TP2v5

TP2 : Autour de RSA

Clément LESAEGE et Antoine POUILLAUDE

1. Manipulation de grands entiers

1.

```
time a=10^10
```

    Time: CPU 0.00 s, Wall: 0.00 s

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

    Time: CPU 0.00 s, Wall: 0.00 s

```
time a=10^(10^3)
```

    Time: CPU 0.00 s, Wall: 0.00 s

```
time a=10^(10^4)
```

    Time: CPU 0.00 s, Wall: 0.00 s

```
time a=10^(10^5)
```

    Time: CPU 0.00 s, Wall: 0.00 s

```
time a=10^(10^6)
```

    Time: CPU 0.02 s, Wall: 0.02 s

```
time a=10^(10^7)
```

    Time: CPU 0.28 s, Wall: 0.28 s

```
#time a=10^(10^8)
```

```
#time a=10^(10^9)
```

Sage peut gérer des entiers d'environ 10^8 nombres dans un temps raisonnable. Il peut gérer 10^9 chiffres en un temps irraisonnable. Et il "plante" à 10^10 chiffre.

2.

Nombre de chiffres d'un entier.

```
(ln(543545454654543546545454545454546546546546546546546546546546546546546546)/ln(10)).n(8)
```

    72.

On crée une fonction, basexn qui retourne le nombre de chiffres d'un entier.

```
def basex(n,b):
    return 1+floor(ln(n)/ln(b))
```

```
basex(554888,10)
```

    6

```
basex(5,10)
```

    1

Il existe également une méthode des entiers qui s'utilise de la façon suivante.

```
a=117
a.ndigits()
```

    3

3.

La fonction factorielle, calcule la factorielle d'un nombre entier, ou non.

Si le nombre n'est pas entier, sage utilise la notion de factorielle généralisée. Il utilise la relation :

$$= \Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$$

avec

```
factorial(4)
```
```
24
```
```
factorial(2.8)
```
```
4.69417420574042
```
```
#factorial??
```

Morceaux du code de la factorielle :

```
if isinstance(x, Rational):
        return gamma(x+1)
```

La fonction factorielle fait appel à la fonction gamma.

4.

```
basex(12^34,10)
```
```
37
```
```
basex((12^34)^3,10)
```
```
111
```
```
basex(12^(34^3),10)
```
```
42417
```
```
basex(factorial(100),10)
```
```
158
```
```
basex(factorial(1000),10)
```
```
2568
```
```
basex(factorial(10000),10)
```
```
35660
```
```
basex(factorial(100000),10)
```
```
456574
```
```
basex(factorial(factorial(6)),10)
```
```
1747
```
```
basex(factorial(factorial(7)),10)
```
```
16474
```
```
basex(factorial(factorial(10)),10)
```
```
22228104
```

5.

```
basex(factorial(factorial(100)),10)
```
```
floor(log(factorial(933262154439441526816992388562667004907159682643\
816214685929638952175999932299156089414639761565182862536979208272237\
758251185210916864000000000000000000000000))/log(10)) + 1
```

Sage n'est pas capable de donner une numérique, il s'arrête donc à une valeur littérale.

On va alors utiliser la forumule de Stirling qui permet de calculer ln(n!)

Pour calculer le nombre de chiffres de 100!!, on doit calculer $1 + floor(ln(100!!)/ln(10) = 1 + floor(stirling(100!))/ln(10)$, ce qui sera a porté de calcul.

```
def stirling(x):
    return round(x*ln(x)-x+(1/2)*ln(x)+ln(sqrt(2*pi))+(1/(12*x)))
```

```
stirling(factorial(100))
```
```
3385309311310028745264010917201411316406912793619799071960165526636\
3671204927277541302673373067867243410323467667170794194631369161860\
5189745008657206672554593
```
```
def nombre_chiffre_stir(n,b):
    return 1+floor(n/ln(b))
```

```
C100=nombre_chiffre_stir(stirling(factorial(100)),10)
```

```
C100.n()
```
```
1.47022115343764e160
```

On a mesuré que un écran du Worksheet Sage faisait :

121 caractères de largeurs

42 lignes de hauteur.

```
def nb_ecran(x,l,h):
    return x//(l*h)
```

```
nb_ecran(C100,121,42).n()
```

```
2.89299715355695e156
```

Ce qui fait beaucoup d'écrans...

6.

```
C100_2=nombre_chiffre_stir(stirling(factorial(100)),2)
```

```
C100_2.n()
```

```
4.88396895530221e160
```

```
(C100/(8*10^9)).n()
```

```
1.83777644179705e150
```

Il faudrait donc environ 6,105*10^150 Gigaoctets pour pouvoir stocker le nombre. Bref instockable.

2 Bézout et Euclide

```
#gcd?
#gcd??
```

```
#xgcd?
#xgcd??
```

gcd retourne le pgcd de deux nombres, si on l'écrit sous la forme gcd(a,b), ou d'un nombre quelconque de nombres si on l'utilise sous la forme gcd(a) où a est une liste de nombres. gcd correspond à l'algorithme d'euclide.

xgcd correspond quant à lui à l'algorithme d'euclide étendue. Il renvoie le pgcd ainsi que les coefficients de Bezout sour la forme (g, s, t) où pgcd(a,b)=g=a*s+t*b

```
#fibonacci?
```

On veut calaculer pgcd(x,y) avec x>y>o

On a vu en TD que lorsque l'algorithme d'Euclide s'arrête en n itérations. $x \geq F_{n+2}$ et $y \geq F_{n+1}$. En cherchant le pgcd de deux nombres de fibonacci successif,on aura $x=F_{n+2}$ et $y=F_{n+1}$., nous serons donc dans le pire cas et le nombre de divisions à effectuer sera maximal(n).

On crée une fonction dpgcd qui calcule le nombre de divisions qui seront effectuées par l'algorithme d'euclide.

```
def dpgcd (a,b) :
  i=0
  while (b > 0) :
    r = a % b
    a = b
    b = r
    i+=1
  return i
```

```
print dpgcd(fibonacci(20),fibonacci(19))
print dpgcd(fibonacci(8),fibonacci(7))
```

```
18
6
```

```
print dpgcd(fibonacci(20),fibonacci(14))
```

```
5
```

Si les nombres de fibonacci dont on cherche le pgcd ne sont pas successifs, le nombre de divisions ne sera pas maximal.

Nous allons maintenant utiliser xgcd pour obtenir les coefficients de Bézout de quelques couples de nombres de fibonacci.

```
print xgcd(fibonacci(7),fibonacci(6))
print xgcd(fibonacci(6),fibonacci(5))
print xgcd(fibonacci(5),fibonacci(4))
print xgcd(fibonacci(20),fibonacci(19))
```

```
(1, -3, 5)
(1, 2, -3)
```

```
(1, -1, 2)
(1, 1597, -2584)
```

```
print xgcd(fibonacci(20),fibonacci(14))
print xgcd(fibonacci(15),fibonacci(8))
```

```
(1, -18, 323)
(1, 1, -29)
```

Pour les calculer le pgcd de deux nombres de fibonacci, on pourrait utiliser la relation PGCD (F_m ; F_n) = $F_{PGCD(m\ ;\ n)}$

Par exemple :

```
print gcd(fibonacci(4),fibonacci(8))
print fibonacci(gcd(4,8))
```

```
3
3
```

3 Nombres premiers

3.1 Sage et les nombres premiers

```
#is_prime?
```

```
is_prime(2)
```

```
True
```

```
is_prime(1)
```

```
False
```

```
is_prime(1234)
```

```
False
```

```
is_prime(11)
```

```
True
```

is_prime est une fonction qui permet de vérifier si un nombre est premier ou non. Bien sûr il est à penser qu'une telle fonction a ses limites lorsqu'il s'agit de très grand nombre tel que 100!!. Plus le nombre à tester est grand plus cela prend du temps.

```
#factor?
##factor??
```

```
factor(420)
```

```
2^2 * 3 * 5 * 7
```

```
factor(11)
```

```
11
```

```
factor(100)
```

```
2^2 * 5^2
```

factor permet de renvoyer la décomposition en facteur premier d'un nombre donné en paramètre.

```
#prime_pi?
```

```
##prime_pi??
```

```
prime_pi(2)
```

```
1
```

```
prime_pi(1)
```

```
0
```

```
prime_pi(11)
```

```
5
```

```
prime_pi(100)
```

```
25
```

prime_pi donne le nombre de nombres premiers inférieurs ou égales au nombre donné en paramètre.

