PKE/KEM - Kyber

Kyber is among the first post-quantum schemes to be standardized and already found its way into products. As a lattice-based system, Kyber is fast and its security guarantees are linked to an NP-hard problem. Also, it has all the nice mathematical ingredients to confuse the hell out of you: vectors of odd-looking polynomials, algebraic rings, error terms and a security reduction to "module lattices".

https://media.ccc.de/v/rc3-2021-cwtv-230-kyber-and-post-quantum

Objetivo:

Implementar o esquema Kyber/Crystalis em classes Python/SageMath apresentando, para cada um, as versões KEM-IND-CPA e PKE-IND-CCA.

KYBER-CPAPKE

Versão que permite obter uma segurança do tipo IND-CPA (segurança contra ataques Chosen Plaintext Attacks).

KYBER-CCAKEM

Versão que permite obter uma segurança do tipo IND-CCA (segurança contra ataques Chosen Ciphertext Attacks).

Nota: Implementação baseada no documento Kyber (https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf)

In [117]:

```
import math, os, numpy as np
from hashlib import sha3_512 as G, shake_128 as XOF, sha3_256 as H, shake_256 as PRF
```

In [118]:

```
# parametros do kyber
    n = 256
    _n = 9
    q = 7681
    Qq = PolynomialRing(GF(q), 'x')
    y = Qq.gen()
    RQ = QuotientRing(Qq, y^n+1)



    # Definir função MOD
    def modMm(r,a):
        _r = r % a
        # Testar se a é par
        if mod(a,2)==0 :
            # Cálculo dos limites -a/2 e a/2
            inf_bound, sup_bound = -a/2, a/2
        else :
            # Se for impar -> calcular os limites -a-1/2 e a-1/2
            inf_bound, sup_bound = (-a-1)/2, (a-1)/2
        # Queremos garantir que o módulo se encontre no intervalo calculado
        while _r > sup_bound :
            _r-=a
        while _r < inf_bound :
            _r+=a
        return _r


    # Converte um array de bytes num array de bits
```

```
    def bytesToBits(bytearr):
        bitarr = []
        for elem in bytearr :
            bitElemArr = []
            # Calculamos cada bit pertencente ao byte respetivo
            for i in range(0,8) :
                bitElemArr.append(mod(elem // 2**(mod(i,8)),2))

            for i in range(0,len(bitElemArr)) :
                bitarr.append(bitElemArr[i])
        return bitarr


    # Converte um array de bits num array de bytes
    def bitsToBytes(bitarr):
        bytearr = []
        bit_arr_size = len(bitarr)
        byte_arr_size = bit_arr_size / 8
        for i in range(byte_arr_size) :
            elem = 0
            for j in range(8) : # Definir macro BYTE_SIZE = 0
                elem += (int(bitarr[i*8+j]) * 2**j)
            bytearr.append(elem)
        return bytearr
```

Passo 1, transformar uma stream de bytes na sua representação NTT (NTT significa "Transformada de Fourier de Números Teoria"), é usada na função de geração de chaves e na função de cifra para a matriz gerada nessas funções. Esta representação permite que o algoritmo seja mais eficiente na multiplicação de polinómios que é mais rápido tendo um tempo de computação de O(n log n) (que cresce linearmente com o valor de n) comparativamente ao habitual tempo O(n^2) (que cresce exponencialmente com o valor de n).

