# TP3_Ex2_KYBER

May 2, 2023

## 0.1 Pergunta 2 - KYBER

O objetivo é a criação de um protótipo para o algoritmo *KYBER*, seguindo o problema *RLWE* (*Ring Learning With Errors*), para a criação de um esquema KEM/PKE. É espectável que o algoritmo **KEM** seja **IND-CPA seguro** e o algoritmo **PKE** seja **IND-CCA seguro**, para a técnica pós-quântica (baseada em reticulados) **KYBER**.

A aplicação do *RLWE* em esquemas **PKE/KEM** é essencialmente baseada na abordagem de *Lyubashesvky*, *Peikert* e *Regev* (**LPR**) publicada em 2013.

O criptosistema *RLWE* é baseado no **LWE**, contudo: - a_i, s e b_i são todos polinómios - Envés do produto interno usa-se multiplicação polinomial (a_i * s = b_i gera tantas equações aproximadas quanto o grau dos b_i).

O algoritmo de RLWE é usado em esquemas criptográficos de reticulados, como o Kyber, para fornecer segurança contra ataques de computação quântica.

### 0.1.1 Estratégia

Normalmente a construção de um esquema de cifra que seja **IND-CCA** seguro é algo mais complicado. No entanto existe um mecanismo, designado por transformação de *Fujisaki-Okamoto* (FOT), que permite (dentro de um contexto bastante lato) converter um esquema **PKE** com segurança **IND-CPA** num esquema **PKE** com segurança **IND-CCA**.

Sendo assim, o processo começará na definição de um esquema *PKE*, com base no algoritmo *KYBER*, que seja **IND-CPA seguro**.

Considerando a transformação de *Fujisaki-Okamoto*, transformar-se-à esse num **PKE IND-CCA seguro**.

Para o **KEM**, utilizar-se o algoritmo anterior de *PKE*, *IND-CPA* seguro, para o criar, sendo este, consequentemente, também *IND-CPA* seguro.

É de salientar que, no *paper* fornecido pela equipa docente, são apenas dadas as étapas necessárias para implementar o *PKE IND-CPA* e um *KEM IND-CCA* (o oposto do pedido no enunciado).

[1]:
```
#imports
from cryptography.hazmat.primitives import hashes
import random
from pickle import dumps, loads
```

## 0.2 NTT

O algoritmo *Number-Theoretic Transform* (NTT) é um algoritmo eficiente para computar a **Transformada de Fourier Discreta** (DFT) em corpos finitos, que são conjuntos finitos de números inteiros equipados com operações aritméticas adequadas. A *NTT* é frequentemente usada em algoritmos de criptografia, como o *KYBER*, que usa o problema de *RLWE*, como mencionado anteriormente.

O algoritmo funciona por meio da divisão do polinômio em dois polinômios menores, calculando a transformada em cada um deles e combinando os resultados para obter a transformada do polinômio original. Esse processo é repetido recursivamente até que os polinômios sejam reduzidos a um único coeficiente, que é a transformada final do polinômio original.

A transformação é definida sobre um conjunto finito de raízes da unidade, que são usadas para expressar as expansões dos polinômios em termos de coeficientes nesses pontos. Estas raízes são escolhidas de modo a formar um grupo multiplicativo que possa ser usado para calcular a transformada e sua inversa de forma eficiente.

O algoritmo é descrito no **capítulo 2a**, dos *papers* fornecidos pela equipa docente, e reinforçado no **capítulo 7** no âmbito do algoritmo *RLWE*, seguindo-se essa estratégia de perto.

### 0.2.1 Classe NTT

**Método 'init'** A classe *NTT* é definida com **dois parâmetros opcionais**, **n** e **q**. O parâmetro $n$ indica o tamanho do vetor que será transformado, que deve ser uma **potência de 2**, e $q$ é o módulo que define o **corpo finito** sobre o qual a transformada será realizada.

Como especificado no *paper*: - O primeiro passo é a escolha de um N da forma 2ˆd e um primo q que verifique q  1 mod 2N.

De seguida, define-se **o anel (*Galois*) dos inteiros módulo q, que é usado para realizar a aritmética modular**, na forma do valor $F$.

Este é usado para criar o corpo finito, que será utilizado nas operações matemáticas da implementação do algoritmo, o valor *R*. $R$ é um anel de polinômios, ou seja, é o anel formado por polinômios com coeficientes no anel $F$; e o gerador do anel é denotado por $w$, para seguir o algoritmo definido no *paper* (usada para construir polinómios em $R$).

São trabalhados na transformada polinómios do tipo *wˆn+1*, tal como definido na var *g*. É encontrado o último zero do polinômio *g,* que é a raiz primitiva *xi* que será usada na transformada.

Em seguida, é definido um conjunto de $n$ raízes da unidade *rs*, que são as potências de *xi* com **expoente ímpar**. Essas raízes são usadas para definir a base **CRT** (*Chinese Remainder Theorem*) que será usada na inversa da transformada.

Mais uma vez, é reiterado que o algoritmo de inicialização (a parametrização), segue próximo os passos do **capítulo 2a/7**, dos *papers.*

**Método 'ntt'** Para o algoritmo *NTT*, dividiu-se o trabalho em dois métodos auxiliares adicionais: 1. ntt_alg; 2. expand.

Começando pelo mais simples, o *expand.* O método *expand(f)* serve para **expandir o polinômio f com zero coeficientes à direita**, tornando o seu comprimento igual a $n$. O passo é necessário para que a transformada seja aplicada corretamente a *f*, já que é esperado um polinômio de comprimento *n*.

A expansão é realizada pelo método *list()*, que retorna a lista de coeficientes do polinômio *f*, e a adição de zeros à direita é feita com o comando **return u + [0]*(self.n-len(u))**.

Por exemplo, se *f = x^3 + x^2 + 2x + 1 e n=8*, então *expand(f)* retorna [1, 2, 1, 1, 0, 0, 0, 0].

O segundo, *ntt_alg*, recebe como *input*: - a raiz primitiva xi do corpo finito; - o tamanho N do vetor; - vetor f de coeficientes do polinômio a ser transformado.

É feita a divisão do vetor em dois e chama, recursivamente, o próprio método *ntt* para cada metade. Após essa divisão, ele multiplica cada valor correspondente do vetor resultante da chamada recursiva pela potência correspondente de *xi* e soma as duas parcelas para produzir os dois valores correspondentes do vetor resultante. Esse processo é repetido para todas as potências de *xi* até que todos os valores do vetor resultante tenham sido calculados.

Deste modo, o método *ntt_alg* segue o algoritmo:

$\mathrm{NTT}(x, N, f)$
  **if** $N = 1$ **then return** $(f_0)$

  $f^+, f^- \leftarrow \mathbf{split}(f)$
  $z \leftarrow x^2$
  $\bar{f}^+ \leftarrow \mathrm{NTT}(z, N/2, f^+)$ ; $\bar{f}^- \leftarrow \mathrm{NTT}(z, N/2, f^-)$
  $s \leftarrow x$
  **for** $i \in \{0, \cdots, N/2-1\}$ **do**
    $a \leftarrow \bar{f}_i^+$ ; $b \leftarrow s \times \bar{f}_i^-$
    $\bar{f}_i \leftarrow a + b$ ; $\bar{f}_{i+N/2} \leftarrow a - b$
    $s \leftarrow s \times z$
  **return** $(\bar{f}_0, \bar{f}_1, \cdots, \bar{f}_{N-1})$

Com estas duas, o método **ntt** é calculado, usando o método *ntt_alg*, com o polinómio expandido no seu *input*.

**Método 'ntt_inv'** No âmbito da **transformada NTT inversa**, seguiu-se o algoritmo nos mesmo capítulos; sendo esta a **abordagem 1**.

