Pour avoir la dernière version aller sur
https://github.com/flashline/Atelier-crypto

et cliquer sur
download zip

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.

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.

Utilisation de grands nombres pour construire les clés (>512 ou 2048 bits).

RSA est basé sur le fait qu'il est très difficile de factoriser un nombre entier très grand, en 2 nombres premiers. 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 les facteurs 1ers de 283189 (503 x 563) nécessite déjà d'écrire un petit programme. Pour un nombre de plusieurs centaines de chiffres cela peut prendre des millions d'années aux plus rapides des ordinateurs actuels. (le terme 'actuels' est important).

Avantages/Inconvénients :
Le temps de calcul pour retrouver la clé privée (dépendant donc de la longueur de cet clé) 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 car il ne nécessite pas d'échange de clé préalable.

Principe de base du chiffrement/déchiffrement :
Avec la clé publique composée d'un nombre 'e' et 'n' on chiffre ainsi :
$C = M^e$ modulo n (C est le reste de la division de M^e par n)
Avec la clé privé composée d'un nombre 'd' et du même nombre 'n' on déchiffre ainsi :
$M = C^d$ modulo n

Liens utiles :
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 suivantes.
Liste nombres 1ers jusqu'à 1000 milliard :
http://compoasso.free.fr/primelistweb/page/prime/liste_online.php
Conversion binaire,hexa,décimal : http://sebastienguillon.com/test/javascript/convertisseur.html
Conversion ascii<=>binaire<=>décimal :
http://ourhouzz.site/ascii-table-decimal-binary-hexadecimal/

Construction de la clé publique :

Exemples pris dans index.js :

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 Phi = (p-1 x q-1)	= 238x292	phi =69 496

et d'un nombre e qui doit être

1/ supérieur à 2

2/ premier par rapport à phi (soit e et phi n'ont aucun diviseur commun sauf 1)

soit (PGCD(e,phi)==1) => e=295

nb : Il n'y a pas d'expression mais un algorithme pour calculer e .

On a donc la clé publique (e,n)

e=295 et n=70 027

NB :

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.

Notion importante qui n'est pas qu'une question de vocabulaire :

On emploi le terme "chiffrer" plutôt que coder ou crypter.

Coder c'est stocker ou afficher une même donnée selon différents formats (texte ou nombre binaire,décimal, etc), en les groupant par 2, 3, etc ou pas. Bref une donnée codée reste une donnée "clair".

Chiffrer c'est remplacer une donnée numérique par une autre qui pourra ensuite être déchiffrée.

Crypter c'est comme chiffrer mais sans possibilité de retrouver le message clair. (Ex : le hashage).

Se rappeler qu'un caractère est déjà codé en un nombre dans un ordinateur. C'est le code ascii.

Ex :

à 'A' correspond le nombre **65**, à 'B' le nombre **66**. Soit 01000001 et 01000010 en binaire.

Ce qui regroupés donnent 0100000101000010. Ce nombre binaire (sur 16 bits) est égal à **16706** en décimal.

On peut dire que 'AB' et 16706 sont 2 représentations d'une seule et même chose. Il n'y a pas chiffrement.

On regroupera les octets par 2 ou 3, etc, dans notre programme index.js pour éviter de retrouver plusieurs fois un même nombre dans la liste des nombres chiffrés. (Voir support.part1.pdf -1ère page- pour comprendre les risques d'un tel cas)

En ligne

http://www.pixaline.net/intra/Atelier-2019/crypto/done/rsa/ voir le fonctionnement du programme terminé avec N = 97902343182121179204054 7

soit une longueur de 24 chiffres ; soit + ou – une clé de 70 bits.

Le programme que vous avez écrit durant l'atelier doit être complété ou remplacé par celui dans **done/rsa/js/index.js** pour pouvoir tester de plus grands nombres comme p=989456129273 et q=989456129339.

Pratique :

Note :

Dans le prog. principal, on a

```
const MAX_POW=10000;
const START_E= 788 ;      // On utilisera ces constantes plus tard

var n,phi,e,d,plainTxt,arrBy,cipherArr;  // variables utilisées
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. Pour d'autres p et q on
                      // regroupera jusqu'à 6 caractères.

var n   = computeN(p,q);
var phi = phiOf (p,q) ;
var e   = primeOf(phi,START_E);

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

// 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 !");
```

On écrit les fonctions :

```
/**
* @param pp         First integer
* @param qq         Second integer
* @return           N part of public key (p*q)
*/
function computeN (pp,qq) {
        if (pp*qq<1000000000000000) // <=1 000 000 000 000 000 soit maximum 15 chiffres
                return pp*qq;
        else    return new BigNumber(pp.toString()).times(new BigNumber(qq.toString())).toFixed() ;
}
```

Noter dans index.html

```
<script type="text/javascript" src="js/bignumber.js"></script>
```

Pour détailler l'API BigNumber :

https://mikemcl.github.io/bignumber.js/

```
/**
* @param pp          First integer
* @param qq          Second integer
* @return Phi
*/
function phiOf (pp,qq) {
    if ((pp-1)*(qq-1)<100000000000000) return (pp-1)*(qq-1); //  pas plus de 14 chiffres
    else return new BigNumber(pp.toString()).minus(1).times(new
    BigNumber(qq.toString()).minus(1)).toFixed();
}
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); if (min<3) min=3; var gcdPhi;
     if (min>=phii) error("Minimum for e : "+min+" is greater than phi : "+phii+" !");

     // On choisit ci-dessous la fonction à utiliser selon la taille de phi.
     if (phi<100000000000) var gcdPhi=easyGCD ;  // 11 digits max
     else gcdPhi=GCD ;

     for (var ee=min;ee<phii;ee++) {
               if (gcdPhi(phii,ee)==1) break;
     }
     return ee;
}
```

