CIMPA Kinshasa 2018 - TD RSA

TP : le cryptosystème RSA

1. Mise en oeuvre sur SageMath : un exemple simple

Choisir deux petits nombres premiers distincts et affecter ces valeurs respectivement à p et q dans Sage.

```
p = 23
q = 29
```

Vérifier à l'aide de Sage que les nombres p et q sont bien premiers.

```
p.is_prime()
```

True

```
q.is_prime()
```

True

Remarque : l'aide de la commande *is_prime* indique que, par défaut, cette fonction exécute un test déterministe de primalité. En l'appelant par is. prime(Proof = False), c'est un test de pseudoprimalité qui est effectué. On peut aussi directement demander un test de pseudoprimalité par la commande *is_ pseudoprime*.

Calculer pq et l'affecter à n.

```
n = p*q
```

```
n
```

667

Dans Sage, la fonction qui calcule l'indicatrice d'Euler φ est *euler_phi*.

```
euler_phi(n)
```

616

On affecte ce résultat à une nouvelle variable, qu'on choisit de nommer *phi.*

```
phi = euler_phi(n)
```

Trouver un entier $e \in Z$ tel que pgcd(e, φ(n)) = 1.

Il s'agit de prendre un nombre n'ayant aucun facteur premier commun avec $\varphi(n)$. Pour cela, on peut s'aider par exemple de la factorisation de φ.

```
factor(phi)
```

$2^3 \cdot 7 \cdot 11$

On choisit par exemple de prendre $e = 5$. En pratique, pour une véritable implémentation de RSA, il vaut mieux éviter de choisir e trop petit.

```
e = 13
```

Trouver un entier $d \in N$ tel que $de \equiv 1 \bmod \varphi(n)$.

Comme e et $\varphi(n)$ sont premiers entre eux, e est inversible modulo $\varphi(n)$, autrement dit il existe un entier d comme voulu. Pour trouver d en pratique, on écrit l'algorithme d'Euclide étendu : il fournit deux entiers d et s tels que $de + s\varphi(n) = 1$. Modulo n, on aura donc $de \equiv 1 \bmod \varphi(n)$. Si jamais d est négatif, on sait qu'on peut le remplacer par un multiple approprié de $\varphi(n)$ de manière à avoir un entier naturel.

Dans Sage, le résultat de l'algorithme d'Euclide étendu entre deux entiers a et b s'obtient par la commande *xgcd*(a, b). Consulter l'aide de Sage pour comprendre à quoi correspondent les trois valeurs retournées par cette fonction.

```
xgcd(e,phi)
```

$(1, 237, -5)$

On peut donc prendre $d = 237$. Vérifions-le.

```
d = 237
(d*e).mod(phi)
```

1

Transmettre votre clé publique (n, e) à votre voisin.

```
print "Ma cle publique est", (n,e)
```

```
Ma cle publique est (667, 13)
```

Pour information, voici votre clé privée.

```
print "Ma cle privee est", (p,q,d)
```

```
Ma cle privee est (23, 29, 237)
```

Mon voisin souhaite m'envoyer un message confidentiel. Il le crypte *à l'aide de ma clé publique (n, e)* (et non pas avec sa propre clé !). Je reçois le message c suivant.

```
c = 158
c
```

158

Je déchiffre ce message à l'aide de ma clé privée.

```
(c^d).mod(n)
```

444

Le message qu'a voulu m'envoyer mon voisin est donc 444, dans sa forme déchiffrée.

2. Mise en oeuvre sur SageMath : un exemple grandeur nature

Obtenir de grands nombres premiers de manière aléatoire.

D'après le théorème des nombres premiers, la probabilité pour qu'un entier naturel d'au plus k décimales et choisi au hasard soit premier est de l'ordre de $1/\ln(10^k)$ c'est-à-dire environ $1/(2,3k)$. Pour fabriquer au hasard de grands nombres premiers, on peut donc procéder ainsi : on tire au hasard un entier et on teste s'il est premier ou non. S'il ne l'est pas, on en essaye un autre, etc. jusqu'à en trouver un. L'heuristique précédente assure que cette méthode va aboutir après un nombre suffisant et raisonnable de lancers avec une probabilité élevée. Pour accélérer la recherche, on a intérêt à utiliser un test de pseudoprimalité (*is_pseudoprime*), puis lorsqu'un candidat a réussi ce test, à confirmer sa primalité par un test déterministe (*is_prime*).