Neste caso, segue-se:

1. Constrói-se a base CRT a partir dos módulos $g_i(w) \equiv (w - x_i)$. A base é formada por polinómios

$$\mu_i(w) \equiv \prod_{j \neq i} g_j(w)/g_j(x_i)$$

Uma vez construída a base CRT o polinómio $f(w)$ recupera-se como

$$f(w) = \sum_{i=0}^{N-1} \bar{f}_i * \mu_i(w)$$

[2]:
```python
class NTT:

    def __init__(self, n, q):

        # Se n for escolhido arbitrariamente
        if n not in [32,64,128,256,512,1024,2048]:
```

```
            raise ValueError("O tamanho do vetor escolhido não é uma potência de␣
↪2 válida!",n)

        self.n = n

        # Caso q não seja fornecido pelo utilizador
        if not q:

            # o valor de q, por definição, deve ser q = 1 mod 2n
            self.q = 1 + 2*n

            while True:

                # Verifica-se se q é um valor primo (obrigatório)
                if (self.q).is_prime():
                    break
                self.q += 2*n
        else:
            # Verifica-se se o q fornecido pelo utilizador é válido
            if q % (2*n) != 1:
                raise ValueError("O q não se enquadra no valor obrigatório,␣
↪segundo a transformada NTT!")

            self.q = q

        # anel infinito (para construir R)
        self.F = GF(self.q)

        # anel de polinómios
        self.R = PolynomialRing(self.F, name="w")

        # Gerador do anel polinomial R
        w = (self.R).gen()

        # Temos que phi (variável root) é um polinómio do tipo w^N + 1
        g = (w^n + 1)

        root = g.roots(multiplicities=False)[-1]

        self.root = root

        # Para o cálculo da inversa
        rs = [root^(2*i+1)  for i in range(n)]
        self.base = crt_basis([(w - r) for r in rs])

    ## ---------------------------- Métodos principais␣
↪---------------------------
```

```
    # Método para calcular a transformada NTT
    def ntt(self,f):
        return self.aux_ntt(self.root,self.n, self.poli_expand(f))

    # Método para calcular a inversa da transformada NTT
    def ntt_inv(self,ff):
        return sum([ff[i]*self.base[i] for i in range(self.n)])

    ## ---------------------------- Métodos auxiliares␣
↪----------------------------

    # Método auxiliar com o algoritmo NTT
    def aux_ntt(self, x,N,f):
        if N==1:
            return f

        # f+,f- split(f)
        f_mais = [f[2*i]   for i in range(N/2)]
        f_menos = [f[2*i+1] for i in range(N/2)]

        # z <- x^2
        z =  x^2

        # f+ <- NTT(z, N/2, f+) ; f- <- NTT(z, N/2, f-)
        ntt_f_mais = self.aux_ntt(z,N/2,f_mais)
        ntt_f_menos = self.aux_ntt(z,N/2,f_menos)

        # s <- x
        s = x
        res = [self.F(0) for i in range(N)]
        for i in range(N/2):
            a = ntt_f_mais[i]
            b = s*ntt_f_menos[i]
            res[i] = a + b
            res[i + N/2] = a - b
            s = s * z
        return res

    # Método para expandir o polinómio (no lado direito)
    def poli_expand(self,f):
        u = f.list()
        return u + [0]*(self.n-len(u))
```

### 0.2.2 KYBER-CPAPKE

Como mencionado na introdução, o processo começa com a criação de um **PKE IND-CPA**, segundo o algoritmo/técnica *KYBER*; neste caso, vai-se seguir o *paper* fornecido pela equipa docente, o *CRYSTALS_ KYBER - Algorithm Specifications And Supporting Documentation (version 3.0)*.

Este descreve, entre outros, os métodos para poder **gerar um par de chaves**, **cifrar uma mensagem** e a respetiva **decifragem** (para um *PKE IND-CPA* seguro).

*IND-CPA* significa "**indistinguível sob ataque de texto cifrado escolhido**". Um esquema criptográfico é considerado *IND-CPA* seguro se um atacante não conseguir distinguir entre o texto cifrado de uma mensagem escolhida por ele e o texto cifrado de uma mensagem aleatória escolhida pelo algoritmo de cifragem, mesmo tendo acesso ao texto cifrado de ambas as mensagens. Em outras palavras, o atacante não pode aprender informações sobre a mensagem original a partir do texto cifrado escolhido.

Adicionalmente, dado a ser a primeira opção (a mais pequena), escolheu-se implementar o *KYBER* **512**; pelo que o tamanho do módulo do **reticulado** a usar no processo será 512. Quanto maior o tamanho, maior será a segurança; contudo, num contexto académico, tal não será necessário para o protótipo.

Como prefácio para a explicação a seguir, para cada um dos métodos: - *keyGen* - **geração das chaves**; - *enc* - **Cifragem da mensagem**; - *dec* - **Decifragem da mensagem**;

… utilizou-se os algoritmos **4**, **5** e **6** do *paper*, relativos ao *KYBER.CPAPKE*.

**Método 'init'** Seguindo o *paper*, alguns parâmetros são apresentados como necessários, antes de se efetuar as ações: - **n** Tamanho do polinómio (**256** por *default*) - **k** Tamanho das matrizes a serem criadas (kxk) - **q** Módulo do polinómio (**7681** por *default*) - **n1|n2** Parâmetros para gerar uma matriz densa (*noise*) - **du** Diferença finita para a frente - **dv** Diferença finita para trás - **M** Matriz para a transformada NTT

Tem-se que $n$ é escolhido com valor 256 uma vez que se pretende **encapsular chaves com 256 bits de entropia**; valores mais pequenos iriam requerer que fosse realizado um *encoding* de mútiplos *bits* da chave num único coeficiente polinomial (menos security). Valores maiores reduziriam a capacidade de escalar a segurança com o parâmetro $k$.

$q$ é uma valor primo "pequeno" para permitir uma multiplicação mais rápida com base na transformada *NTT*.

O $k$ é selecionado para fixar a dimensão da rede(*lattice*) como um múltiplo de $n$.

Os restantes parâmetros (n1, n2, du e dv) foram escolhidos para balançar a segurança (maioritariamente usados para *noise* no algoritmo).

Não obstante, todos os valores escolhidos foram retirados diretamente do *paper* (para o *KYBER* 512), uma vez que estão testados e documentados, como se visualiza em:

| | $n$ | $k$ | $q$ | $\eta_1$ | $\eta_2$ | $(d_u, d_v)$ | $\delta$ |
|---|---|---|---|---|---|---|---|
| Kyber512 | 256 | 2 | 3329 | 3 | 2 | (10, 4) | $2^{-139}$ |
| Kyber768 | 256 | 3 | 3329 | 2 | 2 | (10, 4) | $2^{-164}$ |
| Kyber1024 | 256 | 4 | 3329 | 2 | 2 | (11, 5) | $2^{-174}$ |

**Funções auxiliares** Seguindo o **algoritmo 4** do *paper*, tem-se que serão necessárias, para os 3 algoritmos especificados, algumas funções auxiliares:

- **BytesToBits** Percorre cada elemento do *array* de *bytes*, converte cada num vetor de *bits* de *8* dígitos (1 *byte*) usando operações de divisão e módulo. Adiciona cada *bit* do vetor de *bits* ao *bitarray* final.

- **G** Operação de *hashing* usando o algoritmo **SHA3-512**. (para cálculo de ρ, σ).

- **XOF** Função para criar um *extendable ouput*. - **PRF** Função para criar um *output* pseudo-aleatório.

- **Parse** O *KYBER* utiliza uma estratégia determinística para dar *sample* dos elementos em *Rq* (que são estatisticamente próximos de uma distribuição normal). Para este *sampling*, criou-se a função *parse*:

**Algorithm 1** Parse: $\mathcal{B}^* \rightarrow R_q^n$

**Input:** Byte stream $B = b_0, b_1, b_2 \cdots \in \mathcal{B}^*$
**Output:** NTT-representation $\hat{a} \in R_q$ of $a \in R_q$

$i := 0$
$j := 0$
**while** $j < n$ **do**
  $d_1 := b_i + 256 \cdot (b_{i+1} \bmod^+ 16)$
  $d_2 := \lfloor b_{i+1}/16 \rfloor + 16 \cdot b_{i+2}$
  **if** $d_1 < q$ **then**
    $\hat{a}_j := d_1$
    $j := j + 1$
  **end if**
  **if** $d_2 < q$ **and** $j < n$ **then**
    $\hat{a}_j := d_2$
    $j := j + 1$
  **end if**
  $i := i + 3$
**end while**
**return** $\hat{a}_0 + \hat{a}_1 X + \cdots + \hat{a}_{n-1} X^{n-1}$

- **CBD** Para a especificação do *KYBER* é necessário definir como um polinómio f pertencente a Rq é *sampled* relativamenmte à distribuição binomial *B*. Para tal criou-se o método *CBD* (*Center binomial distribuition*):

