MT10 – TP2

Antoine COLLAS, Tran Thi Son Nu

11 mai 2017

1 Manipulation de grands entiers :

1.

10^k comporte $k+1$ chiffres :
$10^0 = 1$ $10^1 = 10$ $10^2 = 100$...
Donc 10^{10^k} comporte $10^k + 1$ chiffres.

Nous exécutons successivement

```
time a=10^10
time a=10^(10^2)
...
```

Nous obtenons 2.14s pour 10^{10^8} et 28.6s pour 10^{10^9}. Une évaluation du nombre de chiffres des plus grands entiers manipulables par Sage en temps raisonnable est 10^8.

2.

Sage permet de compter le nombre de chiffres d'un entier en base B avec la méthode **digits(B)** qui renvoie une liste de longueur égale au nombre de chiffres. Par exemple :

```
a=10^20
L=a.digits(10)
len(L) //on obtient 21
```

3.

Pour calculer $n!$ on utilise : **factorial(n)**.

4.

```
len((12^34).digits(10))
```

37

```
len(((12^34)^3).digits(10))
```

111

```
len((12^(34^3)).digits(10))
```

42417

```
len(factorial(100).digits(10))
```

158

```
len(factorial(1000).digits(10))
```

2568

```
len(factorial(1000).digits(10)
```

42417

```
len(factorial(10000).digits(10))
```

35660

```
len(factorial(100000).digits(10))
```

456574

```
len(factorial(factorial(6)).digits(10))
```

1747

```
len(factorial(factorial(7)).digits(10))
```

16474

```
len(factorial(factorial(10)).digits(10))
```

22228104

6.

Nous commençons par calculer 100! à l'aide de la fonction **factorial**. Puis nous mettons ce nombre dans la formule de Stirling, ce qui nous donne $ln(n!!)$. Pour calculer la formule de Stirling nous utilisons la fonction **float** qui permet d'arrondir les calculs. Enfin nous divisons $ln(n!!)$ par $ln(10)$:

$$\frac{ln(n!!)}{ln(10)} = log_{10}(n!!) \approx nombres\ de\ chiffres\ de\ n!! \tag{1}$$

En effet,

$$\forall x \in \mathbb{R},\ x > 1,\ x = a \times 10^m\ avec\ 1 \leq a < 10\ et\ m \in \mathbb{N} \tag{2}$$

$$1 \leq a < 10 \implies log_{10}(1) \leq log_{10}(a) < log_{10}(10) \implies 0 \leq log_{10}(a) < 1 \tag{3}$$

$$log_{10}(x) = \frac{ln(x)}{ln(10)} = \frac{ln(a \times 10^m)}{ln(10)} = \frac{ln(a)}{ln(10)} + m = log_{10}(a) + m \tag{4}$$

Pour m grand, donc pour un nombre ayant beaucoup de chiffres :

$$log_{10}(x) \approx m \tag{5}$$

Pour calculer le nombres de chiffres de 100!! , nous utilisons le programme suivant qui utilise la formule de Stirling :

```
def q5(n):
    facto_n=factorial(n)
    ln_facto_n=float(log(n))*n-n+float(log(n))*0.5+
            float(log(sqrt(2*3.1416)))+float(1/(12*n))
    return floor(float(ln_facto_n/log(10))) + 1
```

```
q5(factorial(100))
```

1.47×10^{160}

100!! a donc 10^{160} chiffres.

Pour calculer le nombre d'écrans nous le nombre de chiffres de 100!! par le nombre de chiffres que peut afficher un écran :

```
def nb_ecran (m, lg, high):
    return float((m/(lg*high)))
```

En considérant qu'un écran peut afficher 120 caractères en largeur et 45 en hauteur :

```
nb_ecran(q5(factorial(100)), 120, 45)
```

2.7×10^{156} écrans

Calcul du nombre de Go nécessaires pour stocker 100!! :

```
def q6_Go(m):
    z_1=float((m-1)*log(10)/ log(2) + 1)
    return float (z_1/ (2^32))
```

```
q6_Go(q5(factorial(100)))
```

1.13×10^{151} go

2 Bézout et Euclide

La commande **gcd** permet de calculer le pgcd de 2 nombres :

```
xgcd(32,20)
```

4

En effet les diviseurs de 20 sont $1, 2, 4, 5, 10, 20$ et ceux de 32 sont $1, 2, 4, 8, 16, 32$. Donc pgcd(32,20)=4.