In [119]:

```
# Função parse que como input recebe uma byte stream e retorna a representação NTT,
    # descrita no algoritmo 1 na página 6 do Documento
    def parse(b):
        coefs = [0]*n # O poly terá n=256 coeficientes
        i,j = 0,0
        while j<n :
            d = b[i] + 256*b[i+1]
            d = mod(d,2**13)
            if d<q :
                coefs[br(8,j)] = d
                j+=1
            i+=2
        return RQ(coefs)


    class NTT(object):

        def __init__(self, n=256):
            if not any([n == t for t in [32,64,128,256,512,1024,2048]]):
                raise ValueError("improper argument ",n)
            self.n = n
            self.q = 2*n+1
            while True:
                if (self.q).is_prime():
                    break
                self.q += 2*n
            #print(self.q)
            self.F = GF(self.q) ;  self.R = PolynomialRing(self.F, name="xbar")
            w = (self.R).gen(); self.w = w

            g = (w^n + 1)
            #print(len(g.roots(multiplicities=False)))
            xi = g.roots(multiplicities=False)[-1]
            self.xi = xi
            rs = [xi^(2*i+1)  for i in range(n)]
            self.base = crt_basis([(w - r) for r in rs])
```

```
        def ntt(self,f): #Função NTT com as equacoes (4) e (5) na pagina 6 do documento
            def _expand_(f):
                u = f.list()
                return u + [0]*(self.n-len(u))

            def _ntt_(xi,N,f):
                if N==1:
                    return f
                N_ = N/2 ; xi2 =  xi^2
                f0 = [f[2*i]   for i in range(N_)] ; f1 = [f[2*i+1] for i in range(N_)]

                ff0 = _ntt_(xi2,N_,f0) ; ff1 = _ntt_(xi2,N_,f1)

                s  = xi ; ff = [self.F(0) for i in range(N)]
                for i in range(N_):
                    a = ff0[i] ; b = s*ff1[i]
                    ff[i] = a + b ; ff[i + N_] = a - b
                    s = s * xi2
                return ff

            return _ntt_(self.xi,self.n,_expand_(f))


        def ntt_inv(self,ff):    # transformada inversa
            return sum([ff[i]*self.base[i] for i in range(self.n)])

        def random_pol(self,args=None):
            return (self.R).random_element(args)
```

In [120]:

```
# Função que implementa o bit reversed order. Recebe como input "_bits" que sao:
    # o nº de bits usados para representar nr e "nr" -> o valor a ser bitreversed
    def br(_bits, nr):
        res = 0
        for i in range(_bits) :
            res += (nr % 2) * 2**(_bits-i-1)
            nr = nr // 2
        return res

    # Definição da função compress
    def compress(q,x,d):
        return mod(round((2**d)/q * int(x)),2**d)


    # Definição da função decompress
    def decompress(q,x,d):
        return round((q/2**d) * ZZ(x))
```

Passo 2, funcão CBD, Centered Binomial Distribution (Distribuição Binomial Centralizada). É a função usada para introduzir ruído "indistinguível" do ruído aleatório que também é introduzido. Este ruído vai ser usado tanto na criação das chaves pública e privada (na função keygen) como também na cifra da mensagem (na função encryption). Este processo é fundamental para assegurar a segurança do algoritmo Kyber

In [121]:

```
# Definição da função CBD. Recebe como input o n (comprido) e o array de bytes
    def cbd(noise, btarray):
        f = []
        bitArray = bytesToBits(btarray)
        for i in range(256) :
            a, b = 0, 0
            # Cálculo do a e do b
            for j in range(256) :
                a+=bitArray[2*i*noise + j]
                b+=bitArray[2*i*noise + noise + j]
            f.append(a - b)
        return RQ(f)
```

Passo 3, as funções de codificação e descodificação: É necessário serializar os polinómios (vetores de polinómios) em matrizes de bytes, portanto, precisamos definir como serializamos e des-serializamos os polinómios. Na função Decode des-serializamos uma matriz num polinómio $f = f_0 + f_1 X + \cdots + f_{255} X^{255}$ (n = 256). Definimos a função Encode como o inverso de Decode. Sempre que aplicamos Encode a um vetor de polinómios, codificamos cada polinómio individualmente e concatenamos as matrizes de bytes de saída.

In [122]:

```
# Função decode
    def decode(l, btarray):
        f = []
        bitArray = bytesToBits(btarray)
        for i in range(256) :
            fi = 0
            for j in range(l) :
                fi += int(bitArray[i*l+j]) * 2**j
            f.append(fi)
        return RQ(f)


    # Função encode
    def encode(l, poly):
        bitArr = []
        coef_array = poly.list()
        # Percorremos cada coeficiente
        for i in range(256) :
            actual = int(coef_array[i])
            for j in range(l) :
                bitArr.append(actual % 2)
                actual = actual // 2
        return bitsToBytes(bitArr)
```

