TP3 v5

TP n°3 : Corps finis

et

codes correcteurs

Antoine POUILLAUDE et Clément LESAEGE

1 Construction de corps finis

1.2 Rudiments de Sage pour les corps finis

On essaie les quelques fonctionalités suivantes :

```
F5=GF(5)
```

```
#F5.
```

```
F8.<a>=GF(8,name='a')
```

```
F8.modulus()
```

```
x^3 + x + 1
```

```
R.<X>=GF(2)['X']
```

```
V=VectorSpace(F5,6)
```

```
V
```

```
Vector space of dimension 6 over Finite Field of size 5
```

```
F8.vector_space()
```

```
Vector space of dimension 3 over Finite Field of size 2
```

1.3 D'enombrement des polynômes irréductibles et unitaires de Fp[X]

1.3.1 Factorisation de $X^q - X$ dans Fp[X]

On crée une fonction pour factoriser $X^q - X$ avec $q = p^n$ dans $F_p[X]$

```
def pfactor(p,n):
  A=[]
  R.<X>=GF(p)['X']
  for deg in range(1,n+1):
      if n%deg == 0:
          for k in R.polynomials(deg):
              if (((((X**(p**n))-X) % k) == 0) and
k.is_irreducible()):
                  A.append(k)
  return A
```

```
f = pfactor(2,6)
f
```

```
[X, X + 1, X^2 + X + 1, X^3 + X + 1, X^3 + X^2 + 1, X^6 + X +
+ X^3 + 1, X^6 + X^4 + X^2 + X + 1, X^6 + X^4 + X^3 + X + 1,
X^5 + 1, X^6 + X^5 + X^2 + X + 1, X^6 + X^5 + X^3 + X^2 + 1,
X^5 + X^4 + X + 1, X^6 + X^5 + X^4 + X^2 + 1]
```

On vérifie cette factorisation :

```
R.<X>=GF(2)['X']
print X^(2^6) - X
produit = 1
for k in f :
  produit*=k
print produit
```

```
X^64 + X
X^64 + X
```

1.3.2 La fonction de Möbius

```
#moebius?
```

```
#moebius??
```

1. On vérifie que $\mu(n) \in -1, 0, 1$ pour les 100 premiers entiers naturels.

```
for i in range(100):
  #print moebius(i)
  if (moebius(i) not in [-1,0,1]):
      print false
      break
```

2.On vérifie la formule d'Euler pour la fonction Möbius pour les 100 premiers entiers naturels.

Pour cela on calcule

$\sum_{d|n} \mu(d)$

```
def formule_d_Euler_mais_pour_Moebius(n):
  return sum([moebius(d) for d in range(1,n+1) if n%d==0])
```

```
#for i in range(100):
  print formule_d_Euler_mais_pour_Moebius(i+1)
```

3. Calcul de la formule d'inversion de Möbius pour le calcul de $\phi(100)$

On crée F(n) lorsque f est la fonction d'euler :

```
#F(n) désigne F(n) pour la fonction euler_phi
def F(n) :
  somme = 0
  for d in range(1,n+1):
      if (n%d == 0) :
         somme+=euler_phi(d)
  return somme
```

En utilisant la formule d'Euler on peut simplifier à :

```
def F(n) :
  return n
```

Car $F(n) = \sum_{d|n} \phi(d) = n$

On défini ensuite f(n) par la formule d'inversion de Möbius

```
#f(n) désigne le f(n) obtenu par la formule d'inversion de
Möbius pour la fonction F
def f(n):
  somme =0
  for d in range(1,n+1):
      if (n%d ==0) : #si d|n
          somme+=moebius(n/d) * F(d)
  return somme
```

On vérifie que l'on retrouve bien $\phi(n)$

```
print euler_phi(100)
print f(100)
```

```
40
40
```