  **Algorithm 2** $\mathsf{CBD}_\eta$: $\mathcal{B}^{64\eta} \rightarrow R_q$

  **Input:** Byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
  **Output:** Polynomial $f \in R_q$
  $(\beta_0, \ldots, \beta_{512\eta-1}) := \mathsf{BytesToBits}(B)$
  **for** $i$ from 0 to 255 **do**
    $a := \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
    $b := \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
    $f_i := a - b$
  **end for**
  **return** $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

- **Decode** Existem dois tipos de dados que o *KYBER* precisa de serializar para *byte arrays*: *byte arrays* e polinómios. A função *decode* tem como objetivo a desserialização de polinómio (para *byte arrays*), como se visualiza em:

**Algorithm 3** $\mathsf{Decode}_\ell \colon \mathcal{B}^{32\ell} \to R_q$

**Input:** Byte array $B \in \mathcal{B}^{32\ell}$
**Output:** Polynomial $f \in R_q$
$(\beta_0, \ldots, \beta_{256\ell-1}) \coloneqq \mathsf{BytesToBits}(B)$
**for** $i$ from 0 to 255 **do**
  $f_i \coloneqq \sum_{j=0}^{\ell-1} \beta_{i\ell+j} 2^j$
**end for**
**return** $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$

- **Compress** and **Decompress** Os métodos *Compress* e *Decompress*, como especificados no documento, são usados no âmbito de remover alguns *low-order bits* no criptograma, que não têm muito efeito na correção da decifragem (e reduz o tamanho do criptograma). Também são usados para realizar correção de erros *LWE* na cifragem/decifragem.

Como nota adicional, as funções de *hashing* usadas nos métodos **G**, **XOF** e **PRF** encontram-se todas detalhadas no documento do *KYBER*, mencionado anteriormente. As funções são, respetivamente, *SHA-512*, *SHAKE-128* e *SHAKE-256*.

Adicionalmente, como estão ser efetuadas operações com matrizes, criou-se um conjunto de **métodos auxiliares** para executar essas ações; nomeadamente, soma, subtração e multiplicação de matrizes/vetores.

**Método 'keyGen'** O método *KeyGen* tem como objetivo gerar uma chave **pública** e outra **privada** para serem utilizadas no processo de cifragem/decifragem (*Enc* e *Dec*). Este foi implementando seguindo o **algoritmo 4**, como anteriormente referido. O algoritmo:

**Algorithm 4** KYBER.CPAPKE.KeyGen(): key generation

**Output:** Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
**Output:** Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
$d \leftarrow \mathcal{B}^{32}$
$(\rho, \sigma) \coloneqq \mathrm{G}(d)$
$N \coloneqq 0$
**for** $i$ from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  **for** $j$ from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}[i][j] \coloneqq \mathsf{Parse}(\mathsf{XOF}(\rho, j, i))$
  **end for**
**end for**
**for** $i$ from 0 to $k-1$ **do** ▷ Sample $\mathbf{s} \in R_q^k$ from $B_{\eta_1}$
  $\mathbf{s}[i] \coloneqq \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N \coloneqq N + 1$
**end for**
**for** $i$ from 0 to $k-1$ **do** ▷ Sample $\mathbf{e} \in R_q^k$ from $B_{\eta_1}$
  $\mathbf{e}[i] \coloneqq \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N \coloneqq N + 1$
**end for**
$\hat{\mathbf{s}} \coloneqq \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} \coloneqq \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \coloneqq \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
$pk \coloneqq (\mathsf{Encode}_{12}(\hat{\mathbf{t}} \bmod^+ q) \| \rho)$ ▷ $pk \coloneqq \mathbf{As} + \mathbf{e}$
$sk \coloneqq \mathsf{Encode}_{12}(\hat{\mathbf{s}} \bmod^+ q)$ ▷ $sk \coloneqq \mathbf{s}$
**return** $(pk, sk)$

- Determina a matriz A pertencente a Rq, no domínio $NTT$;

- Determina as *samples* (vetores) $s$ e $e$ pertencente a Rq. (com recursos aos métodos auxiliares *Parse*, *XOF*, *CBD* e *PRF*);
- Retorna a chave pública $pk$ e a privada $sk$, num tuplo.

**Método 'Enc'** O método *Enc* tem como objetivo **cifrar** uma mensagem. Para tal recebe a mensagem, $m$, a chave pública, $pk$ e um valor $r$, as *coins*, que é um valor gerado aleatoriamente. A partir disto, vai-se conseguir gerar um criptograma da mensagem $m$ (com base no anel $Rq$). Seguiu-se o **algoritmo 5**, que se visualiza:

**Algorithm 5** $\text{KYBER.CPAPKE.Enc}(pk, m, r)$: encryption

**Input:** Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
**Input:** Message $m \in \mathcal{B}^{32}$
**Input:** Random coins $r \in \mathcal{B}^{32}$
**Output:** Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

$N := 0$
$\hat{\mathbf{t}} := \mathsf{Decode}_{12}(pk)$
$\rho := pk + 12 \cdot k \cdot n/8$
**for** $i$ from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  **for** $j$ from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}^T[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
  **end for**
**end for**
**for** $i$ from 0 to $k-1$ **do** ▷ Sample $\mathbf{r} \in R_q^k$ from $B_{\eta_1}$
  $\mathbf{r}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
  $N := N + 1$
**end for**
**for** $i$ from 0 to $k-1$ **do** ▷ Sample $\mathbf{e}_1 \in R_q^k$ from $B_{\eta_2}$
  $\mathbf{e}_1[i] := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
  $N := N + 1$
**end for**
$e_2 := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$ ▷ Sample $e_2 \in R_q$ from $B_{\eta_2}$
$\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
$v := \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ ▷ $v := \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
$c_1 := \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
$c_2 := \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
**return** $c = (c_1 \| c_2)$ ▷ $c := (\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$

- Determina a matriz A pertencente a $Rq$, no domínio $NTT$;
- Determina os vetores r, e1, e2 pertencente a $Rq$;
- Determina-se u, com recurso ao método *Compress* e aos vetores r e e1 - $u = A\hat{}T$ r + e1*.
- Determina-se v, com a adição do método *Decompress* e os vetores r e e2 - $v = t\hat{}T$ r + e2 + Decompress(m, 1).*
- Cria-se o criptograma $c$, da seguinte forma, c = (*Compress(u, du),Compress(v,dv)*).

**Método 'Dec'** O método *Dec* tem como objetivo **decifrar** uma mensagem. Recebe, para tal, o **criptograma** $c$ e a **chave pública** *sk*. Neste caso, o resultado final deverá ser a mensagem original $m$, antes de ter sido cifrada, a partir do método *Enc*.

O **algoritmo 6**, para a **decifragem**:

**Algorithm 6** KYBER.CPAPKE.Dec$(sk, c)$: decryption

**Input:** Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
**Input:** Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
**Output:** Message $m \in \mathcal{B}^{32}$

1: $\mathbf{u} := \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
2: $v := \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
3: $\hat{\mathbf{s}} := \mathsf{Decode}_{12}(sk)$
4: $m := \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ ▷ $m := \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
5: **return** $m$

- Determinar os vetores $u$ e $v$ (utilizando o método *Decompress*);
- Com o *decoding* da chave privada feito, obtém-se a mensagem através da expressão: *Compress(v - s^T u , 1)**.