In [123]:

```
# Função de multiplicação entre duas entradas de vetores/matrizes de forma pointwise (coe
ficiente a coeficiente).
    # Recebe como input e1 e e2 -> emento/entrada da matriz/vetor
    def pointwise_mult(e1,e2) :

        mult_vector = []
        for i in range(n) :
            mult_vector.append(e1[i] * e2[i])
        return mult_vector


    # Função de soma entre duas entradas de vetores/matrizes de forma pointwise (coeficie
nte a coeficiente).
    # Recebe como input e1 e e2 -> emento/entrada da matriz/vetor
    def pointwise_sum(e1,e2) :

        sum_vector = []
        for i in range(n) :
            sum_vector.append(e1[i] + e2[i])
        return sum_vector



    def multMatrixVector(M,v,k) :
        T = NTT()
        As = []
        for i in range(k) :
            As.append([0] * n)
            for j in range(k) :
                As[i] = pointwise_sum(As[i],pointwise_mult(M[i][j], v[j]))
            #As[i] += T.ntt_inv(M[i][j] * v[j])
        return As
```

Teste ao funcionamento das funcoes encode, decode e conversao de arrays bytes/bits

In [124]:

```
arr = [32,42,34,5,35,3,5,7,54,34,21,43,5,2,46,7,3,3,21,43,53,0,0,0,0,0,0,0,0,0,0,0,32,42,34,5,35,3,5,7,54,34,21,43,5,2,46,7,3,3,21,43,53,0,0,0,0,0,0,0,0,0,0,0]
poly = decode(2,arr)
print(len(poly.list()))
print(poly.list())
#print(len(encode(2,poly)))
print(arr)
print(encode(2,poly))
if arr == encode(2,poly) :
    print('\n Os arrays correspondem!')
else :
    print('\n Algo correu mal com o decode/encode!')
```

```
256
[0, 0, 2, 0, 2, 2, 2, 0, 2, 0, 2, 0, 1, 1, 0, 0, 3, 0, 2, 0, 3, 0, 0, 0, 1, 1, 0, 0, 3, 1,
0, 0, 2, 1, 3, 0, 2, 0, 2, 0, 1, 1, 1, 0, 3, 2, 2, 0, 1, 1, 0, 0, 2, 0, 0, 0, 2, 3, 2, 0,
3, 1, 0, 0, 3, 0, 0, 0, 3, 0, 0, 0, 1, 1, 1, 0, 3, 2, 2, 0, 1, 1, 3, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 2, 2, 0, 2, 0, 2, 0, 1, 1, 0, 0, 3, 0, 2, 0, 3, 0,
0, 0, 1, 1, 0, 0, 3, 1, 0, 0, 2, 1, 3, 0, 2, 0, 2, 0, 1, 1, 1, 0, 3, 2, 2, 0, 1, 1, 0, 0,
2, 0, 0, 0, 2, 3, 2, 0, 3, 1, 0, 0, 3, 0, 0, 0, 3, 0, 0, 0, 1, 1, 1, 0, 3, 2, 2, 0, 1, 1,
3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[32, 42, 34, 5, 35, 3, 5, 7, 54, 34, 21, 43, 5, 2, 46, 7, 3, 3, 21, 43, 53, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 32, 42, 34, 5, 35, 3, 5, 7, 54, 34, 21, 43, 5, 2, 46, 7, 3, 3, 21, 43, 5
3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[32, 42, 34, 5, 35, 3, 5, 7, 54, 34, 21, 43, 5, 2, 46, 7, 3, 3, 21, 43, 53, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 32, 42, 34, 5, 35, 3, 5, 7, 54, 34, 21, 43, 5, 2, 46, 7, 3, 3, 21, 43, 5
3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

 Os arrays correspondem!
```

PKE

Utilizamos, para parâmetros, os valores especificados no documento referenciado anteriormente,e ainda recorremos ao uso das seguintes funções:

- **XOF com SHAKE-128;**
- **H com SHA3-256;**
- **G com SHA3-512;**
- **PRF(s,b) com SHAKE-256($s||b$);**
- **KDF com SHAKE-256.**

Geração de chaves

A função keygen() não recebe parâmetros como input e produz um par de chaves (chave pública,chave privada) como output.

