EC - Trabalho 3: Criptossistemas pós-quânticos PKE/KEM

Exercício 1

Alínea ii.

Nesta alínea é pedido que seja implementado o *NewHope* numa classe Python/SageMath.

Começamos por criar a classe NTT - que contém as funções que fazem o NTT(Number Theoretic Transform) e o seu inverso - que foi partilhada pelo professor.

In [22]:

```
class NTT(object):
#
    def __init__(self, n=128):
        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 = 1 + 2*n
        while True:
            if (self.q).is_prime():
                break
            self.q += 2*n

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

        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])


    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))

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

Aqui está definida a classe que implementa o NewHope-CPA-PKE, no qual as versões CPA-KEM e CCA-KEM se baseiam.

Todas as funções desta classe, com a exceção de `polyBitRev`, foram baseadas na documentação do NewHope atualizada a 10 de Abril de 2020, sendo que `polyBitRev` baseia-se na implementação do NewHope feita em C e presente no repositório do [GitHub do NewHope (https://github.com/newhopecrypto/newhope/blob/master/ref/ntt.c)](https://github.com/newhopecrypto/newhope/blob/master/ref/ntt.c).

Ao criar a classe são estabelecidas as variàveis `n` e `q`, que se referem ao nível de segurança e módulo utilizado respetivamente, e também `bitrev_table` que será utilizada na função `polyBitRev`.

Os três principais funções desta classe são:

```
- keyGenPKE() - responsável por gerar o par de chaves;
- encryptionPKE() - responsável pela encriptação;
- decryptionPKE() - responsável pela desencriptação.
```

In [98]:

```
# Na nossa notação duas letras repetidas, como por exemplo "aa", servem para representar o
# "s'" é representado por "sl", e "s''" é representado "sll"

import hashlib

class NewHope_CPA_PKE:

    # Parâmetros do NewHope1024 (pág.19 da documentação 2020, Tabela 2 e parte final da pág
    def __init__(self):
        self.n = 1024       # Dimension n
        self.q = 12289      # Modulus q
        #self.k = 8          # Noise parameter k
        #self.y = 7          # NTT parameter y(letra grega que aqui será representada por "y
        #self.w = 49         # n-th root of unity

        self.bitrev_table = [0,512,256,768,128,640,384,896,64,576,320,832,192,704,448,960,3
  16,528,272,784,144,656,400,912,80,592,336,848,208,720,464,976,48,560,304,816,176,688,432,
  8,520,264,776,136,648,392,904,72,584,328,840,200,712,456,968,40,552,296,808,168,680,424,9
  24,536,280,792,152,664,408,920,88,600,344,856,216,728,472,984,56,568,312,824,184,696,440,
  4,516,260,772,132,644,388,900,68,580,324,836,196,708,452,964,36,548,292,804,164,676,420,9
  20,532,276,788,148,660,404,916,84,596,340,852,212,724,468,980,52,564,308,820,180,692,436,
  12,524,268,780,140,652,396,908,76,588,332,844,204,716,460,972,44,556,300,812,172,684,428,
  28,540,284,796,156,668,412,924,92,604,348,860,220,732,476,988,60,572,316,828,188,700,444,
  2,514,258,770,130,642,386,898,66,578,322,834,194,706,450,962,34,546,290,802,162,674,418,9
  18,530,274,786,146,658,402,914,82,594,338,850,210,722,466,978,50,562,306,818,178,690,434,
  10,522,266,778,138,650,394,906,74,586,330,842,202,714,458,970,42,554,298,810,170,682,426,
  26,538,282,794,154,666,410,922,90,602,346,858,218,730,474,986,58,570,314,826,186,698,442,
  6,518,262,774,134,646,390,902,70,582,326,838,198,710,454,966,38,550,294,806,166,678,422,9
  22,534,278,790,150,662,406,918,86,598,342,854,214,726,470,982,54,566,310,822,182,694,438,
  14,526,270,782,142,654,398,910,78,590,334,846,206,718,462,974,46,558,302,814,174,686,430,
  30,542,286,798,158,670,414,926,94,606,350,862,222,734,478,990,62,574,318,830,190,702,446,
  1,513,257,769,129,641,385,897,65,577,321,833,193,705,449,961,33,545,289,801,161,673,417,9
  17,529,273,785,145,657,401,913,81,593,337,849,209,721,465,977,49,561,305,817,177,689,433,
  9,521,265,777,137,649,393,905,73,585,329,841,201,713,457,969,41,553,297,809,169,681,425,9
  25,537,281,793,153,665,409,921,89,601,345,857,217,729,473,985,57,569,313,825,185,697,441,
  5,517,261,773,133,645,389,901,69,581,325,837,197,709,453,965,37,549,293,805,165,677,421,9
  21,533,277,789,149,661,405,917,85,597,341,853,213,725,469,981,53,565,309,821,181,693,437,
  13,525,269,781,141,653,397,909,77,589,333,845,205,717,461,973,45,557,301,813,173,685,429,
  29,541,285,797,157,669,413,925,93,605,349,861,221,733,477,989,61,573,317,829,189,701,445,
  3,515,259,771,131,643,387,899,67,579,323,835,195,707,451,963,35,547,291,803,163,675,419,9
  19,531,275,787,147,659,403,915,83,595,339,851,211,723,467,979,51,563,307,819,179,691,435,
  11,523,267,779,139,651,395,907,75,587,331,843,203,715,459,971,43,555,299,811,171,683,427,
  27,539,283,795,155,667,411,923,91,603,347,859,219,731,475,987,59,571,315,827,187,699,443,
  7,519,263,775,135,647,391,903,71,583,327,839,199,711,455,967,39,551,295,807,167,679,423,9
  23,535,279,791,151,663,407,919,87,599,343,855,215,727,471,983,55,567,311,823,183,695,439,
  15,527,271,783,143,655,399,911,79,591,335,847,207,719,463,975,47,559,303,815,175,687,431,
  31,543,287,799,159,671,415,927,95,607,351,863,223,735,479,991,63,575,319,831,191,703,447,


