Cryptage RSA en JAVA

Francois Jolain

June 23, 2013

Contents

Part I

Le cryptage RSA et ses algorithmes

Chapter 1

Le cryptage RSA

1.1 Introduction

Dans ce monde de malfrats et d'insécurité, l'homme a toujours appris à crypter ses données. On se souvient du cryptage CESAR, où il suffit de décaler les lettres dans l'alphabet. Mais nous n'utilisons plus seulement le monde réel, la Matrice nous a envie, et elle se nomme internet. Et comme dans la Matrice nous sommes surveillé, mais heureusement nous avons le cryptage RSA qui nous vient en aide ! Vous n'avez peut-être jamais entendu parlé du RSA, mais pourtant vous l'utilisez tout le temps sur internet : échange bancaire, mots de passe...

Commençons par un exemple : Vous avez remarqué le casque de vos rêves sur Amazon. Au moment de passer en caisse, Amazon demande votre numéro de carte bancaire, mais un méchant pirate écoute votre conversation et souhaite plus que tout votre numéro. Il vous faut donc établir une connexion cryptée entre vous et Amazon, alors que vous vous êtes jamais vu auparavant pour communiquer un indice permettant le cryptage.

Je ne vais pas vous laissez plus longtemps en suspense, la solution est : cryptage *asymétrique* ! Mais si je vous parle de cryptage *asymétrique*, c'est qu'il doit exister des cryptages *symétriques* ? Un exemple le plus simple est le fameux cryptage CESAR. On le dit *symétrique* car il utilise le même nombre pour crypter et décrypter, dans celui-ci le nombre est le décalage dans l'alphabet. Mais revenons à notre situation, si Amazon vous envoid ce nombre pour permettre de crypter votre numéro de carte bancaire, le pirate la également. Il est donc en mesure de décrypter votre numéro de carte bancaire, que vous avez précédemment crypter, bref pas génial comme sécurité.

Alors qu'avec un cryptage *asymétrique*, il y a deux nombres différents : l'un pour crypter, l'autre pour décrypter. Amazon vous envoi donc le nombre pour crypter et garde précieusement le nombre pour décrypter. Ainsi votre numéro de carte bancaire circule crypter sur internet, et seul Amazon peut le décrypter.

Vous l'avez compris, le cryptage RSA fonctionne sur un système *asymétrique*, il est donc très prisé sur internet. Par contre il utilise plusieurs nombres pour crypter et décrypter, on parlera donc de *clefs*. La clef publique rassemble tout les nombres nécessaires au cryptage, elle peut-être partagée sans limitation. La

clef privée rassemble tout les nombres nécessaires au décrypter, il ne faut surtout pas la partager, seul le destinataire doit la connaitre.

Pour terminer cette introduction, nous pouvons imaginer un cryptage *symétrique* comme un coffre fort, il faut la même clef pour déposer un objet et le reprendre. Alors qu'un cryptage *asymétrique* ressemble plus à une boite au lettre : tout le monde peut y déposer un objet, mais seul le propriétaire à la clef pour récupérer l'objet.

1.2 Créer les clefs

Je vais donc vous expliquer comment créer ces fameuses clefs capables de crypter et de décrypter. En y regardant de plus près, on pourrai croire ces clefs magiques. Car il faut bien qu'elles aient un lien entre elles pour se compléter. Mais il ne faut pas retrouver la clef privée en partant de la clef publique, sinon notre cryptage tombe à l'eau ! Je vous rassure donc, il n'y a pas de magie, juste des maths. Cependant à par cela je ne sais plus, le fonctionnement des clefs restent mystérieux, à vous d'aller chercher sur internet plus de théorie.

Tout le cryptage RSA commence par seulement **deux nombres premiers** P et Q. On va construire tous les autres nombres à partir de ces deux là. Et tout de suite deux nouveau nombres N et M tel que

$$N = P.Q$$

$$M = (P-1)(Q-1)$$

Juste pour l'anecdote qui nous servira plus tard, le nombre N doit-être grand pour un bon niveau de sécurité. En effet, mathématiciens comme vous êtes, vous savez qu'un nombre peut se décomposer en un couple unique de deux nombres premiers. Autrement dit P et Q est **l'unique couple de nombres premiers vérifiant** $N = P.Q$. Donc en théorie si on connait N, on peut retrouver P et Q, et si on connait P et Q on connait toutes les clefs. Cependant dans la pratique, il est difficile de retrouver P et Q, le seul moyen et d'envoyer un ordinateur qui va tester toutes les possibilités... Voilà pourquoi N doit-être grand. En générale N comporte une centaine de chiffres !

