

# Déchiffre et Delettre Partie 2

## Chiffrement asymétrique (RSA)

RSA (Rivest, Shamir et Adleman).

Chiffrement avec clé publique / déchiffrement avec clé privée:

Les clés sont construites par le destinataire qui donne la clé publique à (aux) expéditeur(s).

Seule la clé privée permet le décodage.

Utilisation de grand nombre pour construire les clés (>512 ou 2048 bits).

RSA est basé sur le fait qu'il est très difficile de factoriser en 2 nombres 1ers, un nombre entier très grand. Même avec un ordinateur le calcul prend un temps "exponentiellement" long.

Ex : A partir de 15 il est facile de dire que  $5 \times 3 = 15$  mais donner moi les facteurs de 221 ( $17 \times 13$ ). Avec 283.189 ( $503 \times 563$ )

Avantages/Inconvénients :

Le temps de calcul pour retrouver la clé privée doit être supérieur au temps durant lequel le secret doit être conservé.

Nous ne sommes pas à l'abri d'un(e) mathématicien(e) "génial(e)" trouvant un algorithme rapide de factorisation.

Par contre RSA est adapté aux échanges fréquents et anonymes sur Internet.

Théorie :

Base du chiffrement/déchiffrement général :

Avec la clé publique composée d'un nombre 'e' et 'n' on chiffre ainsi :

$$C = M^e \text{ modulo } n$$

Avec la clé privé composée d'un nombre 'd' et du même nombre 'n' on déchiffre ainsi :

$$M = C^d \text{ modulo } n$$

Une analogie pourrait être que le récepteur des messages donnerait des cadenas au émetteurs potentiels mais garderait pour lui seul la clé ouvrant les cadenas.

Liens :

Wikipedia rsa : [https://fr.wikipedia.org/wiki/Chiffrement\\_RSA](https://fr.wikipedia.org/wiki/Chiffrement_RSA)

Pdf rsa: <http://culturemath.ens.fr/math/pdf/nombres/RSA.pdf>

Vidéos : <https://www.youtube.com/watch?v=6KfJXI-Kvws&t=413s> et suivants

Liste nombres 1ers : [http://compoasso.free.fr/primelistweb/page/prime/liste\\_online.php](http://compoasso.free.fr/primelistweb/page/prime/liste_online.php)

Conversion binaire,hexa,décimal : <http://sebastienguillon.com/test/javascript/convertisseur.html>

## Construction de la clé publique :

|  |    |                                    |
|--|----|------------------------------------|
| Choix de 2 nombre premiers p et q.           | => | p=239 et q=293                     |
| Calcul de $n = p \times q$                   | => | $239 \times 293$<br>n=70 027       |
| Calcul d'un nombre $\phi = (p-1 \times q-1)$ | => | $238 \times 292$<br>$\phi=69\ 496$ |
| et d'un nombre e qui doit être               |    |                                    |
| 1/ > à p et q                                |    |                                    |
| 2/ être premier par rapport à $\phi$         |    |                                    |
| soit $(\text{PGCD}(e,\phi)=1)$               | => | e=295                              |

nb : Il faut un algo. pour le calculer (voir plus avant).

On a donc la clé publique (e,n) => **e=295 et n=70 027**

### Variables de départ et calculs intermédiaires

p=239 et q=293

n=70 027

$\phi=69\ 496$

Clé publique

**e=295 et n=70 027**

Note :

On emploi le terme "chiffrer" plutôt que coder ou crypter pour bien faire la différence avec un format (coder) et indiquer qu'on transforme une suite de chiffres (binaires) en une autre suite de chiffres.

n doit être plus grand que le nombre que l'on veut chiffrer. Ex : si on veut chiffrer une suite de nombres sur 16 bits alors n doit être plus grand que  $2^{16}$  soit 65536.

**En ligne** (ou sur mon pc)

Avant la pratique faire la démonstration expliquée du programme terminé...

## Pratique :

Note :

Dans le prog. principal, on a

```
var p,q;
```

```
p=239 ; q=293 ;
```

```
const by=2 ; // On va regrouper les caractères du message clair par 2. POURQUOI ?  
// pour ne pas avoir le même code qui se répète.
```

```
var n = p*q ;
```

```
var phi= phiOf (p,q) ;
```

```
var e=primeOf(phi);
```

```
// On affiche la clé publique créée :
```

```
log("=> Clé publique = (" +e+" , "+n+"");
```

On écrit les fonctions :

```
/**
```

```
* @param pp      First integer
```

```
* @param qq      Second integer
```

```
* @return Phi
```

```
*/
```

```
function phiOf (pp,qq) {  
    return (pp-1)*(qq-1) ;
```

```
}
```