- **Calculamos ρ e σ, usando a função G com um array de bytes d gerado aleatoriamente**
- **De seguida, geramos a matriz A, a partir de ρ, e a sua representação NTT Â**
- **Depois são gerados s e e, também a partir de ρ**
- **São calculadas as representações NTT dos arrays ^s e ê**
- **Calculamos ^t, a representação NTT da multiplicação da matriz Â com o vetor ^s e adicionamos o vetor ê**
- **Finalmente, são calculadas as chaves:**
  - **a chave pública pk através do encode da concatenação de ^t modulo q com ρ**
  - **a chave privada sk através do encode da concatenação de ^s modulo q**
- **Devolvemos, então, o par de chaves**

Cifragem

A função de cifragem (*encryption*) recebe como argumentos a chave pública pk, uma mensagem m e um array r gerado aleatoriamente. Como output, devolve a mensagem cifrada c.

- Começamos por calcular o decode da chave pública pk e a sua representação NTT ^t
- Obtemos ρ a partir da chave pública\n",
- Geramos a matriz Â da mesma forma que na geração de chaves
- Geramos r e e_1, a partir de ρ
- Geramos ainda e_2, também a partir de ρ
- Calculamos a representação NTT de r, ^r
- Multiplicamos a Â por ^r e calculamos o NTT inverso do resultado, adicionando ao resultado o e_1, obtemos u
- Calculamos decompressed_m, fazendo o $decode_1(m)$ e o $decompress_q$ do resultado com 1
- Calculamos v, fazendo a multiplicação de ^t com ^r e calculando o NTT inverso do resultado, adicionando ainda e_2 e decompressed_m
- Finalmente, obtemos c1 e c2:
  - c1 é obtido do $encode_{du}$ de u $compress_q$ com d_u
  - c2 é obtido do $encode_{dv}$ de v $compress_q$ com d_v
- Devolvemos c, que é a concatenação de c1 com c2

Decifragem

A função de decifra recebe como argumentos a chave privada sk e o texto a decifrar c. Devolve a mensagem m original.

- Obtemos u a partir do $decompress q$ do $decode du$ de c com d_u
- Obtemos v a partir do $decompress q$ do $decode dv$ da 2ª componente de c (c2) com dv
- Obtemos ^s calculando o $decode 12$ de sk
- Calculamos û, a representação NTT de u
- Obtemos mult, que é o resultado da multiplicação de û com ^s, e calculamos o seu NTT inverso (que podemos representar por ^mult)
- Calculamos a diferença dif entre v e ^mult
- Finalmente, obtemos m através do $encode_1$ do $compress_q$ de dif com 1