    # "Using a similar notation, by r<--Rq we declare that a variable r is a polynomial in
    def genA(self, ps): # "ps" representa a publicSeed
        aa = [0] * self.n
        extSeed = bytearray(33)
        extSeed[0:32] = ps[0:32]
        for i in range(0, self.n/64):
            ctr = 0
            extSeed[32] = i
            state = hashlib.shake_128(extSeed)
            while ctr < 64:
```

```python
            buf = state.digest(int(168*1)) # buf é um output de tamanho 168*j; neste ca
            j = 0
            while (j < 168) and (ctr < 64):
                int1 = int(buf[j])
                int2 = (int(buf[j+1]) << 8) % 2**32
                val = int1|int2
                if val < (5*self.q):
                    aa[i*64+ctr] = val % self.q
                    ctr = ctr + 1
                j = j + 2
        return aa

    # Com base em: https://github.com/newhopecrypto/newhope/blob/master/ref/ntt.c
    def polyBitRev(self, poly):
        for i in range(0, self.n):
            r = self.bitrev_table[i]
            if i < r:
                tmp = poly[i]
                poly[i] = poly[r]
                poly[r] = tmp
            i += 1
        return poly

    def sample(self, s, nonce):
        r = [0] * self.n
        extSeed = bytearray(34)
        extSeed[0:32] = s[0:32]
        extSeed[32] = nonce
        for i in range(0, (self.n/64)):
            extSeed[33] = i
            buf = hashlib.shake_256(extSeed).digest(int(128))
            for j in range(0,63):
                a = buf[2*j]
                b = buf[2*j+1]
                # "To compute the Hamming weight, the sum of all bits that are set to one i
                r[64*i+j] = (bin(a).count("1") + self.q - bin(b).count("1")) % self.q
        return r

    # "Addition or subtraction of polynomials in Rq (denoted as + or -, respectively) is th
    def addPoly(self, e, f):
        h = [0]*self.n
        for i in range(0, self.n):
            h[i] = (e[i]+f[i]) % self.q
        return h

    def subPoly(self, e, f):
        h = [0]*self.n
        for i in range(0, self.n):
            h[i] = (e[i]-f[i]) % self.q
        return h

    def mulPoly(self, e, f):
        h = [0]*self.n
        for i in range(0, self.n):
            h[i] = (e[i]*f[i]) % self.q
        return h

    def encodePolynomial(self, sS):
        # valor = (7*n)/4 = 1792
        r = [0] * 1792 # Nesta linha dá TypeError se se utilizar "valor" em vez de 1792
        for i in range(0, (self.n/4)):
```

