NTRU PRIME - KEM

Parâmetros

In [1]:
```
p = 761
q = 4591
t = 143

Zx.<x> = ZZ[]
#R.<xp> = Zx.quotient(x^p - x - 1)
R.<x> = Zx.quotient(x^p - x - 1)
print(R)

F3 = GF(3)
#F3x.<x3> = F3[]
#R3.<xp3> = F3x.quotient(x^p - x - 1)
F3x.<x> = F3[]
R3.<x> = F3x.quotient(x^p - x - 1)
print(R3)

Fq = GF(q)
#Fqx.<xq> = Fq[]
#Rq.<xqp> = Fqx.quotient(x^p - x - 1)
Fqx.<x> = Fq[]
Rq.<x> = Fqx.quotient(x^p - x - 1)
print(Rq)
```

```
Univariate Quotient Polynomial Ring in x over Integer Ring with mo
dulus x^761 - x - 1
Univariate Quotient Polynomial Ring in x over Finite Field of size
3 with modulus x^761 + 2*x + 2
Univariate Quotient Polynomial Ring in x over Finite Field of size
4591 with modulus x^761 + 4590*x + 4590
```

Validação dos Parâmetros

In [2]:
```
def params_validation():
    try:
        assert p.is_prime()
        #print('p primo!')
        assert q.is_prime()
        #print('q primo!')
        assert t > 1
        #print('t>1!')
        assert p > 3*t
        #print('p>3t')
        assert q > 32*t + 1
        #print('q>32t + 1')
    except:
        print('Parâmetros inválidos!')
        return
    print('Parâmetros válidos!')

params_validation()
```

```
Parâmetros válidos!
```

Métodos de transformação

In [3]:
```
q12 = 2295 # q12 = (q//2)

def Rq_fromR(r):
    assert r in R
    ret = Rq([r[i] for i in range(p)])
    assert ret in Rq
    return ret

def R3_fromR(r):
    assert r in R
    return R3([r[i] for i in range(p)])

def nicelift(u):
    return lift(u + q12) - q12

def nicemod3(u): # r in {0,1,-1} with u-r in {...,-3,0,3,...}
    return u - 3*round(u/3)
```

Funções de verificação

In [4]:
```
'''
Função que verifica se um elemento de R é Small:
    Todos os coeficientes em {-1,0,1}
'''
def is_Small(r):
    assert r in R
    return all( abs(r[i]) <= 1 for i in range(p) )

'''
Função que verifica se um elemento tem Hamming Weight de 2t :
    (#coeficientes != 0) == (2*t)
'''
def is_HammingWeight(r):
    assert r in R
    return (2*t) == len([i for i in range(p) if r[i] != 0])

'''
Função que verifica se um elemento de R é tsmall:
    Small and HammingWeight
'''
def is_tSmall(r):
    assert r in R
    return is_Small(r) and is_HammingWeight(r)
```

Métodos de codificação

In [5]:
```
import itertools

def concat(l):
    return list(itertools.chain.from_iterable(l))

q12 = 2295 # (q12 = ((q-1)/2))
```

In [6]:

```
# ------------------------------------------------- ENCODING -----
------------------------------------------------- #
def blocks_encode(u,radix,batch):
    l =[]
    for i in range(0, len(u), batch):
        total = 0
        #print('posição do bloco: ' + str(i))
        for t in range(0,batch):
           # print('u[%d + %d] = %d',i,t,u[i+t])
            total += u[i+t] * radix^t
            #print('total: %d', total)
        l.append(total)
        #print(l)
    return l

def block_decode(u,radix,batch):
    return concat([(u[i]//radix^j)%radix for j in range(batch)] for
i in range(len(u)))

# ------------------------------------------------- ENCODING -----
------------------------------------------------- #
```

In [7]:

```
# ------------------------------------------------- Rq -------
------------------------------------------- #
'''
Este método começa por transformar os elementos de h (que estão em
Rq).
Vamos transformar esses elementos de forma a eles estarem no interv
alo [0,4590]:
    Recebe o h e transforma os elementos no intervalo ]-(q/2)-1, (q
/2)-1[.
    Adiciona q/2.

Após essa tranformação, pegamos na lista resultante(h1) e codificam
os
'''
def encodeRq(h):
    h1 = [q12 + nicelift(h[i]) for i in range(p)]+[0]*(-p % 5)
    #print(h1)
    return blocks_encode(h1,6144,5)
    #return seq2str(h1,6144,5,8)[:1218]
'''
Este método começa por transformar a sequencia em uma lista de inte
iros.
Depois verifica se a lista tem algum valor fora dos admitiveis
Por fim percorre essa lista e subtrai por q/2 (para anular a soma e
fectuada no encode)
'''
def decodeRq(hstr):
    h = block_decode(hstr,6144,5)
    #print(h)
    if max(h) >= q:
        raise Exception("pk out of range")
    return Rq([h[i]-q12 for i in range(p)])

# ------------------------------------------------- Rq -------
------------------------------------------- #
```

In [8]:
```
# ------------------------------------------------- Zx--------
------------------------------------------- #
'''
Este método é responsável por codificar um elemento de Zx.
Para tal vamos pegar no valor que recebemos e a cada coeficiente va
mos adicionar 1 de forma a obtermos coeficientes no
intervalo [0,1,2].
Após isso vamos escrever um conjunto de 4 elementos em radix 4, obt
endo assim um byte.
'''
def encodeZx(m): # assumes coefficients in range {-1,0,1}
    m = [m[i]+1 for i in range(p)] + [0]*(-p % 4)
    return blocks_encode(m,4,4)#seq2str(m,4,4,1)


def decodeZx(mstr):
    m = str2seq(mstr,4,4,1)
    return Zx([m[i]-1 for i in range(p)])
# ------------------------------------------------- Zx--------
------------------------------------------- #
```

In [9]:
```
# ------------------------------------------------ RoundRq------
------------------------------------------- #
q61 = ZZ((q-1)/6)
'''
Neste método vamos codificar elementos de rounded rings.
'''
def encoderoundedRq(c):
    c = [q61 + nicelift(c[i]/3) for i in range(p)] + [0]*(-p % 6)
    return blocks_encode(c,1536,3)

def decoderoundedRq(cstr):
    c = block_decode(cstr,1536,3)
    if max(c) > 1530:
        raise Exception("c out of range")
    c = [ci%(q61*2+1) for ci in c]
    return 3*Rq([c[i]-q61 for i in range(p)])
# ------------------------------------------------ RoundRq------
------------------------------------------- #
```

Geradores

In [10]:

```
def random8():
    return randrange(256)

'''
c0 + 256c1 + 256^2 * c2 + 256^3 *c3
'''
def urandom32():
    c0 = random8()
    c1 = random8()
    c2 = random8()
    c3 = random8()
    return c0 + 256*c1 + 65536*c2 + 16777216*c3

def random32even(): return urandom32() & (-2)
def random321mod4(): return (urandom32() & (-3)) | 1

def randomrange3():
    return ((urandom32() & 0x3fffffff) * 3) >> 30

def randomg():
    g = Zx([randomrange3()-1 for i in range(p)])
    assert R3(g).is_unit()
    return g

def random_tSmall():
    L = [random32even() for i in range(2*t)]
    L += [random321mod4() for i in range(p-2*t)]
    L.sort()
    L = [(L[i]%4)-1 for i in range(p)]
    return Zx(L)

def generateG():
    while True:
        g = randomg()
        print('GenerateG: Random Small Gerado')
        if R3_fromR(g).is_unit():
            print('GenerateG: É irredutível em R3')
            break
        else:
            print('GenerateG: Não é irredutível...')
    return g
```

KeyGen

In [11]:

```
def keyGen():
    g = generateG()
    print('_KeyGen_: Temos **g**.')

    inv_g = 1/R3(g)

    f = random_tSmall()
    print('_KeyGen_: Temos **f**.')

    h = Rq(g)/(3*Rq(f))
    print('_KeyGen_: Temos **h**.')

    pk = encodeRq(h)
    pk = R(pk)
    print('\n_KeyGen_: Temos **pk**.')

    secret = (f,inv_g,pk)
    print('_KeyGen_: Temos o **segredo**.')

    return pk, secret

pk, secret = keyGen()
```

```
GenerateG: Random Small Gerado
GenerateG: É irredutível em R3
_KeyGen_: Temos **g**.
_KeyGen_: Temos **f**.
_KeyGen_: Temos **h**.

_KeyGen_: Temos **pk**.
_KeyGen_: Temos o **segredo**.
```

In [12]:

```python
def encapsulate(pk):
    h = decodeRq(list(pk))
    print('_Encapsulate_: Temos **h**.')

    r = random_tSmall()
    print('_Encapsulate_: Temos **r**.')
    #print(r)

    hr = h*Rq_fromR(r)
    print('_Encapsulate_: Temos **hr**.')

    m = Zx([-nicemod3(nicelift(hr[i])) for i in range(p)]) #m é hr transformado em {-1,0,1}

    c = Rq(m) + hr
    print('_Encapsulate_: Temos **c**.')

    hashR = encodeZx(r)
    C = hashR[0:95]
    print('_Encapsulate_: Temos **Confirmation C**.')

    K = hashR[95:]
    print('\n_Encapsulate_: Temos **Secret Key K**.')

    encoded_c = encoderoundedRq(c)
    print('_Encapsulate_: Temos **encoded c**.')

    return (C,encoded_c), R(K)

cipherText, sessionKey = encapsulate(pk)

#cipherText = C + encoded_c
#sessionKey = K
```

```
_Encapsulate_: Temos **h**.
_Encapsulate_: Temos **r**.
_Encapsulate_: Temos **hr**.
_Encapsulate_: Temos **c**.
_Encapsulate_: Temos **Confirmation C**.

_Encapsulate_: Temos **Secret Key K**.
_Encapsulate_: Temos **encoded c**.
```

In [13]:
```
#cipherText <- C + encoded_c
# secret <- encoded_f + encoded_g_inv + pk

def decapsulate(cipherText,secret):
    (f,ginv,pk) = secret
    print('_Decapsulate_: Temos **f**.')

    h = decodeRq(list(pk))
    print('_Decapsulate_: Temos **h**.')

    (C,rounded_c) = cipherText
    print('_Decapsulate_: Temos **Confirmation C**.')
    print('_Decapsulate_: Temos **Rounded c**.')

    c = decoderoundedRq(rounded_c)
    print('_Decapsulate_: Temos **c**.')

    cX3f = c * Rq_fromR(3*f)
    cX3f = [nicelift(cX3f[i]) for i in range(p)]

    r = R3(ginv) * R3(cX3f)
    r = Zx([nicemod3(lift(r[i])) for i in range(p)])
    print('_Decapsulate_: Temos **r**.')

    hr = h * Rq(r)
    m = Zx([-nicemod3(nicelift(hr[i])) for i in range(p)])
    checkc = Rq(m) + hr

    fullkey = encodeZx(r)
    if sum(r[i]==0 for i in range(p)) != p-2*t:
        print('erro no sum')
        return False
    if checkc != c:
        return False
    if fullkey[:95] != C:
        return False
    return R(fullkey[95:])

fullKey = decapsulate(cipherText,secret)
```

```
_Decapsulate_: Temos **f**.
_Decapsulate_: Temos **h**.
_Decapsulate_: Temos **Confirmation C**.
_Decapsulate_: Temos **Rounded c**.
_Decapsulate_: Temos **c**.
_Decapsulate_: Temos **r**.
```

In [14]:
```
print(fullKey)
print(sessionKey)
```

```
x^95 + 102*x^94 + 85*x^93 + 72*x^92 + 102*x^91 + 85*x^90 + 84*x^89
+ 149*x^88 + 100*x^87 + 41*x^86 + 153*x^85 + 149*x^84 + 21*x^83 +
85*x^82 + 17*x^81 + 85*x^80 + 85*x^79 + 133*x^78 + 84*x^77 + 20*x^
76 + 85*x^75 + 153*x^74 + 18*x^73 + 86*x^72 + 73*x^71 + 105*x^70 +
169*x^69 + 85*x^68 + 85*x^67 + 96*x^66 + 88*x^65 + 145*x^64 + 65*x
^63 + 64*x^62 + 69*x^61 + 84*x^60 + 81*x^59 + 21*x^58 + 89*x^57 +
5*x^56 + 84*x^55 + 85*x^54 + 85*x^53 + 21*x^52 + 89*x^51 + 9*x^50
+ 80*x^49 + 81*x^48 + 145*x^47 + 22*x^46 + 5*x^45 + 97*x^44 + 85*x
^43 + 21*x^42 + 36*x^41 + 85*x^40 + 85*x^39 + 86*x^38 + 24*x^37 +
84*x^36 + 89*x^35 + 85*x^34 + 97*x^33 + 74*x^32 + 85*x^31 + 96*x^3
0 + 85*x^29 + 96*x^28 + 101*x^27 + 148*x^26 + 105*x^25 + 86*x^24 +
85*x^23 + 129*x^22 + 69*x^21 + 102*x^20 + 65*x^19 + 22*x^18 + 149*
x^17 + 5*x^16 + 17*x^15 + 17*x^13 + 72*x^12 + 89*x^11 + 69*x^10 +
85*x^9 + 85*x^8 + 149*x^7 + 65*x^6 + 85*x^5 + 68*x^4 + 165*x^3 + 1
01*x^2 + 153*x + 21
x^95 + 102*x^94 + 85*x^93 + 72*x^92 + 102*x^91 + 85*x^90 + 84*x^89
+ 149*x^88 + 100*x^87 + 41*x^86 + 153*x^85 + 149*x^84 + 21*x^83 +
85*x^82 + 17*x^81 + 85*x^80 + 85*x^79 + 133*x^78 + 84*x^77 + 20*x^
76 + 85*x^75 + 153*x^74 + 18*x^73 + 86*x^72 + 73*x^71 + 105*x^70 +
169*x^69 + 85*x^68 + 85*x^67 + 96*x^66 + 88*x^65 + 145*x^64 + 65*x
^63 + 64*x^62 + 69*x^61 + 84*x^60 + 81*x^59 + 21*x^58 + 89*x^57 +
5*x^56 + 84*x^55 + 85*x^54 + 85*x^53 + 21*x^52 + 89*x^51 + 9*x^50
+ 80*x^49 + 81*x^48 + 145*x^47 + 22*x^46 + 5*x^45 + 97*x^44 + 85*x
^43 + 21*x^42 + 36*x^41 + 85*x^40 + 85*x^39 + 86*x^38 + 24*x^37 +
84*x^36 + 89*x^35 + 85*x^34 + 97*x^33 + 74*x^32 + 85*x^31 + 96*x^3
0 + 85*x^29 + 96*x^28 + 101*x^27 + 148*x^26 + 105*x^25 + 86*x^24 +
85*x^23 + 129*x^22 + 69*x^21 + 102*x^20 + 65*x^19 + 22*x^18 + 149*
x^17 + 5*x^16 + 17*x^15 + 17*x^13 + 72*x^12 + 89*x^11 + 69*x^10 +
85*x^9 + 85*x^8 + 149*x^7 + 65*x^6 + 85*x^5 + 68*x^4 + 165*x^3 + 1
01*x^2 + 153*x + 21
```

In [15]:
```
def test_IND_CCA():
    ct1, sks1 = encapsulate(pk)
    ct2, sks2 = encapsulate(pk)
    if(ct1 == ct2):
        print('\n\n\nNão temos IND_CCA')
    else:
        print('\n\n\nTemos IND_CCA')
    if(sks1 == sks2):
        print('\n\n\nNão temos IND_CCA - k')
    else:
        print('\n\n\nTemos IND_CCA')

test_IND_CCA()
```

```
_Encapsulate_: Temos **h**.
_Encapsulate_: Temos **r**.
_Encapsulate_: Temos **hr**.
_Encapsulate_: Temos **c**.
_Encapsulate_: Temos **Confirmation C**.

_Encapsulate_: Temos **Secret Key K**.
_Encapsulate_: Temos **encoded c**.
_Encapsulate_: Temos **h**.
_Encapsulate_: Temos **r**.
_Encapsulate_: Temos **hr**.
_Encapsulate_: Temos **c**.
_Encapsulate_: Temos **Confirmation C**.

_Encapsulate_: Temos **Secret Key K**.
_Encapsulate_: Temos **encoded c**.



Temos IND_CCA



Temos IND_CCA
```

