PKE Kyber

keypair

Função que gera as chaves pública e privada usadas na cifragem e decifragem

dec

Função de decifragem que recebe a chave privada e um criptograma e retorna a mensagem original.

enc

Função de cifragem que recebe a chave pública e uma mensagem e retorna o criptograma resultante.

CBD

Função que recebe uma bytestream e um inteiro e retorna um polinómio em Rq.

Parse

Função que recebe uma bytestream e retorna um polinómio em Rq.

BytesToBits

Função que transforma um bytearray em bitarray.

compress

Recebe um elemento x ∈ Z e retorna um inteiro em {0, . . . , 2^d − 1} com d < log2(q).

decompress

Recebe um elemento {0, . . . , 2^d − 1} com d < log2(q) e retorna um inteiro em x ∈ Z.

rounding

Função que arredonda x com x ∈ Q, para o inteiro mais próximo (rounded up).

Reverse

Inverte uma lista.

In [300]:

```python
import math
import os
from hashlib import sha3_512
from hashlib import shake_128
from hashlib import shake_256


class Kyber:


    def __init__(self):

        self.q = 3329
        self.n = 256
        self.k = 2
        self.du = 10
        self.dv = 4
        self.nbin1 = 3
        self.nbin2 = 2
        self.R = PolynomialRing(GF(3329),'x')
        self.x = self.R.gen()
        self.fq = (self.x^self.n + 1)
        self.Rq = QuotientRing(self.R,self.fq)

    def rounding(self,rational):
         res = math.ceil(rational)

         if(rational < 0):

            res = res * (-1)

         if(res < rational):
            return rational

         return res

    def Reverse(self,lst):

            return [ele for ele in reversed(lst)]


    def compress(self,x,d):
        h = list(x)
        l = []
        for i in h:


            l.append(self.rounding(((2^d)/self.q) * lift(i)) % 2^d)
        return l

    def decompress(self,x,d):

            l = []
            for i in x:
                l.append(self.rounding(self.q/(2^d)) * i)
            return l


    def BytesToBits(self,bytestream):

        bytes_as_bits = ''.join(format(ord(bytes([byte])), '08b') for byte in bytestream)

        return bytes_as_bits


    def Parse(self,bytestream):

        i = 0
        j = 0
        a = []

        while j < self.n and i < self.n and i + 2 < self.n:

            d1 = bytestream[i] + 256 * (bytestream[i+1] % 16)
            d2 = self.rounding(bytestream[i+1] / 16) + 16 * bytestream[i+2]

            if d1 < self.q :

                a.append(d1)
                j = j + 1

            if d2 < self.q and j < self.n and i < self.n:

                a.append(d2)
                j = j + 1

            i = i + 3

        return self.Rq(a)



    def CBD(self,bytestream,nbin):

        bitstream = self.BytesToBits(bytestream)
        f = []
        i = 0

        while i < self.n - 1:

            j = 0
            while j < nbin- 1:

                a = int(bitstream[2*i*nbin + j])
                b = int(bitstream[2*i*nbin + nbin + j])

                j = j + 1

            i = i + 1
            f.append(a-b)

        return self.Rq(f)

    # Byte array of 32l bytes
    def Decode(self,bytestream):

        l = len(bytestream) / 32
        f = []
        bitstream = self.BytesToBits(bytestream)
        i = 0

        while i < self.n - 1:

            j = 0
            while j < l - 1:

                f.append(bitstream[i*l + j] * 2^j)

                j = j + 1

            i = i + 1

        return self.Rq(f)



    def Encode(self,poly):

        h = list(poly)
        print(len(h) / 32)
        l = len(h) / 32
        res = []
        i = 0

        while i < self.n - 1:

            j = 0
            while j < l - 1:


                res.append(h[i*l + j] / 2^j)

                j = j + 1

            i = i + 1

        return res

    def XOF(self,p,i,j):

        m = shake_128()
        dig = p + bytes([i]) + bytes([j])
        m.update(dig)