La commande **xgcd** permet de calculer le pgcd de 2 nombres a et b et renvoie aussi 2 nombres u et v tels que $u \times a + v \times b = pgcd(a, b)$

```
gcd(32,20)
```

(4,2,-3)
En effet les diviseurs de 20 sont $1, 2, 4, 5, 10, 20$ et ceux de 32 sont $1, 2, 4, 8, 16, 32$. Donc pgcd(32,20)=4.

On peut vérifier que deux nombres de Fibonacci consécutifs sont premiers entre eux :

```
gcd(fibonacci(19),fibonacci(20))
```

1

```
gcd(fibonacci(31),fibonacci(32))
```

1
Pour vérifier que le nombre de divisions est maximal il faut reprogrammer la fonction **gcd** en utilisant l'algorithme d'Euclide et en compter le nombre de divisions :

```
def func_pgcd (a,b) :
    i=0
    while a*b != 0 :
        i+=1
        if a > b:
            a = a-b
        else :
            b = b-a
        return i
```

Le nombre de divisions du calcul du pgcd entre deux nombres de Fibonacci consécutifs est maximal :

```
func_pgcd(fibonacci(19),fibonacci(20))
```

19

Pour calculer les coefficients de Bézout de 2 nombres de Fibonacci nous utilisons xgcd :

```
xgcd(fibonacci(19),fibonacci(20))
```

(1, -2584, 1597)

3 Nombres premiers

3.1 Sage et les nombres premiers

Nous testons les fonctions fournies par Sage pour les nombres premiers. **is_prime** permet de tester si un nombre est premier :

```
is_prime(2)
```

True

```
is_prime(4)
```

False
En effet 2 est premier puisqu'il n'a que 1 et lui même comme diviseur alors que 4 n'est pas premier puisqu'il a 2 comme diviseur.
Ensuite, **prime_range** fournit tous les nombres premiers de 2 jusqu'au nombre nombre fourni en paramètre moins 1 :

```
prime_range(11)
```

[2, 3, 5, 7]

```
prime_range(8,40)
```

[11, 13, 17, 19, 23, 29, 31, 37]

next_prime permet de trouver le nombre premier suivant le nombre saisi en paramètre :

```
next_prime(6)
```

7

factor permet de trouver la factorisation d'un nombre sous forme d'un produit nombres premiers :

```
factor(100)
```

$2^2 * 5^2$

prime_pi compte les nombres premiers inférieurs ou égaux au nombre donné en paramètre :

```
prime_pi(11)
```

5
En effet, les nombres premiers inférieurs ou égaux à 11 sont 2, 3, 5, 7, 11.

3.2 Nombres de Fermat

Nous cherchons un nombre de Fermat qui n'est pas premier. Pour cela nous calculons les nombres de Fermat jusqu'à en trouver un qui n'est pas premier :

```
def nbFermat():
    i=0
    while is_prime((2^(2^i))+1)==True:
        i=i+1
    return i
```

```
nbFermat()
```

5

Donc F_5 n'est pas premier. A l'aide de la commande **factor**, nous obtenons la décomposition de F_5 en un produit de nombres premiers :

```
factor(2^(2^5)+1)
```

641 * 6700417

3.3 Nombres de Mersenne

1.