Continuions notre chemin avec un autre nombre : C, tel que

$$C \textit{ soit premier avec } M$$

. Nous avons donc notre clef publique, elle est constituée de C et N.

$$ClefPublique = (C, N)$$

Et pour finir, le dernier nombre, mais aussi le plus compliqué ! Parmi les théorème d'arithmétiques que l'on a pris soin d'oublier, il existe celui de Bezout.

> Si deux nombres sont premiers entre eux, alors il existe une somme d'eux valant 1

C'est lui qui nous donnera notre dernier nombre : U tel que

$$C.U + M.V = 1$$

. Pour ceux qui se posent des question : V ne nous servira pas. De cela on fait notre clef privée constituée de U[1] et N soit

$$ClefPrivee = (U, N)$$

1.3 Crypter et Décrypter

Nous voila équipé de clefs pour crypter et décrypter, mais nous ne savons toujours pas les utiliser. Sachez que ce cryptage ne fonctionne que pour les nombres. Vous ne pouvez pas lui dire : "crypte moi la lettre A !", mais vous pouvez assigner à A un nombre est crypter celui-ci. Posons donc X le nombre que l'on veut crypter, et X' le nombre une fois crypté. Lorsque Amazon vous donne sa clef publique, pour crypter votre numéro de carte bancaire, il faut faire

$$X' = X^C mod(N)$$

Nous remarquons que les nombres C et N sont "publiques" donc connus de tous. Le fait que N doit-être grand est donc nécessaire pour une bonne sécurité, mais le nombre C peut rester faible. En effet il existe une infinité de nombres premiers avec C, la chance de retrouver M est faible.

Et enfin, lorsque Amazon reçoit votre message crypté, il lui suffit pour retrouver votre numéro de carte bancaire :

$$X = X'^U mod(N)$$

Vous avez remarqué, outre le changement entre X et X', le cryptage et décryptage est identique à une variable près.

Voilà, vous savez maintenant comment utiliser le cryptage RSA, mais je vous ai raconté beaucoup de mathématiques abstraites. Nous allons voir dans le suite, des algorithmes plus concrets qui se cachent derrière le fonctionnement du cryptage.

[1] U doit-être compris en 2 et M

Chapter 2

Les algorithmes

Nos ordinateur sont peut-être plus rapides qu'avant, ils ne sont pas plus intelligents pour autant. C'est donc à nous de leur donner un semblant d'intelligence. Nous allons donc voir des algorithmes pour tirer des nombres premiers, connaitre le PGCD, bref toutes ces choses qui ne représentent rien pour un ordinateur.

Pour recouvrir un langage de programmation universel, je parlerai en *int* pour instancier un nombre. Nous verrons par la suite les limites du *int* en JAVA.

2.1 Nombres premiers entre eux

On va commencer doucement, par un de nous tout premier algo appris : celui d'Euclide. Cette algo sert à déterminer le PGCD[1] de deux nombres. Dans notre cryptage, on a besoin de trouver C tel que C et M sont premiers. Nous allons donc calculer leur PGDC, s'il vaut 1, alors les deux nombres sont premiers entre eux.

L'algo d'Euclide prend deux nombres, tant que leur reste est différent de 0, on décale les nombres : le diviseur devient le nombre à diviser, le reste devient le diviseur.

Pour mieux comprendre, on va trouver le PGCD entre 49 et 14 : on commence par trouver le reste entre 49 et 14.

$$49 = 14 * 3 + 7$$

Puis on décale les nombres

$$14 = 7 * 2 + 0$$

Lorsque le reste vaut 0, le PGCD est le diviseur. Ici $PGCD(49, 14) = 7$

Il ne reste plus qu'à le transcrire sous forme d'algorithme.

```
int a; <-- premier nombre
int b; <-- second nombre
int r = a mod(b); <-- le reste de a/b
while(r != 0)
 a=b;
```

[1] le PGCD est le plus grand diviseur commun au deux nombres

```
b=r;
r = a mod(b);
```

```
retourne b; <-- on retourne le PGCD de a et b
```

Il suffira de tirer un nombre C au sort tant que le PGCD de C et M est différent de 1.

2.2 Trouver U

Nous avons vu le nombre U brièvement tel que $U * C + M * V = 1$. Mais même pour nous ce nombre n'a rien dévident. Nous allons nous appuyer sur l' algo d'Euclide un peu remanié pour le trouver. En fait, on va essayer de répartir le reste que l'on trouve dans l'algo d'Euclide en combinaison linéaire de C et M. Prenons C = 23 et M = 120 :

R =	(Euclide simple) =	C*	U+	M*	V
5	120 - 5*23	23	-5	120	1
3	23 -4*5	23	21	120	-4
2	5 -1*3	23	-26	120	5
1	3 -1*2	23	47	120	-9
0	2 -2*1				

L'algo se finit donc quand $R = 0$, et nous remarquons à l'avant dernière ligne $1 = 47 * 23 - 9 * 120$, on a donc $U = 47$. De plus les valeurs de R, U et V se trouvent par une suite récursive du second ordre, en prenant

$$q = \frac{R_{n+1}}{R_n}$$

, on a :

$$\begin{cases} R_{n+2} &= R_n - q.R_{n+1} \\ U_{n+2} &= U_n - q.U_{n+1} \\ V_{n+2} &= V_n - q.V_{n+1} \end{cases}$$

Il ne nous reste plus qu'a coder cet algo. Dans notre algo, $R_n = r$ et $R_{n+1} = rr$, de même pour U et V. On commence par initialiser les valeurs.

```
int r = C, rr = M;
int u = 1, uu = 0;
int v =0, vv = 1;
```

Ensuite on calcule q avant de décale les lignes du tableau.

```
q = rr/r;
r = rr;
u=uu;
v=vv;
```

Puis on calcule les nouvelles valeurs de rr, uu, et vv.

```
rr = r - q.rr;
uu = u - q.uu;
vv = v - q.vv;
```

Mais dans ce cas on a un problème... les valeurs de r, u et v viennent tout juste d'être changées, elles ne sont plus correctes.

On peut imaginer r et rr comme deux verres, on a besoin de mettre le contenu de r dans rr, mais également le contenu de rr dans r. Une seule solution utiliser un troisièmes verre (intermédiaire) que l'on nommera rs. Et on a donc notre algo pour trouver U:

```
public int atteindreU(int C, int M);
 int r = C, rr = M ;
 int u = 1, uu = 0;
 int v =0,  vv = 1;

 while(rr != 0) {
    q = rr/r;

    //On met le contenu de r dans le troisieme verre (rs)
    rs = r;
    vs =v;
    us = u;

    //On met le contenu de rr dans r, pour liberer rr
    r = rr;
    u=uu;
    v=vv;

    //On recalcule rr en prennant le contenu du troisieme verre
    rr = rs - q.rr;
    uu = us - q.uu;
    vv = vs - q.vv;
 }

 return u

}
```

Mais pour tout vous dire, ce U est peut-être incorrect. Oui il correspond bien à l'équation de bezout, mais il n'est peut-être pas compatible avec le cryptage RSA qui veut un U tel que $2 < U < M$. Par chance la valeur de U vérifie toujours l'équation de bezout, même si on l'additionne plusieurs fois avec M. Il faut donc remanier un peu notre algo pour contrôler la valeur de U.

```
public int atteindreU(int C, int M) {
 int r = C, rr = M ;
 int u = 1, uu = 0;
 int v =0,  vv = 1;

 while(rr != 0) {
    q = rr/r;

    //On met le contenu de r dans le troisieme verre (rs)
    rs = r;
```

```
        vs =v;
        us = u;

        //On met le contenu de rr dans r, pour liberer rr
        r = rr;
        u=uu;
        v=vv;

        //On recalcule rr en prennant le contenu du troisieme verre
        rr = rs - q.rr;
        uu = us - q.uu;
        vv = vs - q.vv;
    }

    // On remanit la valeur de U pour quelle soit conforme
    while(u<2) {
        u = u+M
    }

    return u;
}
```

Je vous l'avez dit que ce U n'avait rien d'évident, mais ça y est : on l'a !

2.3 Trouver un nombre premier

Les nombres premiers sont comme de la mauvaise herbe dans le champs des nombres entiers. De loin on ne fait pas la différence, il faut les examiner de près pour s'apercevoir qu'ils ne sont pas comme le reste. De plus ils apparaissent subitement en plein champs, sans que l'on sache prévoir l'emplacement suivant.

Le moyen le plus radical pour savoir qu'il n'est pas premier, c'est de montrer qu'il n'est divisible par aucun nombre. Si I est premier, alors il n'est divisible pas aucun nombre inférieur ou égal à $\sqrt{I}$. C'est barbare, et ça marche !

Mais je vous rappelle notre situation, nous avons deux nombres premiers à trouver aléatoirement : P et Q, en sachant que $N = P * Q$ doit-être le plus grand possible. Il faut donc utiliser un algorithme un peu plus subtil, car le premier prendra trop de temps.

