Estruturas Criptográficas

Trabalho Prático 2 - Exercício 2

José de Matos Moreira - PG53963

Pedro Freitas - PG52700

Enunciado do problema

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 importante 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.

Resolução

Em primeiro lugar, surge a fase de *import*.

In [4]:
```
from sage.all import *
```

NTT_CRT

Todo o código desenvolvido encontra-se inserido nesta classe. A mesma implementa a transformada **NTT**, num corpo finito, **GF(q)**, recorrendo ao **CRT**, exatamente da mesma forma que o mesmo é descrito no **Capítulo 4**. Deste modo, explica-se cada uma das funções presentes:

- **__init__**: métdo de inicialização. Responsável por inicializar as variáveis **n** e **q**, verificando se o valor de **n** é da forma 2^d e se o valor do parâmetro **q**, se passado na inicialização (atribuindo um quando isso não acontece), verifica a condição **q ≡ 1 mod 2n**. Também inicializa os valores **F** e **R**, sendo, respetivamente, um campo finito e um anel polinomial, e a base, através das raízes

- **_expand_**: função que expande um polinómio até que o mesmo atinja o tamanho **n**
- **_ntt_**: função responsável por aplicar a transformada **NTT**, recursivamente, tal como especificado no **Capítulo 4**
- **ntt**: método que calcula a **NTT** de um polinómio
- **ntt_inv**: função que calcula a inversa da **NTT**
- **random_pol**: função capaz de gerar um polinómio aleatório, pertencente ao anel polinomial já criado

In [5]:
```
class NTT_CRT():

    def __init__(self, n, q):
        if not (n > 0 and (n & (n - 1)) == 0):
            raise ValueError("[ERROR] n value")
        self.n = n

        if q == None:
            self.q = 1 + 2 * self.n
            while True:
                if self.q.is_prime():
                    break
                self.q += 2 * n

        else:
            if q % (2 * n) != 1:
                raise ValueError("[ERROR] q value")

            self.q = q

        self.F = GF(self.q)
        self.R = PolynomialRing(self.F, name="w")
        w = (self.R).gen()

        phi = w ^ n + 1
        self.xi = phi.roots(multiplicities=False)[0]
        rs = [self.xi ^ (2 * i + 1)  for i in range(n)]
        self.base = crt_basis([(w - r) for r in rs])


    def _expand_(self, f):
        u = f.list()
        return u + [0] * (self.n - len(u))


    def _ntt_(self, xi, N, f):
        if N == 1:
            return f

        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 = self._ntt_(xi2, N_, f0)
```

```
        ff1 = self._ntt_(xi2, N_, f1)

        s  = xi
        ff = [None] * N

        for i in range(N_):
            ff[i] = ff0[i] + s * ff1[i]
            ff[i + N_] = ff0[i] - s * ff1[i]
            s *= xi2

        return ff


    def ntt(self, f):
        return self._ntt_(self.xi, self.n, self._expand_(f))

    def ntt_inv(self,ff):
        return sum([ff[i] * self.base[i] for i in range(self.n)])

    def random_pol(self, args):
        return (self.R).random_element(args)
```

Testes de aplicação

In [6]:
```
T = NTT_CRT(2048, None)
pol = T.random_pol(1024)
ff = T.ntt(pol)
inv_ff = T.ntt_inv(ff)

if pol == inv_ff:
    print('Successful test!')

else:
    print('Error!')
```

```
Successful test!
```