Pour former la liste des nombres premiers inférieurs ou égaux à 257 nous utilisons la fonction **prime_range** vue précédemment :

```
prime_range(258)
```

[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,211,223,227,229,233,239,241,251,257]

Ensuite, **len** renvoie le nombre d'éléments contenus dans une liste :

```
len(prime_range(258))
```

55
Il y a donc 55 nombres premiers inférieurs ou égaux à 257.

Nous formons la liste des nombres de Mersenne en calculant les $2^p - 1$ avec p premier pris dans liste formée par **prime_range** :

```
def nbMersenne(n):
    M=[]
    for p in prime_range(n+1):
        M.append((2^p)-1)
    return M
```

[3, 7, 31, 127, 2047, 8191, 131071, 524287, 8388607, 536870911, 2147483647, 137438953471, 2199023255551, 8796093022207, **...**]

2.

Nous calculons chaque nombre de Mersenne et vérifions s'il est premier ou non. S'il est premier nous mettons la valeur de p dans la liste des indices :

```
def nbMersennePremier(n):
    ind=[]
    for p in prime_range(n+1):
        if is_prime((2^p)-1)==True:
            ind.append(p)
    return ind
```

Nous obtenons les nombres de Mersenne premiers suivants : $M_2, M_3, M_5, M_7, M_{13}, M_{17}, M_{19}, M_{31}, M_{61}, M_{89}, M_{107}, M_{127}$. Mersenne avait donc trouver jusqu'à $p = 60$ les nombres de Mersenne premiers.

3.

Pour décomposer $M41$ et $M47$ nous utilisons la fonction **factor** qui permet de décomposer un nombre entier en produit de nombres premiers :

```
factor((2^41)-1)
```

13367 * 164511353

```
factor((2^47)-1)
```

2351 * 4513 * 13264529

3.4 Sur la répartition des nombres premiers

1.

Afin d'évaluer les limites de la commande **prime_pi** nous proposons le programme ci-après. Ce programme calcule les prime_pi(10^i) en mesurant le temps d'execution à l'aide de la fonction **time()** appliquée sur l'objet **time** qui renvoie l'heure actuelle en seconde. Le programme renvoie la puissance de 10 pour laquelle **prime_pi** a mis plus de 60 secondes.

```
def limPrime_pi():
    i=1
    debut=time.time()
    prime_pi(10^i)
    fin=time.time()
    while(fin-debut<60):
        i=i+1
        debut=time.time()
        prime_pi(10^i)
        fin=time.time()
    return i
```

14
Donc **prime_pi(10^14)** met plus de 60 secondes à être exécutée.

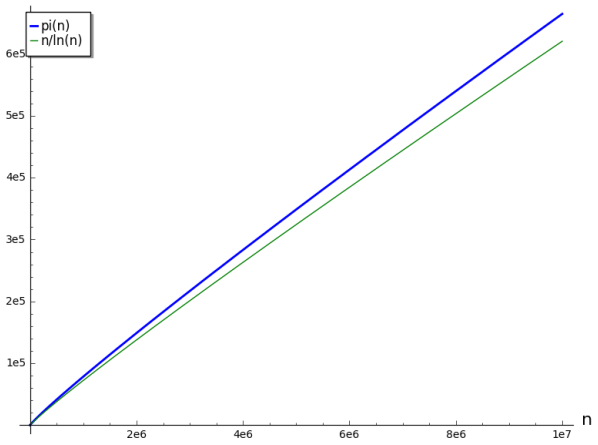
2.

Nous traçons sur un même graphique $\pi(n)$ et $\frac{n}{ln(n)}$ pour montrer que le nombre de nombres premiers se comporte comme $\frac{n}{ln(n)}$.

```
p=plot(prime_pi,2,10^7,thickness=2,
    legend_label="pi(n)")+plot(x/ln(x),2,10^7,color='green',
    legend_label="n/ln(n)")
p.axes_labels(['n',''])
p
```

Nous obtenons la figure suivante :