3.2 Nombres de Fermat

```
def fermat_avait_tort_et_le_tort_tue():
    n=0
```

```
    F=(2**(2**n))+1
    while(is_prime(F)):
        n=n+1
        F=(2**(2**n))+1
        print(is_prime(F))
    return F
```

```
fermat_avait_tort_et_le_tort_tue()
```

```
    True
    True
    True
    True
    False
    4294967297
```

```
factor(fermat_avait_tort_et_le_tort_tue())
```

```
    True
    True
    True
    True
    False
    641 * 6700417
```

4294967297 est un nombre de Fermat et il n'est pas premier. La conjecture qu'avait faite Fermat est donc fa

3.3 Nombres de Mersenne

Liste des nombres premiers inférieurs ou égaux à 257 :

```
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]
```

Nombre de nombres premiers inférieurs ou égaux à 257 :

```
prime_pi(257)
```

```
    55
```

```
def mersenne_list(list):
    mers_l=[]
    for i in range(len(list)):
        mers_l.append((2**(list[i]))-1)
    return mers_l
```

```
mers_l=mersenne_list(prime_range(258))
```

```
def mersenne_list_prem(list):
    mers_prem=[]
    for i in range(len(list)):
        if (is_prime(list[i])):
            mers_prem.append(list[i])
    return mers_prem
```

Liste des nombres de Mersenne pour p≤257 :

```
mersenne_list_prem(mers_l)
```

```
    [3, 7, 31, 127, 8191, 131071, 524287, 2147483647,
    2305843009213693951, 618970019642690137449562111,
    162259276829213363391578010288127,
    170141183460469231731687303715884105727]
```

```
def mersenne_est_rectifie(list):
    result=[]
    mers=[]
    for i in range(len(list)):
        if (is_prime((2**(list[i]))-1)):
            result.append(list[i])
            mers.append((2**(list[i]))-1)
    #print(mers)
    return result
```

Liste des p≤257 pour lesquelles l'affirmation de Mersenne est vraie :

```
mersenne_est_rectifie(prime_range(258))
```
```
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127]
```

La liste des nombres de Mersenne premiers donnée par Mersenne n'est pas la bonne. Mp est premier pour p=61,89,107 alors qu'ils ne sont pas dans la liste donnée par Mersenne. Les Mp pour p=67 et 257 ne sont pas premiers alors qu'ils apparaissent dans la liste donnée par Mersenne.

```
factor((2**(41)-1))
```
```
13367 * 164511353
```
```
factor((2**(47)-1))
```
```
2351 * 4513 * 13264529
```

Voici les factorisation des nombres de Mersenne pour p=41 et 47. Ils n'étaient donc pas premiers.

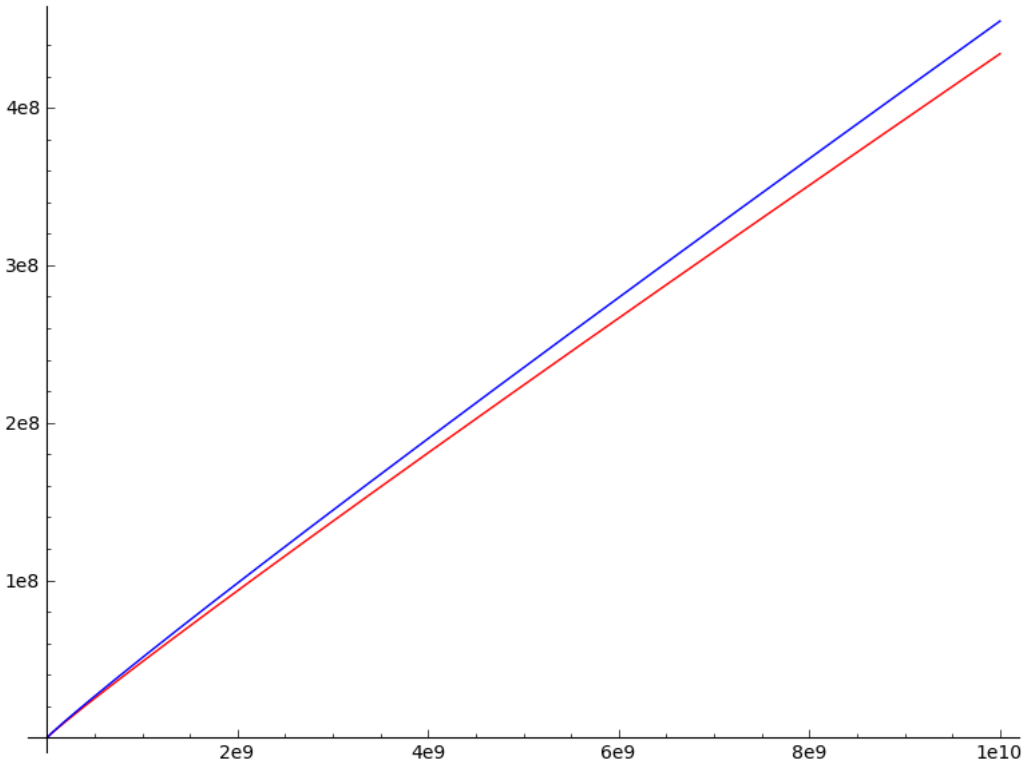
3.4 Sur la répartition des nombres premiers

```
prime_pi(10^10)
```
```
455052511
```
```
prime_pi(10^11)
```
```
4118054813
```
```
prime_pi(10^12)
```
```
37607912018
```
```
prime_pi(10^13)
```
```
Traceback (click to the left of this block for traceback)
...
NotImplementedError: computation of prime_pi() greater than 2^40 not
implemented
```