In [125]:

```
class KyberPKE :       # classe PKE

    def __init__(self,n,k,q,nn,du,dv,dt) :

        self.n = n
        self.k = k
        self.q = q
        self.nn = nn
        self.du = du
        self.dv = dv
        self.dt = dt
        Qq = PolynomialRing(GF(q), 'x')
        y = Qq.gen()
        RQ = QuotientRing(Qq, y^n+1)
        self.Rq = RQ

    # Gerar a chave pública
    def keygen(self) :
        d = os.urandom(32)
        h = G(d)
        digest = h.digest()
        ro,sigma = digest[:32], digest[32:]
        N = 0
        A, s, e = [], [], []
        # Construção da matriz A
        for i in range(self.k) :
            A.append([])
            for j in range(self.k) :
```

```
                xof = XOF()
                xof.update(ro + j.to_bytes(4,'little') + i.to_bytes(4,'little'))
                A[i].append(parse(xof.digest(int(q))))

        # Construção do vetor s
        for i in range(self.k) :
            prf = PRF()
            prf.update(sigma + int(N).to_bytes(4,'little'))
            s.append(cbd(self.nn, prf.digest(int(q+1))))
            N += 1
        # Construção do vetor e
        for i in range(self.k) :
            prf = PRF()
            prf.update(sigma + int(N).to_bytes(4,'little'))
            e.append(cbd(self.nn, prf.digest(int(q))))
            N += 1
        # Calculo do ntt de s
        T = NTT()
        _s = []
        for i in range(self.k) :
            _s.append(T.ntt(s[i]))

        t = multMatrixVector(A,_s,self.k)

        for i in range(self.k) :
            t[i] = T.ntt_inv(t[i])
            t[i] = self.Rq(pointwise_sum(t[i].list(),e[i].list()))

        #print(t[0].list())
        # compress(q,t,dt)
        # percorremos cada um dos polinomios do vetor t
        for i in range(self.k) :
            lst = t[i].list()
            for j in range(len(t[i].list())) :
                lst[j] = compress(self.q,lst[j],self.dt)
            t[i] = self.Rq(lst)

        # Aqui temos que t = compress(q,t,dt)
        # Calculamos agora pk = (encode(dt,t) || ro)
        pk = []
        for i in range(self.k) :
            res = encode(self.dt,t[i])
            #dec = decode(self.dt,res)
            #print(t[i].list()) ; print() ; print(dec.list())
            #print(len(t[i].list()))
            for j in range(len(res)) :
                pk.append(res[j])
        #print(len(pk))
        for i in range(len(ro)) :
            pk.append(ro[i])

        # Para cada polinomio :
        for i in range(self.k) :
            # Para cada coeficiente do polinomio :
            lst = _s[i]
            for j in range(len(_s[i])) :
                lst[j] = mod(_s[i][j],self.q)
            _s[i] = lst
        # agora tratamos do encode :
        sk = []
        for i in range(self.k) :
            res = encode(13,self.Rq(_s[i]))
            for bt in res :
                sk.append(bt)

        return(pk,sk)

    # Função que vai cifrar as mensagens
    # Recebe como input: pk(chave privada gerada), m(mensagem q vai ser cifrada) e r(Rand
om Coins)
    def encryption(self, pk, m, r):
```

```python
def byteArrToBytes(btArray) :
    byts = b''
    for i in btArray :
        byts += i.to_bytes(1,'little')
    return byts

N = 0
t = []

# Implementação do decode(dt,pk)
for i in range(self.k) :
    part = pk[i*32*self.dt:i*32*self.dt+32*self.dt]
    #print(len(part))
    t.append(decode(self.dt,part))
# Implementação do decompress(q,decode(dt,pk),dt)
for i in range(self.k) :
    lst = t[i].list()
    for j in range(len(t)) :
        lst[j] = decompress(self.q,lst[j],self.dt)
    t[i] = self.Rq(lst)

ro = byteArrToBytes(pk[self.dt*self.k*self.n/8:])

At = []
# Construção da matriz A
for i in range(self.k) :
    At.append([])
    for j in range(self.k) :
        xof = XOF()
        xof.update(ro + i.to_bytes(4,'little') + j.to_bytes(4,'little'))
        At[i].append(parse(xof.digest(int(self.q))))
rr, e1 = [], []
# Construção do vetor rr
for i in range(self.k) :
    prf = PRF()
    prf.update(r + int(N).to_bytes(4,'little'))
    rr.append(cbd(self.nn, prf.digest(int(q+1))))
    N += 1
# Construção do vetor e1
for i in range(self.k) :
    prf = PRF()
    prf.update(r + int(N).to_bytes(4,'little'))
    e1.append(cbd(self.nn, prf.digest(int(self.q))))
    N += 1

prf = PRF()
prf.update(r + int(N).to_bytes(4,'little'))
e2 = cbd(self.nn, prf.digest(int(self.q)))

# Cálculo do ^rr :
_rr = []
T = NTT()
for i in range(self.k) :
    _rr.append(T.ntt(rr[i]))

# Cálculo do vetor em Rq u
u = multMatrixVector(At,_rr,self.k)

for i in range(self.k) :
    u[i] = T.ntt_inv(u[i])
    u[i] = self.Rq(pointwise_sum(u[i].list(),e1[i].list()))

# Cálculo do v :
v = [0] * self.n
    # Calculo de NTT(t) transposta :
for i in range(self.k) :
    t[i] = T.ntt(t[i])

# Calculo de v = NTT(t)T . _rr :
for i in range(self.k) :
    v = pointwise_sum(v,pointwise_mult(t[i],_rr[i]))
```

```python
        # Calculo de v = NTT-1(NTT(t)T . _rr)
        v = T.ntt_inv(v)
            # Calculo de v = NTT-1(NTT(t)T . _rr) + e2
        v = pointwise_sum(v.list(),e2.list())
        v = self.Rq(v)

        # Calculamos o decode(1,decompress(q,m,1))
        decompressed_m = []
        for i in range(len(m)) :
            decompressed_m.append(decompress(self.q, m[i], 1))

        decompressed_m = decode(1, decompressed_m)

        # Calculo do valor final de v:
        v = self.Rq(pointwise_sum(v.list(),decompressed_m.list()))

        # Cálculo de c1 :
        c1 = []
        # Calculo de compress(q,u,du) :
        for i in range(self.k) :
            lst = u[i].list()
            for j in range(len(u[i].list())) :
                lst[j] = compress(self.q,lst[j],self.du)
            u[i] = self.Rq(lst)
            # Calculo de encode(du,compress(q,u,du))
        for i in range(self.k) :
            u[i] = encode(self.du,u[i])
            for bt in u[i] :
                c1.append(bt)


        # Cálculo de c2 :
        # Calculo de compress(q,v,dv) :
        lst = v.list()
        for i in range(len(v.list())) :
            lst[i] = compress(self.q,lst[i],self.dv)
        v = self.Rq(lst)
            # Calculo de encode(dv,compress(q,v,dv)) :
        c2 = encode(self.dv,v)

        return c1+c2

    # Função que vai decifrar mensagens
    def decryption(self,sk,ct):

        T = NTT()
        c1 = ct[:self.du*self.k*self.n/8]
        c2 = ct[self.du*self.k*self.n/8:]

        # Calculo de u = decompress(q,decode(du,ct),du):
        u = []
        # Calculo de decompress(q,decode(du,ct),du) :
        for i in range(self.k) :
            part = c1[i*32*self.du:i*32*self.du+32*self.du]
            u.append(decode(self.du,c1))
            lst = u[i].list()
            for j in range(len(u[i].list())) :
                lst[j] = decompress(self.q,lst[j],self.du)
            u[i] = self.Rq(lst)

        # Calculo de v
        v = decode(self.dv,c2)
        lst = v.list()
        for i in range(len(v.list())) :
            lst[i] = decompress(self.q,lst[i],self.dv)
        v = self.Rq(lst)

        # Calculo de _s :
        _s = []
        for i in range(self.k) :
            _s.append(decode(13,sk[i*32*13:i*32*13+32*13]))
```

```
        # Calculo de NTT(u) :
        for i in range(self.k) :
            u[i] = T.ntt(u[i])
            # Calculo de sT . NTT(u) :
        mult = self.Rq([])
        for i in range(self.k) :
            mult = pointwise_sum(mult,pointwise_mult(_s[i],u[i]))
            # Calculo de NTT-1(sT . NTT(u)) :
        mult = T.ntt_inv(mult)
            # Calculo de v - NTT-1(sT . NTT(u)) :
        dif = [0] * self.n
        for i in range(self.n) :
            dif[i] = v.list()[i]-mult.list()[i]
            # Calculo de m = compress(q,v - NTT-1(sT . NTT(u)),1)
        m = []
        for i in range(self.n) :
            m.append(compress(self.q,dif[i],1))

        m = encode(1,self.Rq(m))

        return m
```