FIGURE 1 –



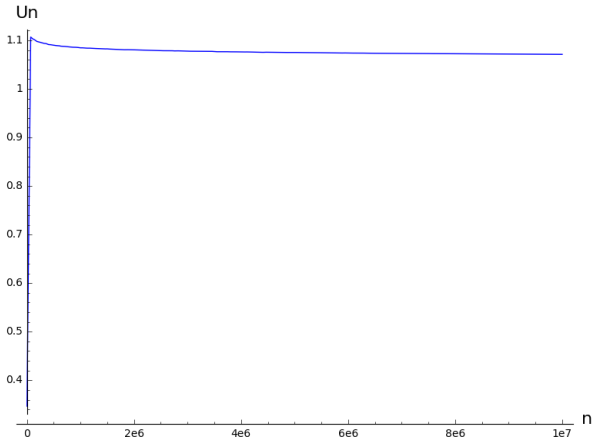
3.

Nous traçons sur un graphique $U_n = \pi(n)\frac{ln(n)}{n}$ pour vérifier le théorème des nombres premiers.

```
p=plot(prime_pi(x)*ln(x)/x,2,10^7, legend_label="Un")
p.axes_labels(['n','Un'])
p
```

Nous obtenons la figure suivante :

FIGURE 2 –



4 Algorithmes d'exponentiation

4.1 Naïf itératif et naïf récursif

Pour calculer x^n nous utilisons simplement une boucle for en procédant par multiplications successives :

```
def expIteratif(x,n):
    res=1
    for i in range(n):
        res=res*x
    return res
```

Comme nous calculons n multiplications la complexité est en $O(n)$.

Nous réalisons le même algorithme mais de manière récursive :

```
def expRecursif(x,n):
    if n==0:
        return 1
    else:
        return x*expRecursif(x,n-1)
```

Chaque appel est en $O(1)$, comme il y a n appels, l'algorithme est en $O(n)$.

4.2 Dichotomique itératif et dichotomique récursif

Afin d'améliorer la complexité de notre algorithme et donc sa rapidité nous programmons un algorithme dichotomique. Pour sa version itérative, nous commençons par nous débarrasser des cas triviaux $n = 0$ et $n = 1$. Puis si n est impaire alors nous le décrémentons pour que n soit paire. Ensuite nous calculons x^n comme indiqué dans l'énoncé : $x^2 = x \times x$, $x^4 = x^2 \times x^2$,... Si le n de départ est impaire alors le résultat de la boucle est x^{n-1}. Donc nous multiplions par x pour obtenir x^n.

```
def expDichotomiqueIteratif(x,n):
    if(n==0):
        return 1
    if(n==1):
        return x
    res=x
    impaire=n%2
    if impaire==1:
        n=n-1
    while n>1:
        res=res*res
        n=n/2
    if impaire==1:
        res=res*x
    return res
```

Pour calculer la complexité de l'algorithme il faut calculer le nombre d'appels de la boucle while. Nous divisons n par 2 à chaque itération. Soit k le nombre d'appels donc $\frac{n}{2^k} = 1$ donc $2^k = n$ donc $k = log_2(n)$. Donc la complexité est $O(log_2(n)) = O(ln(n))$.

Nous réalisons le même algorithme mais de manière récursive. Nous faisons attention à ne pas appeler 2^n fois la fonction ; c'est pourquoi nous utilisons la variable temporaire temp.

```
def expDichotomiqueRecursif(x,n):
    if n==0:
        return 1
    if n%2==0:
        temp=expDichotomiqueRecursif(x,n/2)
        return temp*temp
    else:
        temp=expDichotomiqueRecursif(x,(n-1)/2)
        return temp*temp*x
```

Chaque appel est en $O(1)$, comme il y a environ $log_2(n)$ appels (pour les mêmes raisons que l'algorithme itératif), l'algorithme est en $O(ln(n))$.