1.3.3 Calcul du nombre de polynômes unitaires irréductibles de degré d dans $F_p[X]$

On définie une fonction nous retournant $Irr_p(n)$

```
def Irr(p,n):
  somme=0
  for d in range(1,n+1) :
      if (n%d==0):
          somme+=moebius(n/d)*(p^d)
  somme/=n
  return somme
```

On dresse un tableau de ses valeurs :

```
def tabIrr(pmax,nmax) :
  print "n\p  |",
  for p in range(2,pmax+1):
      print "     ", p,
  print "\n",
  print "_ _ _|",
  for i in range(3*p):
      print "_",
  print "\n",
  for n in range(1,nmax+1):
      print "%3d" % n, "  |",
      for p in range(2,p+1):
          a = Irr(p,n)
          print "%6d"% a ,
      print "\n",
```

```
tabIrr(5,10)
```

```
n\p  |      2      3      4      5
_ _ _| _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
  1  |      2      3      4      5
  2  |      1      3      6     10
  3  |      2      8     20     40
  4  |      3     18     60    150
  5  |      6     48    204    624
  6  |      9    116    670   2580
  7  |     18    312   2340  11160
```

```
      8   |      30    810    8160   48750
      9   |      56   2184   29120  217000
     10   |      99   5880  104754  976248
```

```
def trouvep(p,n):
  A=[]
  nb=0
  R.<X>=GF(p)['X']
  for deg in range(1,n+1):
      for k in R.polynomials(deg):
          if (k.is_irreducible()):
              A.append(k)
              nb+=1
  print nb
  return A
```

```
#trouvep(2,10)
```

On en a alors bien dénombré 226

1.4 Calcul de polynômes unitaires irréductibles de $F_p[X]$

On sait qu'il y a 2+1+2+3+69+18+30+56+99=226 polynômes irréductibles de degrés inférieurs ou égal à 10 dans F_2 [X]

On va tester tous les polynomes unitaires de degrés inférieur ou égal à 10 et les afficher si ils sont irreductibles.

```
def preturn(p,n):
  A=[]
  nb=0
  R.<X>=GF(p)['X']
  for deg in range(1,n+1):
          for k in R.polynomials(deg):
              if (k.is_irreducible()):
                  A.append(k)
                  nb+=1
  return A
```

```
#preturn(2,10)
```

```
q=2**8
c='a'
n=8
k=4
r=n-k
```

```
G=GF(q,c)
c=G.gen()
R=PolynomialRing(G,'X')
X=R.gen()

a=G.multiplicative_generator()
v=[G.one() for i in range(n)]
alpha=[a**i for i in range(n)]
```

```
def codeGRS(y,k,v,a):
    if y.degree() > k:
        print "Attention le polynôme spécifier est trop grand
par rapport au k choisi"
    code = [vi*y(ai) for vi, ai in zip(v,a)]

    return code
```

```
code=codeGRS(a^6*X^2+a^4*X+a^2+1,k,v,alpha)
code
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6
a^3 + 1]
```

```
def Lagrange(a,i):
    L=1
    L=prod([(X-a[j]) for j in range(len(a)) if j<>i])
    return L
```

```
def decodeGRS(code,a,v,k):
    y=0
    for i in range(len(code)):
        Li=Lagrange(a,i)
        y+=(code[i]/(v[i]*Li(a[i])))*Li
    return y
```

```
decodeGRS(code,alpha,v,k)
```

```
a^6*X^2 + a^4*X + a^2 + 1
```

Cas où les α_i ne sont pas deux à deux distincts.

```
alphaNotDistinct=[a**i for i in range(n-1)]
alphaNotDistinct.append(a**(n-2))
```

```
code=codeGRS(a^6*X^2+a^4*X+a^2+1,k,v,alphaNotDistinct)
code
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
```

```
+ a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6
a^3 + a^2]
```

```
decodeGRS(code,alphaNotDistinct,v,k)
```

```
Traceback (click to the left of this block for traceback)
...
ZeroDivisionError: division by zero in finite field.
```

On a une division par zéro dans la fonction decodeGRS. En effet si on a $\alpha_i = \alpha_j$ avec $i \neq j$. On a alors $L_i(\alpha_i) = \prod_{j\in[0,n-1],j\neq i}(X-\alpha_j) = 0$. Il est alors impossible de calculer $L_i(\alpha_i)^{-1}$.

