Estruturas Criptográficas - TP2-2

PG53721 - Carlos Machado

PG54249 - Tiago Oliveira

Enunciado - Implementação Sagemath do NTT-CRT

Neste problema pretende-se uma implementação *Sagemath* do NTT-CRT, ou seja, a aplicação do teorema chinês dos restos na criptografia.

O primeiro passo, após ter escolhido um **N**, passa por gerar um primo que verifique condição: $q \equiv 1 \mod 2N$.

In [1]:
```
def generate_q(n):
    if not  n in [32,64,128,256,512,1024,2048]:
            raise ValueError("improper argument ",n)
    q = 1 + 2*n
    while True:
        if q.is_prime():
            break
        q += 2*n
    return q
```

In [2]:
```
n = 1024
q = generate_q(n)
```

De seguida é necessário calcular:

- Corpo Finito **F**.
- Variável **R** que é o anel de polinómios sobre esse corpo *F*.
- A variável **w** que representa o gerador do anel de polinômios.
- O poliónio **g** utlizado para calcular as raízes.
- O valor de **xi** que representa a última raíz do polínomio *g*.

In [3]:
```
F = GF(q) ;  R = PolynomialRing(F, name="w")
w = R.gen()

g = (w^n + 1)
xi = g.roots(multiplicities=False)[-1]
rs = [xi^(2*i+1)  for i in range(n)]
base = crt_basis([(w - r) for r in rs])
```

A próxima etapa requer a definição de um **f** pertencente a *Rq*.

In [4]:
```
def random_pol(args=None):
    return R.random_element(args)
```

In [5]:
```
f = random_pol(1023)
```

Esta função auxiliar _*expand* permite expandir o polinómio *f* para o tamanho necessário.

In [6]:
```
def _expand_(f):
    u = f.list()
    return u + [0]*(n-len(u))
```

O algoritmo recursivo para calcular o vetor **ff** é o seguinte:

In [7]:
```
def _ntt_(xi,N,f):
    if N==1:
        return f
```

```python
    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 = [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
```

In [8]:
```python
def ntt(f):
    return _ntt_(xi,n,_expand_(f))
```

No que toca à transformada inversa,a reconstrução tem a forma:

$$f = \sum_i ff_i \times \mu_i$$

Sendo *ff* a transformada NT do polinónio *f* e *u* a base.

In [9]:
```python
def ntt_inv(ff):                                    ## transformada inversa
    return sum([ff[i]*base[i] for i in range(n)])
```

Teste

In [10]:
```python
ff = ntt(f)

fff = ntt_inv(ff)

# print(fff)
print("Correto ? ",f == fff)
```

```
Correto ?  True
```