La limite de pime_pi est 2^40 (de l'ordre de 10^13).

```
def den(u):
    return u / ln(u)
def prime_den(u):
    return prime_pi(ceil(u))
```

```
c1=plot(den, (0,10^10),color='red')
c2=plot(prime_den, (0,10^10),color='blue')
show(c1+c2)
```



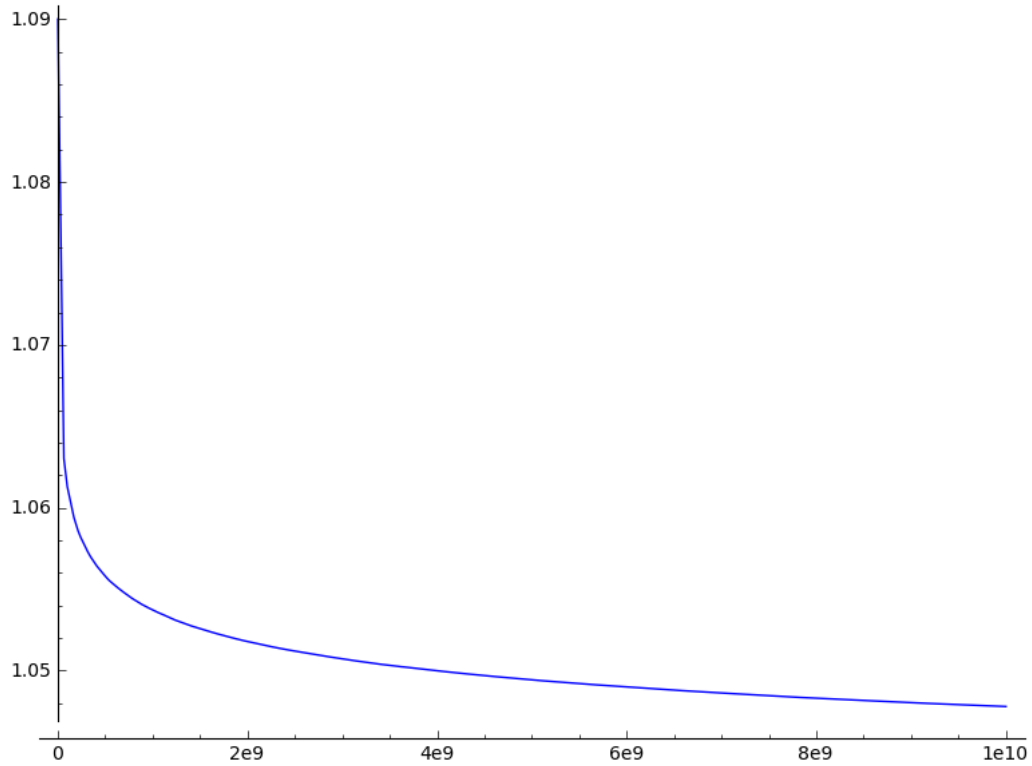
nombre de nombres premiers ≤ n en rouge

n/ln(n) en bleu.

```
#plot?
```

```
def suite(u):
    return (prime_den(u)*(1/(den(u))))
```

```
c=plot(suite, (0,10^10),color='blue')
show(c)
```



On remarque que la suite U_n semble converger vers 1, ce qui vérifierait le théorème des nombres premiers.

4 Algorithmes d'exponentiation

4.1 Naïf itératif et naïf récursif

Algorithme récursif naïf :

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

```
powrec(11,7)
```

```
19487171
```

Algorithme itératif naïf :

```
def powit(x,n):
    result=1
    for i in range(n+1):
        if (i==0):
            result=result*1
        else:
            result=result*x
    return result
```

```
powit(11,7)
```

```
19487171
```

Ces deux algorithmes sont de complexité identique. O(n) dans le modèle des coûts fixes et O(n^2 ln (x^2)) dans celui des coûts variables.