2.3 Simulation d'erreurs de transmission

Fonction simulant les erreurs de transmission :

```
def error_trans(code,Nb_err=0):
    n=len(code)-1
    codeErr=list(code)
    while Nb_err >0:
        rl=randint(0,n)
        rq=G.random_element()
        codeErr[rl]+=rq
        Nb_err-=1
    return codeErr
```

Essai d'une interpolation de Lagrange après une erreur :

Code :

```
code=codeGRS(a^6*X^2+a^4*X+a^2+1,k,v,alpha)
code
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6
a^3 + 1]
```

```
codeErr=error_trans(code,5)
codeErr
```

```
[a^6 + a^4 + a^2 + 1, a^6 + a^5 + a + 1, a^5 + a^4 + 1, a^7 +
a^5 + a^3 + a^2 + a, a^7 + a^5 + a^4 + a^3 + a^2, a^6 + a^5 +
a + 1, a^6 + a^4 + a^3 + a^2, a^7]
```

Décodage après erreur :

```
decodeGRS(codeErr,alpha,v,k)
```

```
(a^6 + a^5 + a^2 + a)*X^7 + (a^6 + a^4 + a^3 + a + 1)*X^6 + (
a^5 + a^2 + a + 1)*X^5 + (a^7 + a^4 + a^3 + a + 1)*X^4 + (a^6
+ 1)*X^3 + (a + 1)*X^2 + (a^6 + a^5 + a^3)*X + a^7 + a^5 + a^
+ a
```

Après une unique erreur l'interpolation de Lagrange donne n'importe quoi.

3. Correction d’erreurs grâce aux GRS

3.1 Le polynôme syndrome

```
def syndrome_bigsum(code,v,a,k):
   n=len(code)
   sum=0
   for i in range(len(code)):
       somme = 0
       Li=Lagrange(a,i)
       for j in range(n-k):
           somme = somme + (a[i]*X)**j
       sum += (code[i]/(v[i]*Li(a[i])))*somme
   return sum
```

```
code=codeGRS(a^6*X^2+a^4*X+a^2+1,k,v,alpha)
code
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6
a^3 + 1]
```

```
S=syndrome_bigsum(code,v,alpha,k)
S
```

```
0
```

On remarque donc que le code est parfaitement celui qui a été envoyé.

```
def is_corrigible(S,k,n,e=R(0)):
   r=n-k
   d=r+1
   if(S==0):
       if(e.hamming_weight()<=d-1):
           return "Unscrambling will succeed"
       if(e.hamming_weight()>=d):
           return "Unscrambling might be a failure"
   if(S<>0 and e.hamming_weight()<=1):
       return "Decoding may be possible y''=y'-e"
```

```
is_corrigible(S,k,n)
```

```
'Unscrambling will succeed'
```

Supposons maintenant qu'il y ai eu une erreur sur le chemin.

```
code=codeGRS(a^6*X^2+a^4*X+a^2+1,k,v,alpha)
```

```
code
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6
a^3 + 1]
```

On génère une erreur sur le chemin.

```
codeErr=error_trans(code,1)
codeErr
```

```
[a^7 + a^2 + a + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a^3
a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6 +
a^3 + 1]
```

```
S=syndrome_bigsum(codeErr,v,alpha,k)
S
```

```
(a^5 + a^3 + a^2)*X^3 + (a^5 + a^3 + a^2)*X^2 + (a^5 + a^3 +
+ a^5 + a^3 + a^2
```