```python
            t0 = sS[(4*i)+0] % self.q
            t1 = sS[(4*i)+1] % self.q
            t2 = sS[(4*i)+2] % self.q
            t3 = sS[(4*i)+3] % self.q
            r[(7*i)+0] = int(t0) & int(0xff) # int(0xff)=255
            r[(7*i)+1] = (int(t0) >> 8)|((int(t1) << 6) % 2**32)&int(0xff)
            r[(7*i)+2] = (int(t1) >> 2) & int(0xff)
            r[(7*i)+3] = (int(t1) >> 10)|((int(t2) << 4) % 2**8)&int(0xff)
            r[(7*i)+4] = (int(t2) >> 4) & int(0xff)
            r[(7*i)+5] = (int(t2) >> 12)|((int(t3) << 2) % 2**8)&int(0xff)
            r[(7*i)+6] = (int(t3) >> 6) & int(0xff)
        return r

    # "The secret key is then either encoded into an array of 869 bytes (n = 512) or 1792 b
    # "The public key is encoded as an array of 928 bytes (n = 512) or 1824 bytes (n = 1024
    # 1824 = 1792 + 32; 1792 = (7*n)/4
    # A notação na documentação é um bocado confusa, por essa parte está esclarecida no com
    def encodePK(self, bB, ps):
        # valor = ((7*n)/4) + 32 #1824
        r = [0] * 1824 # Nesta linha dá TypeError se se utilizar "valor" em vez de 1824
        r[0:1792] = self.encodePolynomial(bB)
        r[1792:] = ps[0:32]
        return r

    def keyGenPKE(self):
        seed = os.urandom(32)
        tSeed = b'0x01'+seed
        z = hashlib.shake_256(tSeed).digest(int(64))
        publicSeed = z[:32]
        noiseSeed = z[32:]
        aa = self.genA(publicSeed)
        s = self.polyBitRev(self.sample(noiseSeed, 0))
        obj = NTT(1024)
        ss = obj.ntt(s)
        e = self.polyBitRev(self.sample(noiseSeed, 1))
        ee = obj.ntt(e)
        mulres = self.mulPoly(aa, ss)
        bb = self.addPoly(mulres, ee)
        pk = self.encodePK(bb, publicSeed)
        sk = self.encodePolynomial(ss)
        return pk, sk

    def decodePolynomial(self, v):
        r = [0] * self.n
        for i in range(0, (self.n/4)):
            r[4*i+0] = int(v[7*i+0]) | (((int(v[7*i+1])&int(0x3f)) << 8) % 2**32)
            r[4*i+1] = (int(v[7*i+1]) >> 6) | ((int(v[7*i+2]) << 2) % 2**32) | (((int(v[7*i
            r[4*i+2] = (int(v[7*i+3]) >> 4) | ((int(v[7*i+4]) << 4) % 2**32) | (((int(v[7*i
            r[4*i+3] = (int(v[7*i+5]) >> 2) | ((int(v[7*i+6]) << 6) % 2**32)
        return r

    # 1824 = 1792 + 32; 1792 = (7*n)/4
    def decodePK(self, pk):
        bb = self.decodePolynomial(pk[:1792])
        seed = pk[1792:]
        return bb, seed

    def encode(self, msg):
        v = [0] * self.n
        for i in range(0,32):
            for j in range(0,8):
```

```
                mask = -((msg[i] >> j)&1)
                v[8*i+j+0] = int(mask) & int(self.q/2)
                v[8*i+j+256] = int(mask) & int(self.q/2)
                # if n == 1024:  # Esta condição está em comentário pois é sempre verdadeir
                v[8*i+j+512] = int(mask) & int(self.q/2)
                v[8*i+j+768] = int(mask) & int(self.q/2)
        return v

    def compress(self, vl):
        kn = 0
        tn = [0] * 8
        hn = [0] * 384 # (3*n/8)
        for l in range(0, self.n/8):
            i = 8 * l
            for j in range(0,8):
                tn[j] = vl[i+j] % self.q
                tn[j] = int((int((int(tn[j]) << 3)) + self.q/2) / self.q) & int(7)
            hn[kn+0] = tn[0]|((tn[1] << 3))|((tn[2] << 6))
            hn[kn+1] = (tn[2] >> 2)|((tn[3] << 1))|((tn[4] << 4))|((tn[5] << 7))
            hn[kn+2] = (tn[5] >> 1)|((tn[6] << 2))|((tn[7] << 5))
            kn = kn + 3
        return hn