        return m.digest(int(self.n))

    def PRF(self,o,N):

        m = shake_256()
        dig = o + bytes([N])
        m.update(dig)

        return m.digest(int(self.n))



    def mult_mat_vec(self, matrix, vector):
        res = [None] * (self.k)
        for i in range(self.k):
            res[i] = 0
            for j in range(self.k):
                mult = self.Rq(vector[j]) * self.Rq(matrix[i * (self.k) + j])
                res[i] += self.Rq(mult)
        return res

    def sum_vec(self,vec1,vec2):

        res = [None]*self.k

        i = 0
        while i < self.k:

                res[i] = self.Rq(vec1[i]) + self.Rq(vec2[i])

                i = i + 1

        return res

    def sub_vec(self,vec1,vec2):

        res = [None]*self.k

        i = 0
        while i < self.k:

                res[i] = self.Rq(vec1[i]) - self.Rq(vec2[i])

                i = i + 1

        return res


    def mult_vec(self,vec1,vec2):

        res = [0]*self.k

        i = 0
        while i < self.k:

            res[i] = res[i] + self.Rq(vec1[i]) * self.Rq(vec2[i])
            i = i + 1

        return res


    def keygen(self):

        d = os.urandom(32)
        p = sha3_512(d).digest()
        o = p[-32:]
        A = [None] * ((self.k)* (self.k))
        s = [None]*self.k
        e = [None]*self.k
        N = 0
        t = [None]*self.k


        i = 0
        while i < self.k:

            j = 0
            while j < self.k:

                index = i*(self.k)+j
                A[index] = self.Parse(self.XOF(p,i,j))
                j = j + 1

            i = i + 1

        i = 0
        while i < self.k:

            s[i] = self.CBD(self.PRF(o,N),self.nbin1)
            N = N + 1
            i = i + 1

        i = 0
        while i < self.k:

            e[i] = self.CBD(self.PRF(o,N),self.nbin1)
            N = N + 1
            i = i + 1

        i = 0

        t = self.sum_vec(self.mult_mat_vec(A,s),e)

        pk = t
        sk = s

        return (pk,sk)

    def enc(self,pub_key,message):

        rcoins = os.urandom(32)
        A = [None] * ((self.k)* (self.k))
        r = [None]*self.k
        e1 = [None]*self.k
        e2 = [None]*self.k
        c1 = [None]*self.k
        c2 = [None]*self.k
        N = 0
        p = sha3_512(rcoins).digest()
        i = 0

        while i < self.k:
            j = 0
            while j < self.k:
                index = i*(self.k)+j
                A[index] = self.Parse(self.XOF(p,i,j))
                j = j + 1
            i = i + 1

        i = 0
        while i < self.k:

            r[i] = self.CBD(self.PRF(rcoins,N),self.nbin1)
            i = i + 1
            N = N + 1

        i = 0
        while i < self.k:

            e1[i] = self.CBD(self.PRF(rcoins,N),self.nbin2)
            i = i + 1
            N = N + 1

        e2 = self.CBD(self.PRF(rcoins,N),self.nbin2)
        u = self.sum_vec(self.mult_mat_vec(A,r),e1)
        v = self.sum_vec(self.sum_vec(self.mult_vec(pub_key,r),e2),self.decompress(message,1
))

        i = 0
        while i < self.k:

            c1[i] = self.compress(u[i],self.du)
            c2[i] = self.compress(v[i],self.dv)

            i = i + 1
        return (c1,c2)

    def dec(self,sec_key,cif):

        i = 0
        u = [None]*self.k
        v = [None]*self.k
        aux = [None]*self.k

        (c1,c2) = cif

        u = self.decompress(c1,self.du)

        while i < self.k:

            u[i] = self.decompress(c1[i],self.du)

            i = i + 1

        v = self.decompress(c2,self.dv)

        while i < self.k:

            aux[i] = self.mult_vec(sec_key[i],u[i])

            i = i + 1

        # message = self.compress(self.sub_vec(v,aux),1)
        return []
```

In [301]:

```python
test = Kyber()
td = test.Parse(os.urandom(256))
(p,s) = test.keygen()
cif = test.enc(p,mess)
test.dec(s,cif)
```

Out[301]: []