Neste exercício foi nos pedido para implementar em sagemath um protótipo da norma FIPS203. Esta norma funciona com 3 possíveis opções para o número de bits de segurança, 512, 768 e 1024. A principal diferencça entre estas opções são os valores da variaveis, assim como os algoritmos de hash e RBG utilizados. Como esta implementação é para motivos académicos decidimos por utilizar 512 bits de segurança. Desta forma as variavéis irão ter os seguintes valores:

- n = 256
- q = 3329
- k = 2
- η1 = 3
- η2 = 2
- du = 10
- dv = 4 Primeiramente começamos por escrever algumas funções de suporte, na sua marioria funções de hash, e a inicialização das variávies, ζ e γ.

In [1]:
```
import hashlib, os
from functools import reduce


N = 256
Q = 3329


def bit_rev_7(r):
    return int('{:07b}'.format(r)[::-1], 2)


def G(c):
    G_result = hashlib.sha3_512(c).digest()
    return G_result[:32], G_result[32:]


def H(c):
    return hashlib.sha3_256(c).digest()


def J(s, l):
    return hashlib.shake_256(s).digest(l)


def XOF(rho, i, j):
    return hashlib.shake_128(rho + bytes([i]) + bytes([j])).digest(1536)


def PRF(eta, s, b):
    return hashlib.shake_256(s + b).digest(64 * eta)


def vector_add(ac, bc):
    return [(x + y) % Q for x, y in zip(ac, bc)]


def vector_sub(ac, bc):
    return [(x - y) % Q for x, y in zip(ac, bc)]


def compress(d, x):
    return [(((n * 2**d) + Q // 2 ) // Q) % (2**d) for n in x]

def decompress(d, x):
    return [(((n * Q) + 2**(d-1) ) // 2**d) % Q for n in x]


ZETA = [pow(17, bit_rev_7(k), Q) for k in range(128)]
GAMMA = [pow(17, 2 * bit_rev_7(k) + 1, Q) for k in range(128)]


def bits_to_bytes(b):
    B = bytearray([0] * (len(b) // 8))

    for i in range(len(b)):
        B[i // 8] += b[i] * 2 ** (i % 8)

    return bytes(B)


def bytes_to_bits(B):
    B_list = list(B)
    b = [0] * (len(B_list) * 8)

    for i in range(len(B_list)):
        for j in range(8):
            b[8 * i + j] = B_list[i] % 2
            B_list[i] //= 2

    return b


def byte_encode(d, F):
    b = [0] * (256 * d)
    for i in range(256):
        a = F[i]
        for j in range(d):
```

```
            b[i * d + j] = a % 2
            a = (a - b[i * d + j]) // 2

    return bits_to_bytes(b)


def byte_decode(d, B):
    m = 2 ** d if d < 12 else Q
    b = bytes_to_bits(B)
    F = [0] * 256

    for i in range(256):
        F[i] = sum(b[i * d + j] * (2 ** j) % m for j in range(d))

    return F
```

Após termos estas funções definidas podemos começar a escrever os algoritmos utilizados para a realização do ML-KEM.
Começamos por definir as várias funções do NTT, que irão ser utilizadas tanto no K-PKE como no ML-KEM.

In [2]:
```
def sample_ntt(B):
    i, j = 0, 0
    ac = [0] * 256

    while j < 256:
        d1 = B[i] + 256 * (B[i + 1] % 16)
        d2 = (B[i + 1] // 16) + 16 * B[i + 2]

        if d1 < Q:
            ac[j] = d1
            j += 1

        if d2 < Q and j < 256:
            ac[j] = d2
            j += 1

        i += 3

    return ac

def ntt(f):
    fc = f
    k = 1
    len = 128

    while len >= 2:
        start = 0
        while start < 256:
            zeta = ZETA[k]
            k += 1
            for j in range(start, start + len):
                t = (zeta * fc[j + len]) % Q
                fc[j + len] = (fc[j] - t) % Q
                fc[j] = (fc[j] + t) % Q

            start += 2 * len

        len //= 2

    return fc


def ntt_inv(fc):
    f = fc
    k = 127
    len = 2
    while len <= 128:
        start = 0
        while start < 256:
            zeta = ZETA[k]
            k -= 1
            for j in range(start, start + len):
                t = f[j]
                f[j] = (t + f[j + len]) % Q
                f[j + len] = (zeta * (f[j + len] - t)) % Q

            start += 2 * len

        len *= 2

    return [(felem * 3303) % Q for felem in f]

def base_case_multiply(a0, a1, b0, b1, gamma):
    c0 = a0 * b0 + a1 * b1 * gamma
    c1 = a0 * b1 + a1 * b0

    return c0, c1

def multiply_ntt_s(fc, gc):
    hc = [0] * 256
    for i in range(128):
        hc[2 * i], hc[2 * i + 1] = base_case_multiply(fc[2 * i], fc[2 * i + 1], gc[2 * i], gc

    return hc
```

