Exercicio2

May 2, 2024

[ ]:
```
import random
import numpy as np
from pickle import dumps, loads
%pip install sagemath-standard
from sage.all import *
from hashlib import shake_128, shake_256,sha256,sha512
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: sagemath-standard in /usr/lib/python3/dist-
packages (9.5)
Requirement already satisfied: cysignals>=1.10.2 in
/home/fura/.sage/local/lib/python3.10/site-packages (from sagemath-standard)
(1.11.4)
Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
#class NTT:

    #Inicialização da classe
    #def __init__(self, n=128, q=None):

        #if not  n in [32,64,128,256,512,1024,2048]:
            #raise ValueError("improper argument ",n)
        #self.n = n
        #if not q:
            #self.q = 1 + 2*n
            #while True:
                #if (self.q).is_prime():
                    #break
                #self.q += 2*n
        #else:
            #if q % (2*n) != 1:
                #raise ValueError("Valor de 'q' não verifica a condição NTT")
            #self.q = q

        #self.F = GF(self.q) ;  self.R = PolynomialRing(self.F, name="w")
        #w = (self.R).gen()

        #g = (w^n + 1)
```

```
        #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])

    # Método para calcular o ntt para um polinómio
    #def ntt(self,f):

        #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))

    # Método para calcular o inverso de ntt, retornando o polinomio original
    #def invNtt(self,ff):
        #return sum([ff[i]*self.base[i] for i in range(self.n)])
```

[ ]:
```
n = 256

q = 7681

#q = next_prime(3*n)
#while q % (2*n) != 1:
#    q = next_prime(q+1)

#print(q)

coins = os.urandom(32)

Z.<w> = ZZ[]
x = w^n +1
```

```
R.<w> = QuotientRing(Z ,Z.ideal(x))

Zq.<w> = GF(q)[]
Rq1.<w> = QuotientRing(Zq ,Zq.ideal(x))

Rq = lambda x : Rq1(R(x))

ntt = NTT(n,q)
```

[ ]:
```
def bytesToBits(array):
    bits = []

    for b in array:
        arr = []
        for i in range(0,8):
            arr.append(mod(b//2**(mod(i,8)),2))
            for i in range(0,len(arr)):
                bits.append(arr[i])

    return bits
```

[ ]:
```
def compress(x,d) :
    c = x.list()
    c2 = []
    p = int(2 ** d)
    for x in c:
        new = round(p / q * int(x)) % p
        c2.append(new)

    return Rq(c2)

def decompress(x,d):
    c = x.list()
    c2 = []
    p = 2 ** d
    for x in c:
        new = round(q / p * int(x))
        c2.append(new)

    return Rq(c2)
```

[ ]:
```
#SHA3-512 (input tamanho variável -> Duplo de 32 bytes)
def G(a):
    digest = sha512(str(a).encode()).digest()
    return digest[:32],digest[32:]

#SHAKE-256 (input de 32 bytes e variável -> output variável)
```

```
def PRF(b,b1):
    return shake_256(str(b).encode() + str(b1).encode()).digest(int(q))

#SHAKE-128 (3 inputs -> output variável)
def XOF(p,i,j):
    return shake_128(str(p).encode() + str(i).encode() + str(j).encode()).
 ↪digest(int(q))

#SHA3-256 (input variável -> output de 32 bytes)
def H(x):
    return sha256(str(s).encode()).digest()
```

[ ]:
```
def Parse(b):
    x = []
    i = 0
    j = 0

    while j < n:
        d1 = b[i] + 256 * mod(b[i+1],16)
        d2 = b[i+1]//16 + 16 * b[i+2]

        if d1 < q :
            x.append(d1)
            j = j+1
        if d2 < q and j<n:
            x.append(d2)
            j = j+1
        i = i+3

        return Rq(x)

def decode(array,l):

        f = []

        bitArray = bytesToBits(array)

        for i in range(len(array)):

            fi = 0

            for j in range(l):

                fi += int(bitArray[i*l+j]) * 2**j

            f.append(fi)

```

```
        return Rq(f)


def CBD(array,b):

        f=[0]*n

        bitArray = bytesToBits(array)

        for i in range(256):

            a = 0
            b = 0

            for j in range(b):

                a += bitArray[2*i*b + j]
                b += bitArray[2*i*b + b + j]

            f[i] = a-b

        return Rq(f)
```

[ ]:
```
# Multiplica uma matriz por um vetor

def mulMatrizVetor(M, v, k, n):
    res = [[0] * n] * k

    for i in range(len(M)):
        for j in range(len(M[i])):
            M[i][j] = [M[i][j][x] * v[j][x] for x in range(n)]

    for i in range(len(M)):
        for j in range(len(M[i])):
            res[i] = [res[i][x] + M[i][j][x] for x in range(n)]

    return res

# Soma duas matrizes

def somaMatrizes(m1, m2, n):
    res = []
    for i in range(len(m1)):
        res.append([m1[i][j] + m2[i][j] for j in range(n)])
    return res
```

```python
# Soma elementos de vetores

def somaV(v1, v2, n):
    res = [0] * n
    for i in range(n):
        res[i] = v1[i] + v2[i]

    return res

def sumVet(v1, v2, n):
    return [v1[i] + v2[i] for i in range(n)]

# Subtrai elementos de vetores

def subVet(v1, v2, n):
    return [v1[i] + v2[i] for i in range(n)]

# Multiplica elementos de dois vetores

def multV(v1, v2,n):

    res = []
    for i in range(n):
        res.append((v1[i] * v2[i]))

    return res


# Multiplicação de matrizes

def mulMatrizes(m1, m2, n):

    for i in range(len(m1)):
        m1[i] = multV(m1[i], m2[i],n)

    tmp = [0] * n
    for i in range(len(m1)):
        tmp = somaV(tmp, m1[i],n)

    return tmp
```