4.2 Dichotomique itératif et dichotomique récursif

Algorithme dichotomique récursif.

```
def powdirec(x,n,y=1):
    if (n==1):
        return x*y
    else:
        if ((n%2)==0):
            return powdirec((x**2),(n/2),y)
        elif ((n%2)!=0):
            return powdirec((x**2),(n-1)/2,(y*x))
```

```
powdirec(11,7)
```

```
    19487171
```

```
powdirec(5,2)
```

```
    25
```

Algorithme dichotomique itératif :

```
def powdicit(x,n):
    p=1
    while n>0:
        if ((n%2)!=0):
            p=p*x
            n=n-1
        n=n/2
        if (n>0):
            x=x*x
    return p
```

```
powdicit(11,7)
```

```
    19487171
```

```
11^7
```

```
    19487171
```

Ces deux algorithmes sont de même complexité :

O(ln(n)) dans le domaine des coûts fixes mais O(n^2 ln (x^2)) dans le domaine des coûts variables.

Dans le domaine des coûts variables, il n'y a donc pas de gain de complexité par rapport à l'algorithme naïf.

4.3 Algorithme d'exponentiation modulaire

```
def exmod(x,n,N=0,y=1):
    if (n==1):
        return (mod(x,N)*y)
    else:
        if ((n%2)==0):
            return exmod((mod(x,N)**2),(n/2),N,y)
        elif ((n%2)!=0):
            return exmod((mod(x,N)**2),(n-1)/2,N,(y*mod(x,N)))
```

L'élévation à une puissance conserve le modulo. Il suffit donc d'appliquer l'algorithme dichotomique d'exponentiation et de rabaisser à chaque étape la taille du calcul en utilisant un $x_{[N]}$ plutôt qu'un simple x.

Cela aura pour effet de diminuer grandement le temps du calcul.

```
time exmod(12^8,12^8,7)
```

```
    1
    Time: CPU 0.01 s, Wall: 0.01 s
```

```
def expomod(x,n,N=0):
    r=1
    e=0
    while(e<n):
        e=e+1
        r=mod(r*x,N)
    return r
```

Il existe un algorithme drastiquement plus efficace pour calculer cette quantité mais il faudrait faire intervenir les valeurs binaires des nombres et repose sur l'exponentiation binaire.

```
time expomod(12^4,12^4,7)
```

```
    1
```

```
Time: CPU 0.28 s, Wall: 0.29 s
```

```
time mod((10^8)^(10^8),7)
```

```
2
Time: CPU 35.97 s, Wall: 36.11 s
```

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

La matrice A est inversible et A^{-1} vaut
$\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$

Pour n=0, l'algorithme renvoie bien le bon résultat.
Pour n>0, on utilise la relation donnée et on obtient directement la validité de l'algorithme.

```
def fibo(n):
    M=Matrix([[1,1],[1,0]])
    M^=n
    return M[0][1]
```

```
fibo(12)
```

```
144
```

```
fibonacci(12)
```

```
144
```

```
#power??
```

```
#power_mod??
```

5 Cryptosystème RSA

5.2 L'ensemble des messages, codage, decodage

On essaie différents types de codage :

-alphabet usuel

-alphabet hexadécimal

-alphabet binaire

```
S1=AlphabeticStrings()
```

```
S2=HexadecimalStrings()
```

```
S3=BinaryStrings()
```

```
original="Que j'aime à faire connaître un nombre utile aux sages ..."
```

```
ori_num3=S3.encoding(original)
```

```
ori_num3
```

```
01010001011101010110010100100000011010101110001010000000100110010110\
00010110100101110110101100101001000001100001110100000001000000110011 0\
01100001011010010111001001100101001000000110001101101111011011001 10\
11100110000111000011101011100111010001110010011001010010000000111010 1\
01101110001000000110111001101111011011010110001001110010011001010010\
00000111010101110100011010010110110001100101001000000110000101110101\
01111000001000000111001101100001011001110110010101110011001000000010\
11100010111000101110
```

```
print ori_num3.decoding()
```

```
Que j'aime à faire connaître un nombre utile aux sages ...
```

```
ori_num2=S2.encoding(original)
```

```
print ori_num2
```

```
517565206ae2809961696d6520c3a020666169726520636f6e6e61c3ae7472652075\
6e206e6f6d627265207574696c652061757820736167657320e2e2e
```

```
print ori_num2.decoding()
```

```
Que j'aime à faire connaître un nombre utile aux sages ...
```