[3]:
```
class KYBER_CPAPKE:

    ## -------------------------------- Inicialização␣
 ↪-------------------------------- ##

    def __init__(self):

        self.n = 256
        self.q = 7681
        self.k = 2
        self.n1 = 3
        self.n2 = 2
        self.du = 10
        self.dv = 4

        # Anel de polinómios (que tem coeficientes inteiros na var w)
        Z.<w> = GF(self.q)[]

        # Polinónimo irredutível f
        f = w^self.n + 1

        # Anel quociente de polinómios inteiros
        Rq.<w> = QuotientRing(Z ,Z.ideal(f))
        self.Rq = Rq

        # Matriz da transformada NTT
        self.M = NTT(self.n,self.q)

    ## -------------------------------- Funções Principais␣
 ↪-------------------------------- ##

    # Gerador de chaves privada/pública (a usar no processo de cifra/decifrar)
    def KeyGen(self):
```

```
# d ← B32
d = bytearray(os.urandom(32))

# (ρ, σ) := G(d)
ro,sigma = self.G(d)

# N := 0
N = 0

# Generate matrix Â  Rq(k x k) in NTT domain
mat_a = []

# Temos que são matrizes (k x k)
for i in range(self.k):
    mat_a.append([])
    for j in range(self.k):
        mat_a[i].append(self.M.ntt(self.Parse(self.XOF(ro,j,i))))


# Sample s  Rq from Bη1
sample_s = []

for i in range(self.k):
    sample_s.insert(i,self.CBD(self.PRF(sigma, N), self.n1))
    N = N+1

# Sample e  Rq from Bη1
sample_e = []

for i in range(self.k):
    sample_e.insert(i,self.CBD(self.PRF(sigma, N), self.n1))
    N = N+1

# ŝ := NTT(s), ê := NTT(e)
for i in range(self.k) :
    sample_s[i] = self.M.ntt(sample_s[i])
    sample_e[i] = self.M.ntt(sample_e[i])

# t1 := Â o ŝ
t1 = m_matrix_vec(mat_a, sample_s, self.k, self.n)
# t := t1 + ê
t = sum_matrix(t1, sample_e, self.n)

# pk := As + e
pk = t, ro
# sk := s
```

```
        sk = sample_s

        return pk,sk


    # Cifrar uma mensagem m, através da chave pública pk e um valor aleatório r
    def Enc(self,pk ,m ,r):

        # N := 0
        N = 0

        t, ro = pk

        # Generate matrix Â  Rq in NTT domain
        mat_a = []

        for i in range(self.k):
            mat_a.append([])
            for j in range(self.k):
                mat_a[i].append(self.M.ntt(self.Parse(self.XOF(ro,i,j))))

        # Sample r  Rq from Bη1, ^r:= NTT(r)
        sample_r = []

        for i in range(self.k):
            sample_r.insert(i,self.M.ntt(self.CBD(self.PRF(r, N), self.n1)))
            N += 1

        # Sample e1  Rq from Bη2
        sample_e1 = []

        for i in range(self.k):
            sample_e1.insert(i,self.CBD(self.PRF(r, N), self.n2))
            N += 1

        # Sample e2  Rq from Bη2
        sample_e2 = self.CBD(self.PRF(r, N), self.n2)

        # Â o r
        u1 = m_matrix_vec(mat_a, sample_r, self.k, self.n)

        # NTT-1( Â o r)
        u2 = []
        for i in range(len(u1)) :
            u2.append(self.M.ntt_inv(u1[i]))

        # u := NTT( Â o r) + e1
```

```
        u3 = sum_matrix(u2, sample_e1, self.n)

        u = []
        for i in range(len(u3)) :
            u.append(self.Rq(u3[i]))

        # t o r
        v1 = mult_matrix(t, sample_r,self.n)

        # NTT-1( t o r)
        v2 = self.M.ntt_inv(v1)
        # NTT-1( t o r) + e2
        v3 = self.Rq(sum_vecs(v2, sample_e2, self.n))

        # Decompress(m, 1)
        m_decom = self.Decompress(m, 1)
        # v := NTT-1( t o r) + e2 + Decompress(m, 1)
        v = self.Rq(sum_vecs(v3, m_decom, self.n))

        # Compress(u, du)
        c1 = []

        for i in range(len(u)):
            c1.append(self.Compress(u[i],self.du))


        # Compress(v, dv)
        c2 = self.Compress(v,self.dv)

        return (c1,c2)


    def Dec(self,sk, c):

        c1, c2 = c

        # Decompress(c1,du)
        u = []

        for i in range(len(c1)):
            u.append(self.Decompress(c1[i],self.du))

        # Decompress(c2,dv)
        v = self.Decompress(c2,self.dv)

        # NTT(u)
        ntt_u = []
```

```python
        for i in range(len(u)) :
            ntt_u.append(self.M.ntt(u[i]))

        # ŝ ∘ NTT(u)
        m1 = mult_matrix(sk, ntt_u, self.n)

        # v - NTT-1(ŝ ∘ NTT(u))
        m2 = sub_vecs(v, self.M.ntt_inv(m1), self.n)

        # Compress(v  NTT-1(ŝ ∘ NTT(u)), 1)
        m = self.Compress(self.Rq(m2), 1)

        return m


    ## -------------------------------- Métodos Auxiliares --------------------------------- ##

    # Método auxiliar para converter um array de bytes em bits
    def BytesToBits(self, arr):
        res = []

        for e in arr:
            elemArr=[]
            for i in range(0,8):
                elemArr.append(mod(e//2**(mod(i,8)),2))
                for i in range(0,len(elemArr)):
                    res.append(elemArr[i])

        return res

    # Extendable output funtion - XOF (como inserido no paper)
    def XOF(self,value, value1, value2):

        dig = hashes.Hash(hashes.SHAKE128(int(self.q)))
        dig.update(value)
        dig.update(bytes(value1))
        dig.update(bytes(value2))
        res = dig.finalize()
        return res

    # Pseudorandom function (PRF) - Criar um valor pseudo-aleatório
    def PRF(self,value, value1):

        dig = hashes.Hash(hashes.SHAKE256(int(self.q)))
        dig.update(value)
        dig.update(bytes(value1))
```

```
        return dig.finalize()

    # Hashing de um valor - retornando um tuplo com os valores
    def G(self, value):

        dig = hashes.Hash(hashes.SHA3_512())
        dig.update(bytes(value))
        res = dig.finalize()

        return res[:32],res[32:]

    # Algoritmo 1 - Sampling estatístico (parsing de um vetor de bytes b para um polinómio em Rq)
    def Parse(self,b):

        i = 0
        j = 0
        res=[]

        while j < self.n:
            d1 = b[i] + 256 * mod(b[i+1],16)
            d2 = b[i+1]//16 + 16 * b[i+2]

            if d1 < self.q :
                res.append(d1)
                j = j+1
            if d2 < self.q and j<self.n:
                res.append(d2)
                j = j+1
            i = i+3

        sample = self.Rq(res)

        return sample


    # Algoritmo 2 - Smapling from a binomial distribution (retorna um anel Rq)
    def CBD(self,B,nn):

        f=[0]*self.n

        # Utilizar o método para transformar um array de bytes em bits
        b_arr = self.BytesToBits(B)

        for i in range(256):
            a = 0
            b = 0
```

```
            for j in range(nn):
                a += b_arr[2*i*nn + j]
                b += b_arr[2*i*nn + nn + j]
            f[i] = a-b

        sample = self.Rq(f)

        return sample

    # Algoritmo 3 - Criar um polinómio f do anel a Rq (segundo um array de bytes
↪B)
    def Decode(self,B,l):

        f = []

        # Utilizar o método para transformar um array de bytes em bits
        b_arr = self.BytesToBits(B)

        for i in range(len(B)):
            fi = 0
            for j in range(l):
                fi += int(b_arr[i*l+j]) * 2**j
            f.append(fi)

        poli = self.Rq(f)

        return poli

    # Remove low-order bits no x (não todos)
    def Compress(self,x,d) :

        res = []

        for coef in x.list():
            new = mod(round( int(2 ** d) / self.q * int(coef)), int(2 ** d))
            res.append(new)

        no_low = self.Rq(res)

        return no_low

    # Repõem o x parcialmente
    def Decompress(self,x,d) :

        res = []

        for coef in x.list():
```

```python
            new = round(self.q / (2 ** d) * int(coef))
            res.append(new)

        new_x = self.Rq(res)

        return new_x
```

### 0.2.3 KYBER-CCAPKE (PKE IND-CCA)

Para o *PKE* da técnica *KYBER*, como referido anteriormente, é pedido que esta seja *IND-CCA* seguro.

*IND-CCA* significa "**indistinguível para o adversário com acesso a um criptograma escolhido**" (em inglês, *indistinguishability under chosen ciphertext attack*).

Um sistema criptográfico é considerado *IND-CCA* seguro se, mesmo que um atacante consiga escolher textos cifrados e obter as suas decifragens, ele não seja capaz de distinguir a decifragem de um texto cifrado escolhido aleatoriamente de outra decifragem qualquer.

Em outras palavras, o atacante não deve ser capaz de deduzir qualquer informação sobre a chave secreta a partir da análise do comportamento do sistema criptográfico ao ser submetido a criptogramas escolhidos por ele.

Neste caso, no **capítulo 2a**, a equipa docente forneceu um modo de tornar um *PKE IND-CPA* (o que temos implementado), para um *PKE IND-CCA*, na forma da **transformação de Fujisaki-Okamoto**.