    # 7n/4 + 3n/8 = 1792 + 384 = 2176
    def encodeC(self, uu, h):
        c = [0] * 2176
        c[0:1792] = self.encodePolynomial(uu)
        c[1792:2176] = h
        return c

    def decodeC(self, c):
        uu = self.decodePolynomial(c[0:1792])
        h = c[1792:]
        return uu, h

    # n/8 = 1024/8 = 128
    def decompress(self, a):
        kn = 0
        r = [0] * self.n
        for l in range(0, 128):
            i = 8 * l
            r[i+0] = a[kn+0]&7
            r[i+1] = (a[kn+0] >> 3)&7
            r[i+2] = (a[kn+0] >> 6)|(((a[kn+1] << 2))&4)
            r[i+3] = (a[kn+1] >> 1)&7
            r[i+4] = (a[kn+1] >> 4)&7
            r[i+5] = (a[kn+1] >> 7)|(((a[kn+2] << 1))&6)
            r[i+6] = (a[kn+2] >> 2)&7
            r[i+7] = (a[kn+2] >> 5)
            kn = kn + 3
            for j in range(0,8):
                r[i+j] = (r[i+j] * self.q + 4) >> 3
        return r

    # "The decoding function Decode (see Algorithm 11) maps from bn=256c coefficients back
    # For example, for n = 1024, take 4 = [1024=256] coefficients (each in the range {0,...
    # accumulate their absolute values, and set the key bit to 0 if the sum is larger than
    def decode(self, v):
        u = [0] * 32
        for i in range(0,256):
            tn = abs(int(v[i+0] % self.q) - int(self.q/2))
```

```python
            tn = tn + abs(int(v[i+256] % self.q) - int(self.q/2))
            #if n == 1024 # Esta condição está em comentário pois é sempre verdadeira
            tn = tn + abs(int(v[i+512] % self.q) - int(self.q/2))
            tn = tn + abs(int(v[i+768 % self.q]) - int(self.q/2))
            tn = tn - self.q
        # As próximas 2 linhas seriam usadas se n = 512
        #else:
            #tn = tn - q/2
            tn = tn >> 15
            u[i >> 3] = u[i >> 3] | -(tn << (i&7))
        return u

    def encryptionPKE(self, pk, u, coin):
        bb, publicSeed = self.decodePK(pk)
        aa = self.genA(publicSeed)
        sl = self.polyBitRev(self.sample(coin, 0))
        el = self.polyBitRev(self.sample(coin, 1))
        ell = self.sample(coin, 2)
        obj = NTT(1024)
        tt = obj.ntt(sl)
        ntt_el = obj.ntt(el)
        mulres = self.mulPoly(aa, tt)
        uu = self.addPoly(mulres, ntt_el)
        v = self.encode(u)
        multemp = self.mulPoly(bb, tt)
        inverNtt = obj.ntt_inv(multemp)
        vltemp = self.addPoly(inverNtt, ell)
        vl = self.addPoly(vltemp, v)
        h = self.compress(vl)
        c = self.encodeC(uu, h)
        return c

    def decryptionPKE(self, c, sk):
        uu, h = self.decodeC(c)
        ss = self.decodePolynomial(sk)
        vl = self.decompress(h)
        multemp = self.mulPoly(uu, ss)
        obj = NTT(1024)
        invtemp = obj.ntt_inv(multemp)
        subtemp = self.subPoly(vl, invtemp)
        u = self.decode(subtemp)
        return u
```

A célula abaixo mostra um teste da classe NewHope_CPA_PKE.

In [102]:

```
# Teste do NewHope-CPA-PKE

nh = NewHope_CPA_PKE()
coin = os.urandom(32)
m = [randrange(0,255) for _ in range(32)]
pk, sk = nh.keyGenPKE()
ct = nh.encryptionPKE(pk, m, coin)
res = nh.decryptionPKE(ct, sk)
print(m)
print(res)
m == res
```

```
[24, 82, 120, 236, 99, 188, 52, 180, 18, 189, 212, 166, 119, 232, 146, 92, 3
9, 59, 72, 28, 118, 13, 161, 187, 133, 47, 171, 128, 36, 27, 242, 11]
[24, 82, 120, 236, 99, 188, 52, 180, 18, 189, 212, 166, 119, 232, 146, 92, 3
9, 59, 72, 28, 118, 13, 161, 187, 133, 47, 171, 128, 36, 27, 242, 11]
```

Out[102]:

```
True
```

Exercício 2

Neste exercício é pedido que sejam implementadas as versões KEM-IND-CPA e PKE-IND-CCA de cada algoritmo.

NewHope-CPA-KEM

Abaixo encontra-se uma classe que implementa o NewHope-CPA-KEM, que se baseia na implementação do NewHope-CPA-PKE implementado anteriormente.

A função *keyGen_CPA_KEM* utiliza a função de geração de chaves do NewHope-CPA-PKE para gerar um par de chaves.

A função *encapsulation_CPA_KEM* é responsável por criar e encapsular um *shared secret*, fazendo uso da função de encriptação da versão CPA-PKE.

A função *decapsulation_CPA_KEM* desencapsula e devolve o *shared secret*, usando para isso a função de desencriptação da versão CPA-PKE.

In [105]:

```
# Parte dedicada a expandir o CPA-PKE para CPA-KEM
class NewHope_CPA_KEM:
    def __init__(self):
        self.nh = NewHope_CPA_PKE()

    def keyGen_CPA_KEM(self):
        pk, sk = self.nh.keyGenPKE()
        return pk, sk

    def encapsulation_CPA_KEM(self, pk):
        coin = os.urandom(32)
        shake_in = b'0x02'+coin
        kcoin = hashlib.shake_256(shake_in).digest(int(64))
        kn = kcoin[:32]
        coinl = kcoin[32:]
        c = self.nh.encryptionPKE(pk, kn, coinl)
        ss = hashlib.shake_256(kn).digest(int(32))
        return c, ss

    def decapsulation_CPA_KEM(self, c, sk):
        kl = self.nh.decryptionPKE(c,sk)
        ss = hashlib.shake_256(bytes(kl)).digest(int(32))
        return ss
```

De seguida encontra-se um teste da classe NewHope_CPA_KEM.

In [106]:

```
# Teste do NewHope-CPA-KEM
nh = NewHope_CPA_KEM()
pk, sk = nh.keyGen_CPA_KEM()
c, ss = nh.encapsulation_CPA_KEM(pk)
res = nh.decapsulation_CPA_KEM(c, sk)
print(ss)
print(res)
ss == res
```

```
b'\x95\xb4m\xab\x1c\xbe\xc7\x1cE\xaavLXis-\xfe<\x0fM\xc7W\xd2\x04\xf5d\xda\x
95\x8f\xdd\x0f='
b'\x95\xb4m\xab\x1c\xbe\xc7\x1cE\xaavLXis-\xfe<\x0fM\xc7W\xd2\x04\xf5d\xda\x
95\x8f\xdd\x0f='
```

Out[106]:

```
True
```

NewHope-CCA-KEM

De seguida encontra-se definida a classe que implemeta o NewHope-CCA-KEM, que é necessária para a conversão para NewHope-CCA-PKE.

Mais uma vez, esta classe tem por base o NewHope-CPA-PKE implementado no início.

In [133]:

```python
# Parte dedicada a expandir o CPA-PKE para CCA-KEM e este para CCA-PKE
class NewHope_CCA_KEM:
    def __init__(self):
        self.nh = NewHope_CPA_PKE()

    def keyGen_CCA_KEM(self):
        pk, sk = self.nh.keyGenPKE()
        s = os.urandom(32)
        shakeout = hashlib.shake_256(bytes(pk)).digest(int(32))
        res1 = sk + pk
        skt = bytes(res1) + shakeout + s
        return skt

    def getKeys(self, skt):
        sk = skt[:1792]
        pk = skt[1792:3616]
        return pk, sk

    def encapsulation_CCA_KEM(self, pk):
        coin = os.urandom(32)
        shakein1 = b'0x04'+coin
        u = hashlib.shake_256(shakein1).digest(int(32))
        tempshake = hashlib.shake_256(pk).digest(int(32))
        shakein2 = b'0x08' + u + tempshake
        k_coinl_d = hashlib.shake_256(shakein2).digest(int(96))
        k = k_coinl_d[:32]
        coinl = k_coinl_d[32:64]
        d = k_coinl_d[64:]
        c = self.nh.encryptionPKE(pk, u, coinl)
        ct = c + list(d) #2208 bytes
        cbytes = b''
        for elem in c:
            cbytes += (int(elem)).to_bytes(2, 'big')
        shin = cbytes + d
        tshake = hashlib.shake_256(shin).digest(int(32))
        ss = hashlib.shake_256(k+tshake).digest(int(32))
        return ct, ss

    # Tamanho de c = 7*n/4+3*n/8 = 2176
    # Tamanho de pk = 7*n/4+32 = 1824
    # Tamanho de sk = 7*n/4 = 1792
    def decapsulation_CCA_KEM(self, ct, skt):
        c = ct[:2176]
        d = ct[2176:] # d é de tamanho 32
        sk = skt[:1792]
        pk = skt[1792:3616]
        h = skt[3616:3648] # h é de tamanho 32
        s = skt[3648:] # s é de tamanho 32
        ul = self.nh.decryptionPKE(c, sk)
        shakein = b'0x08' + bytes(ul) + h
        shaketemp = hashlib.shake_256(shakein).digest(int(96))
        kl = shaketemp[:32]
        coinll = shaketemp[32:64]
        dl = shaketemp[64:]
        cbytes = b''
        if c == self.nh.encryptionPKE(pk, ul, coinll) and bytes(d) == dl:
            fail = 0
            for elem in c:
                cbytes += int(elem).to_bytes(2, 'big')
```

```
        else:
            fail = 1
        kn = [0] * 2
        kn[0] = kl
        kn[1] = s
        c_d = cbytes + bytes(d)
        parte2 = hashlib.shake_256(c_d).digest(int(32))
        parte12 = kn[fail] + parte2
        ss = hashlib.shake_256(parte12).digest(int(32))
        return ss
```

Abaixo está um teste da classe NewHope_CCA_KEM.

In [134]:

```
# Teste do NewHope-CCA-KEM
nh = NewHope_CCA_KEM()
skt = nh.keyGen_CCA_KEM()
pk, sk = nh.getKeys(skt)
ct, ss = nh.encapsulation_CCA_KEM(pk)
res = nh.decapsulation_CCA_KEM(ct, skt)
print(ss)
print(res)
ss == res
```

```
b'\x05\xb4\x00dH\xad3\xced\x0b\x91\\\x03\x96\x07`!\xae\x92\n\xa5\xe1\x87\x8c
\x85X\x1c\xf6\xdc\xd8\xea$'
b'\x05\xb4\x00dH\xad3\xced\x0b\x91\\\x03\x96\x07`!\xae\x92\n\xa5\xe1\x87\x8c
\x85X\x1c\xf6\xdc\xd8\xea$'
```

Out[134]:

```
True
```

NewHope-CCA-PKE

Para transformar o NewHope-CCA-KEM em NewHope-CCA-PKE devem ser seguidas as "standard conversion techniques as specified by NIST" (Q13 no seguinte [link (https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/faqs)](https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/faqs)). Assim sendo, será necessário utilizar o AES no modo GCM. Para tal foi implementada a classe *AES*, como pode ser visto abaixo, que encripta e desencripta texto limpo.

In [3]:

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

class AES:
    def __init__(self):
        pass

    def genNounce(tamanho): #tamanho em bytes
        nounce = os.urandom(tamanho)
        return nounce

    def encrypt(chave, textolimpo):
        iv = self.genNounce(12) #gera um IV de 12 bytes

        #Criação de um objeto AES-GCM através da chave e do IV
        encryptor = Cipher(
            algorithms.AES(chave), #Cifra AES
            modes.GCM(iv), #Modo GCM
            backend=default_backend()
        ).encryptor()

        #Encripta o texto limpo
        ciphertext = encryptor.update(textolimpo) + encryptor.finalize()

        return (iv, ciphertext, encryptor.tag)

    def decrypt(chave, iv, ciphertext, tag):
        #Criação de um objeto AES-GCM através da chave, do IV
        decryptor = Cipher(
            algorithms.AES(chave),
            modes.GCM(iv,tag),
            backend=default_backend()
        ).decryptor()

        #Retorna o texto limpo
        return decryptor.update(ciphertext) + decryptor.finalize()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-3-d05bc41717bd> in <module>()
----> 1 from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
      2 from cryptography.hazmat.primitives import hashes, hmac
      3 from cryptography.hazmat.primitives.ciphers import Cipher, algorithm
s, modes
      4 from cryptography.hazmat.backends import default_backend
      5 from cryptography.hazmat.primitives.ciphers.aead import AESGCM

ModuleNotFoundError: No module named 'cryptography'
```

NewHope-CCA-PKE

A classe que implementa o NewHope-CCA-PKE é a seguinte.

A geração de chaves será feita de forma idêntica à de NewHope-CCA-KEM.

A encriptação é feita concatenando o criptograma da mensagem a encriptar ao criptograma gerado pelo NewHope-CCA-KEM.

Para desencriptar é feito o desencapsulamento seguido da desencriptação da mensagem.

In [148]:

```
# "NewHope-CCA-KEM can be converted to an IND-CCA-secure public key encryption scheme using
# conversion techniques as specified by NIST. In particular, shared secret ss can be used a
# in an appropriate data encapsulation mechanism in the KEM/DEM (key encapsulation mechanis
# encapsulation mechanism) framework [47]."

# https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/faqs
# "To convert a KEM to a public key encryption scheme, NIST will construct the encryption f
# by appending to the KEM ciphertext, an AES-GCM ciphertext of the plaintext message, with
# The AES key will be the symmetric key output by the encapsulate function.
# (The key generation function will be identical to that for the original KEM, and the decr
# constructed by decapsulation followed by AES decryption.)"

class NewHope_CCA_PKE:
    def __init__(self):
        self.nh = NewHope_CCA_KEM()
        self.aes = AES()

    def keyGen_CCA_PKE(self):
        skt = self.nh.keyGen_CCA_KEM()
        return skt

    def encryption_CCA_PKE(self, msg, skt):
        pk, sk = self.nh.getKeys(skt)
        kemct, ss = self.nh.encapsulation_CCA_KEM(pk) # "ss" é a chave usada no AES-GCM; ke
        self.iv, mct, self.tag = self.aes.encrypt(ss, msg) # mct = Message Ciphertext
        ct = kect + mct
        return ct

    def decryption_CCA_PKE(self, ct, skt):
        kemct = ct[:2208]
        mct = ct[2208:]
        ss = self.nh.decapsulation_CCA_KEM(skt)
        msg = self.aes.decrypt(ss, self.iv, mct, self.tag)
        return msg
```

In [150]:

```
nh = NewHope_CCA_PKE()
skt = nh.keyGen_CCA_PKE()
msg = [randrange(0,255) for _ in range(32)]
ct = nh.encryption_CCA_PKE(msg, skt)
res = nh.decryption_CCA_PKE(ct, skt)
print(msg)
print(res)
msg == res
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-150-493bf0dfb012> in <module>()
----> 1 nh = NewHope_CCA_PKE()
      2 skt = nh.keyGen_CCA_PKE()
      3 msg = [randrange(Integer(0),Integer(255)) for _ in range(Integer(32)
)]
      4 ct = nh.encryption_CCA_PKE(msg, skt)
      5 res = nh.decryption_CCA_PKE(ct, skt)

<ipython-input-148-cc232f850ba7> in __init__(self)
     20     def __init__(self):
     21         self.nh = NewHope_CCA_KEM()
---> 22         self.aes = AES()
     23 
     24     def keyGen_CCA_PKE(self):

NameError: name 'AES' is not defined
```