NTRU PRIME - PKE

In [16]:
```
def encrypt(pk,m=None):
    enc,k = encapsulate(pk)
    #print(k)
    m = R(m)
    (C,encoded_c) = enc
    if(m == None):
        return (enc,k,k)
    return(enc, m + k,k)
```

In [17]:
```
message = random_tSmall()
encripted_message = encrypt(pk,message)
```

```
_Encapsulate_: Temos **h**.
_Encapsulate_: Temos **r**.
_Encapsulate_: Temos **hr**.
_Encapsulate_: Temos **c**.
_Encapsulate_: Temos **Confirmation C**.

_Encapsulate_: Temos **Secret Key K**.
_Encapsulate_: Temos **encoded c**.
```

In [18]:
```
def decrypt(sk,c,pk):
    (enc, mm,k) = c
    #print(mm)
    k1 = decapsulate(enc,sk)
    assert k == k1
    return mm - k1
```

In [19]:
```
decripted_message = decrypt(secret,encripted_message,pk)
```

```
_Decapsulate_: Temos **f**.
_Decapsulate_: Temos **h**.
_Decapsulate_: Temos **Confirmation C**.
_Decapsulate_: Temos **Rounded c**.
_Decapsulate_: Temos **c**.
_Decapsulate_: Temos **r**.
```

In [20]:
```
if(message == decripted_message):
    print('yey')
else:
    print 'no :('
```

```
yey
```

PKE-IND-CCA

In [21]:

```
# ADVERSÁRIO
m_test1 = random_tSmall()
m_test2 = random_tSmall()
m_test3 = random_tSmall()
test1 = encrypt(pk,m_test1)
test2 = encrypt(pk,m_test2)
test3 = encrypt(pk,m_test3)

messageList=[m_test1,m_test2,m_test3]
def game_encrypt(pk, messageList):
    b = randint(0,len(messageList)-1)
    return encrypt(pk,messageList[b])

cipher = encrypt(pk)
cipherMessage = game_encrypt(pk,messageList)
#print(cipherMessage)

cipher_enc, cipher_message, cipher_k = cipher

testTry1 = (cipher_enc, m_test1 + cipher_message,cipher_k)
#print(testTry1)

testTry2 = (cipher_enc, m_test2 + cipher_message,cipher_k)
testTry3 = (cipher_enc, m_test3 + cipher_message,cipher_k)

#if(cipherMessage == testTry1):
if(test1 == testTry1):
    print('Foi a mensagem1!')
#elif(cipherMessage == testTry2):
elif(test2 == testTry2):
    print('Foi a mensagem2!')
#elif(cipherMessage == testTry3):
elif(test3 == testTry3):
    print('Foi a mensagem3!')
else:
    print('PKE-IND-CCA')
```

Encapsulate: Temos **h**.
Encapsulate: Temos **r**.
Encapsulate: Temos **hr**.
Encapsulate: Temos **c**.
Encapsulate: Temos **Confirmation C**.

Encapsulate: Temos **Secret Key K**.
Encapsulate: Temos **encoded c**.
Encapsulate: Temos **h**.
Encapsulate: Temos **r**.
Encapsulate: Temos **hr**.
Encapsulate: Temos **c**.
Encapsulate: Temos **Confirmation C**.

Encapsulate: Temos **Secret Key K**.
Encapsulate: Temos **encoded c**.
Encapsulate: Temos **h**.
Encapsulate: Temos **r**.
Encapsulate: Temos **hr**.
Encapsulate: Temos **c**.
Encapsulate: Temos **Confirmation C**.

Encapsulate: Temos **Secret Key K**.
Encapsulate: Temos **encoded c**.
Encapsulate: Temos **h**.
Encapsulate: Temos **r**.
Encapsulate: Temos **hr**.
Encapsulate: Temos **c**.
Encapsulate: Temos **Confirmation C**.

Encapsulate: Temos **Secret Key K**.
Encapsulate: Temos **encoded c**.
Encapsulate: Temos **h**.
Encapsulate: Temos **r**.
Encapsulate: Temos **hr**.
Encapsulate: Temos **c**.
Encapsulate: Temos **Confirmation C**.

Encapsulate: Temos **Secret Key K**.
Encapsulate: Temos **encoded c**.
PKE-IND-CCA