Antes de entrar nas alterações significantes, a classe tem um método *KeyGen* que, tal como o *KEM*, utilizará simplesmente o método da classe *KYBER_ CPAPKE*.

**Método 'Enc'** Tem-se, segundo o *paper*, a seguinte transformação para o método de cifra $E$: **E'(x) forall r <- h . forall (y, r' ) <- (x 'xor' g(r), h(r||y)) . (y , f(r, r '))**

O método *Enc*, no final, irá devolver um par (**tuplo**), onde $y$ é o resultado da operação de *xor* entre a mensagem original $x$ e a saída da função $g$ aplicada ao valor aleatório $r$, ou seja, $y$ = $x$ *'xor'* $g(r)$ e $c$ é o criptograma gerado pelo método *Enc*. Segue-se, então, os seguintes passos: - Gera um valor r, aleatório Rq e a sua respetiva *hash*, via o "hashing" $g$; - Primeiro elemento do tuplo, $y$, será o resultado da operação de *XOR* com a mensagem $x$ e o valor $r$ (uma **ofuscação** da mensagem); - O segundo, é o resultado de aplicar a função dde cifra, do *KYBER_ CPAPKE*, com a chave pública *pk*, a mensagem a ser o valor aleatório $r$ e a variável *coins* ser a concatenação de $y$ com $r$.

**Método 'Dec'** Relativamente à decifra, o seguinte algoritmo:

**D'(y, c) forall r <- D(c) . if c != f(r, h(r||y)) then None else y 'xor' g(r)**

… rejeita o criptograma se detetar algum sinal de fraude, para além de recuperar a mensagem $x$.

Para este:

- Utiliza-se o método de decifra do *KYBE_ CPAPKE* com o criptograma criado, $c$, guardado na variável $r$;

- Se $c$ não for igual ao resultado de aplicar a função $Enc$ em $r$ com a concatenação desse $r$ com $y$, então ocorreu um erro.

[4]:
```
class KYBER_CCAPKE:

    def __init__(self):
        self.cpa = KYBER_CPAPKE()

    ## -------------------------------- Métodos principais␣
 ↪-------------------------------- ##

    # Retorna as chaves pública/privada a serem usadas no encapsulamento/
 ↪desencapsulamento
    def KeyGen(self):

        pk, sk = self.cpa.KeyGen()
        return pk, sk

    def Enc(self, x, pk):

        #  r←h
        r = self.cpa.Rq([choice([0, 1]) for i in range(self.cpa.n)])

        #  y← x  g(r)
        y = self.XOR(x, self.g(bytes(r)))

        # c←Enc(r,h(ry))
        c = self.cpa.Enc(pk,r,self.g(bytes(r)+y))

        return (y, c) # (y,Enc(r,r'))


    def Dec(self, y, c, sk):

        #  r←Dec(c)
        r = self.cpa.Dec(sk, c)

        # f(r,h(ry))
        f = self.cpa.Enc(pk,r,self.g(bytes(r)+y))

        if c[0] != f[0]: # c  f(r,h(ry))
            return None  #
        else:
            return self.XOR(y, self.g(bytes(r))) # y  g(r)

    ## -------------------------------- Métodos auxiliares␣
 ↪-------------------------------- ##
```

```python
    # Método para realizar um xor
    def XOR(self,k, value):
        return bytes(a ^^ b for a, b in zip(k, value))

    # Função de hashing
    def g(self, value):
        digest = hashes.Hash(hashes.SHA3_256())
        digest.update(value)

        return digest.finalize()
```

### 0.2.4 KYBER-CPAKEM (KEM IND-CPA)

Seguindo, mais uma vez o documento, tem-se que o algoritmo *KEM* desejado pelos autores vai ser *IND-CCA* seguro. Deste modo, utilizaram o esquema *PKE IND-CPA*, *tweaked* com uma transformação *Fujisaki-Okamoto*, para construirem um *KEM IND-CCA* seguro.

Isto é o contrário do pedido pela equipa docente, pelo que, neste caso, será apenas utilizado o esquema *PKE IND-CPA* seguro, criado em cima, poara construir um protótipo de um *KEM IND-CPA* seguro.

Mesmo assim, seguiu-se os algoritmos definidos pela documentação.

Tem-se, então, 3 novos métodos, para **gerar chaves pública/privada**, **encpasular chave partilhada** e o respetivo **desencapsulamento**.

Antes de tudo, foi criado uma única **função auxiliar**, o método **H**.

**Funções auxiliares** O método *H* é usado na implementação do algoritmo de *KYBER* como uma função de *hash* criptográfico para transformar um *input* arbitrário de *bytes* num *hash output* de comprimento fixo. Ele é usado para transformar o *output* da função de **encapsulamento** numa **chave secreta de tamanho fixo** (chave partilhada).

Utiliza o algoritmo *SHA3-256* da biblioteca *hashlib* do *Python* para calcular o *hash* criptográfico de um *input value*. Ele retorna o *hash* calculado como uma *string* de *bytes* de tamanho fixo de **32 bytes** (**256 bits**).

... A escolha da função de *hash*, tal como na classe a cima, estava definida na documentação dada.

**Método 'KeyGen** Neste caso, tal como exemplificado no **algoritmo 7**:

**Algorithm 7** KYBER.CCAKEM.KeyGen()

**Output:** Public key $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$
**Output:** Secret key $sk \in \mathcal{B}^{24\cdot k\cdot n/8+96}$
1: $z \leftarrow \mathcal{B}^{32}$
2: $(pk, sk') := \text{KYBER.CPAPKE.KeyGen}()$
3: $sk := (sk' \| pk \| \mathrm{H}(pk) \| z)$
4: **return** $(pk, sk)$

. . . utiliza-se diretamente o método *KeyGen* da classe *KYBER_ CPAPKE*, dado que segue o mesmo algoritmo. Tal como se visualiza na imagem em cima.

**Método 'Enc'** Relativamente ao **algoritmo 8**:

**Algorithm 8** KYBER.CCAKEM.Enc($pk$)

**Input:** Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
**Output:** Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
**Output:** Shared key $K \in \mathcal{B}^*$
$m \leftarrow \mathcal{B}^{32}$
$m \leftarrow \mathrm{H}(m)$ ▷ Do not send output of system RNG
$(\bar{K}, r) := \mathrm{G}(m \| \mathrm{H}(pk))$
$c :=$ KYBER.CPAPKE.Enc$(pk, m, r)$
$K :=$ KDF$(\bar{K} \| \mathrm{H}(c))$
**return** $(c, K)$

O método *Enc(pk)* é utilizado para encapsular uma **chave secreta** utilizando a **chave pública** *pk*. Este método gera um *nonce* aleatório *m*, calcula uma *hash key* da serialização de *m* utilizando a função *H*, e cifra *m* utilizando a função *Enc* da classe *KYBER_ CPAPKE*. A função Enc recebe como *input* a chave pública *pk*, o vetor *m* e um vetor aleatório *r*, relativo às *coins*.

O nonce está a ser criado aleatoriamente usando a função *choice* do módulo *random*. A função choice([0, 1]) escolhe aleatoriamente um dos dois valores possíveis (0 ou 1), e esta escolha é repetida **n** vezes para criar um vetor aleatório de *n bits*. Em seguida, este vetor é usado como *input* para o anel *Rq*, que codifica o vetor como um polinômio de grau **n-1** com coeficientes inteiros módulo *q*. O polinômio resultante é, então, usado como o *nonce* para criar a chave secreta.

**Método 'Dec'** Por fim, o **algoritmo 9**:

**Algorithm 9** KYBER.CCAKEM.Dec($c$, $sk$)

**Input:** Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
**Input:** Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$
**Output:** Shared key $K \in \mathcal{B}^*$
$pk := sk + 12 \cdot k \cdot n/8$
$h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
$z := sk + 24 \cdot k \cdot n/8 + 64$
$m' :=$ KYBER.CPAPKE.Dec$(\mathbf{s}, (\mathbf{u}, v))$
$(\bar{K}', r') := \mathrm{G}(m' \| h)$
$c' :=$ KYBER.CPAPKE.Enc$(pk, m', r')$
**if** $c = c'$ **then**
    **return** $K :=$ KDF$(\bar{K}' \| \mathrm{H}(c))$
**else**
    **return** $K :=$ KDF$(z \| \mathrm{H}(c))$
**end if**
**return** $K$

O método *Dec(c, sk)* é utilizado para desencapsular a **chave secreta** utilizando a **chave privada** *sk*. Este método decifra o vetor *m* cifrado no passo anterior utilizando a função *Dec* da classe

*KYBER_ CPAPKE*, calcula uma *hash key* da serialização de $m$ utilizando a função $H$ e retorna a *key*.