On a bien le polynôme Syndrôme qui ne vaut pas 0, signe qu'il y a bien eu des problèmes lors de la transmission.

```
def clef(S,n,k):
    if S==0:
        print "Le polynôme syndrôme est nul vous n'avez pas
besoin de cet algo"
    r=n-k
    T = [X^r,1+0*X,0+0*X]
    Y = [S,0+0*X,1+0*X]
    Z = [0,0,0]
    ct=0
    while Y[0].degree() >= (r/2) and ct < 10:
        Z = [T[i]-Y[i]*(T[0]//Y[0]) for i in range(len(T))]
        T=Y
        Y=Z
        ct = ct+1
    (locT,evaT) = (Y[2],Y[0])

    #verification
    if((Y[1]*(X**r)+locT*S)<>evaT or evaT.degree()>=(r/2)):
        print "Error"
    return [locT/locT(0), evaT/locT(0)]
```

On récupére les clés.

```
key=clef(S,n,k)
key
```

```
[X + 1, a^5 + a^3 + a^2]
```

```
def Erreur(key,a,v,poly=0):
    Errors=0
    Errb=[0 for i in range(len(a))]
    u=[]
    for i in range(len(a)):
        Li=Lagrange(a,i)
        u.append(1/(v[i]*Li(a[i])))
    primeLoc=key[0].derivative()

    for b in range(n):
        if(key[0](1/a[b])==0):
            Errb[b]=(-(a[b]*key[1]
(1/a[k]))/(u[b]*primeLoc(1/a[b])))

    if(poly==0):
       return Errb
    else:
        for i in range(len(a)):
            Errors += Errb[i]*X^(i)
        return Errors
```

```
e=Erreur(key,alpha,v)
ePoly=Erreur(key,alpha,v,1)
print ePoly
print e
```

```
a^7 + a^6 + a^4 + a
[a^7 + a^6 + a^4 + a, 0, 0, 0, 0, 0, 0, 0]
```

Voyons si l'erreur peut être correctible.

```
is_corrigible(S,k,n,ePoly)
```

```
"Decoding may be possible y''=y'-e"
```

On effectue la correction nécessaire.

```
codeCor=[codeErr[i]-e[i] for i in range(n)]
codeCor
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6
a^3 + 1]
```

Regardons si le polynôme syndrôme vaut bien 0.

```
syndrome_bigsum(codeCor,v,alpha,k)
```

```
0
```

Parfait décodons alors.

```
decodeGRS(codeCor,alpha,v,k)
```

```
a^6*X^2 + a^4*X + a^2 + 1
```

Nous avons donc corrigé l'erreur.

Testons sur un plus grand nombres d'erreurs.

```
code=codeGRS(a^6*X^2+a^4*X+a^2+1,k,v,alpha)
code
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^6 + a^5 + a^4 + a + 1, a^6 + a^4 + a^3 + a^2, a^6
a^3 + 1]
```

```
codeErr=error_trans(code,2)
codeErr
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^6 + a, a^6 + a^4 + a^3 + a^2, a^7 + a^6 + a^3 + a^
```

```
S=syndrome_bigsum(codeErr,v,alpha,k)
S
```

```
(a^6 + a^4 + a^3 + 1)*X^3 + (a^5 + a^2 + a + 1)*X^2 + (a^4 +
1)*X + a^7 + a^5 + a^4 + a^3 + a^2 + a
```

```
key=clef(S,n,k)
key
```

```
[(a^7 + a^6 + a^3 + a^2 + 1)*X^2 + (a^7 + a^5)*X + 1, (a^5 +
a^7 + a^5 + a^4 + a^3 + a^2 + a]
```

```
e=Erreur(key,alpha,v)
ePoly=Erreur(key,alpha,v,1)
print ePoly
print e
```

```
(a^5 + a^3 + 1)*X^7 + (a^7 + a^5 + a^4 + a + 1)*X^5
[0, 0, 0, 0, 0, a^7 + a^5 + a^4 + a + 1, 0, a^5 + a^3 + 1]
```

```
is_corrigible(S,k,n,ePoly)
```

```
"Decoding may be possible y''=y'-e"
```

```
codeCor=[codeErr[i]-e[i] for i in range(n)]
codeCor
```

```
[a^6 + a^4 + a^2 + 1, a^5 + a^4 + a^3, a^5 + a^4 + 1, a^6 + a
+ a + 1, a^7 + a^6 + a^5 + a^4 + 1, a^6 + a^4 + a^3 + a^2, a^
+ a^5 + a^2]
```

```
syndrome_bigsum(codeCor,v,alpha,k)
```

```
0
```

```
decodeGRS(codeCor,alpha,v,k)
```

```
(a^7 + a^5 + a^3 + a^2 + a + 1)*X^7 + (a^7 + a^6 + a^5 + a^4
1)*X^6 + (a^4 + a)*X^5 + a^3*X^4 + (a^6 + a^4 + 1)*X^3 + (a^5
1)*X^2 + (a^7 + a^3 + 1)*X + a^7 + a^6 + a^5 + a^3
```

