Estruturas criptograficas: TP2 problema 2

Descrição

Uma das aplicações mais importantes do teorema chinês dos restos (CRT) em criptografia é a transformada NTT “Number Theoretic Transform”. Esta transformada é uma componente importantes de “standards” PQC como o Kyber e o Dilithium mas também de outros algoritmos submetidos ao concurso NIST PQC. A transformação NTT tem várias opções e aquela que está apresentada no Capítulo 4: Problemas Difíceis usa o CRT. Neste problema pretende-se uma implementação Sagemath do NTT-CRT tal como é descrito nesse documento.

Esta versão particular de NTT que deriva das propriedades do CRT só se aplica a corpos primos em que o primo q tem uma forma particular. Também não se aplica a qualquer elemento de q mas apenas aos conjunto de polinómios de graus inferior a um certo limite N da forma de uma potência de 2.

Passos para calcular a transformada

Escolha de um N da forma 2^d e um primo q que verifique q = 1 mod 2N.

Calculo das raízes do polinómio da forma *omega.nth_root(2)* omega^i* em que *omega = primitive_root(q)^2* Este passo é efetuado apenas uma vez, pois apenas depende de N e q.

Os N resíduos calculam-se como f(s_i).

Inversa da transformada

Para obter o polinómio a partir da transfornada é calculado o vetor *mu* (base), atravéz dos módulos CRT (calculado apenas uma vez pois so depende de N e q)

Posteriormente é feito o somatório do produto dos resíduos com o vetor mu.

In [9]:
```
from sage.all import *
```

In [10]:
```
def ntt_transform(f, s):

    # Calcula a transformada NTT de f
    trans = [f(s_i) for s_i in s]

    return trans

def ntt_inverse(trans, s, mu):
```

```
    # Calcula a transformada inversa
    f_inv = [mu[i] * trans[i] for i in range(len(trans))]

    return sum(f_inv)

def raizes(omega, N):

    s = [omega.nth_root(2) * omega^i for i in range(N)]

    return s

def base_mu(x, s, N):

    mods = [x - s[i] for i in range(N)]

    mu = CRT_basis(mods)

    return mu


# Escolha de valores
q = 17
N = 8
omega = primitive_root(q)^2

# Polinômio de exemplo
f = PolynomialRing(GF(q), 'x')
x = f.gen()
# f = 1 + x -2*x^2 - x^3


# Cálculo das raízes
s = raizes(omega, N)

mu = base_mu(x, s, N)
```

In [11]:

```
p1 = f.random_element(7)
print(p1)

trans1 = ntt_transform(p1, s)
print(trans1)

orig = ntt_inverse(trans1, s, mu)
print(orig)
```

```
5*x^7 + 7*x^6 + 12*x^5 + 9*x^3 + 15*x^2 + 9*x + 13
[9, 2, 5, 4, 4, 5, 0, 7]
5*x^7 + 7*x^6 + 12*x^5 + 9*x^3 + 15*x^2 + 9*x + 13
```

Teste

Para testar o bom funcionamento do algoritmo, foi efetuado um teste com um N e q "pequenos"

Primeiro é calculada a transformada de um polinómio e posteriormente obtido o polinómio original pelo processo inverso.

No segundo teste é multiplicado um polinomio por ele mesmo a nível dos residuos e obtido o polinomio resultante atravéz da transformada.

In [12]:
```
# Calcula a transformada NTT de p1
p2 = f.random_element(3)
print(p2)
trans2 = ntt_transform(p2, s)
print(trans2)

# Multiplica as transformadas ponto a ponto
trans_produto = [int(trans2[i]) * int(trans2[i]) for i in range(N)]
print(trans_produto)

# Calcula a transformada inversa do resultado da multiplicação
produto_orig = ntt_inverse(trans_produto, s, mu)

print(produto_orig)

print(p2*p2)
```

```
13*x^3 + 7*x^2 + 15*x + 6
[6, 1, 11, 12, 13, 0, 11, 11]
[36, 1, 121, 144, 169, 0, 121, 121]
16*x^6 + 12*x^5 + 14*x^4 + 9*x^3 + 3*x^2 + 10*x + 2
16*x^6 + 12*x^5 + 14*x^4 + 9*x^3 + 3*x^2 + 10*x + 2
```