Avant d'écrire la fonction suivante easyGCD() voir quelques explications en ouvrant *easyGCD.pdf.*

Pratique :

```
/**
* Compute the Greatest Common Denominator between a and b
* @param a  First integer
* @param b  Second integer
* @return the GCD
*/
function easyGCD (a,b) {
      while (a != b) {
            if (a > b)       a-=b ;  // log("a>b => a=a-b : "+a+"-"+b+" ="+(a-=b)); //
            else             b-=a;  // log("a<=b => b=b-a: "+b+"-"+a+" ="+(b-=a)); //
      }
      // log("a=b=pgcd="+a);
      return a;
}
```

NB : En enlevant les commentaires on peut afficher la progression des valeurs ; ce qui aide à la compréhension de l'algorithme.

La fonction ci-dessous GCD() est déjà écrite mais voir une explication géométrique de l'algorithme (dit d'Euclide) en ouvrant gcd.pdf.

```
/**
* 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;
}
```

Tester la création de la clé publique par Alice...

Création de la clé privée 'd' :

Théorie :

calcul du nombre 'd' :

A partir de **phi** et e on calcul
le nombre **d** supérieur à e (quand e est petit par rapport à phi) et inférieur à **phi**
par une boucle qui vérifie :

(e * d) % phi est égal à 1

Dans notre exemple

Avec :
phi = 69496 et e = 295

La clé privé est :

d=66237 et n=70027

Pour une 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 de connus.

Ex : si on a : $c=m^3 \%100 = \mathbf{11}$

il très difficile de retrouver **m** avec uniquement 100, 11 et 3 (donc sans connaître m^3)

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

vérifions : $71^3 \% 100 = 11$.

Pratique

Dans la fonction alicePrivate() , 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) {
        if (phi<1000000)     return dComputeEasy(ee,phii);
        else return extendedEuclide(ee,phii);
}
```

On écrit la fonction *dCompute* en appelant dComputeEasy() que l'on va écrire.
Mais avec une valeur de N importante, il faut lui substituer l'algo optimisé (Euclide étendu) soit la fonction extendedEuclide() déjà écrite.
Je n'expliquerai pas ici cet algorithme. Je dirai juste pour les matheux que l'un des coefficients de Bezout donne d et que si il est négatif il faut lui ajouter phi.

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

Note :
Explication mathématique 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

Tester la création de la clé d...

Chiffrement du message :

Théorie (Rappel) :

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 d'un dragon dans une tour... il envoie un message à Alice...

// Dans la fonction bobSend() , on a :

var plainTxt="**Help me Alice. I'm lost!**";

// Les octets de son message sont regroupés 2 par 2 dans un tableau.
var arrBy= stringToArrayBy(plainTxt,by); // Déjà écrite

NB : faire un essai avec by=1 ; 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) {
                if (ee<MAX_POW) {
                        arrOut.push (new BigNumber(arr[i].toString()).pow(ee).mod(nn).toFixed()) ;
                }
                else arrOut.push(powMod(arr[i],ee,nn));
        }
        return arrOut;
}
```

Au lieu de Math.pow(arr[i],ee) % nn ; qui ne marche qu'avec de très petits nombres e et arr[i], on utilise l'API BigNumber qui fait la même chose sans boguer.

Mais si ee >= 10000 on utilise la fonction powMod() qui donne le même résultat mais avec des exponentiations et MODulo successifs sur de petits nombres, cela va beaucoup plus vite. (Voir explication page suivante).

Quelques explications théoriques pour comprendre l'algorithme de powMod() utilisée pour le Chiffrement/Déchiffrement du message :

Théorie :

Pour chiffrer ou déchiffrer nous devrions élever le nombre m ou c à des puissances élevées avant le calcul du modulo qui seul nous intéresse.

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 donc en particulier a x a % c = a%c x a%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 tous les '8 élevé à chacune des puissances de 2'.
Noter qu'à partir de la 2ème ligne, on reprend le résultat de la ligne précédente.

			avec % 10	avec % 100 c'est moins monotone
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 modulos 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

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

Voir la fonction implémentant cet algorithme powMod() sur la page suivante...

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;
    ed=new BigNumber(ed.toString());                    // L'équivalent en JS sans BigNumber :
    ic=new BigNumber(ic.toString());
    nn=new BigNumber(nn.toString());
    while(ed.gt(0)) {                                    // while(ed>0) {
            if (!(ed.mod(2).eq(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=ed.dividedToIntegerBy(2) ;                // ed=Math.floor(ed/2);
    }
    return oc.toFixed();                                 // return oc
}
```

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

et l'API sur https://mikemcl.github.io/bignumber.js/

Tester le chiffrement par Bob ...

Alice va déchiffrer le message ...

Pratique :

Dans la fonction aliceReceive() , on a :

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

// et

// Convertion de l'array déchiffré en chaîne de caractères clair.
   var unciphText=arrayByToString(uncipherArr,by);
```

La fonction *arrayByToString()* déjà écrite (C'est cadeau !) est l'inverse de *stringToArrayBy()*.

On va écrire la fonction uncipher.

```
/**
* 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.
}
```

Noter que le chiffrement étant similaire au déchiffrement (seul les éléments e ou d diffèrent), les clés publique ou privée pourraient être inversées du moment qu'on en garde une secrète.

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

**** On teste tout le programme ***
**** puis on continue à tester avec ***
**** des valeurs différentes pour p,q et by ***