Estimons le nombre d'entiers à tester. Notons $q = 1 - 1/\ln(10^k)$ la probabilité qu'un entier naturel d'au plus k décimales et choisi au hasard ne soit pas premier. La probabilité d'avoir n échecs successifs est égale à q^n. Elle est inférieure à 5% dès lors que $n \geq \frac{\log(0,05)}{\log q}$.

```
k = 310
log(0.05)/log(1-1/(ln(10^k))).n()
```

2136.85961134156

Dans Sage, la commande *.n()* permet d'obtenir une valeur approchée (elle signifie *numerical approximation*).

Par exemple pour $k = 310$, nous avons 95% de chance de trouver un nombre premier en testant 2136 nombres entiers. Pour obtenir un premier à exactement 310 décimales, le calcul est similaire et l'estimation assez proche.

Trouvons de cette manière un nombre premier à au moins 310 chiffres. Dans le programme ci-dessous, la commande # permet d'introduire des commentaires, qui ne sont pas interprétés par Sage.

```
p = ZZ.random_element(10^312)      # On choisit au hasard un nombre entier inferieur a 10^312
N = 1       # On initialise le compteur pour le nombre de tests
while (len(p.digits()) < 310) or (not p.is_pseudoprime()):
    p = ZZ.random_element(10^310)
    N = N+1
```

La boucle *while* fait recommencer le test dès lors que le nombre a moins de 310 décimales ou qu'il n'est pas premier.

La valeur finale de N retourne le nombre de tests effectués.

```
N
```

1454

Comptons le nombre de décimales de p.

```
len(p.digits())
```

310

On peut aussi afficher le nombre p.

```
p
```

6391853333868611242170737847325195783993823783334077440428939500197135818387148630767522327283885130884196764522235032567842350619907832748365683143039883749821863032658996730095527

Il reste à confirmer sa primalité par un test déterministe. La commande *% time* permet d'afficher le temps utilisé pour exécuter une commande.

```
%time
p.is_prime()
```

True

```
CPU time: 21.61 s,  Wall time: 21.62 s
```

Même chose avec un autre nombre premier q.

```
q = ZZ.random_element(10^312)
while (len(q.digits()) < 310) or (not q.is_pseudoprime()):
    q = ZZ.random_element(10^312)
```

```
len(q.digits()),q
```

(312, 1312826663478509391198324185933004770014966479439696708043395916707204347243801192934335461438202265968924579887952711268562920982070233936640517172564518770824848214039903726710

```
%time
q.is_prime()
```

True

```
CPU time: 21.94 s,  Wall time: 21.94 s
```

On termine en calculant notre clé publique n et par vérifier qu'elle a au moins 617 décimales.

```
n = p*q
n
```

```
8391395485746715624921742592901268413687340501782412944539707697754649636546736405592532858465901126030971961661039900090261008448998266139554009997041182102448101181470966499086325231
```

```
len(n.digits())
```

```
621
```

Variantes. On aurait aussi pu utiliser les commandes *next_prime* (qui retourne le nombre premier suivant immédiatemment un entier donné) ou la commande toute faite *random_prime* (qui retourne un nombre premier au hasard d'une taille donnée).

Calcul de $\varphi(n)$.

```
# euler_phi(n) # Trop long !
```

Le calcul de $\varphi(n)$ par la commande est trop long. C'est normal car Sage va chercher à le factoriser n afin de calculer $\varphi(n)$. Comme n a plus de 610 décimales, c'est une tâche impossible en un temps raisonnable. La sécurité de RSA réside ici.

Pour calculer $\varphi(n)$, on se rappelle que comme $n = pq$ avec p et q premiers distincts, on a $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$. On demande à Sage de calculer directement ce nombre, qui suppose de connaître les facteurs p et q de n (clé privée).

```
phi = (p-1)*(q-1); phi
```

```
8391395485746715624921742592901268413687340501782412944539707697754649636546736405592532858465901126030971961661039900090261008448998266139554009997041182102448101181470966499086325231
```

Choix de e

On doit trouver e entier naturel tel que son pgcd avec $\varphi(n)$ est égal à 1. Là encore, il n'est pas question de factoriser $\varphi(n)$, qui est très grand...

La majorité des entiers plus petit que $\varphi(n)$ n'ont aucun facteur commun avec lui. On procède donc en tirant des entiers au hasard, jusqu'à en trouver un qui réalise la condition souhaitée.

```
e = ZZ.random_element(phi)
while gcd(e, phi) != 1:
    e = ZZ.random_element(phi)
```

```
e
```

```
1712218806381493539836766609650311123538483836971420220382607944162182712432098798417883372303524552940761861462383088948808846235969601283596833818288481940635004715193764401165055
```

Vérification :

```
e.gcd(phi)
```

```
1
```

Choix de d.

On cherche un entier $d \in \mathbb{N}$ tel que $de \equiv 1 \bmod \varphi(n)$. À nouveau, c'est l'algorithme d'Euclide étendu qui permet de le faire.

```
e.xgcd(phi)
```

```
(1, 1729803298463647757342564583839101006756105127155000089044906680960535502643097200313033709813283588188915146112673407783857773985304194607188163417114393527038237430057736712771897
```

On extrait de ce résultat la deuxième composante de la liste. Remarque : en Python (et donc en Sage), la numérotation des listes commence à 0. On demande donc ici la 1ère composante.

```
d = e.xgcd(phi)[1]
d
```

```
1729803298463647757342564583839101006756105127155000089044906680960535502643097200313033709813283588188915146112673407783857773985304194607188163417114393527038237430057736712771896751
```

Clés publique et privée

Notre clé publique est donc :

```
(n,e)
```

```
(8391395485746715624921742592901268413687340501782412944539707697754649636546736405592532858465901126030971961661039900090261008448998266139554009997041182102448101181470966499086325231
```

et notre clé privée est :

```
(p,q,d)
```

```
(6391853333868611242170737847325195783993823783334077440428939500197135818387148630767522327283885130884196764522235032567842350619907832748365683143039883749821863032658996730095527
```

Chiffrage et déchiffrage d'un message

On choisir un message numérique m au hasard, avec $m < n$.

```
m = ZZ.random_element(n)
m
```

```
5229483346533717070381155799145152066368563616137837413137572897429797825107893198885022415826244674876899958389373071430055670232165282219431633476270173764704323070961995940066001131
```

On vérifie que m est bien strictement inférieur à n.

```
m < n
```

True

Chiffrons le message avec notre clé privée. Il s'agit de calculer m^e mod n.

```
# (m^e).mod(n) # Trop long !
```

La commande ci-dessus prend beaucoup trop de temps : elle fait calculer m^e, puis le réduire modulo n. Ce n'est pas une manière adaptée de calculer des puissances modulo n. Il vaut mieux utiliser l'exponentiation rapide (*square and multiply*), qui s'obtient dans Sage par la commande *power_mod*.

```
%time
c = power_mod(m,e,n)
c
```

309592762402277889565898176145957673305797901018438172437817471415120241563904638949258531364759627543586642150217469715845897148378080973803274629432056067189133502295197824583644766

CPU time: 0.06 s, Wall time: 0.07 s

On déchiffre ensuite le message à l'aide de notre clé privée. Pour cela, on évalue c^d mod n, en utilisant encore l'exponentiation rapide.

```
power_mod(c,d,n)
```

522948334653371707038115579914515206636856361613783741313757289742979782510789319888502241582624467487689995838937307143005567023216528221943163347627017376470432307096199594006600113

Vérifions que le message initial, et le message chiffré puis déchiffré coïncident.

```
power_mod(c,d,n) == m
```

True