ET

```
/**
```

```
* Creation of number e used to cipher
```

```
* @param phii    (p-1) * (q-1)
```

```
* @param min     Start value for e
```

```
* @param pp      first prime number
```

```
* @param qq      second prime number
```

```
* @return the e number
```

```
*/
```

```
function primeOf(phii,min=0) {  
    min=Math.abs(min); min+=2;min+=2;  
    if (min>=phii) error("Minimum for e : "+min+" is too great !");  
    for (var ee=min;ee<phii;ee++) {  
        if (GCD(phii,ee)==1) break;  
    }  
    return ee;
```

```
}
```

Note :

Dans le prog. principal :

```
// On vérifie que la valeur d'un bloc de 'by' octets est plus petit que n
```

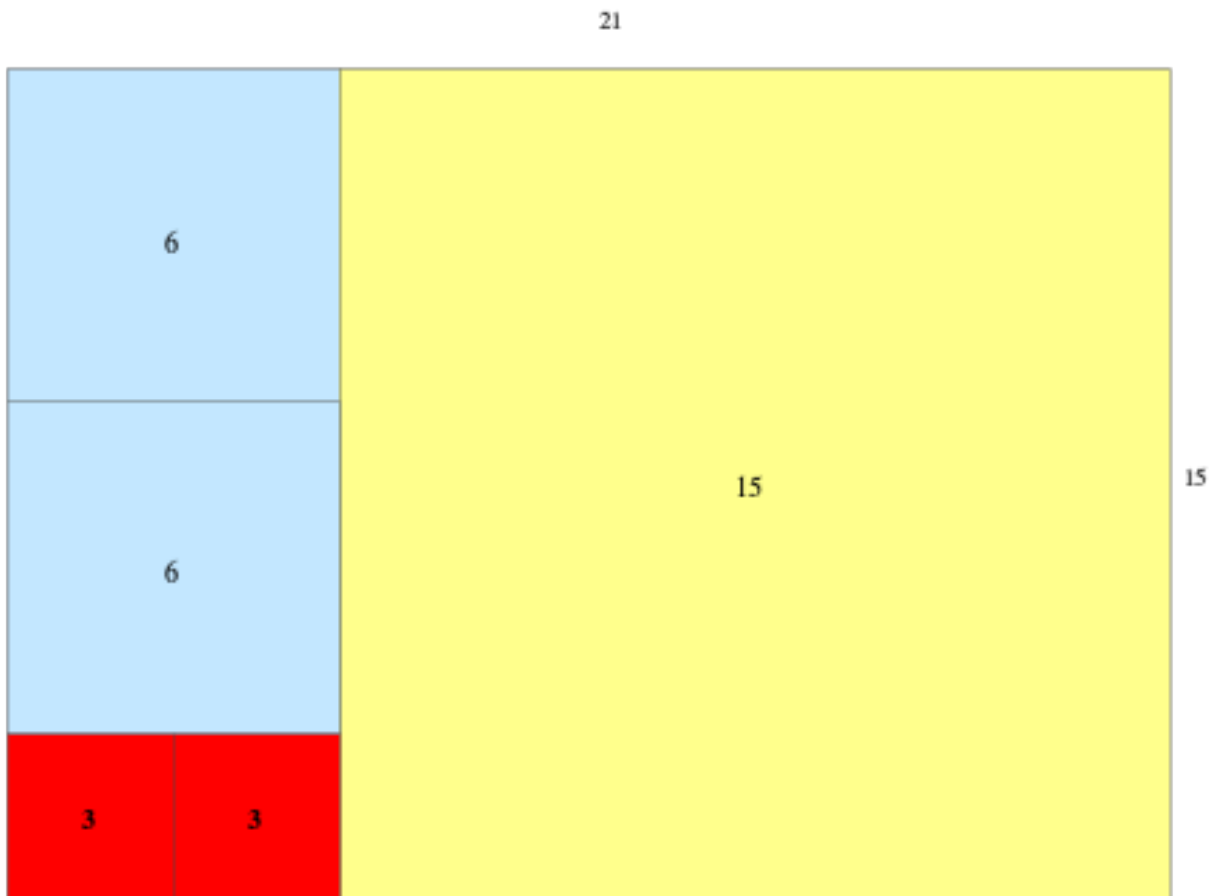
```
if (Math.pow(256,by)>n) error("Public key's n="+n+" too small !");
```

avant de revenir à la pratique ... quelques explications sur GCD(phii,ee)

## Théorie

explication de l'algo de GCD()

[https://fr.wikipedia.org/wiki/Algorithme\\_d%27Euclide](https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide)



$$21 \% 15 = 6$$

$$15 \% 6 = 3$$

$$6 \% 3 = 0$$

$$\Rightarrow \text{PGCD} = 3$$

21

5

1:

16

```

var r=a%b;
  while (r!=0) {
    a=b;
    b=r;
    r=a%b;
  }
return b;