4.3 Algorithme d'exponentiation modulaire

Pour calculer $x^n[N]$ nous utilisons l'algorithme dichotomique récursif conçu précédemment. A chaque itération nous calculons le reste de la division du carré de la variable temporaire par N.

```
def expModulaireRecursif(x,n,N):
    if n==0:
        return 1
    if n%2==0:
        temp=expModulaireRecursif(x, n/2, N)
        return mod(temp*temp,N)
    else:
        temp=expModulaireRecursif(x,(n-1)/2, N)
        return mod(temp*temp*x,N)
```

Nous comparons notre algorithme avec **power_mod**. **power_mod(10^15, 10^110, 110)** met 0.39 ms alors que **expModulaireRecursif(10^15, 10^110, 110)** met 2.6ms.

4.4 Calcul des nombres de Fibonacci par exponentiation dans les matrices 2*2

Pour calculer F_n, nous implémentons la matrice donnée dans l'énoncé puis nous nous calculons la puissance n de cette matrice à l'aide des fonctions déjà implémentées dans Sage. Enfin, nous extrayons F_n de la matrice.

```
def fibo(n):
    A=matrix([[1,1],[1,0]])
    A=A^n
    return A[0,1]
```

```
fibo(25)
```

75025

Nous pouvons aussi utiliser directement la fonction **fibonacci** :

```
fibonacci(25)
```

75025

Les commandes **power??** et **power_mod??** ne donnent rien d'intéressant. **power??** retourne des exemples avec la fonction **generic_power** mais **generic_power??** ne donne rien non plus...

5 Cryptosystème RSA

5.1 Introduction

5.2 L'ensemble des messages, codage, décodage

Pour pouvoir appliquer un chiffrement RSA il faut que notre ensemble de départ soit $\mathbb{Z}/N\mathbb{Z}$. Après avoir codé notre chaine de caractères en chiffres binaires nous souhaitons regrouper ces chiffres par paquet de façon à les transformer en chiffres compris entre 0 et $N-1$. Un paquet de k chiffres binaires génère 2^k possibilités. De plus, comme il faudra faire l'opération inverse (**alphabetise**) il faut que le codage soit injectif. Donc nous souhaitons $2^k \leqslant N$ donc $k \leqslant log_2(N)$. Les paquets seront donc de taille

partie entière inférieure de $log_2(N)$. Chaque paquet est converti en un chiffre en base 10.

Cependant le dernier paquet pose problème. En effet, si par exemple nous choisissons $N = 8$ alors $log_2(N) = 3$ donc nous allons former des parquets de 3 bits. Cependant, le codage en binaire ne contient pas forcément un nombre de caractères multiple de 3. Le dernier paquet peut donc ne contenir que 2 caractères : 01 par exemple. $(01)_2 = 1$ en base 10, lors de l'utilisation de la procédure **alphabetise** nous ne saurons pas si le dernier paquet est 1, 01 ou 001. Nous avons donc choisi d'ajouter un dernier chiffre à la fin de la liste qui indique la taille du paquet.

```
def numerise(original, N):
    S3=BinaryStrings()
    ori_num=S3.encoding(original)
    res=[]
    taille=int(log(N,2))
    while (len(ori_num)>taille):
        res.append(int(str(ori_num[0:taille]),2))
        ori_num=ori_num[taille:]
    res.append(int(str(ori_num[0:taille]),2))
    res.append(len(str(ori_num[0:])))
    return res
```

```
t=numerise("je suis en train de faire un tp de mt10 sage", 19801)
t
```

[6809, 4615, 3541, 10611, 2073, 5858, 465, 12897, 6747, 8710, 4500, 8294, 6234, 5926, 5249, 13678, 2077, 1794, 401, 9504, 7005, 787, 129, 13153, 6617, 1,2]

