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 5x3=15 mais donner moi les facteurs de 221 (17x13). Avec 283.189 (503 x 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 :

$C = M^{e}$ modulo n

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

$M = C^{d}$ modulo n

Liens :
Wikipedia rsa :https://fr.wikipedia.org/wiki/Chiffrement_RSA
Pdf rsa: http://culturemath.ens.fr/maths/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
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 x q => 239x293 n=70 027
Calcul d'un nombre φ = (p-1 x q-1) => 238x292 φ=69 496
et d'un nombre e qui doit être
1/ > à p et q
2/ être premier par rapport à φ
soit (PGCD(e,φ)==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
φ=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.

Pratique :

Note :

Dans le prog. principal, on a

```
var p = 239 ;
var q = 293 ;
var n = p*q ;

var phi= phiOf (p,q) ;
var e=primeOf(phi,p,q);
var by=2 ;      // On va regrouper les caractères du message clair par 2. POURQUOI ?
                // pour ne pas avoir le même code qui se répète.

var len16=5; // long de la string hexa (16^len16 > n) (i.e. 16^5 = 1,048,576 > 70 027)
                // alors que 16^4 =65536 < 70 027)
                // cette string ne sert que pour la transmission.
                // On pourrait convertir en base64 ou un array en json.

// On affiche la clé publique crée :
log("Clé publique  = ("+e+" , "+n+")");
```

On écrit les functions :

```
/**
* @param pp         First integer
* @param qq         Second integer
* @return Phi
*/
function phiOf (pp,qq) {
        return (pp-1)*(qq-1) ;
}
```

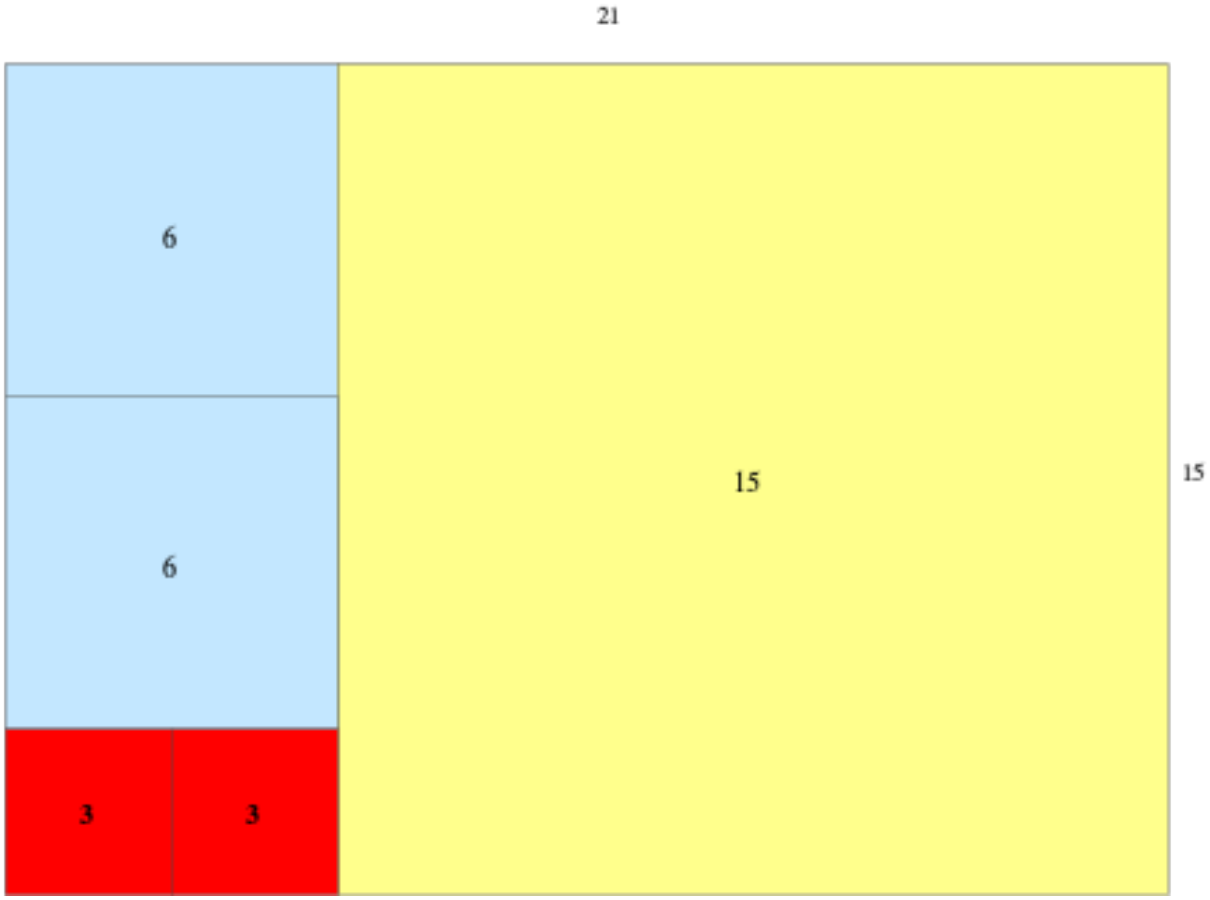
ET

```
/**
* Creation of number e used to cipher
* @param phii       (p-1) * (q-1)
* @param pp                 first prime number
* @param qq                 second prime number
* @return the e number
*/
function primeOf(phii,pp,qq) {
        var start=Math.max(pp,qq)+1;
        for (var ee=start;ee<phii;ee++) {
                if (GCD(phii,ee)==1) break;
        }
        return ee;
}
```

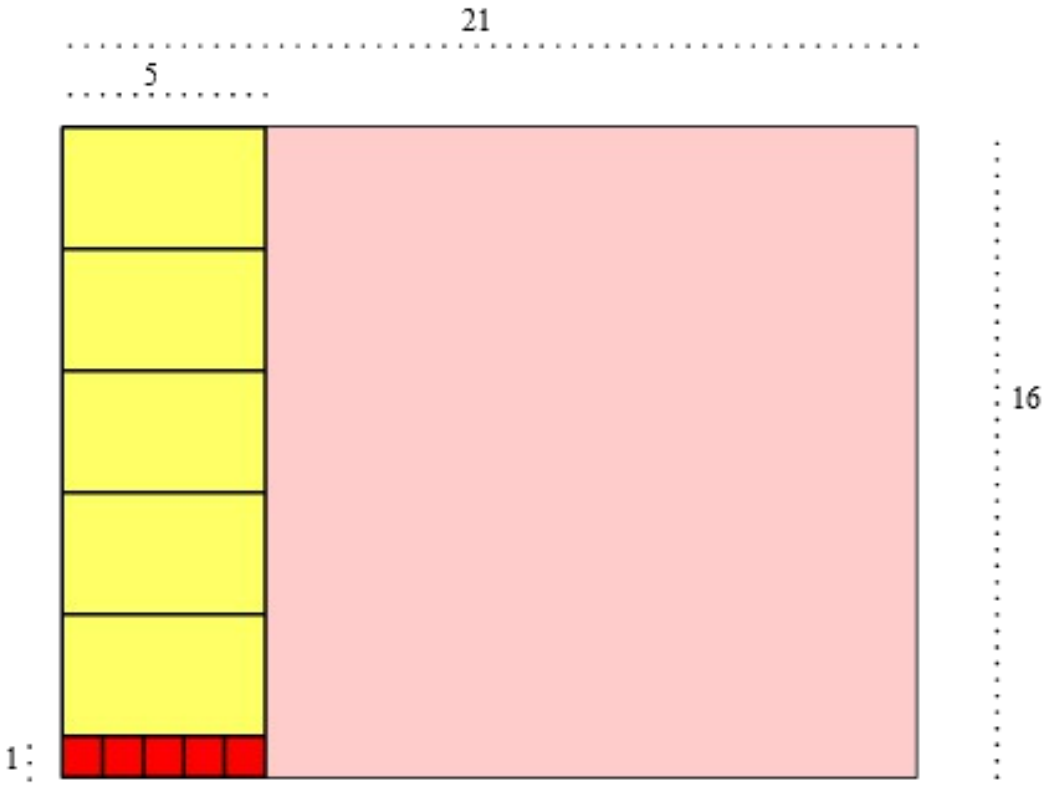
avant de revenir à la pratique … quelques explications sur GCD(phii,ee)

Théorie

explication de l'algo de GCD()
https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide



21%15 = 6
15%6 = 3
6%3 = 0 => PGCD = 3




```
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

Chiffrement du message :

Théorie au tableau

Rappel du principe :

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

Pratique :

Note :
Dans le prog. principal, on a :

```
// On vérifie que la valeur d'un bloc de 'by' octets est plus petit que n
if (Math.pow(256,by)>n) {
        var str="Public key n="+n+" too small !";
        log(str );
        throw(str);
}
// On vérifie que 16^len16 est plus grand que n donc que les valeurs chiffrées tiendront  bien
// dans la chaine hexa.
if (Math.pow(16,len16)<n) {
        var str="Hexa string length len16="+len16+" too small !";
        log(str );
        throw(str);
}
// 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 !";

// 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
// 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);

// puis
// Conversion du texte chiffrer en une string hexa (cadeau)
var cipheredTxt=toHexaString(cipherArr,len16);
// On aurait pu remplacer par une transmission en json ou base64 ou CSV. Ce serait juste un autre
format.
```

On va écrire la fonction *cipher* et étudier la fonction *toHexaString* :

Note :
On utilise l'API externe dans BigNumber.js
car JS avec Math.pow(n,e) ; ne fonctionne pas avec des grands nombres.

```
/**
* 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
*/
```

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

```
function cipher (arr,ee,nn) {
        BigNumber.config({ RANGE: 10000000 }); // 10 millions
        BigNumber.config({ POW_PRECISION: 0 }) ; // No limit of significative digits
        var arrOut=[];
        for (var i in arr) {
                var pc=new BigNumber(arr[i]);    // conversion en BigNumber
                var cc=pc.pow(ee).mod(nn);       // équivalent var cc=Math.pow(pc,ee) % nn ;
                arrOut.push(cc.toNumber());      // reconversion en nombre standard js
        }
        return arrOut;
}
```

Cette fonction est déjà écrite. Explication.

```
/**
* Creation of an hexa string before send ciphered text
* @param  arr             A text string
* @param  len16           Number hexa digits (len16=5)
* @return Ciphered text as hexa string
*/
function toHexaString (arr,len16) {
        var hxChars="0123456789abcdef"; // Hexa digits from 0 to f
        var hxStr="";
        for (var i in arr) {                    // for each value in array
                var v=arr[i]; var str16="";             // store in v
                while (v>0) {                           // while value hasn't been processed
                        var d=v%16;                     // extract right value from 0 to 15
                        var ch=hxChars.substr(d,1); // find corresponding hexa digit
                        str16=ch+str16;                 // store right to left the hexa digit
                        v=Math.floor(v/16);             // put in v the left rest
                }
                for (var j=str16.length;j<len16;j++) {
                        str16="0"+str16;                // zero left padding
                }
                hxStr+=str16;
        }
        return hxStr;
}
```

On teste le chiffrement par bob
et la création de la chaîne hexa

Création de la clé privée 'd' :

Théorie au tableau :

calcul du nombre 'd' :

On calcul à partir de φ , e , p et q
le 1er nombre **d** supérieur à p et q et inférieur à n (par une boucle)
qui vérifie :

(e * d) % φ est égal à 1 … Par un petit algorithme.

Exemple

Avec :
p =239 et q=293
n =70027
φ =69496
La clé privé est :
d=16255 et n=70027

Je ne fais pas ici la démonstration mathématique
Voir
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^3 % 100 = **11** impossible de retrouver rapidement avec 11 et 3, la valeur de c.
par contre en faisant : 11^7 % 100 on a **71** et effectivement c était bien égal à 71 .
vérifions que 71^3 % 100 = 11.

Pratique
Dans le prog. principal, on a :

```
// Alice aura crée sa clé privée (d,n) en même temps qu'elle a créé la clé publique...
// nous le faisons ici par :
var d=dCompute(phi,p,q,e);
```

On écrit la fonction *dCompute* . Il existe un algo qui optimise ce calcul (Euclide étendu) mais nous ne l'utiliserons pas ici car l'algo classique n'est pas très long à l'exécution.

```
/** Compute of number d of private key (d,n)
* @param phii        (p-1) * (q-1)
* @param pp          first prime number p
* @param qq          second prime number q
* @param ee          Number e (idem in public key)
* @return the d number
*/
function dCompute(phii,pp,qq,ee) {
        var start=Math.max(pp,qq)+1;
        for (var dd=start;dd<(p*q);dd++) {
                if ((ee*dd)%phii==1) break;
        }
        return dd;
}
```

On teste ...

Déchiffrement du message :

Théorie :

Pour déchiffrer nous devrions élever le nombre chiffré à la puissance d alors que d devient très grand dès que les nombres p et q augmentent.
Nous devons utiliser une astuce pour éviter ça. (voir l'exponentiation modulaire que nous devons à Bruce Schneier).
L'algo est basé sur :

a x b % c = a%c x b%c

Exemple : Calculer 8^{26} mod 10 sans calculer 8^{26}

On va décomposer 26 en puissance de 2 :

26 = 16 + 8 + 2

qui s'écrit donc en binaire

16	8	4	2	1
1	1	0	1	0

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

			avec % 10	avec %100
8^1			modulo 10 = 8	= 8
8^2	= 8^1 x 8^1	=> 8 x 8	modulo 10 = **4**	**= 64**
8^4	= 8^2 x 8^2	=> 4 x 4	modulo 10 = 6	= 96
8^8	= 8^4 x 8^4	=> 6 x 6	modulo 10 = **6**	**= 16**
8^{16}	= 8^8 x 8^8	=> 6 x 6	modulo 10 = **6**	**= 56**

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} mod 10 = 4 x 6 x 6 = 144 modulo 10 = 4 = 44

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

On va écrire la fonction de cet algorithme.

Jmd : Afficher cette page en même temps que j'expliquerai powMod()

Pratique :

Dans le prog. principal, on a :

// Conversion de la string hexa en array
var decArr=toDecArray(cipheredTxt,len16);
// (qui au passage est égal à l'array chiffré de Bob)
La fonction *toDecArray* déjà écrite, est l'inverse de *toHexaString*

// déchiffrement
uncipherArr=uncipher(decArr,d,(p*q)) ;

// et
// Convertion de l'array déchiffré en text clair.
var unciphText=arrayByToString(uncipherArr,by);
La fonction *arrayByToString* déjà écrite, est l'inverse de *stringToArrayBy.*

Voir rapidement .

```
/**
* Convert hexa string ciphered text into array
* @param  str         hexa string
* @param  len16     Number hexa digits (len16=5)
* @return Ciphered array
*/
function toDecArray (str,len16) {
        var arr=[];
        for (var i=0;i< str.length ; i+=len16) { // i target the first left digit of hexa string
                var str16=str.substr(i,len16);        // get the hexa string of 5 digits
                var v=Number('0x'+str16) ;            // convert to decimal
                arr.push(v);                          // store in returned array
        }
        return arr;
}
```

On ne va écrire que la fonction *uncipher* car la fonction *arrayByToString* a 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) {
        var arrOut=[];
        for (var i in arr) {
                arrOut.push(powMod(arr[i],dd,nn)); // arrOut.push(Math.pow(arr[i],dd) % nn) ;
        }
        return arrOut;
}
/**
* 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);
        while(ed>0) {
                if (ed%2!=0) oc=ic.times(oc).mod(nn);    // (ic * oc) % nn
                ic=ic.times(ic).mod(nn);                 // ic * ic
                ed=Math.floor(ed/2);
        }
        return oc;
}
```

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

**** On teste ***