À partir de deux erreurs l'algorithme de corrections ne foncitonne plus. Il faut rajouter de la redondance, autrement dit augmenter N.

4. Conclusion : une chaîne de transmission cryptée robuste

On reprends nos fonctions RSA du TP précédant :

```
q=2**8
c='a'
n=100
k=70
r=n-k

G=GF(q,c)
c=G.gen()
R=PolynomialRing(G,'X')
X=R.gen()

a=G.multiplicative_generator()
v=[G.one() for i in range(n)]
alpha=[a**i for i in range(n)]
```

```
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
```

```
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
```

```
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
```

```
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
```

On crée une fonction qui va découper les messages numériques en paquets

```
def splitering(X, c, k) :
  kbin = bin(k)[2:]
  sum = 0
  l = ceil(len(kbin)/8.0)
  for i in range(l) :
    pack = int(kbin[-8:], base = 2)
    p = 0
    while pack > 0 :
      if (pack & 1) :
        sum += X**i * c**p
      pack >>= 1
      p += 1
    kbin = kbin[:-8]
  return sum
```

Et une autre qui va se charger de les regrouper.

```
def gathering(poly) :
  coeff = poly.coeffs()
  d = 0
  n = 0
  for c in coeff :
    n += int(c.int_repr()) << 8*d
    d += 1
  return n
```

On génère la clef RSA :

```
key1=cleRSA(128)
key1
```

```
(26557571031376865984509743006411285096016399127701284191551
557965309045475519244075150975953291300582361161252273202716
204134756454677124491310669451392246832211915585618641413132
451036907077796879106456505358864721496559217464065460302427
100040611717843942052525248045985315183831313703234514002427
846138577620285800388508474996745794619220431610182791342936
```

1. Message alphabétique clair :

```
message="Message top secret qui ne doit pas tomber entre les
mains de Malory : Le code de la salle contenant vous savez
quoi est sf54ds45sfd4 . Terminé"
```

2. Message numérique crypté RSA :

```
cryptedMessage=encryption(numerise(message,key1[0]),key1)
```

3. Message codé GRS envoyé :

```
splittedMessage = [splitering(X, a, m) for m in
cryptedMessage]
```

```
GRScodedMessage=[codeGRS(m,k,v,alpha) for m in
splittedMessage]
```

4. Message reçu avec erreurs :

Attention, à cause d'une erreur cela ne fonctionne pas. Passez à l'étape 5.

```
GRScodedMessagewithErrors=[error_trans(m,1) for m in
```

```
GRScodedMessage]
```

```
S=[syndrome_bigsum(m,v,alpha,k) for m in
GRScodedMessagewithErrors]
```

```
listKey=[clef(m,n,k) for m in S]
```

```
e=[Erreur(m,alpha,v) for m in listKey]
```

```
GRScorrectedMessage=[[GRScodedMessagewithErrors[i][j]-e[i][j]
for j in range(len(e[i]))]for i in range(len(e))]
```

```
S=[syndrome_bigsum(m,v,alpha,k) for m in GRScorrectedMessage]
S
```

```
[0, 0, 0]
```

