Dilithium

gen

Função de geração de chaves pública e privada. Começa por se gerar uma matriz de polinómios em Rq, de seguida calculam-se os vectores para a chave secreta s1 e s2. Por fim, a segunda parte da chave pública, t, é calculada e o valor retornado pela função é um par onde um elemento é a chave pública e o outro a chave privada.

sign

Função que recebendo uma mensagem e uma chave privada, faz a assiantura da mensagem, retornando esta assiantura no final. Para tal necessita da chave privada e de efetuar um hash. Deste hash resulta "c", e por fim, com "c", é possível calcular "z".

verify

Função que recebe uma mensagem, uma chave pública e o resultado da função sign e verifica se a chave corresponde à mensagem passada como argumento.

Imports e variáveis globais

In [1]:

```
import hashlib

n = 256
q = 8380417
k = 3
l = 2
peso = 60

neta = 7
gama1 = (q-1)/16
gama2 = gama1/2
beta = 375

Zx.<x>  = ZZ[]
Gq.<z>  = PolynomialRing(GF(q))

R.<x>  = Zx.quotient(x^n+1)
Rq.<z> = Gq.quotient(z^n+1)
```

Funções auxiliares

In [2]:

```
def S(limit,size):
    lista = []
    for i in range(size):
        poly = []
        for j in range(n):
            poly.append(randint(1,limit))
        lista.append(Rq(poly))
```

```
    res = matrix(Rq,size,1,lista)
    return res

def Decompose(C,alfa):

    r = mod(C,int(q))
    r0 = int(mod(r, int(alfa)))

    if (r-r0) == (q-1):
        r1 = 0
        r0 = r0 - 1
    else:
        r1 = (r-r0)/(int(alfa))

    return (r1,r0)

def auxHB(r):

    res = Decompose(r,2*gama2)

    return res[0]

def HighBits(polys):

    lista = polys.list()

    for i in range(len(lista)):
        poly = lista[i]
        polyL = poly.list()

        for j in range(len(polyL)):
            polyL[j] = auxHB(int(polyL[j]))

        lista[i] = polyL

    return lista


def LowBits(poly):
    lista = poly.list()
    for i in range(len(lista)):
        f = lista[i]
        F = f.list()
        for j in range(len(F)):
            F[j] = auxLB(int(F[j]))

        lista[i] = F

    return lista

def auxLB(C):
    res = Decompose(C,2*gama2)
    return res[1]

def normal(v):

    for i in range(2):
        norma = auxNormal(v[i],q)
        v[i] = norma
    return max(v)

def auxNormal(poly,number):
    lista = poly.list()
    for i in range(len(lista)):
        f = lista[i]
        F = f.list()
        for j in range(len(F)):
            F[j] = abs(int(F[j]))
        lista[i]=F

    List = []
    for i in range(len(lista)):
```

```
        List.append(max(lista[i]))

    return max(List)

def H(value):
    H = []
    contador = 0
    contador_ = 0
    for i in range(0,n,2):
        u=value[i]+value[i+1]
        contador = contador + 1
        if u == '11':
            H.append(0)
        if u == '01':
            H.append(1)
            contador_ = contador_ + 1
        if u == '00':
            pass
        if u == '10':
            H.append(-1)
            contador_ = contador_ + 1
        if contador_ >= peso:
            break

    for i in range(n-contador):
        H.append(0)

    return H
```

Funções principais

In [3]:

```
def gen():

    auxA = []

    for i in range(k*l):

        auxA.append(Rq.random_element())

    A = matrix(Rq,k,l,auxA)

    s1 = S(neta,l)
    s2 = S(neta,k)

    t = A*s1 + s2

    pk = (A,t)
    sk = (A,t,s1,s2)

    return (pk,sk)
```

In [4]:

```
def sign(keys,m):

    pk,sk = keys
    z = None
    c = None
    A,t = pk
    A,t,s1,s2 = sk

    flag = True

    while z == None and flag == True:

        y = S(gama1-1, l)
        Ay = A*y
```

```
        w1 = HighBits(Ay)

        string = ''
        string = string + m[2:]

        # w1 to string to be hashed
        for i in range(len(w1)):
            for j in range(len(w1[i])):
                k = bin(w1[i][j])
                if w1[i][j] >= 0:
                    string = string + k[2:]
                if w1[i][j] < 0:
                    string = string + k[3:]

        c = H(string)
        cQ = Rq(c)

        z = y + cQ*s1

        if (int(normal(z)[0])) >= (gama1-beta) and (normal(LowBits(Ay-cQ*s2))) >= (gama2
-beta):

            flag = True

        else:

            flag = false

    return (z,c)
```

In [5]:

```
def verify(pk,m,cripto):

    (z,c) = cripto
    (A,t) = pk

    cQ = Rq(c)

    w1 = HighBits(A*z-cQ*t)

    string = ''
    string = string + m[2:]

    for i in range(len(w1)):
        for j in range(len(w1[i])):
            k = bin(w1[i][j])
            if w1[i][j] >= 0:
                string = string + k[2:]
            if w1[i][j] < 0:
                string = string + k[3:]

    hashC = H(string)

    if (int(normal(z)[0])) < (gama1-beta) and hashC == c:
        print ('Passed!')
        return 1
    else:
        print ('Failed verify!')
        return 0
```

Teste

Devido a um erro de implementação, a verificação falha.

In [6]:

```
keys = gen()
pk,sk = keys
```

```
text = bin(1024)
cripto = sign(keys,text)

verify(pk,text,cripto)
```

```
Failed verify!
```

Out[6]:

```
0
```