SIMULATIONS NUMÉRIQUES - SAGEMATH/PYTHON

Ines Abdeljaoued Tej [1]

Codage et décodage Al kindi/César - RSA/Miller-Rabin

Le système de chiffrement RSA est une méthode qui repose sur certains concepts de la théorie des nombres. C'est une méthode à clé publique, nommée à partir des noms de famille de ses inventeurs Ronald Rivest, Adi Shamir et Leonard Adleman. Ce TP a pour but d'étudier le principe de fonctionnement de l'algorithme RSA. Pour cela, nous allons d'abord étudier un code permettant de décrypter et de décrypter un message. A la suite, nous allons voir comment générer des clés publiques et privées pour crypter un message.
Mais avant cela, la première section présentera un petit exemple ludique de décryptage d'un message, permettant d'introduire un certain nombre de vocabulaire lié à la cryptographie. Cet exemple comprend une brève explication du cryptage de Jules César ainsi que de l'algorithme de décriptage d'Al Kindi.

1 Algorithme d'Al Kindi

Pour décoder un message texte, on va utiliser l'algorithme de Abu Yusuf Yaqub Ibn Ishaq Al Kindi qui a vécu à Bagdad de 801 à 873. Cet algorithme consiste à comparer la fréquence d'apparition des lettres codées avec la fréquence des lettres dans la langue française. En effet, une lettre qui apparaît souvent dans la langue française, va apparaître souvent (via son code) dans un message codé. Ci-dessous les fréquences des lettres dans la langue française :

Lettres	E	S	A	I	T	N	R	U	L	O	D	C	P
Fréquences (%)	14.7	7.9	7.6	7.5	7.2	7.1	6.6	6.3	5.5	5.4	3.7	3.3	3

Soit le message codé suivant [2] :

```
code = "YR PNEER QR Y ULCBGRAHFR "
code += "RFG RTNY FV WR AR Z NOHFR "
code += "N YN FBZZR QRF PNEERF "
code += "QRF QRHK NHGERF PBGRF"
```

On crée l'objet `message` composé des lettres de `code` et l'ensemble `lettres` des différentes lettres apparaissant dans le message codé :

1. MA à l'ESSAI, membre du Laboratoire BIMS, `inestej@gmail.com`

2. TP inspiré des supports du Laboratoire d'informatique et de mathématiques de l'Institut de recherche sur l'enseignement des mathématiques de la Réunion : http ://irem.univ-reunion.fr/IMG/pdf/TP11.pdf

```
message = [i for i in code]
lettres = set(message)
```

1. Si une lettre apparaît souvent dans un message, on s'attend à ce qu'elle apparaisse autant dans un message codé. On calcule donc le tableau d'effectifs en comptant les lettres du message :

   ```
   for j in lettres:
   print(j+'-->'+str(message.count(j)))
   ```

   Comparer ce résultat avec le tableau ci-dessus, et donnez le décodage de la lettre E, S et A.

2. Pour continuer le décodage, et vu que les fréquences d'apparition des lettres sont trop proches, on peut utiliser les propriétés lingustiques, comme le fait qu'un mot long commence souvent par une consonne ou qu'il y a peu de mots de deux lettres. On peut également tester si le procédé de chiffrement utilisé est bien celui proposé par Jules César.

   Ce procédé date de 120 avant J.C., il utilise la technique suivante : il suffit de procéder à une permutation circulaire de toutes les lettres de l'alphabet, en remplaçant chaque lettre par celle qui est située à s rangs plus loin (ici s correspond à la clé de chiffrement). Dans cet exemple, la lettre A correspond à N, la lettre E correspond à R et la lettre F correspond à S. Vérifiez, grâce au code suivant, que la clé de chiffrement est bien $s = 13$ et en déduire le message déchiffré.

   ```
   s = 13 # s est la cle de chiffrement selon la méthode de Jules César
   n=len(message)
   MessageDechiffre=""
   for i in range(n):
       caractere = message[i]
       p = ord(caractere) #Varie entre 65 et 91
       q = p + s
       if q > 91:
           q = q - 26
       if q == 91:
           q = 65
       r = chr(q)
       print caractere, p, r, q
       MessageDechiffre = MessageDechiffre + r
   print MessageDechiffre
   ```

3. Pourquoi est-ce que le codage de César est facile à casser ? Indication : trouver le nombre de clés différentes qu'on peut utiliser pour le codage de César.