Com o NTT feito podemos então começar a realização do K-PKE(public-key encryption)
componente necessário para a realização do ML-KEM.
Este é consittuido por 3 algoritmos:

KeyGen()
Esta função através dos valores mencionados previamente, __k__ e η, seeds geradas aleatóriamente e o uso do NTT cria um par de chaves, sendo uma para cifragem,ek, e outra para decifragem,dk.
Encript()
Esta função recebe a chave de cifragem calculada em KeyGen(), uma mensagem ,m, e uma aleatoriadade de cifragem,r, ambas um array de 32 bytes irá gerar um texto cifrado,c. Isto irá ser feito utilzando várias funções do ntt, assim como multiplicação e adição das matrizes e arrays calculados.
Decript()
Esta função recebe o texto cifrado e a chave de decifragem e através destes reconstroi a mensagem utilizada em Encript().
Esta reconstrução é feita calculando os coeficientes e constante das equações feitas em Encript(), através da chave de decifragem.

In [3]:
```
def sample_poly_cbd(B, eta):
    b = bytes_to_bits(B)
    f = [0] * 256

    for i in range(256):
        x = sum(b[2 * i * eta + j] for j in range(eta))
        y = sum(b[2 * i * eta + eta + j] for j in range(eta))
        f[i] = (x - y) % Q

    return f


def k_pke_keygen(k, eta1):
    d = os.urandom(32)
    rho, sigma = G(d)
    N = 0
    Ac = [[None for _ in range(k)] for _ in range(k)]
    s = [None for _ in range(k)]
    e = [None for _ in range(k)]

    for i in range(k):
        for j in range(k):
            Ac[i][j] = sample_ntt(XOF(rho, i, j))

    for i in range(k):
        s[i] = sample_poly_cbd(PRF(eta1, sigma, bytes([N])), eta1)
        N += 1

    for i in range(k):
        e[i] = sample_poly_cbd(PRF(eta1, sigma, bytes([N])), eta1)
        N += 1

    sc = [ntt(s[i]) for i in range(k)]
    ec = [ntt(e[i]) for i in range(k)]
    tc = [reduce(vector_add, [multiply_ntt_s(Ac[i][j], sc[j]) for j in range(k)] + [ec[i]]) f

    ek_PKE = b"".join(byte_encode(12, tc_elem) for tc_elem in tc) + rho
    dk_PKE = b"".join(byte_encode(12, sc_elem) for sc_elem in sc)

    return ek_PKE, dk_PKE


def k_pke_encrypt(ek_PKE, m, rand, k, eta1, eta2, du, dv):
    N = 0
    tc = [byte_decode(12, ek_PKE[i * 384 : (i + 1) * 384]) for i in range(k)]
    rho = ek_PKE[384 * k : 384 * k + 32]
    Ac = [[None for _ in range(k)] for _ in range(k)]
    r = [None for _ in range(k)]
    e1 = [None for _ in range(k)]

    for i in range(k):
        for j in range(k):
            Ac[i][j] = sample_ntt(XOF(rho, i, j))

    for i in range(k):
        r[i] = sample_poly_cbd(PRF(eta1, rand, bytes([N])), eta1)
        N += 1

    for i in range(k):
        e1[i] = sample_poly_cbd(PRF(eta2, rand, bytes([N])), eta2)
        N += 1

    e2 = sample_poly_cbd(PRF(eta2, rand, bytes([N])), eta2)
    rc = [ntt(r[i]) for i in range(k)]
    u = [vector_add(ntt_inv(reduce(vector_add, [multiply_ntt_s(Ac[j][i], rc[j]) for j in rang
    mu = decompress(1, byte_decode(1, m))
    v = vector_add(ntt_inv(reduce(vector_add, [multiply_ntt_s(tc[i], rc[i]) for i in range(k)

    c1 = b"".join(byte_encode(du, compress(du, u[i])) for i in range(k))
    c2 = byte_encode(dv, compress(dv, v))

    return c1 + c2


def k_pke_decrypt(dk_PKE, c, k, du, dv):
    c1 = c[:32 * du * k]
    c2 = c[32 * du * k : 32 * (du * k + dv)]
    u = [decompress(du, byte_decode(du, c1[i * 32 * du : (i + 1) * 32 * du])) for i in range(
```

```
    v = decompress(dv, byte_decode(dv, c2))
    sc = [byte_decode(12, dk_PKE[i * 384 : (i + 1) * 384]) for i in range(k)]
    w = vector_sub(v, ntt_inv(reduce(vector_add, [multiply_ntt_s(sc[i], ntt(u[i])) for i in ra

    return byte_encode(1, compress(1, w))
```

