un super-crypteur-de-la-mort-qui-tue-en-un-coup, mais plutôt de découvrir une bibliothèque qui sert à gérer de grands nombres (la bibliothèque [GMP](#)), et comment implémenter ce qu'on a vu précédemment en C++.

Étant donné que l'on va coder notre programme en C++, il est vivement conseillé d'avoir quelques bases (notamment utilisation / création d'objets). Vous trouverez un cours sur le C++ [ici](#), n'hésitez pas à vous y référer !

Voilà : sur ce... à l'attaque ! 🤖

GMP - "Arithmetic without limitation"

Cette partie présente l'intérêt d'une telle bibliothèque et explique comment l'installer (sous GNU/Linux, et Windows), et finit par la création d'un petit programme d'exemple...

L'intérêt de cette bibliothèque



Euh... une bibliothèque ? mais c'est pour quoi faire ?

La bibliothèque GMP est une bibliothèque qui va nous permettre de gérer de très très grand nombre, et d'effectuer tout un tas de calculs rapidement.



Un "double" ou un "float" peut très bien pu suffir pour gérer nos *grands* nombres, non ? Pas la peine de prendre cette... *GMP* !

La réponse est non (vous deviez vous en douter ^^). Le type "double", le plus grand des types primitifs, a pour extréma -1.7×10^{308} et 1.7×10^{308} . Or, je sais pas si vous vous rappelez, mais dans le chapitre sur le [déchiffrage](#), on avait à un moment donné ce calcul : 386^{4279} . Autant vous dire que le résultat dépasse largement le type "double". (Même un "unsigned double" ne ferait pas l'affaire pour ceux qui y auraient pensé...)

Pour la petite histoire, j'ai calculé 386^{4279} avec la bibliothèque GMP, et le résultat est composé d'exactly 11069 chiffres, un double aurait rendu l'âme bien avant d'afficher le résultat 😊. On ne peut donc pas se passer d'une telle bibliothèque pour faire notre programme.

Passons à l'installation 🤖.

Installation sous Windows

Préparation...

Pour installer la bibliothèque GMP sous Windows, nous allons installer une sorte de paquetage, composé d'un shell (une console en gros 🤖) et de plusieurs petits programmes, qui va nous permettre de compiler les sources de la bibliothèque, pour que l'on puisse ensuite s'en servir... Ce paquetage se nomme MSYS, et est disponible en téléchargement [ici](#).



Pour continuer, vous devez avoir MinGW d'installé sur votre machine ! Si ce n'est pas le cas, vous devez le faire. (Si vous avez Dev-Cpp ou Code-Blocks, il doit être présent...)

Allez dans le répertoire d'installation de MinGW, puis dans le dossier "bin". (Ce répertoire est pour ma part "C:\MinGW", mais il peut être "C:\Dev-Cpp", tout dépend de votre machine) 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 maintenant 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.

Voici un screen de ce que j'ai :


```

c:\ C:\WINDOWS\system32\cmd.exe
C:\msys\1.0\postinstall>PATH ..\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\system32\Wbem
C:\msys\1.0\postinstall>.\bin\sh.exe pi.sh
This is a post install process that will try to normalize between
your MinGW install if any as well as your previous MSYS installs
if any. I don't have any traps as aborts will not hurt anything.
Do you wish to continue with the post install? [yn ] y
Do you have MinGW installed? [yn ] y
Please answer the following in the form of c:/foo/bar.
Where is your MinGW installation? c:/MinGW_

```

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. Extrayez ensuite l'archive dans un dossier facile d'accès... "C:\gmp" par exemple.

Compilation...

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, grâce à la commande "cd", ce qui donne : `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/dev-cpp --enable-cxx`

Note : Remplacez "c/dev-cpp" par le chemin correspondant à votre IDE. Si le chemin de votre IDE est "C:\Code-Blocks\" tapez "c/Code-Blocks"

Laissez travailler. Une fois cela fini (Il ne devrait pas y avoir d'erreur...) On peut compiler.. La compilation se fait avec la commande : `make`. La compilation peut être longue... Il faut être patient 😊. A la fin de la compilation, tapez `make check` puis `make install`, pour installer la bibliothèque. (Encore un peu de patience... 😊)

Normalement, rendu à ce point, la bibliothèque devrait être (correctement) installée.

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

Installation sous GNU/Linux

Normalement, la bibliothèque GMP est déjà installée sur votre machine si vous avez suivi le tutoriel dans l'ordre. Si vous l'avez installé, vous pouvez aller directement à la dernière partie de ce chapitre. Sinon ouvrez une console et copiez-y :

Code : Console

```

cd ~
mkdir RSA
cd RSA
wget http://ftp.sunet.se/pub/gnu/gmp/gmp-4.2.3.tar.gz
tar xzf gmp-4.2.3.tar.gz
cd gmp-4.2.3

```

```
./configure --enable-cxx
make
make check
sudo make install
cd ..
rm -rf gmp-4.2.3 gmp-4.2.3.tar.gz
cd /usr/lib/
sudo ln -s /usr/local/lib/libgmpxx.so.4 libgmpxx.so.4
cd ~/RSA
```

Voilà, c'est installé 😊

Créer un projet avec notre nouvelle bibliothèque

Notre premier programme

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 : (Je vous expliquerais plus en détail comment utiliser cette bibliothèque, ce code est juste fait pour vérifier si le tout marche, et montrer comment on link la bibliothèque...)

Code : C++

```
/* 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

#define MAX 10 //On affichera MAX nombres premiers...

int main(int argc, char **argv)
{
    mpz_class a(0); //On crée 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;
}
```

Compiler notre code

Avec un IDE

Si vous utilisez un IDE, vous devez linker la bibliothèque avec le fichier "libgmpxx.a" situé dans le dossier "lib" de votre IDE, puis avec "libgmp.a" (L'ordre est important !). Avec Dev-Cpp, j'ai ceci : (Cliquez sur l'image pour la voir en plein écran)



Comme vous pouvez le constater, j'ai placé sur l'image des petits numéros, ils vous permettent de vous repérer quant à l'ordre des opérations à effectuer.

Une fois cela fait, vous pouvez ensuite compiler... Tout devrait bien se passer.

Avec un makefile

Si vous utilisez makefile ou la ligne de commande pour compiler, vous devez linker la bibliothèque avec votre programme en tapant : `-lgmpxx -lgmp`. Ce qui donne par exemple : `g++ -o executable main.cpp -lgmpxx -lgmp`

Et voilà, votre programme devrait compiler et fonctionner correctement. Il devrait afficher ceci :

Code : Console

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

Nous venons d'installer la bibliothèque GMP, dans le prochain chapitre, nous verrons quelques fonctions de bases de cette bibliothèque, et nous attaquerons notre programme de chiffrement.

Vous pouvez, si le cœur vous en dit, d'ores et déjà consulter la documentation de GMP, disponible au format PDF, [ici](#).

Comment utiliser GMP ?

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

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. Voyons ensemble comment on se sert de ce nouvel objet !

Code : C++

```
#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 :

Code : Console

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

On commence donc par inclure la bibliothèque GMP (*ligne 2*). On peut ensuite créer nos objets. Je vous présente ici 3 façons distinctes de créer un objet (*ligne 8*).

- 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.

Je ne serais pas plus long sur la création des objets `mpz_class`, l'essentiel à savoir est vu.

Les opérateurs usuels

Sommes, différences, produits, quotients.

Comme vous le savez (normalement !), le C++ offre un concept qui s'appelle *la surcharge des opérateurs*. L'objet `mpz_class` nous en fait donc profiter. Je vais être assez rapide sur le sujet, car il n'y a pas vraiment de différence d'utilisation entre les variables de type primitif (long, int, double, etc...) et l'objet `mpz_class`. Je vais juste vous montrer quelques exemples.

Code : C++

```
#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 :

Code : Console

```

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

```



Ein ? $16/15 = 1$? Eh oui, `mpz_class` ne gère pas les nombres à virgule. (Pour utiliser des virgules, il faut se servir de "`mpf_class`" (le "f" pour float)). Ceci dit ; `mpz_class` offre une alternative assez intéressante à cette lacune, chose que nous allons voir tout de suite.

Les divisions

GMP nous donne tout un panel de fonctions sur la division. (Vous trouverez la liste exhaustive à la partie 5.6 de la documentation officielle (dont je vous ai fourni le lien, dans la conclusion du chapitre précédent.)) Voici les prototypes des trois fonctions sur la division qui me semblent primordiaux.

- `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)`

En voyant ces prototypes, plusieurs questions devraient vous venir à l'esprit :

- Pourquoi utilise-t-on des fonctions, et non des méthodes ?
- "`mpz_t`" ? Une nouveau type ? Un nouvel objet ? 🤔
- A quoi vont bien pouvoir nous servir ces fonctions ?

Alors, prenons les questions dans l'ordre.

- Il faut savoir, qu'à l'origine, GMP est en C, et non en C++. Certaines fonctions ont été implémentées en C++, des objets

(*mpz_class* ; *mpf_class* ; *mpq_class* (nous ne verrons pas ce dernier)) 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.

- "*mpz_t*" découle du fait que la bibliothèque était à l'origine en C. Il s'agit d'un des trois types de variables créés à l'origine de la GMP.
- 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".
 - Et la dernière permet d'obtenir le quotient et le reste de la division euclidienne de "n" par "d".

Maintenant que l'on sait à quoi servent ces fonctions, et que l'on sait pourquoi le prototype utilise des "*mpz_t*" et non des "*mpz_class*", on va voir comment on les utilise. Pour s'en servir, il va falloir que l'on "transforme" notre objet "*mpz_class*" en une variable du type "*mpz_t*". C'est pour cela que la méthode `get_mpz_t()` a vu le jour !

Je ne fais l'exemple que d'une fonction, les autres, vous pourrez les tester vous même : c'est le même principe.

Code : C++

```
#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 :

Code : Console

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

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. C'est pour cela qu'elle regorge de fonctions mathématiques. Certaines sont les mêmes que celles disponibles dans la bibliothèque mathématique standard. (`Ceil()`, `floor()`, `sqrt()`, `abs()`, etc.) Ce n'est pas sur celles-ci que je veux appuyer, car leur utilisation est assez intuitive et nous ne nous en servons pas durant ce tutoriel. Nous allons voir ensemble seulement les fonctions dont nous ferons usage plus tard.

La fonction "modulo"

Vous vous en rappelez de cette fonction ? On l'a vue dans la partie théorique du tutoriel. C'est sur cette fonction que repose, quasiment, tout le système RSA.

Je vous donne juste le prototype, avec l'exemple vu avant (pour la division) vous devriez vous en sortir... 🤪

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

Cette fonction effectue le calcul suivant : $r = n \bmod(d)$



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"

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

Voici son prototype :

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

Cette fonction effectue le calcul suivant : $rop = base^{exp}$

L'exposant doit être un entier de type "unsigned long int", et non un "mpz_class" !



La méthode `get_ui()` permet de transformer un `mpz_class` en "unsigned long int" ; cependant, attention à vérifier que votre objet loge dans "unsigned long int" !

Une fonction idéale pour RSA !

Nous pourrions très bien faire notre programme RSA en utilisant seulement les deux dernières fonctions que l'on vient de voir. Cependant, calculer une puissance puis un modulo (l'un après l'autre) demande de nombreux calculs, et donc du temps. L'astuce consiste en fait à calculer le modulo et la puissance en même temps ! 🤔

Et grâce à un algorithme et un peu de magie mathématique (🧙), on gagne du temps !

Voici le prototype de la fonction "RSA" :

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

Cette fonction effectue le calcul suivant : $rop = base^{exp} \bmod (modulo)$

Vous l'aurez deviné ; c'est cette fonction qui va nous servir lors du chiffage et du déchiffage de nos messages secrets. Voilà, je pense que les présentations de bases sont désormais faites.

A vous de manipuler les objets `mpz_class` (et pourquoi pas `mpf_class`, si vous désirez travailler avec des flottants).

Au prochain chapitre, nous attaquons les choses sérieuses ! La création du programme commencera... 😎

Ce tutoriel n'est pas fini ! Venez de temps en temps vérifier s'il n'y a pas de nouveaux chapitres ! 🤪