Pour décoder nous fabriquons la procédure **alphabetise**. Cette procédure prend chaque nombre de la liste, le transforme en binaire puis complète avec des 0 pour obtenir un nombre binaire de la taille du paquet. Comme dit précédemment le dernier nombre de la liste indique la taille du dernier paquet et nous sert donc à compléter le dernier paquet avec le bon nombre de 0.

```
def alphabetise(code, N):
    taille=int(log(N,2))
    res=''
    while (len(code)>2):
        temp=str(bin(code[0]))[2:]
        while len(temp)<taille:
            temp='0'+temp
        res=res+temp
        code=code[1:]
    temp=str(bin(code[0]))[2:]
    while len(temp)<code[1]:
        temp='0'+temp
    res=res+temp
    return S3(res).decoding()
```

```
alphabetise(t,19801)
```

'je suis en train de faire un tp de mt10 sage'

5.3 La génération de clés RSA

Pour évaluer les limites de **factor**, nous augmentons N jusqu'à ce que factor(N) ne soit plus capable de retrouver $p \times q$ en un temps raisonnable. Nous nous rendons compte que si p et q sont proches alors **factor** arrive toujours à factoriser N. Pour ne pas avoir p et q trop proches, nous utilisons **randint** qui génère un nombre aléatoire. Ainsi nous générons des nombres premiers aléatoires p et q et demandons au programme de s'arrêter quand le calcul de factor(p*q) dépasse 1 minute.

```
def limFactor():
    i=1
    debut=0
    fin=1
    while(fin-debut<60):
        p=next_prime(randint(10^i,10^(i+1)))
        q=next_prime(randint(10^i,10^(i+1)))
        debut=time.time()
        factor(p*q)
        fin=time.time()
        i+=1
    return i
```

35

Donc à partir de 10^{35} (pour p et q) **factor** met plus de 1 minute à trouver p et q.

Comme dit précédemment si p et q sont trop proches alors **factor** arrive facilement à factoriser N. Aussi prendre un N grand augmente la difficulté à le factoriser. Enfin, e ne doit pas être trop faible. En effet, si une personne intercepte le même message envoyé à plusieurs personnes, alors il est possible de retrouver le message originel avec le théorème des restes chinois.

De plus il ne faut pas utiliser les nombres de Mersenne premiers. En effet, ces nombres ont une forme bien définie ce qui réduit énormément le domaine des p et q. Nous ne connaissons aujourd'hui que 45 nombre de Mersenne premiers. Le nombre de possibilités de N est donc beaucoup trop faible.

Dans le programme cleRSA nous commençons par déterminer de façon aléatoire le nombre de chiffres de p et le nombre de chiffres de q tout en respectant les conditions sur m et sur le fait que p et q ne doivent pas être trop proches. Nous commençons par déterminer de p et q de manière aléatoire. Puis nous déterminons e de manière à ce qu'il soit premier avec l'indicatrice d'Euler de N et enfin nous déterminons d. Pour calculer d il faut trouver l'inverse de e dans $\mathbb{Z}/\phi(N)\mathbb{Z}$. Cela revient à trouver une relation de Bézout entre e et $\phi(N)$. Une telle relation est donnée par **xgcd**.

```
def cleRSA(m):
    nbChiffresP=randint(int(0.6*m),int(0.65*m))
    nbChiffresQ=randint(int(0.5*m),int(0.55*m))

    p=next_prime(randint(10^nbChiffresP,10^(nbChiffresP+1)))
    q=next_prime(randint(10^nbChiffresQ,10^(nbChiffresQ+1)))

    N=p*q

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

    e=randint(10^20,10^100)
    while (gcd(e,phi)!=1):
        e=randint(10^20,10^100)

    temp=xgcd(e,phi)
    d=temp[1]

    return [N,e,d]
```

```
[N,e,d]=cleRSA(30)
factor(N)
```

77866604446172761 * 64551797310311765119

La factorisation se fait de manière quasi instantanée.

Une clé de 2048 bits signifie que la clé vaut au maximum $2^{2048}-1$. Nous utilisons la commande suivante qui permet de générer une clé d'une longueur égale au nombre de chiffres de $2^{2048}-1$.

```
[N,e,d]=cleRSA(len((2^2048).digits(10)))
```

La commande renvoie les clés après quelques secondes. Cette fois-ci il n'est plus possible de factoriser N.