[5]:
```
class KYBER_CPAKEM:

    #Função de inicialização das variaveis a usar nos métodos
    def __init__(self):
        self.cpa = KYBER_CPAPKE()

    ## ------------------------------- Métodos principais␣
 ↪------------------------------- ##

    # Retorna as chaves pública/privada a serem usadas no encapsulamento/
 ↪desencapsulamento
    def KeyGen(self):

        pk, sk = self.cpa.KeyGen()
        return pk, sk

    # Método para encapsualar a chave secreta (com a chave pública)
    def Enc(self,pk):

        # Criar um valor aleatório (nonce)
        m = self.cpa.Rq([choice([0, 1]) for i in range(self.cpa.n)])
        # chave secreta
        k = self.H(dumps(m))

        # variável aleatória, coins
        r = os.urandom(256)
        c = self.cpa.Enc(pk,m,r)

        return c, k

    # Método para desencapsular a chave secreta (partilhada), através da chave␣
 ↪privada
    def Dec(self,c,sk):

        # Mensagem decifrada
        m = self.cpa.Dec(sk,c)

        # Chave secreta (partilhada)
        k = self.H(dumps(m))

        return k

    ## ------------------------------- Métodos auxiliares␣
 ↪------------------------------- ##
```

```
    # Método de hash para criar a chave sereta (partilhada)
    def H(self, value):

        digest = hashes.Hash(hashes.SHA3_256())
        digest.update(value)

        return digest.finalize()
```

### 0.2.5 Funções auxiliares (matrizes)

Todas as funções presentes na seguinte célula são utilizadas para executar operações entre vetores e matrizes, como **soma**, **subtração** e **multiplicação**.

- As funções *sum_vecs*, *sub_vecs* e *mult_vecs* são utilizadas para realizar operações elemento a elemento entre vetores, nomeadamente a **soma**, **subtração** e **multiplicação**.
- As funções *sum_matrix*, *sub_matrix* e mult_matrix são utilizadas para realizar operações elemento a elemento entre matrizes, de modo análogo ao primeiro ponto.
- A função *m_matrix_vec* é utilizada para multiplicar uma matriz por um vetor e retornar o vetor resultante.

[6]:
```
## Métodos auxiliares (para executar operações entre matrizes)

## -------------------------------------------- Entre vetores␣
 ↪-------------------------------------------- ##

def sum_vecs(ff1, ff2,n):
    return list(map(lambda x, y: x + y, ff1, ff2))

def mult_vecs(ff1, ff2,n):
    return list(map(lambda x, y: x * y, ff1, ff2))

def sub_vecs(ff1, ff2,n):
    return list(map(lambda x, y: x - y, ff1, ff2))

## -------------------------------------------- Entre matrizes␣
 ↪-------------------------------------------- ##

def sum_matrix(e1,e2,n):
    return list(map(lambda x, y: sum_vecs(x, y, n), e1, e2))

def sub_matrix(e1,e2,n):
    return list(map(lambda x, y: sub_vecs(x, y, n), e1, e2))

def mult_matrix(vec1, vec2, n):
    res = []
    for i in range(len(vec1)):
```

```
        res.append(mult_vecs(vec1[i], vec2[i],n))
    return reduce(lambda x, y: sum_vecs(x, y, n), res)


# Multiplicação entre matriz e vetor
def m_matrix_vec(M,v,k,n):
    res = []
    for i in range(len(M)):
        row = list(map(lambda x: mult_vecs(x, v[i],n), M[i]))
        res.append(reduce(lambda x, y: sum_vecs(x, y, n), row))
    return res
```

## 0.3 Teste do algoritmo PKE (IND-CCA)

[7]:
```
PKE = KYBER_CCAPKE()

pk, sk = PKE.KeyGen()

print("Chaves pública/privada criadas.")
print()

mens = "12345678910"
print("A mensagem a enviar: ", mens)
print()

print("A cifrar mensagem...")
print()

(y, c) = PKE.Enc(mens.encode(), pk)

print("O valor 'y': ")
print()
print(y)
print()
print("O valor 'c' (criptograma): ")
print()
print(c)
print()


print("A testar correção da decifragem...")
print()
decifra = PKE.Dec(y, c, sk)

if decifra == None:
    raise ValueError("Ocorreu um erro no processo do PKE IND-CCA!")
```

```python
print("O processo de cifra/decifra ocorreu com sucesso!", decifra.decode())
```

```
Chaves pública/privada criadas.

A mensagem a enviar:  12345678910

A cifrar mensagem...

O valor 'y':

b'Lh\x81\xde$\xc6\x86*\xc7\xafP'

O valor 'c' (criptograma):

([994*w^255 + 47*w^254 + 934*w^253 + 753*w^252 + 182*w^251 + 17*w^250 +
843*w^249 + 142*w^248 + 427*w^247 + 864*w^246 + 515*w^245 + 198*w^244 +
599*w^243 + 507*w^242 + 83*w^241 + 142*w^240 + 431*w^239 + 889*w^238 + 616*w^237
+ 266*w^236 + 30*w^235 + 195*w^234 + 410*w^233 + 272*w^232 + 338*w^231 +
681*w^230 + 793*w^229 + 837*w^228 + 1015*w^227 + 428*w^226 + 254*w^225 +
1003*w^224 + 878*w^223 + 914*w^222 + 366*w^221 + 141*w^220 + 347*w^219 +
590*w^218 + 969*w^217 + 87*w^216 + 278*w^215 + 243*w^214 + 982*w^213 + 345*w^212
+ 629*w^211 + 37*w^210 + 938*w^209 + 20*w^208 + 519*w^207 + 1003*w^206 +
333*w^205 + 831*w^204 + 666*w^203 + 931*w^202 + 522*w^201 + 680*w^200 +
690*w^199 + 376*w^198 + 258*w^197 + 194*w^196 + 318*w^195 + 281*w^194 +
848*w^193 + 658*w^192 + 480*w^191 + 308*w^190 + 170*w^189 + 485*w^188 +
189*w^187 + 158*w^186 + 556*w^185 + 245*w^184 + 10*w^183 + 859*w^182 + 701*w^181
+ 275*w^180 + 937*w^179 + 363*w^178 + 22*w^177 + 950*w^176 + 645*w^175 +
963*w^174 + 914*w^173 + 688*w^172 + 217*w^171 + 734*w^170 + 661*w^169 + 95*w^168
+ 524*w^167 + 916*w^166 + 17*w^165 + 843*w^164 + 767*w^163 + 563*w^162 +
719*w^161 + 614*w^160 + 651*w^159 + 497*w^158 + 45*w^157 + 617*w^156 + 944*w^155
+ 33*w^154 + 869*w^153 + 521*w^152 + 267*w^151 + 294*w^150 + 729*w^149 +
750*w^148 + 319*w^147 + 115*w^146 + 315*w^145 + 749*w^144 + 155*w^143 +
876*w^142 + 56*w^141 + 77*w^140 + 252*w^139 + 873*w^138 + 763*w^137 + 809*w^136
+ 455*w^135 + 536*w^134 + 489*w^132 + 983*w^131 + 585*w^130 + 926*w^129 +
877*w^128 + 329*w^127 + 555*w^126 + 654*w^125 + 8*w^124 + 590*w^123 + 804*w^122
+ 303*w^121 + 242*w^120 + 167*w^119 + 554*w^118 + 884*w^117 + 783*w^116 +
366*w^115 + 402*w^114 + 281*w^113 + 369*w^112 + 696*w^111 + 436*w^110 +
190*w^109 + 987*w^108 + 207*w^107 + 794*w^106 + 596*w^105 + 346*w^104 +
572*w^103 + 708*w^102 + 558*w^101 + 682*w^100 + 438*w^99 + 213*w^98 + 894*w^97 +
718*w^96 + 154*w^95 + 892*w^94 + 436*w^93 + 456*w^92 + 141*w^91 + 500*w^90 +
570*w^89 + 509*w^88 + 942*w^87 + 551*w^86 + 913*w^85 + 714*w^84 + 452*w^83 +
111*w^82 + 515*w^81 + 50*w^80 + 927*w^79 + 585*w^78 + 602*w^77 + 807*w^76 +
215*w^75 + 845*w^74 + 891*w^73 + 170*w^72 + 675*w^71 + 522*w^70 + 841*w^69 +
749*w^68 + 910*w^67 + 570*w^66 + 314*w^65 + 873*w^64 + 807*w^63 + 771*w^62 +
694*w^61 + 172*w^60 + 53*w^59 + 71*w^58 + 925*w^57 + 359*w^56 + 509*w^55 +
827*w^54 + 627*w^53 + 532*w^52 + 87*w^51 + 239*w^50 + 633*w^49 + 849*w^48 +
1004*w^47 + 111*w^46 + 25*w^45 + 200*w^44 + 495*w^43 + 992*w^42 + 451*w^41 +
155*w^40 + 443*w^39 + 455*w^38 + 745*w^37 + 1019*w^36 + 739*w^35 + 455*w^34 +
```

364*w^33 + 497*w^32 + 656*w^31 + 331*w^30 + 101*w^29 + 510*w^28 + 281*w^27 +
370*w^26 + 775*w^25 + 871*w^24 + 582*w^23 + 267*w^22 + 434*w^21 + 721*w^20 +
217*w^19 + 900*w^18 + 210*w^17 + 573*w^16 + 668*w^15 + 689*w^14 + 736*w^13 +
612*w^12 + 615*w^11 + 440*w^10 + 505*w^9 + 196*w^8 + 444*w^7 + 214*w^6 + 308*w^5
+ 985*w^4 + 830*w^3 + 293*w^2 + 812*w + 885, 1022*w^255 + 810*w^254 + 113*w^253
+ 877*w^252 + 772*w^251 + 129*w^250 + 447*w^249 + 163*w^248 + 571*w^247 +
863*w^246 + 773*w^245 + 719*w^244 + 901*w^243 + 455*w^242 + 300*w^241 +
238*w^240 + 423*w^239 + 212*w^238 + 384*w^237 + 54*w^236 + 17*w^235 + 74*w^234 +
800*w^233 + 982*w^232 + 730*w^231 + 181*w^230 + 64*w^229 + 546*w^228 + 382*w^227
+ 97*w^226 + 442*w^225 + 515*w^224 + 197*w^223 + 525*w^222 + 537*w^221 +
195*w^220 + 948*w^219 + 316*w^218 + 360*w^217 + 234*w^216 + 217*w^215 +
697*w^214 + 805*w^213 + 635*w^212 + 935*w^211 + 980*w^210 + 528*w^209 +
347*w^208 + 41*w^207 + 320*w^206 + 464*w^205 + 314*w^204 + 542*w^203 + 265*w^202
+ 978*w^201 + 326*w^200 + 773*w^199 + 999*w^198 + 62*w^197 + 726*w^196 +
344*w^195 + 648*w^194 + 806*w^193 + 562*w^192 + 416*w^191 + 930*w^190 + 4*w^189
+ 905*w^188 + 457*w^187 + 352*w^186 + 474*w^185 + 809*w^184 + 476*w^183 +
489*w^182 + 471*w^181 + 570*w^180 + 289*w^179 + 257*w^178 + 737*w^177 +
440*w^176 + 885*w^175 + 283*w^174 + 307*w^173 + 347*w^172 + 277*w^171 +
458*w^170 + 906*w^169 + 51*w^168 + 989*w^167 + 894*w^166 + 261*w^165 + 588*w^164
+ 858*w^163 + 210*w^162 + 416*w^161 + 381*w^160 + 989*w^159 + 588*w^158 +
59*w^157 + 884*w^156 + 732*w^155 + 655*w^154 + 353*w^153 + 517*w^152 + 245*w^151
+ 88*w^150 + 281*w^149 + 436*w^148 + 345*w^147 + 675*w^146 + 483*w^145 +
913*w^144 + 846*w^143 + 276*w^142 + 117*w^141 + 699*w^140 + 779*w^139 + 71*w^138
+ 680*w^137 + 34*w^136 + 698*w^135 + 235*w^134 + 196*w^133 + 531*w^132 +
211*w^131 + 526*w^130 + 495*w^129 + 44*w^128 + 692*w^127 + 216*w^126 + 200*w^125
+ 439*w^124 + 285*w^123 + 4*w^122 + 430*w^121 + 913*w^120 + 335*w^119 +
965*w^118 + 366*w^117 + 601*w^116 + 282*w^115 + 787*w^114 + 654*w^113 +
459*w^112 + 89*w^111 + 880*w^110 + 277*w^109 + 204*w^108 + 320*w^107 + 466*w^106
+ 255*w^105 + 9*w^104 + 1021*w^103 + 101*w^102 + 409*w^101 + 735*w^100 +
482*w^99 + 172*w^98 + 35*w^97 + 754*w^96 + 56*w^95 + 643*w^94 + 961*w^93 +
329*w^92 + 967*w^91 + 390*w^90 + 737*w^89 + 751*w^88 + 242*w^87 + 452*w^86 +
432*w^85 + 796*w^84 + 1018*w^83 + 392*w^82 + 906*w^81 + 549*w^80 + 407*w^79 +
549*w^78 + 417*w^77 + 216*w^76 + 995*w^75 + 439*w^74 + 177*w^73 + 580*w^72 +
675*w^71 + 658*w^70 + 911*w^69 + 438*w^68 + 511*w^67 + 294*w^66 + 624*w^65 +
227*w^64 + 520*w^63 + 666*w^62 + 577*w^61 + 100*w^60 + 891*w^59 + 561*w^58 +
226*w^57 + 497*w^56 + 561*w^55 + 424*w^54 + 970*w^53 + 754*w^52 + 498*w^51 +
760*w^50 + 39*w^49 + 538*w^48 + 218*w^47 + 873*w^46 + 269*w^45 + 269*w^44 +
450*w^43 + 755*w^42 + 732*w^41 + 250*w^40 + 714*w^39 + 114*w^38 + 854*w^37 +
159*w^36 + 598*w^35 + 696*w^34 + 793*w^33 + 991*w^32 + 145*w^31 + 762*w^30 +
704*w^29 + 491*w^28 + 565*w^27 + 234*w^26 + 395*w^25 + 918*w^24 + 81*w^23 +
73*w^22 + 770*w^21 + 211*w^20 + 860*w^19 + 300*w^18 + 920*w^17 + 897*w^16 +
233*w^15 + 113*w^14 + 455*w^13 + 602*w^12 + 431*w^11 + 41*w^10 + 860*w^9 +
200*w^8 + 941*w^7 + 583*w^6 + 246*w^5 + 43*w^4 + 282*w^3 + 260*w^2 + 53*w +
757], 2*w^254 + 5*w^253 + 9*w^252 + 12*w^251 + 14*w^250 + 11*w^248 + 8*w^247 +
8*w^245 + 10*w^244 + 15*w^243 + 4*w^242 + 2*w^241 + 2*w^240 + 8*w^239 + 4*w^238
+ w^237 + 15*w^236 + 4*w^234 + 15*w^233 + 9*w^232 + 6*w^231 + 8*w^230 + 2*w^229
+ 9*w^228 + 7*w^227 + 7*w^226 + 3*w^225 + 10*w^224 + 10*w^223 + 10*w^222 +
13*w^221 + 3*w^220 + 5*w^219 + w^218 + 7*w^217 + 9*w^215 + 5*w^214 + 3*w^213 +

```
4*w^212 + 6*w^211 + 13*w^210 + 10*w^209 + 13*w^208 + 8*w^207 + 2*w^206 + w^205 +
w^204 + 7*w^203 + 6*w^202 + 15*w^201 + 3*w^200 + 11*w^199 + 9*w^198 + 2*w^197 +
6*w^196 + 2*w^195 + 2*w^194 + 12*w^193 + 13*w^192 + 10*w^191 + 8*w^190 +
13*w^189 + 2*w^188 + 7*w^187 + 13*w^186 + 8*w^185 + 7*w^184 + 5*w^181 + 7*w^180
+ 2*w^179 + 2*w^178 + 2*w^177 + 13*w^176 + 7*w^175 + 9*w^174 + w^173 + 3*w^172 +
15*w^171 + 5*w^170 + 3*w^169 + 15*w^168 + 9*w^167 + w^166 + 14*w^165 + 10*w^164
+ w^163 + w^162 + 2*w^160 + w^159 + 14*w^158 + 7*w^156 + 14*w^155 + 5*w^154 +
7*w^153 + 8*w^151 + 2*w^150 + 3*w^149 + 14*w^148 + 12*w^147 + 7*w^146 + 10*w^145
+ 4*w^144 + 8*w^143 + 9*w^142 + 8*w^141 + 12*w^140 + 3*w^139 + 14*w^138 +
2*w^137 + w^136 + 2*w^135 + 15*w^134 + 8*w^132 + 15*w^131 + 15*w^130 + 8*w^129 +
11*w^128 + 14*w^127 + 3*w^126 + 9*w^125 + 5*w^124 + 13*w^122 + 8*w^120 + 2*w^119
+ 6*w^118 + 7*w^117 + 7*w^116 + 4*w^115 + 4*w^114 + 9*w^113 + 6*w^112 + 4*w^111
+ 2*w^110 + 13*w^109 + 7*w^108 + 4*w^107 + 14*w^105 + 13*w^104 + 14*w^103 +
5*w^102 + 4*w^101 + 5*w^100 + 15*w^99 + 4*w^98 + 2*w^97 + w^96 + 6*w^95 + 6*w^94
+ 14*w^93 + 12*w^92 + 3*w^91 + 15*w^90 + 2*w^89 + 5*w^87 + 5*w^86 + 9*w^85 +
3*w^84 + 11*w^83 + 2*w^82 + 3*w^81 + 10*w^79 + 5*w^77 + 2*w^76 + 8*w^75 +
12*w^74 + 2*w^73 + 12*w^72 + 3*w^71 + 7*w^70 + 8*w^69 + 5*w^68 + 10*w^66 +
3*w^65 + 3*w^64 + 14*w^63 + w^61 + 2*w^60 + 4*w^59 + 15*w^58 + 8*w^57 + w^56 +
2*w^55 + 9*w^54 + 7*w^53 + 12*w^52 + 13*w^51 + w^50 + 11*w^49 + 13*w^48 +
11*w^47 + 8*w^46 + 7*w^45 + 5*w^44 + 12*w^43 + 4*w^42 + 12*w^41 + 13*w^40 +
6*w^39 + 13*w^38 + 3*w^37 + 2*w^36 + 13*w^34 + 5*w^33 + w^32 + 12*w^31 + 11*w^30
+ 15*w^29 + w^28 + 3*w^27 + 9*w^25 + 6*w^24 + 8*w^23 + 7*w^22 + 4*w^21 + 14*w^20
+ 13*w^19 + 14*w^18 + w^17 + 12*w^16 + 3*w^15 + 13*w^14 + 6*w^13 + 14*w^12 +
3*w^11 + w^10 + 8*w^9 + w^7 + 11*w^6 + 10*w^5 + 9*w^4 + 13*w^3 + 15*w^2 + 2*w)

A testar correção da decifragem...

O processo de cifra/decifra ocorreu com sucesso! 12345678910
```