2 Algorithme de Miller-Rabin

Le but de cette section est de construire des nombres premiers, le plus grand possible. Pour cela, on va utiliser l'algorithme de Miller-Rabin qui permet de tester si un nombre donné est premier ou pas, en un temps raisonnable.

L'algorithme se base sur le théorème suivant : **Théorème.** Soit p un nombre premier. On pose s la puissance maximale de 2 divisant $p-1$:

$$p - 1 = 2^s.d \text{ avec } d \text{ pair.}$$

Pour tout $1 \leq n \leq p-1$, on a l'une des possibilités suivantes :

$$n^d = 1 \bmod p \qquad \text{ou alors} \qquad n^{2^j d} = -1 \bmod p \qquad \text{pour un certain } 0 \leq j < s.$$

Il s'agit donc de choisir au hasard un nombre n jusqu'à ce que l'une des deux conditions précédente n'est pas satisfaite. Cet n est appelé "témoin" du fait que p est un nombre composé. Si p n'est pas premier, alors il existe $3/4$ de chance que n soit "témoin". L'idée est donc d'utiliser le petit théorème de Fermat i.e. le fait que $a^{p-1} = 1 \bmod p$.

1. On désigne par t la taille en bits de la clé RSA que l'on souhaite utiliser. Il nous faut donc trouver deux nombres premiers de taille $t/2$. Pour cela, on utilise la fonction `random` pour trouver un nombre de $t/2$ bits au hasard puis `netx_prime` le nombre premier suivant.
2. Déterminez le plus petit nombre premier de 10 chiffres (pensez à la conversion en écriture binaire et à la commande `random_prime(2^1023, 2^1024)`).

   ```
   import random
   randint(10\^9, 10\^10)
   random.getrandbits(2048)
   ```

3. Implémentez l'algorithme de Miller-Rabin. Le pseudo-code obtenu de Wikipédia est donné ci-dessous. La fonction `Témoin_de_Miller(a,n)` renvoie Vrai si a est un témoin de Miller que n est composé, Faux si n est fortement pseudo-premier en base a. Elle prend en entrée un nombre n entier, impair ≥ 3 et a un entier > 1. On calcule s et d tels que $n - 1 = 2^s d$ avec d impair. Ici $s > 0$ car n est impair. On note x le reste de la division de a^d par n, puis dans la boucle `Tant que`, le reste de la division de x^2 par n.

   ```
   Témoin_de_Miller(a, n):
           x := a^d % n #x entier reste de la division de ad par n
           si x = 1 ou x = n - 1
                  renvoyer(Faux)
           Tant que s > 1
                   x := x^2 % n
                   si x = n - 1
                          renvoyer(Faux)
                   s := s - 1
           Fin de boucle tant que
           renvoyer(Vrai)
   ```

Le test de Miller-Rabin peut alors être décrit comme suit, `Miller-Rabin(n, k)` renvoie Vrai si n est fortement pseudo-premier en base a pour k entiers a, Faux s'il est composé. C'est un algorithme qui a une complexité polynomiale en terme de temps de calcul : $O(k.log(n)^3)$ obtenue en remarquant que la décomposition $n-1 = 2^s d$ se calcule en $O(log(n))$.

```
Miller-Rabin(n,k):
        répéter k fois :
                choisir a aléatoirement dans l'intervalle [2, n-2]
                si Témoin_de_Miller(a,n)
                        renvoyer(Faux)
        Fin de boucle répéter
        renvoyer(Vrai)
```

4. La fonction `is_prime` est un test de primalité qui prouve si un nombre donné est premier ou pas. La fonction `is_pseudoprime` est un test probabiliste de primalité : l'option > 0 utilise le test de primalité probabiliste de Miller-Rabin. Ce test s'appuie sur le petit théorème de Fermat. Testez ces deux fonctions et expliquez à quoi servent les fonctions `netx_prime` et `random_prime` ou encore `next_probable_prime` [3].