5. Message décodé GRS :

```
GRSdecodedMessage=[decodeGRS(m,alpha,v,k) for m in
GRScorrectedMessage]
```

```
gatheredMessage=[gathering(m) for m in GRSdecodedMessage]
```

6. Message numérique décrypté :

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

```
decryptedMessage=decrypt(gatheredMessage,key1)
```

7. Message alphabétique reconstitué

```
message_final=alphabetise(decryptedMessage,key1[0])
print message_final
```

```
Message top secret qui ne doit pas tomber entre les mains de
: Le code de la salle contenant vous savez quoi est sf54ds45s
Terminé
```

Augmentons le nombre d'erreurs.

```
key1=cleRSA(128)
key1
```

```
(341851405055587978423884490779842140848391298524268394508426
998993781538408295210295858738690647832320999972197793902023 1
321872829895055529378570628531427950926850029539187464171237 2
464491584266278699791843271132331555051769884748861999975585 6
188480547055890051962233975340459289444826666705821794504213
071970999549437244298589010373344286363806787252400272337125 6
```

```
message="Message top secret qui ne doit pas tomber entre les
mains de Malory : Le code de la salle contenant vous savez
quoi est sf54ds45sfd4 . Terminé"
```

```
cryptedMessage=encryption(numerise(message,key1[0]),key1)
```

```
splittedMessage = [splitering(X, a, m) for m in
cryptedMessage]
```

```
GRScodedMessage=[codeGRS(m,k,v,alpha) for m in
splittedMessage]
```

```
GRScodedMessagewithErrors=[error_trans(m,5) for m in
GRScodedMessage]
```

```
S=[syndrome_bigsum(m,v,alpha,k) for m in
GRScodedMessagewithErrors]
```

```
listKey=[clef(m,n,k) for m in S]
```

```
e=[Erreur(m,alpha,v) for m in listKey]
```

```
GRScorrectedMessage=[[GRScodedMessagewithErrors[i][j]-e[i][j]
for j in range(len(e[i]))]for i in range(len(e))]
```

```
S=[syndrome_bigsum(m,v,alpha,k) for m in GRScorrectedMessage]
S
```

```
[(a^7 + a^6 + 1)*X^29 + (a^7 + a^4 + a^2 + a + 1)*X^28 + (a^6
+ a^2 + a + 1)*X^27 + (a^6 + a^5 + a^3)*X^26 + (a^5 + a^4 + a
+ (a^7 + a^4 + 1)*X^24 + a^5*X^23 + (a^5 + a^4 + a^2 + 1)*X^2
(a^6 + a^4 + a^3 + a^2 + a + 1)*X^21 + (a^5 + a^3 + a^2 + a)*
(a^7 + a^4 + a^2)*X^19 + (a^7 + a^6 + a^5 + a^2 + a)*X^18 + (
a)*X^17 + (a^6 + a^4 + a^3 + a)*X^16 + (a^6 + a^5 + a)*X^15 +
a^6 + a^5 + a)*X^14 + (a^3 + a^2 + 1)*X^13 + (a^7 + a^3 + a^2
a)*X^12 + (a^7 + a^6 + 1)*X^11 + (a^7 + a^5 + a^4 + 1)*X^10 +