Cette recherche des plus grands nombres premiers possibles a passionné les mathématiciens. Il existe donc plusieurs algorithmes pour vérifier si le nombre est premier, mais dès lors que le nombre devenait grand on a utilisé des algorithmes à *primalité probable*. Vous avez bien entendu : le nombre trouvé est probablement premier, on en est pas sûr... Mais rassurez vous cette probabilité est forte.

Pour ce faire nous allons utiliser le petit théorème de Fermat qui est :

> Si p est un nombre premier et si a est un entier non divisible par p, alors $a^{p-1} - 1$ est un multiple de p ou encore
>
> $$a^{p-1} - 1 \ mod(p) = 0$$

Nous allons nous en servir pour formuler : soit p un nombre aléatoire et a un certain nombre premier avec p. Si $a^{p-1}\ mod(p) = 1$, alors p est *probablement* premier. En effet la réciproque du théorème de Fermat n'est pas toujours vraie. Pour éviter les erreurs, il faut la tester avec plusieurs a différents, et nous aurons une réciproque *probablement* vraie.

Un problème persiste, les valeurs à mettre pour que a soit premier avec p. Afin d'éviter toutes erreurs, le plus simple est d'utiliser uniquement des nombres premiers connus pour a, et c'est ce que nous allons faire !

Il faut donc d'abord une liste de nombres premiers, la variable a va prendre successivement les valeurs de la liste. Pour que notre probabilité soit forte et éviter les erreurs, une liste de 10-15 nombres premiers est suffisante. On a donc

```
int premier[] = { 2, 3, 5, 7, 11, 13, 17, 23, 29, 31, 37, 41 };
```

Le petit théorème de Fermat doit donc donner 1 avec toutes les valeurs de a. On va donc placer chaque résultats dans une variable appelée "produit", tout simplement parce que nous allons multiplier chaque résultat entre eux. Ainsi si un résultat est autre que 1, la variable "produit" sera différente de 1.

Nous pouvons donc savoir si p est premier :

1. on test pour chaque valeurs de a le petit théorème de Fermat
2. on multiplie au fur et à mesure le résultat
3. on compare : si le produit final est 1, p est "probablement" premier
4. on répété l'algo en prenant $p = p + 1$, pour trouver un p premier

Ce qui nous donne le code suivant :

```
private int atteindrePremier(int p)
   {
        int premier[] = {
            2, 3, 5, 7, 11, 13, 17, 23, 29, 31,
            37, 41
        };
        int produit;
        do
        {
            p = p +1;
            produit = 1;

            for(int i = 0; i < premier.length; i++)
            {
                int nbrePremier = premier[i];
                produit = produit*( a^(p-1) mod (p) )
            }

        } while(produit != 1);
        return p;
}
```

Part II

Le Programme JAVA

Les maths sont en partie finies. Il est temps de mettre les mains dans le cambouis. Nous venons de voir le cryptage RSA et ses algorithmes mathématiques, il nous reste à le coder, pour construire une classe JAVA, qui soit la plus polymorphe possible. En effet le but ultime de cette seconde partie est de créer un classe qui va s'adapter à ses futurs programmes. Il faut donc que toute ces entrées et sorties soient "standard comme des int ou String. De plus on pourra modifier le niveau de cryptage à chaque création des clefs, ainsi que le nombre de coeurs utilisés lors du cryptage ou décryptage. Bref un beau programme ! Pour suivre la suite il est recommandé de **connaitre le langage JAVA**.

Chapter 3

Les limites du int

Depuis le début, nous utilisons ces fameux int dans nos algo pour plus de clarté. Mais le int est limité au niveau de la mémoire, et donc de la taille du nombre enregistré. Essayons le code suivant

```
int i =2147483647;
System.out.println(i);
i++;
System.out.println(i);
```

Normalement la console nous renvoie : 2147483647 et -2147483648... Alors soit pour JAVA $2147483647 + 1 = -2147483648$, soit le int i est reparti à zéro. Connaissant JAVA, on peut dire que le int est reparti à zero, 2147483647 est donc la limité maximale des int, et lorsque l'on augmente, le int repart à sa valeur minimale : -2147483648.

Même pour les long, il existe ce problème de taille maximale. Or dans le cryptage RSA nos nombres sont immenses de l'ordre de 10^{100}. Il faut donc se tourner vers un objet qui ne connait pas de limite, et cet objet est **BigInteger**. Comme son nom l'indique, c'est un "gros" integer.