3 Algorithme RSA

Le principe du système de chiffrement RSA est le suivant [4] :

Trouver deux nombres premiers p et q	p et q doivent rester secrets
Calculer $n = pq$	n est clé publique
Choisir e tel que son PGCD avec $(p-1)(q-1)$ soit 1	e est clé publique
Trouver d tel que $ed \equiv 1\,(\text{modulo}\,(p-1)(q-1))$	d est la clé privée.

Le cryptage fonctionne de la manière suivante : il y a deux personnes Ali et Besma qui souhaitent communiquer en toute sécurité. Soit M un nombre qui correspond au message à envoyer. B génère n ainsi que les exposants de déchiffrements e et d. B envoie à A les nombres n et e qui constituent la clé publique [5].

Ici Ali est l'émetteur et il ne connaît que la clé publique, c'est-à-dire les nombres e et n. Il calcule $M^e \equiv N(\text{modulo}\,n)$ et envoie au destinataire le message N. Le récepteur qui est Besma calcule $N^d \equiv W(\text{modulo}\,n)$. On constate que W est exactement le message M.

Exercice à faire : Les messages qu'on peut chiffrer sont les entiers M compris entre 0 et n-1. Le codage du message M s'obtient par le calcul de N et le déchiffrement par le calcul de W.

3. *http : //sametmax.com/les − nombres − en − python/*, voir aussi http ://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf pour une bonne utilisation des outils de génération de nombre pseudo-aléatoires

4. Le nombre e est appelé *la graine* et on pose $\phi = (p-1)(q-1)$.

5. Le code python pour l'algorithme RSA a été inspiré de https ://gist.github.com/JonCooperWorks/5314103 alors que la structure du TP a suivi les travaux pratiques du master Informatique de l'Université de Lille 1 : https ://www.irisa.fr/prive/sgambs/TP7_intro_securite.pdf

1. Calculez $N = M^e$ (modulo n). Pour cela utilisez la commande `help(mod)` ou encore la commande %.
2. Le décodage ou le déchiffrement d'un message codé s'obtient par le calcul de `W = N`^{d} modulo n. Donnez deux fonctions de cryptage et de décryptage, par clés RSA : on appellera les fonctions `N = encrypt(e, n, M)` et `W = decrypt(d, n, N)`.
3. Les attaques actuelles du RSA se font essentiellement en factorisant l'entier n de la clé publique. La sécurité du RSA repose donc sur la difficulté à factoriser de grands entiers. Pour garantir une certaine fiabilité des transactions, il faut choisir des clés plus grandes : des clés de 2^{1024} voire 2^{2048} bits, i.e. des clés à `1024*ln(2)/ln(10)` ou `2048*ln(2)/ln(10)` chiffres.
   La comande `factor` factorise les entiers qu'on lui passe en paramètre. Générez des clés RSA de plus en plus grande et constatez l'augmentation du temps de calcul de la factorisation de ces données.

Le RSA est une méthode de chiffrement assez sûre, mais il n'est pas exclu que des services secrets comme la NSA ou le FSB aient réussi le décryptage du RSA.

4 Correction

4.1 Algorithme d'Al Kindi

Soit le message chiffré suivant :

```
YR PNEER QR Y ULCBGRAHFR RFG RTNY FV WR AR Z NOHFR N YN FBZZR QRF PNEERF QRF
QRHK NHGERF PBGRF
```

Il y a 20 lettres distinctes : qui apparaissent avec une fréquence détaillée ci-dessous :

Lettres	A	C	B	E	G	F	H	K	L	O	N	Q	P	R	U	T	W	V	Y	Z
Fréquences (%)	2	1	3	5	4	9	4	1	1	1	7	4	3	17	1	1	1	1	4	3

Les trois premières fréquences indiquent les lettres E, S et N. Ainsi, la lettre R correspond très probablement à la lettre E, la lettre F correspond à la lettre S et la lettre N correspond à la lettre A. En tenant compte de la clé de chiffrement $s = 13$, on peut en déduire le message déchiffré :

```
LE CARRE DE L HYPOTENUSE EST EGAL SI JE NE M ABUSE A LA SOMME DES CARRES DES
DEUX AUTRES COTES
```

.