5.4 Fonctions de chiffrement et déchiffrement RSA

Pour chiffrer notre liste d'entiers compris entre 0 et $N-1$ nous calculons chaque nombre puissance e modulo N. Cela nous donne le programme suivant :

```
def chiffrerRSA(liste, N, e):
    for i in range(len(liste)):
        liste[i]=power_mod(liste[i],e,N)
    return liste
```

Pour déchiffrer, le principe est le même :

```
def dechiffrerRSA(liste, N, d):
    for i in range(len(liste)):
        iste[i]=power_mod(liste[i],d,N)
    return liste
```

Un petit essai :

```
[N,e,d]=cleRSA(5)
original="un test"
t=numerise(original,N)
chif=chiffrerRSA(t,N,e)
dech=dechiffrerRSA(chif,N,d)
alphabetise(dech,N)
```

'un test'

5.5 Signature avec RSA

Protocole 1 :

En suivant l'énoncé nous obtenons une fonction de chiffrement et une fonction de déchiffrement pour le protocole 1 :

```
def protocole1Chiffrement (message, signature, Na, da, Nb, eb) :
    m1c = numerise(message, Na)
    s1c = numerise(signature, Na)

    m2c = chiffrerRSA(m1c, Nb, eb)
    s2c = chiffrerRSA(s1c, Na, da)

    return [m2c, s2c]
```

```
def protocole1Dechiffrement (message, signature, Na, ea, Nb, db) :
    m1c = dechiffrerRSA(message, Nb, db)
    s1c = dechiffrerRSA(signature, Na, ea)

    m1 = alphabetise(m1c, Na)
    s1 = alphabetise(s1c, Na)

    return [m1, s1]
```

Illustration du protocole 1 :

```
[Na,ea,da]=cleRSA(30)
[Nb,eb,db]=cleRSA(35)
m=protocole1Chiffrement("Je fais un tp...","signature
    une",Na,da,Nb,eb)
protocole1Dechiffrement(m[0],m[1],Na,ea,Nb,db)
```

['Je fais un tp...', 'signature une']

Protocole 2 :

En suivant l'énoncé nous obtenons une fonction de chiffrement et une fonction de déchiffrement pour le protocole 2 :

```
def protocole2Chiffrement (message, Na, da, Nb, eb) :
    if Na > Nb :
        m1c = numerise(message, Na)
        m2c = chiffrerRSA(m1c, Nb, eb)
        m3c = chiffrerRSA(m2c, Na, da)

    elif Na < Nb :
        m1c = numerise(message, Na)
        m2c = chiffrerRSA(m1c, Na, da)
        m3c = chiffrerRSA(m2c, Nb, eb)

    return m3c
```

```
def protocole2Dechiffrement (message, Na, ea, Nb, db) :
    if Na > Nb :
        m2c = dechiffrerRSA(message, Na, ea)
        m1c = dechiffrerRSA(m2c, Nb, db)
        m1 = alphabetise(m1c,Na)

    elif Na < Nb :
        m2c = dechiffrerRSA(message, Nb, db)
        m1c = dechiffrerRSA(m2c, Na, ea)
        m1 = alphabetise(m1c,Na)

    return m1
```

Illustration du protocole 2 :

```
[Na,ea,da]=cleRSA(30)
[Nb,eb,db]=cleRSA(35)
m=protocole2Chiffrement("Je fais un tp...",Na,da,Nb,eb)
protocole2Dechiffrement(m,Na,ea,Nb,db)
```

'Je fais un tp...'

Il faut différencier deux cas pour conserver la bijection et ainsi pouvoir décoder le message. Pour $N_A > N_B$, si nous commençons à chiffrer avec N_A on aura une liste dans $\mathbb{Z}/N_A\mathbb{Z}$ et donc des nombres seront supérieurs à N_B. Lors du chiffrement avec N_B nous perdrons de l'information.