En utilisant S2 et S3, on arrive à encoder/décoder sans perdre d'information.

```
ori_num1=S1.encoding(original)
```

```
print ori_num1
```
```
QUEJAIMEFAIRECONNATREUNNOMBREUTILEAUXSAGES
```
```
print ori_num1.decoding()
```
```
QUEJAIMEFAIRECONNATREUNNOMBREUTILEAUXSAGES
```
```
ori_num1bis=S1.encoding("Que j'aime a faire connaitre un nombre utile aux sages")
print ori_num1bis
```
```
QUEJAIMEAFAIRECONNAITREUNNOMBREUTILEAUXSAGES
```

Le décodage ne marche pas, les caractères qui n'appartiennent pas aux 26 lettres de l'alphabet sont éliminés.

Deplus, on remarque que cette fonction n'est pas injective, avec deux entrées différentes, on peut avoir le même résultat. (on a essayé une 2nd fois sans les accents et on a obtenue le même résultat)

```
def numerise(original, n):
    S3 = BinaryStrings()
    code = S3.encoding(original)
    res = []
    l = int(log(n, 2))
    while(len(code) > 0) :
        res.append(code[0:l])
        code = code[l:]
    return res
```

```
list=numerise("Bonjour je m appelle Antoine",8)
print list
```
```
[010, 000, 100, 110, 111, 101, 101, 110, 011, 010, 100, 110, 111,
101, 110, 101, 011, 100, 100, 010, 000, 001, 101, 010, 011, 001,
010, 010, 000, 001, 101, 101, 001, 000, 000, 110, 000, 101, 110,
000, 011, 100, 000, 110, 010, 101, 101, 100, 011, 011, 000, 110,
010, 100, 100, 000, 010, 000, 010, 110, 111, 001, 110, 100, 011,
011, 110, 110, 100, 101, 101, 110, 011, 001, 01]
```

Numerise encode une chaîne de caractère avec l'aide de l'objet BinaryStrings et découpe le code obtenus en paquets de longueur $log_2(n)$ afin que les paquets aient une valeur inférieure à n.

Alphabetise effectue l'opération inverse : elle récupere une liste dont elle va concaténer tous les éléments en une chaîne de caractères et complète la chaîne afin qu'elle soit congrue soit divisible par 8.

```
def alphabetise(list,t):
    res = ""
    k = 0
    digits = int(log(t,2))
    for i in range(len(list)-1):
        list[i] = '0'*(digits-len(list[i]))+str(list[i])
        k += digits
    last= list[len(list)-1]
    while ((len(last)+k) % 8 != 0):
        last = '0' + last
    list[len(list)-1]=last
    S3 = BinaryStrings()
    while len(list) > 0:
        res= str(list.pop()) + res
    res = S3(res).decoding()
    #retour=unicode(retour, "utf8")
    return res
```

```
alphabetise(list,8)
```
```
'Bonjour je m appelle Antoine'
```

5.3 La genration des clés RSA

```
def N_prime(N):
    return next_prime(randint(10**N,10**(N+1)))
```

```
N=30
p=N_prime(N)
q=N_prime(N)
```

```
time factor(p*q)
```
```
29643040075540707704246926091 41 * 94897958713179084747089352095 83
Time: CPU 17.02 s, Wall: 21.19 s
```

```
N=40
p=N_prime(N)
```

```
q=N_prime(N)
```

```
time factor(p*q)
```

```
    Traceback (click to the left of this block for traceback)
    ...
    __SAGE__
```

Sur les ordinateurs de l'UTC la factorisation renvoie une erreur, sur un de nos ordinateurs personnels, la factorisation se fait en environ 3 minutes.

La limite de factorisation se trouve donc vers des nombres de taille 10^40 pour des nombres qui sont le produit deux nombres premiers élevés.

Quelles sont les recommandations de la societe RSA Data-Security ?

Il est recommandé d'utiliser des clefs de grande taille pour lutter contre l'augmentation de la puissance de calcul des ordinateurs. Il est également demandé d'utiliser des nombres premiers aléatoires. Les entiers p et q ne doivent pas être trop proches et l'exposant e ne doit pas être trop faible.

Pourquoi les nombres de Mersenne premiers doivent être evités ?

Il faut éviter les nombres de Mersenne (2^p -1 avec p premier,), car ils ont des diviseurs premiers de la forme 1+2kp. On peut donc les rechercher en testant les entiers de 2p en 2p, ce qui provoque un gain de temps considérable et donc une baisse de la difficulté de factorisation.

Pourquoi l'exposant de chirement e ne doit pas être choisi trop petit ?