```

```

a=21 ; b=16 ; r=5

```

```

?(r!=0)

```

```

  a=16

```

```

  b=5

```

```

  r=1

```

```

?(r!=0)

```

```

  a=5;

```

```

  b=1

```

```

  r=0

```

```

?(r!=0)

```

```

return 1

```

## Pratique :

```
/**
 * Compute the Greatest Common Denominator between a and b
 * @param a First integer
 * @param b Second integer
 * @return the GCD
 */
function GCD (a,b) {
    var r=a%b;
    while (r!=0) {
        a=b; b=r; r=a%b;
    }
    return b;
}
```

On teste la création de la clé publique par Alice

# Création de la clé privée 'd' :

## Théorie au tableau :

calcul du nombre 'd' :

On calcul à partir de  $\varphi$  , e

le 1er nombre **d** supérieur à e et inférieur à  $\varphi$  (par une boucle)  
qui vérifie :

$(e * d) \% \varphi$  est égal à 1 ... Par un petit algorithme.

Exemple

Avec :

p =239 et q=293

n =70027

$\varphi$  =69496

La clé privé est :

**d=66237 et n=70027**

*Je ne fais pas ici la démonstration mathématique*

*Voir*

[https://fr.wikipedia.org/wiki/Chiffrement\\_RSA](https://fr.wikipedia.org/wiki/Chiffrement_RSA) ou

<http://culturemath.ens.fr/maths/pdf/nombres/RSA.pdf>

On note juste qu'il est quasiment impossible de retrouver le dividende d'une division si on a que le diviseur et le reste.

Ex : si on a :  $c=m^3 \% 100 = 11$  impossible de retrouver m avec uniquement 11 et 3.

par contre en faisant :  $11^7 \% 100$  on a **71** et effectivement m était bien égal à 71 .

vérifions que  $71^3 \% 100 = 11$ .

## Pratique

Dans le prog. principal, on a :

```
var d=dCompute(e,phi);
```

```
/**
 * Compute of number d of private key (d,n)
 * @param ee      Number e (of public key)
 * @param phii    (p-1) * (q-1)
 * @return the d number
 */
function dCompute(ee,phii) {
    // return extendedEuclide(ee,phii);
    return dComputeEasy(ee,phii);
}
```

On écrit la fonction *dCompute* en appelant *dComputeEasy()* que l'on va écrire.  
Mais avec une valeur de N importante, il faudra lui substituer un algo optimisé (Euclide étendu) qui est déjà écrit dans le source.

```
function dComputeEasy(ee,phii) {
    var min=3;
    for (var dd=min;dd<phii;dd++) {
        if ((ee*dd)%phii==1) break;
    }
    return dd;
}
```

Note :

Explication du théorème d'Euclide étendu

<https://www.youtube.com/watch?v=M7vOxKVLsVY>

<http://www.bibmath.net/crypto/index.php?action=affiche&quoi=complements/algoeuclid>

[https://fr.wikipedia.org/wiki/Algorithme\\_d%27Euclide\\_%C3%A9tendu](https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide_%C3%A9tendu)

On teste la création de la clé d...



# Chiffrement du message :

## Théorie au tableau

Rappel du principe :

Pour chaque groupe de 2 caractères (16 bits) ,  
si la valeur en clair est dans  
 $m$   
on calcul la valeur chiffrée  $c$  par  
$$c = (m^e) \% n$$
  
ou  $e$  et  $n$  sont les composants de la clé publique.

## Pratique :

```
// Bob a reçu (comme tout le monde) la clé publique,  
// Prisonnier dans une tour par un dragon il envoie un message à Alice...  
// il écrit son message à Alice  
var plainTxt="Help me. I'm lost!";  
  
// Les octets de son message sont regroupés 2 par 2 dans un tableau.  
var arrBy= stringToArrayBy(plainTxt,by);  
// On a déjà vu cette fonction dans le programme avec clé secrète  
// On va la regarder rapidement.  
// faire un essai avec by=1 et on verra que les caractères Identiques donnent des chiffrements  
identiques
```

```
puis  
// Chiffrement  
var cipherArr=cipher (arrBy,e,n);
```

On va écrire la fonction *cipher*

```
/**  
 * Ciphering with public key (e,n)  
 * @param arr      16 bits elements array with plain text  
 * @param ee       Number e of public key  
 * @param nn       Number n (p*q)  
 * @return 16 bits elements array with ciphered text  
 */  
function cipher (arr,ee,nn) {  
    var arrOut=[];  
    for (var i in arr) {  
        arrOut.push(powMod(arr[i],ee,nn));    // arrOut.push(Math.pow(arr[i],ee) % nn) ;  
    }  
    return arrOut;  
}
```

Au lieu de `Math.pow(arr[i],ee) % nn` ; qui ne marche plus si on choisit un nombre  $e$  grand,  
on va écrire la fonction `powMod()` qui donne le même résultat mais avec de exponentiations  
successives plus petites.