Tendo o K-PKE feito podemos finalmente passar para as funções consituintes do ML-KEM

KeyGen()

Através da função criada em K-PKE gera um par de chaves de encapsulação e de desencapsulação. A chave de desencapsulação é criada concatenando a chave de cifragem, de decifragem, H da chave de cifragem e z, um array de 32 bytes aleatórios.

Encaps()

Utilizando a chave de encapsulamento criado em KeyGen(), ek, e m, um array de 32 bytes aleatórios, utilizando as funções de hash, H e G, é gerado o segredo partilhado K, e o valor aleatório,r. Depois são utilzados os valores calculados préviamente, ek e r, para cifrar a mensagem criada, m. Finalmente o segredo criade e a mensagem cifrada são returnados.

Decaps()

Através da chave de encapsulação criada anteriormente,dk, e a mensagem cifrada c, iremos decifrar a mensagem recebida, ml. A chave de encapsulação é uma concetenação de valores utilzados também para a cifragem, iremos então dar parse a este de maneira a recuperá-los. Após ter os valores estes iremos utilzar a mensagem decifrada e a hash da mensagem de cifragem, h, para gerar o segredo partilhado. A mensagem decifrada é então cifrada com os valores extraidos, e utilizando outra vez o K-PKE.Encript(). Isto irá servir como garantia de que nada foi alterado, de maneira a garantir que os segredos são iguais.

In [4]:
```
def ml_kem_keygen(k, eta1):
    z = os.urandom(32)
    ek_PKE, dk_PKE = k_pke_keygen(k, eta1)
    ek = ek_PKE
    dk = dk_PKE + ek + H(ek) + z

    return ek, dk


def ml_kem_encaps(ek, k, eta1, eta2, du, dv):
    m = os.urandom(32)
    K, r = G(m + H(ek))
    c = k_pke_encrypt(ek, m, r, k, eta1, eta2, du, dv)

    return K, c


def ml_kem_decaps(c, dk, k, eta1, eta2, du, dv):
    dk_PKE = dk[0: 384 * k]
    ek_PKE = dk[384 * k : 768 * k + 32]
    h = dk[768 * k + 32 : 768 * k + 64]
    z = dk[768 * k + 64 : 768 * k + 96]
    ml = k_pke_decrypt(dk_PKE, c, k, du, dv)
    Kl, rl = G(ml + h)
    Kb = J((z + c), 32)
    cl = k_pke_encrypt(ek_PKE, ml, rl, k, eta1, eta2, du, dv)
    if c != cl:
        Kl = Kb

    return Kl


def ml_kem_exec():
    #valores para 512 bits de segurança
    k = 4
    eta1 = 2
    eta2 = 2
    du = 11
    dv = 5

    ek, dk = ml_kem_keygen(k, eta1)
    #verificações de tipo
    if type(ek) != bytes or len(ek) != 384 * k + 32:
        raise ValueError('invalid ek (type check)')

    if b''.join([byte_encode(12, decoded_ek_elem) for decoded_ek_elem in [byte_decode(12, ek[:
        raise ValueError('invalid dk (type check)')

    K, c = ml_kem_encaps(ek, k, eta1, eta2, du, dv)

    if type(c) != bytes or len(c) != 32 * (du * k + dv):
        raise ValueError('invalid c (type check)')

    Kl = ml_kem_decaps(c, dk, k, eta1, eta2, du, dv)

    print('Equal shared keys?', K == Kl)


def main():
    ml_kem_exec()


if __name__ == '__main__':
    main()
```

```
Equal shared keys? True
```

In [ ]: