TP2_Exercicio2

April 2, 2024

1 TP2 - Exercício 2

1.0.1 Autores

Afonso Ferreira - pg52669
Tiago Rodrigues - pg52705

1.0.2 Enunciado

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.

1.0.3 Number Theoretic Transform - CRT

```
Classe que representa a Transformada Numérica Teórica (NTT).

A classe NTT fornece métodos para realizar a NTT direta e inversa em polinómios.

Args:
    n (int): O tamanho do polinómio. Deve ser um dos seguintes [32, 64, 128, 256, 512, 1024, 20
    q (int, opcional): O valor do módulo. Se não fornecido, um valor adequado de q é escolhido

Raises:
    ValueError: Se o valor de n não estiver entre os tamanhos permitidos.

Attributes:
    n (int): O tamanho do polinómio.
    q (int): O valor do módulo.
    F (FiniteField): O campo finito usado para cálculos.
    R (PolynomialRing): O anel de polinómios usado para cálculos.
    xi (Element): A raiz primitiva da unidade.
    base (list): A base do Teorema Chinês dos Restos (CRT).

Métodos:
    ntt(f): Realiza a NTT direta no polinómio de entrada f.
```

```
ntt_inv(ff): Realiza a NTT inversa no polinómio de entrada ff.
random_pol(args): Gera um polinómio aleatório.
```

O primeiro passo é a escolha de um N da forma 2^d e um primo q que verifique $q \equiv 1 \bmod 2N$. O corpo $\mathbb{F}_q$ contém todas as raízes do polinómio $\phi \equiv w^N + 1$. Seja ξ uma qualquer destas raízes; então todas as raízes têm a forma $\xi^{2,i+1}$, com ; $i = 0, \cdots, N-1$.

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

class NTT(object):
    # Construtor
    # O primeiro passo é a escolha de um $N$ da forma $2^d$  e um primo $\,q\,$␣
 ↪que  verifique $\,q \equiv 1 \bmod 2N\,$.
    def __init__(self, n=128, q=None):
        if not  n in [32,64,128,256,512,1024,2048]:
            raise ValueError("improper argument ",n)
        self.n = n

        # Se q não for fornecido, escolhe um valor de q de acordo com as regras␣
 ↪do NTT
        if not q:
            self.q = 1 + 2*n
            while True:
                if (self.q).is_prime():
                    break
                self.q += 2*n
        else:
            # Se q for fornecido, verifica se satisfaz a condição NTT
            if q % (2*n) != 1:
                raise ValueError("Valor de 'q' não verifica a condição NTT")
            self.q = q

        # Define o campo finito e o anel de polinómios
        self.F = GF(self.q) ;  self.R = PolynomialRing(self.F, name="w")
        w = (self.R).gen() # variável w do anel de polinómios R

        # Calcula a raiz primitiva da unidade xi
        g = (w**n + 1) #
        xi = g.roots(multiplicities=False)[-1] # obtemos raíz primitiva da␣
 ↪unidade xi
        self.xi = xi
        raizes = [xi**(2*i+1)  for i in range(n)] # obtemos as raízes de␣
 ↪unidade xi^(2i+1)
        self.base = crt_basis([(w - raiz) for raiz in raizes]) # construção da␣
 ↪base do teorema chinês do resto
        # E = crt_basis(X)
        # X - lista de inteiros que são coprimos em pares
```

```
        # E - lista de inteiros de tal modo que E[i] = 1 (mod X[i]) e E[i] = 0␣
↪(mod X[j]), sendo que j != i

    # Função que aplica a transformada NTT a um polinómio f
    def ntt(self,f):
        def _expand_(f):
            u = f.list() # lista dos coeficientes do polinómio f
            return u + [0]*(self.n-len(u)) # expande o polinómio f para o␣
↪tamanho n

        def _ntt_(xi,N,f):
            if N==1:
                return f
            N_ = N//2 ; #  N / 2 coeficientes
            xi2 =  xi**2 # xi^2
            f0 = [f[2*i]   for i in range(N_)] ; f1 = [f[2*i+1] for i in␣
↪range(N_)] # divide f em f0 par e f1 ímpar (split)
            ff0 = _ntt_(xi2,N_,f0) ; ff1 = _ntt_(xi2,N_,f1) # recursão

            s  = xi ; ff = [self.F(0) for _ in range(N)] # inicializa ff␣
↪(transformada) com zeros (polinómio de tamanho N)
            for i in range(N_):
                a = ff0[i] ; b = s*ff1[i]
                ff[i] = a + b ; ff[i + N_] = a - b # calcula ff[i] e ff[i + N/2]
                s = s * xi2 # atualiza s
            return ff

        return _ntt_(self.xi,self.n,_expand_(f))

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

    def random_pol(self,args=None):
        return (self.R).random_element(args)
```

1.0.4 Teste

```
[ ]: T = NTT(n=2048,q=343576577)

# Temos o polinómio f
f = T.random_pol(1023)

# Aplicamos a transformada NTT a f
ff = T.ntt(f)

# Obtemos o polinómio f que é a transformada inversa de ff
fff = T.ntt_inv(ff)
```

```python
# Verificamos se f e fff são iguais
print("Correto ? ",f == fff)
```

```
Correto ?  True
```