Avant quelques explications théoriques.

# Chiffrement/Déchiffrement du message :

## Théorie :

Pour chiffrer ou déchiffrer nous devrions élever le nombre  $m$  ou  $c$  à des puissances élevées dès que  $n$  ( $p \times q$ ) devient grand donc  $e$  et surtout  $d$ .

Nous devons utiliser une astuce pour éviter ça. (voir l'[exponentiation modulaire](#) que nous devons à [Bruce Schneier](#)).

L'algo est basé sur :

$$a \times b \% c = a \% c \times b \% c$$

Exemple : Calculer  $8^{26} \bmod 10$  sans calculer  $8^{26}$

On va décomposer 26 en puissance de 2 :

$$26 = 16 + 8 + 2$$

qui s'écrit donc en binaire

$$\begin{array}{ccccccc} & 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{array}$$

On va calculer le modulo 10 pour les nombres : 8 élevé à chacune des puissances de 2.

|          |                    |                          | <u>avec % 10</u>     | <u>avec % 100</u> |
|----------|--------------------|--------------------------|----------------------|-------------------|
| $8^1$    |                    |                          | modulo 10 = 8        | = 8               |
| $8^2$    | $= 8^1 \times 8^1$ | $\Rightarrow 8 \times 8$ | modulo 10 = <b>4</b> | <b>= 64</b>       |
| $8^4$    | $= 8^2 \times 8^2$ | $\Rightarrow 4 \times 4$ | modulo 10 = 6        | = 96              |
| $8^8$    | $= 8^4 \times 8^4$ | $\Rightarrow 6 \times 6$ | modulo 10 = <b>6</b> | <b>= 16</b>       |
| $8^{16}$ | $= 8^8 \times 8^8$ | $\Rightarrow 6 \times 6$ | modulo 10 = <b>6</b> | <b>= 56</b>       |

On va multiplier les résultats qui correspondent à la décomposition de 26 soit 2 8 16 dont les modulo obtenus sont respectivement **4 6 6** ; puis prendre le modulo 10

On a donc :

$$8^{26} \bmod 10 = 4 \times 6 \times 6 = 144 \bmod 10 = \mathbf{4} \qquad \qquad \qquad = \mathbf{44}$$

Vérifiez sur calculatrice :  $8^{26} \bmod 10$  est égal à 4 et  $8^{26} \bmod 100$  est égal à 44

On va écrire la fonction() JS de cet algorithme.

Jmd : [Afficher cette page en même temps que j'expliquerai powMod\(\)](#)

## Pratique :

```
/**
 * Cipher or uncipher one char
 * @param ic      input char (unciphered or ciphered )
 * @param ed      key          (e or d)
 * @param nn      Number n (p*q)
 * @return        output char (ciphered or unciphered)
 */
function powMod (ic,ed,nn) {
    var oc=1;
    ic=new BigNumber(ic);           // nothing
    nn=new BigNumber(nn.toString()); // nothing
    while(ed>0) {
        if (ed%2!=0) oc=ic.times(oc).mod(nn); // if (ed%2!=0) oc = (ic * oc) % nn ;
        ic=ic.times(ic).mod(nn);             // ic = (ic * ic) % nn ;
        ed=Math.floor(ed/2);
    }
    return oc.toFixed();              // return oc
}
```

Voir

<script type="text/javascript" src="js/bignumber.js"></script> dans index.html

et l'API :

On teste le chiffrement de Bob ...

Alice va déchiffrer le message ...

### Pratique :

Dans le prog. principal, on a :

```
// déchiffrement
```

```
var uncipherArr=uncipher(cipherArr,d,n) ;
```

```
// et
```

```
// Conversion de l'array déchiffré en text clair.
```

```
var unciphText=arrayByToString(uncipherArr,by);
```

La fonction *arrayByToString* déjà écrite, est l'inverse de *stringToArrayBy*.

On ne va écrire que la fonction *uncipher* après avoir regarder rapidement la fonction *arrayByToString* déjà été vue dans l'atelier précédent.

```
/**
 * Unciphering with private key (d,n)
 * @param arr      16 bits elements array with ciphered text
 * @param dd       Number d of private key
 * @param nn       Number n (p*q)
 * @return 16 bits elements array with unciphered text
 */
function uncipher (arr,dd,nn) {
    return cipher (arr,dd,nn) ; // on appelle cipher() avec le paramètre dd au lieu de ee.
}
```

Alice lit le message et décide sur son cheval blanc, de partir terrasser le dragon et délivré Bob !!

\*\*\*\* On teste \*\*\*