```

```
a^3 + a^2 + a)*X^9 + (a^3 + 1)*X^8 + (a^5 + a^3 + 1)*X^7 + (a
a^6 + a^5 + a^4 + a^3)*X^6 + (a^7 + a^4 + a^3 + a^2 + a)*X^5
+ a + 1)*X^4 + (a^5 + a^4 + a^3 + a^2 + a)*X^3 + (a^6 + a^5 +
a^3)*X^2 + (a^5 + a^4 + a^2)*X + a^7 + a + 1, (a^6 + a^2)*X^2
(a^7 + a^3 + a^2 + a + 1)*X^28 + (a^4 + a^3 + 1)*X^27 + (a^7
a^2)*X^26 + a^7*X^25 + (a^6 + a^5 + a^4 + a + 1)*X^24 + (a^4
a^2)*X^23 + (a^6 + a^5 + a)*X^22 + (a^5 + a^2)*X^21 + (a^6 +
1)*X^20 + (a^4 + a^3 + a)*X^19 + (a^2 + a + 1)*X^18 + (a^7 +
a^5 + a^4)*X^17 + (a^7 + a^5 + a^4 + a^3 + a^2 + 1)*X^16 + (a
a^3 + a^2 + a + 1)*X^15 + (a^7 + a^4 + a^3 + a^2 + a + 1)*X^1
(a^7 + a^5 + a^3 + a + 1)*X^13 + (a^7 + a^6 + a^5 + a^4 + a^2
a)*X^12 + (a^6 + a^5 + a^4 + a^2 + a + 1)*X^11 + (a^4 + a^3 +
1)*X^10 + (a^7 + a^5 + a^3 + a^2 + 1)*X^9 + (a^6 + a^5 + a^4)
(a^6 + a^4)*X^7 + (a^5 + a^4 + a^3 + a^2 + a + 1)*X^6 + (a^7
a^4 + a^3 + a^2 + 1)*X^5 + (a^6 + a^5 + a^3 + a)*X^4 + (a^7 +
a^5 + a + 1)*X^3 + (a^6 + a^3 + a^2 + a + 1)*X^2 + (a^7 + a^6
+ a + 1)*X + a^7 + a^4 + a^2 + a + 1, (a^7 + a^5 + a^2 + a +
+ (a^6 + a^5 + a^4 + a + 1)*X^28 + (a^7 + a^6 + a^5 + a^4 +
a^2)*X^27 + (a^7 + a^5 + a^4)*X^26 + (a^7 + a^2)*X^25 + (a^7
a^2)*X^24 + (a^7 + a^5 + a^4)*X^23 + (a^6 + a^5 + a^4 + a^3 +
1)*X^22 + (a^5 + a^4 + a^2 + a)*X^21 + (a^7 + a^6 + a^5 + a^4
a^3)*X^20 + (a^7 + a^6 + a^5 + a^4 + a^3 + a^2 + 1)*X^19 + (a
a^5 + a^2)*X^18 + (a^7 + a^5 + a^2 + 1)*X^17 + (a^7 + a^6 + a
a^4 + a^2 + a + 1)*X^16 + (a^7 + a^5 + a^4 + a^3 + a^2 + a)*X
(a^6 + a^2 + a)*X^14 + (a^5 + a^4 + a^3 + a^2 + a)*X^13 + (a^
+ a^5 + a^3 + a)*X^12 + (a^6 + a^5 + a^3 + a^2 + a + 1)*X^11
+ a^6 + a^4 + a^3 + a^2 + a)*X^10 + (a^7 + a^6 + a^3 + a + 1)
(a^7 + a^5 + a^4 + a + 1)*X^8 + (a^5 + a^3 + a^2)*X^7 + a^6*X
a^2*X^5 + (a^6 + a^5 + a^4 + a^3 + a^2 + 1)*X^4 + (a^7 + a^6
a^4)*X^3 + (a^7 + a^6 + a^5 + a^3 + a^2)*X^2 + (a^6 + a^2 + a
a^7 + a^6 + a^5 + a^3 + a]
```

Cela n'augure rien de bon pour la suite.

```
GRSdecodedMessage=[decodeGRS(m,alpha,v,k) for m in
GRScorrectedMessage]
```

```
gatheredMessage=[gathering(m) for m in GRSdecodedMessage]
```

```
decryptedMessage=decrypt(gatheredMessage,key1)
```

```
message_final=alphabetise(decryptedMessage,key1[0])
print message_final
```

```
Ku5yBqh!gV‹],<#[o2i3̈,G+FtJ*WY7'Gک(WP{L"qe'5ٲc e`:X
9:#KO?\Q"uzXFM(vYE=/lbB
```