Sa qualité est donc de ne pas avoir de limite, et donc de stocker n'importe quel nombre. Son défaut est surtout sa syntaxe, qui est bien plus lourde que celle des int. Voici un tableau pour conversion int/BigInteger.

int i = 2147483647	BigInteger i = new BigInteger("2147483647")
int i = 1	BigInteger i = BigIntegr.ONE
int c = a+b	BigInteger c = a.add(b);
int c = a-b	BigInteger c =a.soustract(b)
int c = a*b	BigInteger c =a.muliply(b)
c ==a	c.compareTo(a) == 0
c < a	c.compareTo(a) ==-1
c > a	c.compareTo(a) == 1

Je vous laisse aller voir en Annexe nos précédents algos convertis avec des BigInteger.

Chapter 4

Utilisation du mots de passe

Nous arrivons à l'une de plus grosse problématiques que je me suis posée : Comment gérer le mot de passe ? Il faut donner le choix du mot de passe à l'utilisateur, mais il ne faut pas que le logiciel enregistre celui-ci ou la clef privée quelque part, ça serait trop dangereux.

4.1 Récupéré le clef privée par le mot de passe

Seul le mot de passe correct doit permettre de décrypter le message, mais l'ordinateur ne l'a pas en mémoire. Il ne peut donc pas comparer le mot de passe donné par l'utilisateur au mot de passe correct. Le programme va donc dans tout les cas décrypter le message peu importe celui que l'on donne. Afin d'éviter de révéler le message secret avec un faux mot de passe, nous allons voir comment décrypter en fonction de celui-ci. Grâce à cette façon seul le vrai mot de passe produira un décryptage "complet", c'est à dire, renverra le message secret.

Le programme ne connait ni le mot de passe correct, ni la clef privé, mais il connait à tout moment la clefs privée, et le mot de passe donné par l'utilisateur. Nous devons reconstruire la clef privée (U,N) à l'aide de la clef publique (C, N) et du mot de passe. Dans la suite nous l'associerons à un nombre MDP, qui est unique pour chaque mot de passe.

Nous remarquons que nous connaissons déjà le nombre N, il ne reste plus que le nombre U. On aurait pu tout simplement dire $MDP = U$. Mais U étant lié aux nombres C et M, l'utilisateur ne pourra pas choisir son mot de passe. Il faut donc regarder plus loin, et justement aux nombre C et M. On connait déjà C grâce à la clef publique, il ne reste plus qu'à connaitre M pour retrouver la clef privée. Mais là encore M dépend de deux nombres P et Q, et l'utilisateur n'a toujours pas le choix du mot de passe. On remonte donc à P et Q. Pour rappel P et Q sont deux nombres premiers tel que $N = P * Q$. Donc si on connait Q, on peut retrouver P par $P = \frac{N}{Q}$.

Dans ce cas on peut prendre Q comme étant le MDP, soit

$$Q = MDP$$

. Puis à l'aide du nombre N présent dans la clef publique, on retrouve P, puis

M :

$$P = \frac{N}{Q}$$

$$M = (Q-1)(P-1)$$

Ensuite toujours avec la clef publique, on récupère C, et par conséquent U, car

$$C * U + M * V = 1 \; et \; 2 < U < M$$

On a donc réussit à retrouver les nombres U et N formant la clef privée, avec seulement les nombres C et N présents dans la clef publique ainsi que le mot de passe. On remarque au passage que nos clefs sont dorénavant liées au mot de passe choisit par l'utilisateur. Et seul le mot de passe correct permet de retrouver la bonne clef privée.

4.2 Transformer le mot de passe en un nombre

Nous avons vu le précédemment le nombre MDP, qui est l'image du mot de passe de l'utilisateur, soit une chaine de caractères. Il existe plusieurs méthodes pour changer des lettres en chiffres, la plus simple est de ce référer au code ASCII. Ensuite pour transformer cette suite de nombres en UN nombre, il existe aussi plusieurs façons. La seule condition est de retourner un grand nombre premier pour plus de sécurité. Personnellement j'ai choisi de multiplier chaque nombre ASCII entre eux pour former le MDP, et d'utiliser l'algorithme de primalité pour atteindre le nombre premier. Voici le programme correspondant :

```
BigInteger Q = BigInteger.ONE;
for(int i = 0; i < mdp.length(); i++)
    Q = Q.multiply(BigInteger.valueOf(mdp.charAt(i)));

Q = reachPremier(Q);
```

4.3 Créer les Clefs

Cette section vous dit peut-être déjà quelque chose, car je l'ai traitée dans la partie 1. Mais nous avons désormais un moyen de créer ces clefs avec le mot de passe choisit par l'utilisateur. Et comme c'était trop simple, je me suis permis de rajouter une autre variable à prendre en compte que l'on appellera *degrés de cryptage alias DC*. En résumer notre méthode *creatClefs* prend en paramètres le String mot de passe et le int DC. Puis elle nous renvoie la clef publique, soit le String de C et le String de N, tous les deux rangés dans un tableau.

```
public String[] creatClefs(String mdp, int DC) { }
```

La clef publique est jalousement gardée par notre class, il ne faut surtout pas qu'elle puisse y sortir !

Ensuite on tire au hasard le nombre P. Pour cela on demande un nombre à l'objet *Random()*. Puis on choisit sur combien de byte le nombre tiré sera rangé. Dans notre cas le nombre de byte est le même nombre que DC, ainsi plus on augmente le degré de cryptage, plus le nombre P sera grand, plus le cryptage

sera sécurisé. Dans mon programme, on peut choisir DC tel que $80 < DC < 120$ à titre d'exemple. Il ne reste plus qu'à envoyer P dans la méthode *BigInteger reachPremier(BigInteger b)*, que nous avons créé dans la partie I, et qui nous renverra la valeur du nombre premier le plus proche du P spécifié.

```
P = new BigInteger(DegresChiffrage, new Random());
P = reachPremier(P);
```

Ensuite il faut fabriquer Q à l'aide du mot de passe. Pour cela j'ai suggéré de multiplier entre eux chaque caractère par le nombre ASCII. Sachez que le nombre inscrit d'un caractère peut se trouver par *BigInteger.valueOf(char c)*. Il ne faut pas oublier de transformer Q en nombre premier.

```
for(int i = 0; i < mdp.length(); i++)
    Q = Q.multiply(BigInteger.valueOf(mdp.charAt(i)));

Q = reachPremier(Q);
```

Voilà la suite on la connaît, il faut multiplier, diviser, trouver U... On a juste à mettre bout à bout tous nos précédents algo vus. Vous remarquerez que pour tirer C au sort je divise CD par 5. En effet C est un nombre publique, il n'a pas besoin d'être le plus grand possible, j'ai donc arbitrairement choisi de diviser la taille du nombre par 5.

```
// N = P*Q
N = P.multiply(Q);
// M = (P-1)(Q-1)
M = P.subtract(BigInteger.ONE).multiply(Q.subtract(BigInteger.ONE));

// On tire C au sort tant qu'il n'est pas premier avec M
do
    C = new BigInteger(DegresChiffrage / 5, new Random());
while(PGCD(C, M).compareTo(BigInteger.ONE) != 0);

// On trouve U
U = reachU(C, M);

// On ajoute C et N dans le tableau que l'on va retourner
String strClefPublic[] = {
    String.valueOf(C), String.valueOf(N)
};
return strClefPublic;
```

4.4 Les constructeurs de l'objet RSA

Notre objet RSA dispose de deux constructeurs : l'un avec la clef publique seule uniquement si on veut crypter des messages, l'autre avec le clef publique et le mot de passe si on veut crypter et décrypter des messages.

```
public RSA(String C, String N) {

    // Contructeur a appeler si on veut seulement crypter
```

```
}

public RSA(String C, String N, String mdp) {

    // On constructeur a appeler si on veut crypter et d crypter
}
```

Nous allons nous intéresser au second constructeur. Vous allez voir qu'il est presque semblable à la méthode *creatClef*. Comme dans cette méthode, on commence par convertir le mot de passe en Q.

```
BigInteger Q = BigInteger.ONE;
for(int i = 0; i < mdp.length(); i++)
    Q = Q.multiply(BigInteger.valueOf(mdp.charAt(i)));

Q = reachPremier(Q);
```

Le changement intervient maintenant, on ne vas plus tirer au sort P pour calculer N, mais on va retrouver P par la relation $P = \frac{N}{Q}$

```
P = N.divide(Q);
```

Ensuite, on repart sur le calcul de M et de U, mais cette fois il ne faut plus tirer C au sort.

```
M = P.subtract(BigInteger.ONE).multiply(Q.subtract(BigInteger.ONE));
U = reachU(C, M);
```

Nous avons donc maintenant un programme capable de créer les clefs à partir du mot de passe. Mais également de retrouver les clefs à partir du mot de passe ! Il ne nous reste plus qu'à crypter et décrypter.

Chapter 5

Crypter et décrypter

Maintenant que l'on sait construire et reconstruire nos clefs, il nous reste encore à crypter et décrypter nos données, mais pour cela il faut les convertir ! En effet le cryptage ne marche qu'avec des nombres et des grands, on doit donc utiliser ces fameux BigInteger. Mais l'utilisation des BigInteger doit rester dans la class, il plus facile pour nous de directement donner un String et de retourner le String crypter, que d'utiliser sans cesse ces horribles nombres.

5.1 D'un String à crypter à un tableau de BigInteger

Cette partie est très simple, la stratégie utilisée va être de découper le String en caractères. Puis nous allons convertir ce caractère en nombre, en prenant son code ASCII (comme pour le mot de passe), il nous reste plus qu'à ranger ce nombre dans un tableau.

```
// String a convertir
 String str = "hello world";
// tableau a remplir
 BigIntenger tab[] = new BigInteger[str.length()];

 for(int i =0 ; i<tab.length ; i++ ) {
    // On ranger le code ASCII du caractere dans le tableau
    tab[i] = BigInteger.valueOf(str.charAt(i));
 }
```

5.2 Crypter

Nous voilà au coeur du programme, mais paradoxalement c'est aussi le plus cours et le plus simple. Nous devons crypter chaque nombre un par un par la formule :

$$X' = X^C mod(N)$$

avec X' le nombre crypté, et X le nombre à crypter. Le code est donc des plus simple.

```
// Nombre a crypter
 BigInteger tab;

 for(BigInteger bi : tab)
      bi = bi.modPow(C,N);
```

Vous avez remarqué il existe une méthode qui fait exactement ce que l'on recherche *modPow()* à croire qu'elle nous attendait.

Si vous regardez le code source vous verrez que le programme change, il y a des *ProcessusDeCalculs* qui se glissent dans le code. C'est normal, le programme est conçu pour fonctionner en multi-Thread. On lui indique le nombre de coeurs à utiliser, il découpe le travail (le message à crypter ou à décrypter), et lance des Threads qui s'occupent juste de crypter ou décrypter le tableau qu'en leur donne. Je n'ai pas jugé cette algorithme nécessaire dans le cryptage RSA, c'est pour cela que je n'en parle pas. Mais libre à vous d'aller voir les commentaires dans le code source.

5.3 D'un tableau de BigInteger à un String crypté

Passons aux choses sérieuses, nous venons de chiffrer tout le tableau nombre par nombre, il faut renvoyer un String à l'utilisateur correspondant à son message crypté. Il faut que le message crypté prenne le moins de place possible, et bien sur que l'on puisse retrouver les nombres par la suite, si l'on veut décrypter.

Pour réduire la taille du message le meilleur moyen est de convertir chaque nombre en plusieurs octets, puis toujours avec la table ASCII, on fait correspondre chaque octet à un caractère. Par exemple le nombre présent dans un case du tableau est 17524, on le convertit en binaire : 0100 0100 0111 0111. Puis on regroupe par octet 0100 0100 et 0111 0111. Enfin on fait correspondre chaque octet à son caractère ASCII soit ici 'D' et 'w'. On rajoute au message crypter "Dw" et on passe à la case suivante.

Pour être sur de retrouver les nombres dans le message, c'est plus compliqué. La méthode suivante va ajouter au fur et à mesure des caractères à notre message. Mais tous les nombres n'aurons pas le même nombre d'octets, les nombres ne prendrons donc pas tous la même "taille" dans le String. Si notre String est : "uyffje¡!ùv24=(-", il est impossible de savoir qui est à qui.

Ce que l'on sait par contre, c'est que le nombre crypté est plus petit que N car $X' = X^C mod(N)$. On va donc remplir artificiellement chaque nombre pour que celui-ci fasse la même taille que N, qui est notre majorant. Par exemple avec notre 17524, si N prend 3 octets, alors on va écrire notre nombre 0000 0000 0100 0100 0111 0111. On lui ajoute donc un octet nul, qui ne modifiera pas le nombre, mais ajoutera un caractère nul au String. Ainsi chaque nombre prend le même taille dans le String à renvoyer.

```
// On recupere le nombre d'octet majorant
int ByteParNombre = N.toByteArray().length;

// On instancie le String a retourner
// Il s'agit d'un StringBuffer pour plus de rapidite lors du l'ajout de
// caractere
StringBuffer cryp = new StringBuffer("");
```

```
// Tableau contenant les nombres cryptes
BigInteger biTab;

    // On enumere chaque nombre
    for(int indexTab = 0; indexTab < biTab.length; indexTab++)
    {
        // On le decoupe en octet
        byte NbreTab[] = biTab[indexTab].toByteArray();

        // On le remplit artificiellement
        for(int Remplissage = ByteParNombre - NbreTab.length; Remplissage > 0;
        cryp.append('\0'); // ajout du caract re nul

        // On ajoute chaque octet au String sous forme d'un caractere
        for(int indexOctet = 0; indexOctet < NbreTab.length; indexOctet++)
        {
            byte b = NbreTab[indexOctet];
            cryp.append((char)b);
         }

     }



        return String.valueOf(cryp);
```