Si l'exposant est trop faible il est possible d'utiliser le théorème de Coppersmith pour retrouver le message clair si on en connait déjà une partie. Par exemple avec un e=3, on pourra retrouver le message clair si on connaît déjà 2/3 du message clair.

Si e est faible, il est également possible de déchiffrer un message si un même message est chiffré pour plusieurs destinataires en utilisant des N différents et un même e.
Il est possible de déchiffrer le message si le nombre de destinataires est supérieur ou égal à e.
Ceci est appelé l'attaque de Håstad(car elle utilise une version améliorée du théorème de Håstad).

Il est recommandé par MM K.Lestra et R.Veerheu d'utiliser 65737 (un nombre de Fermat) comme valeur de e.

```
def cleRSA(m):
    key=()
    sup = 10 ** int(m/2);
    p = next_prime(randint(sup, 10*sup - 1));
    q = next_prime(randint(sup, 10*sup - 1));
    N=p*q
    e=0
    while(gcd(e,(p-1)*(q-1))!=1):
        e=ZZ.random_element((p-1)*(q-1))
    _,u,_=xgcd(e,(p-1)*(q-1))
    d=u
    l=1
    while d<=0:
        d=u+l*(p-1)*(q-1)
        l=l+1
    key=(N,e,d)
    return key
```

```
key1=cleRSA(30)
```

```
key1
```

```
    (218122049603907453200053900338 91, 178797975768390589109705986279 53,
    134431261397680949011499524064 01)
```

```
time factor(key1[0])
```

```
    4234365130141733 * 5151233842618727
    Time: CPU 0.05 s, Wall: 0.38 s
```

Si l'on souhaite générer des clés de 2048bits l'algorithme de test de primalité offert par sage ne suffit plus. Il faudrait alors un test de primalité probabiliste car un nombre de 2048bits est assez conséquent.

```
(n,e,d)=cleRSA(30)
Pukey=(n,e)
Prkey=(n,d)
```

5.4 Fonction de cryptage et de décryptage
Théoriquement il s'agit de la même fonction puisqu'il s'agit d'une exponentiation modulaire dans les deux cas. Je vais cependant faire deux fonctions différentes pour respecter les types des objets qui seront manipulés.

```
def encryption(list,key):
    for i in range(len(list)):
        list[i]=power_mod(Integer(str(list[i]),base=2),key[1],key[0])
    return list
```

```
list=numerise("Bonjour je m'appelle Jean",Pukey[0])
print list
```

```
    [01000010011011110110111001101010011011110111010101110010001000000 11\
    01010011001010010000001101101001001,
```

```
    1101100001011100000111000001100101011011000110110001100101001000000 1\
    0010100110010101100001011011 10]
```

```
code=encryption(list,Pukey)
print code
```

```
    [638539648461862564033116189 9703, 6809562852635332885047196310415]
```

```
def decrypt(code, key):
    code = [bin(power_mod(x, key[1], key[0]))[2:] for x in code]
    return code
```

```
code2=decrypt(code,Prkey)
print alphabetise(code2,Pukey[0])
```

```
    Bonjour je m'appelle Jean
```

Créons une fonction pour le protocol 1.

protocol1(m1, s1, PrkeyA, PukeyB) permettra de crypter un message m1 à destination de l'utilisateur B, avec une signature s1 de l'utilisateur A.

protocol1(msg, PrkeyB, PukeyA) permettra à l'utilisateur B de décoder un msg=(message,signature) crypté par l'utilisateur A.

```
def protocol1(a,b,c,d=0):
    if(d):
        return protocol1_encrypt(a, b, c, d)
    else :
        return protocol1_decrypt(a,b,c)
```

```
def protocol1_encrypt(m1, s1, PrkeyA, PukeyB):
    Nc=10
    #Étape 1
    m1c=numerise(m1,Nc)
    s1c=numerise(s1,Nc)
    #Étape 2
    m2c=encryption(m1c,PukeyB)
    s2c=encryption(s1c,PrkeyA)
    return (m2c,s2c)
```

```
def protocol1_decrypt(msg_crypt, PrkeyB, PukeyA):
    Nc=10
    #Étape 3
    m1c=decrypt(msg_crypt[0],PrkeyB)
    s1c=decrypt(msg_crypt[1],PukeyA)
    #Étape 4
    m1=alphabetise(m1c,Nc)
    s1=alphabetise(s1c,Nc)
    return (m1,s1)
```