Les procédures sur Python sont les suivantes. On peut s'amuser à coder et à décoder, grâce à la méthode de déchiffrement de Cesar, n'importe quel texte contenant les lettres majuscules de l'alphabet.

```
def codage_cesar(cle, message):
    #message est sous format string
    n = len(message)
    MessageChiffre = ""
    for i in range(n):
        caractere = message[i]
        p = ord(caractere) #Varie entre 65 et 91
        q = p + s
        if q > 91:
            q = q - 26
        if q == 91:
            q = 65
        r = chr(q)
        #print caractere, p, r, q
        MessageChiffre=MessageChiffre+r
    return(MessageChiffre)

message = "LE CARRE DE L HYPOTENUSE"
codage_cesar(cle, message)

def decodage_cesar(cle,code):
    n = len(code)
    MessageDechiffre = ""
    for i in range(n):
        caractere = code[i]
        p = ord(caractere) #Varie entre 65 et 91
        q = p + s
        if q > 91:
            q = q - 26
        if q == 91:
            q = 65
        r = chr(q)
        #print caractere, p, r, q
        MessageDechiffre=MessageDechiffre+r
    return(MessageDechiffre)

decodage_cesar(cle,code)
```

4.2 Algorithme de Miller-Rabin

```
sage: import random
sage: def decompose(n):
...       s = 0
...       d = n
...       while d % 2 == 0:
```

```
...         d = d/2
...         s += 1
...      return s, d
sage: decompose(33)
(0, 33)
sage: def Temoin_de_Miller(a,n):
...       s,d = decompose(n-1)
...       #print n-1, 2^s*d
...       x = a^d % n
...       if x == 1 or x == n - 1:
...           return(False)
...       while s > 1:
...           x = x^2 % n
...           if x == n - 1:
...               return(False)
...           s = s - 1
...       return(True)
sage: def Miller_Rabin(n,k):
...       i = 0
...       while i<k:
...           a = randint(2,n-2)
...           if Temoin_de_Miller(a,n)==True:
...               return(False)
...           i += 1
...       return(True)
sage: n = 122201
sage: a = 10
sage: k = 10
sage: print Temoin_de_Miller(a, n), Miller_Rabin(n, k)
False True
sage: next_prime(122187)
122201
```

4.3 Algorithme RSA

```
sage: import random
sage: """
sage: Ce que fait Besma :
...       D'abord B génère n ainsi que e et d.
...       Ensuite B envoie les clés publiques n et e à Ali.
...
sage: """
sage: p = 100
sage: p = next_prime(p)
sage: q = next_prime(p)
```

```
sage: print p,q
sage: n = p*q
sage: #la graine est choisie par hasard ;
sage: #il faut vérifier que e et phi sont premiers entre-eux :
sage: phi = (p-1)*(q-1)
sage: e = random.randrange(1,phi)
sage: print e, n, "les exposants de déchiffrements"
101 103
2522 10403 les exposants de déchiffrements
sage: phi = (p-1)*(q-1)
sage: #On cherche d tel que d*e+a*phi=1
sage: #print phi, e, n, d
sage: """
sage: Ali reçoit les clés de déchiffrement n et e.
sage: Pour envoyer le message M codé, A calcule N :
sage: """
sage: # Soit un message M :
sage: M = randint(2,n-1)
sage: print M
sage: def encrypt(e, n, M):
...       return pow(M,e,n)
sage: N = encrypt(e, n, M)
sage: print N
sage: """
sage: Ce que fait B : En recevant N de Ali, Besma calcule W :
...       B calcule d qui est la clé privée
...       Ensuite, elle détermine W qui est le M déchiffré.
sage: """
sage: def decrypt(d, n, N):
...       return(pow(N, d, n))
sage: g, a, d = xgcd(phi, e) #phi*a+d*e=g
sage: while g<>1 and d<=0:
...       g, a, d = xgcd(phi, e)
...       e = random.randrange(1,phi)
...
...
sage: W = decrypt(d, n, N)
sage: print p, q, n, e, d, M, N, W, M==W
101 103 10403 9019 19 3805 9308 4727 False
sage: """
sage: Pour casser le codage, il suffit de factoriser n et de récupérer p et q
sage: """
sage: factor(n)
101 * 103
```