[ ]:
```python
class KYBER_PKE_INDCPA:

    # Parametros retirados secção 1.4 (KYBER 512)
    def __init__(self):
        self.n = 256
        self.q = 7681
```

```
        self.k = 2
        self.n1 = 3
        self.n2 = 2
        self.du = 10
        self.dv = 4

    #Algoritmo de geração de chaves
    def keyGen(self):

        d = Rq1.random_element()
        ro,sigma = G(d)
        N = 0

        A = [0,0]

        # Generate matrix A   Rq
        for i in range(self.k):
            A[i] = []
            for j in range(self.k):

                A[i].append(ntt.ntt(Parse(XOF(ro,j,i))))

        # Sample s   Rq
        s = [0] * self.k
        for i in range(self.k):

            s[i] = ntt.ntt(CBD(PRF(sigma,N), self.n1))
            N += 1

        # Sample e   Rq
        e = [0] * self.k
        for i in range(self.k):

            e[i] = ntt.ntt(CBD(PRF(sigma,N), self.n1))
            N += 1

        # pk := As + e

        # sk := s
        mult = mulMatrizVetor(A, s, self.k,self.n)

        t = somaMatrizes(mult,e,self.n)

        pubk = t, ro

        privk = s
```

```
        return pubk, privk

    # Cifragem
    def cifragem(self,pubk, mensagem, coins):

        N = 0
        t, ro = pubk

        # Generate matrix A (transposta)   Rq
        tA = [0,0]
        for i in range(self.k):

            tA[i] = []

            for j in range(self.k):

                tA[i].append(ntt.ntt(Parse(XOF(ro,i,j))))

        # Sample r   Rq
        r = [0] * self.k
        for i in range(self.k):
            r.insert(i,ntt.ntt(CBD(PRF(coins, N), self.n1)))
            N += 1

        # Sample e1   Rq
        e1 = [0] * self.k
        for i in range(self.k):
            e1.insert(i,CBD(PRF(coins, N), self.n2))
            N += 1

        # Sample e2   Rq
        e2 = CBD(PRF(r, N), self.n2)

        # u := A * r + e1
        aux = mulMatrizVetor(tA, r, self.k, self.n)

        # inverso de ntt, NTT ^ -1
        aux2 = []
        for i in range(len(aux)) :
            aux2.append(ntt.invNtt(aux[i]))

        aux3 = somaMatrizes(aux2, e1, self.n)

        u = []
        for i in range(len(aux3)) :
            u.append(Rq(aux3[i]))
```

```
        # v := t * r + e2 + Decompress(m, 1)

        vAux = mulMatrizes(t,r, self.n)

        vAux1 = ntt.invNtt(vAux)

        vAux2 = Rq(somaV(vAux1,e2, self.n))

        m1 = decompress(mensagem, 1)

        v = Rq(sumVet(vAux2,m1,self.n))

        # c := (Compress(u, du), Compress(v, dv))
        c1 = [0] * len(u)

        for i in range(len(u)):
            c1.append(compress(u[i],self.du))

        c2 = compress(v,self.dv)

        return (c1,c2)

    # Decifragem
    def decifragem(self,privk, ct):

        c1, c2 = ct

        s = privk

        u = []

        for i in range(len(c1)):
            u.append(decompress(c1[i],self.du))


        v = decompress(c2,self.dv)


        untt = []
        for i in range(len(u)) :
            untt.append(ntt.ntt(u[i]))

        aux = mulMatrizes(s,untt,self.n)

        aux1 = subVet(v,ntt.invNtt(aux), self.n)

        # m := Compressq(v - sT u, 1))
```

```
        m = compress(Rq(aux1), 1)

        return m

    # Encapsulamento

    def enc(self, pk):
        # nounce
        n = Rq1.random_element()
        # hashe do nonce
        shared_key = H(n)

        ct = self.cifragem(pk, decompress(n, 1), coins)

        return ct, shared_key

    # Desencapsulamento

    def dec(self, ct):
        # decifra o encapsulamento
        n = self.decifragem(ct)

        # hash de forma a voltar a ter a chave partilhada
        shared_key = H(p)

        return shared_key


# TESTE #

kyber = KYBER_PKE_INDCPA()

pubk, privk = kyber.keyGen()

#print(pubk)

#print(privk)


ct = kyber.cifragem(pubk, decode('criptografia'.encode(),1), coins)

c1, c2 = ct

print(c1)

print(c2)
```

```python
m = kyber.decifragem(privk, ct)

print(m)
```

```
[0, 0, 0, 0]
8*w^11 + 8*w^10 + 8*w^7 + 8*w^6 + 8*w^4 + 8*w^3 + 8*w^2 + 8*w + 8
w^11 + w^10 + w^7 + w^6 + w^4 + w^3 + w^2 + w + 1
```