5.4 D'un String à décrypter à un tableau de BigInteger

Le message crypté à fait le tour du monde, il est allé dans les endroits les plus obscurs d'internet pour arriver à son destinataire. Le destinataire a renseigné le mot de passe et la clef publique pour pouvoir décrypter le message. Il faut donc retrouver notre tableau de nombres dans le String crypté. Mais comme nous avons soigneusement enregistré nos nombres dans la section précédente, nous savons qu'il faut lire les caractères par intervalle régulier. Pour connaitre l'intervalle, il suffit de connaitre la taille du nombre N (son nombre d'octets).

Voyons ensemble un exemple. On prend le message crypté "ePi0(D", et 3 comme la taille de N (N est donc enregistré sur 3 octets). On découpe le String principal en plusieurs Strings de la même taille que N, on a donc le String "ePi" et "0(D". Puis on convertit caractère par caractère le String en binaire, soit le nombre 0110 0101 0101 0000 0110 1001 et 0011 0001 0010 1000 0100 0100, ce qui donne en décimal 6639721 et 3221572. Notre tableau de BigInteger à crypter sera donc 6639721, 3221572. Et voici le code :

```
// On recupere la taille de N
int ByteParNombre = N.toByteArray().length;

// On instancie le tableau a decrypter
```

```
BigInteger tab[] = new BigInteger[cryp.length() / ByteParNombre];
int index = 0;

// On parcourt le String    intervalle r gulier de ByteParNombre caracteres
for(int indexStr = 0; indexStr < cryp.length(); indexStr += ByteParNombre)
{
      // On decoupe le String original par tranches de ByteParNombre
      // caracteres puis on met chaque carctere dans un tableau
      char charTab[] = cryp.substring(indexStr, indexStr +
                        ByteParNombre).toCharArray();

      // On transforme chaque caract re en son equivalent en byte ASCII
      byte byteTab[] = new byte[charTab.length];
      for(int a = 0; a < byteTab.length; a++)
                byteTab[a] = (byte)charTab[a];

      // On ajoute un nouveau BigInteger genere par le tableau de bytes
      tab[index] = new BigInteger(byteTab);
      index++;
}
```

5.5 Décrypter

Il y a de fortes ressemblances dans le code entre crypter et décrypter, il faut dire qu'il n'y a qu'une variable qui change C pour crypter et U pour décrypter. Le code source reste aussi différent à cause des ces histoires de multi-Thread. Mais si vous avez compris le code pour crypter, ce code est quasiment identique. Pour ceux qui veulent juste de la simplicité le code est le suivant :

```
// Tableau a decrypter
 BigInteger tab[];

 for(BigInteger bi : tab)
    bi = bi/modPow(U,N);
```

5.6 D'un tableau de BigInteger au String décrypté

Et nous voici dans les dernières ligne de code, le moment où après tant d'efforts on a réussi à crypter, puis réussi à décrypter les nombres. Il ne manque plus qu'à retrouver le message original, celui que nous avons crypté. Nous devons donc transformer chaque nombre en son caractère ASCII, puis les ajouter les un à la suite des autres. Ce qui nous donne le code suivant :

```
 // j'utilise un StringBuffer pour plus de rapidite
 StringBuffer strbuf = new StringBuffer("");

 for(bigInteger bi : tab)
    strbuf.append( (char) bi.intValue() );
```

```
// On retourne le message decrypte
return String.valueOf(strbuf);
```

Chapter 6

Conclusion

Et bien, voilà un beau travail, pour un beau programme. Je vous remercie de m'avoir lu jusqu'au bout. N'hésitez pas à me donner votre avis sur **mon blog** : `http://2froblog.wordpress.com`. Pour ceux qui on repérer une faille potentielle dans le programme, merci de le préciser également. L'intégralité du logiciel avec toutes les sources et le programme java est sur **mon dossier github**: `https://github.com/FrancoisJ/Source_RSA.git`. Personnellement j'ai bien aimé ce projet, car c'est la première fois que je fais des algos "mathématiques" et surtout un programme un peu utile...