Exemple d'utilisation :

```
(n1,e1,d1)=cleRSA(4)
PukeyA,PrkeyA=(n1,e1),(n1,d1)
(n2,e2,d2)=cleRSA(4)
PukeyB,PrkeyB=(n2,e2),(n2,d2)
m1="Salut Pauline, c'est Antoine. Comment ça va ?"
s1="Antoine"

msg=protocol1(m1,s1,PrkeyA, PukeyB)
(m1, s1) = protocol1(msg,PrkeyB,PukeyA)
print m1
print s1
```

```
    Salut Pauline, c'est Antoine. Comment ça va ?
    Antoine
```

Protocole 2:

La fonction stoi va transformer une liste de chaines de caractères binaires en une liste d'integer.

```
def stoi(list):
    return [Integer(str(x),base=2) for x in list]
```

La fonction dtob va transformer une liste d'integer en une chaîne de caractères base 2.

```
def dtob(list):
    return [bin(x)[2:] for x in list]
```

La fonction protocole va, en fonction du paramètre t, soit crypter ou décrypter. Dans le premier cas on envoi le message à chiffrer, celui-ci sera numérisé puis chiffré grâce à la fonction protocole2crypt. Dans le second cas on inverse l'ordre en faisant la comparaison contraire de celle faite dans le premier cas. On va par la suite faire appel à la même fonction protocole2crypt.

Et on renvoi le résultat

Le cas $N_A = N_B$ n'a pas besoin d'être considéré car il est presque impossible statistiquement puisque N_A et N_B sont générés de façon aléatoire.
Si jamais deux utilisateurs avaient le même N, il leur serait chacun possible de retrouver le d de l'autre utilisateur en utilisant la formule $ed \equiv 1_{[\Phi(N)]}$, ce qui provoquerait un grave problème de sécurité.

```
def protocole2(code,key1,key2,t=0):
    Nc=60
    if t:
        m1c=numerise(code,Nc)
        m1c=stoi(m1c)
        if key1[0]>key2[0]:
            return protocole2crypt(m1c,key2,key1,t)
        elif key2[0]>key1[0]:
            return protocole2crypt(m1c,key1,key2,t)
    else:
        if key1[0]<key2[0]:
            m1c_2=dtob(protocole2crypt(code,key2,key1,t))
        elif key2[0]<key1[0]:
            m1c_2=dtob(protocole2crypt(code,key1,key2,t))
        return alphabetise(m1c_2,Nc)
```

La fonction protocole2crypt va faire les deux exponentiations modulaires nécessaires au chiffrement du message. Dans les deux cas (chiffrage, déchiffrage) on va faire appel à la même fonction puisqu'il s'agit de la même opération mais avec des paramètres différents.

```
def protocole2crypt(code,key1,key2,t=0):
    code1=[exmod(x,key1[1],key1[0]) for x in code]
    code2=[exmod(y,key2[1],key2[0]) for y in code1]
    return code2
```

Petite Application.

On fixe Nc à 60. Et on génère les clé RSA des deux utilisateurs.

```
(n1,e1,d1)=cleRSA(100)
PukeyA,PrkeyA=(n1,e1),(n1,d1)
(n2,e2,d2)=cleRSA(100)
PukeyB,PrkeyB=(n2,e2),(n2,d2)
#print n1,n2,e1,e2,d1,d2
```

On rentre un beau message dans m1.

```
m1="Salut Clément c'est Antoine. Comment ça va ? J'adore le RSA car c'est trop cool. éàèâü"
```

Ensuite on chiffre le message à l'aide du protocole 2. Libre à vous d'afficher m3c.

```
m3c=protocole2(m1,PukeyB,PrkeyA,1)
```

Puis afin de récupérer le message original on fait à nouveau appel à protocole2 mais en utilisant cette fois-ci la clé privé de Bob et la clé publique d'Alice.

Attention on dois utiliser les clé dans le même ordre de personne.

```
m1c=protocole2(m3c,PrkeyB,PukeyA)
```

Enfin on affiche le résultat.

```
print m1c
```

```
Salut Clément c'est Antoine. Comment ça va ? J'adore le RSA car
c'est trop cool. éàèâü
```

Dans le protocole 2 on différencie deux cas car sinon le codage ne fonctionne pas et le message est perdu. Supposons en effet $N_A > N_B$ et que l'on code dans le mauvais ordre. Ainsi après la première exponentiation on aura une liste dans Z/N_AZ les éléments de la liste pourront donc être supérieur à N_B et donc en repassant une nouvelle fois à l'exponentiation on perdra l'information car la fonction RSA ne sera alors plus bijective.

```
```