In [126]:

```
k = KyberPKE(n=256,k=2,q=7681,nn=5,du=11,dv=3,dt=11)
(pk,sk) = k.keygen()

print('Tamanho da chave publica: ',len(pk))
#print('\nChave privada: ')
#print(sk)

m = [32,4,35,78,64,45,2,35,64,45,2,35,53,34,54,32,32,4,35,78,64,45,2,35,64,45,2,35,53,3
4,54,32]
print('Mensagem a cifrar: ',m) ; print()

ct = k.encryption(pk,m,os.urandom(32))

dct = k.decryption(sk,ct)

print('Mensagem decifrada: ',dct) ; print()

#print('Chave pública: ')
#print(pk)
#print('\nChave privada: ')
#print(sk)
```

```
Tamanho da chave publica:  736
Mensagem a cifrar:  [32, 4, 35, 78, 64, 45, 2, 35, 64, 45, 2, 35, 53, 34, 54, 32, 32, 4, 3
5, 78, 64, 45, 2, 35, 64, 45, 2, 35, 53, 34, 54, 32]

Mensagem decifrada:  [10, 237, 155, 168, 192, 186, 187, 169, 94, 224, 19, 22, 175, 228, 1
75, 190, 2, 255, 30, 42, 179, 43, 23, 229, 35, 221, 10, 124, 150, 65, 78, 199]
```

In [ ]:

```

```