### 0.4 Teste do algoritmo KEM (IND-CPA)

[9]:
```python
KEM = KYBER_CPAKEM()

pk, sk = KEM.KeyGen()

print("Criando e encapsulando chave secreta...")
print()

c, k = KEM.Enc(pk)

print("Criptograma: ")
print()
print(c)
print()

print("Chave secreta (partilhada/encapsulada): ")
print()
```

```python
print(k)
print()

print("A desencapsular a chave secreta...")
print()

shared_k = KEM.Dec(c, sk)
print("A chave partilhada é: ")
print()
print(shared_k)
print()

if k != shared_k:
    raise ValueError("Ocorreu um erro! A chave não foi desencapsulada com␣
 ↪sucesso!", shared_k)

print("A chave foi transmitida com sucesso! ", shared_k)
```

```
Criando e encapsulando chave secreta...

Criptograma:

([387*w^255 + 102*w^254 + 982*w^253 + 42*w^252 + 326*w^251 + 822*w^250 +
908*w^249 + 297*w^248 + 207*w^247 + 796*w^246 + 580*w^245 + 607*w^244 +
560*w^243 + 968*w^242 + 773*w^241 + 828*w^240 + 25*w^239 + 582*w^238 + 656*w^237
+ 131*w^236 + 15*w^235 + 752*w^234 + 135*w^233 + 298*w^232 + 869*w^231 +
646*w^230 + 452*w^229 + 914*w^228 + 142*w^227 + 382*w^226 + 103*w^225 +
474*w^224 + 923*w^223 + 39*w^222 + 408*w^221 + 187*w^220 + 729*w^219 + 520*w^218
+ 584*w^217 + 999*w^216 + 793*w^215 + 575*w^214 + 303*w^213 + 454*w^212 +
804*w^211 + 138*w^210 + 891*w^209 + 536*w^208 + 307*w^207 + 964*w^206 +
669*w^205 + 730*w^204 + 895*w^203 + 338*w^202 + 52*w^201 + 480*w^200 + 585*w^199
+ 455*w^198 + 636*w^197 + 176*w^196 + 911*w^195 + 619*w^194 + 187*w^193 +
338*w^192 + 453*w^191 + 1018*w^190 + 918*w^189 + 418*w^188 + 920*w^187 +
399*w^186 + 649*w^185 + 571*w^184 + 417*w^183 + 479*w^182 + 57*w^181 + 238*w^180
+ 749*w^179 + 626*w^178 + 656*w^177 + 823*w^176 + 457*w^175 + 452*w^174 +
381*w^173 + w^172 + 173*w^171 + 647*w^170 + 845*w^169 + 179*w^168 + 206*w^167 +
122*w^166 + 246*w^165 + 521*w^164 + 897*w^163 + 269*w^162 + 528*w^161 +
776*w^160 + 487*w^159 + 331*w^158 + 406*w^157 + 498*w^156 + 242*w^155 +
151*w^154 + 746*w^153 + 322*w^152 + 1001*w^151 + 1022*w^150 + 972*w^149 +
311*w^148 + 20*w^147 + 243*w^146 + 532*w^145 + 867*w^144 + 320*w^143 + 492*w^142
+ 202*w^141 + 104*w^140 + 313*w^139 + 678*w^138 + 1022*w^137 + 144*w^136 +
399*w^135 + 47*w^134 + 209*w^133 + 730*w^132 + 229*w^131 + 959*w^130 + 971*w^129
+ 540*w^128 + 808*w^127 + 121*w^126 + 323*w^125 + 4*w^124 + 897*w^123 +
698*w^122 + 71*w^121 + 545*w^120 + 758*w^119 + 179*w^118 + 615*w^117 + 216*w^116
+ 369*w^115 + 466*w^114 + 702*w^113 + 453*w^112 + 639*w^111 + 77*w^110 +
30*w^109 + 626*w^108 + 337*w^107 + 135*w^106 + 289*w^105 + 1018*w^104 +
680*w^103 + 159*w^102 + 821*w^101 + 808*w^100 + 8*w^99 + 655*w^98 + 248*w^97 +
478*w^96 + 899*w^95 + 112*w^94 + 618*w^93 + 301*w^92 + 977*w^91 + 901*w^90 +
```

2*w^89 + 960*w^88 + 431*w^87 + 107*w^86 + 795*w^85 + 364*w^84 + 1013*w^83 +
297*w^82 + 556*w^81 + 144*w^80 + 15*w^79 + 557*w^78 + 632*w^77 + 281*w^76 +
308*w^75 + 674*w^74 + 981*w^73 + 41*w^72 + 891*w^71 + 472*w^70 + 241*w^69 +
379*w^68 + 255*w^67 + 659*w^66 + 792*w^65 + 249*w^64 + 957*w^63 + 270*w^62 +
451*w^61 + 710*w^60 + 506*w^59 + 710*w^58 + 97*w^57 + 403*w^56 + 806*w^55 +
560*w^54 + 257*w^53 + 118*w^52 + 388*w^51 + 440*w^50 + 714*w^49 + 138*w^48 +
267*w^47 + 806*w^46 + 987*w^45 + 304*w^44 + 128*w^43 + 185*w^42 + 965*w^41 +
25*w^40 + 291*w^39 + 411*w^38 + 619*w^37 + 917*w^36 + 361*w^35 + 385*w^34 +
536*w^33 + 412*w^32 + 243*w^31 + 236*w^30 + 920*w^29 + 620*w^28 + 814*w^27 +
739*w^26 + 510*w^25 + 561*w^24 + 415*w^23 + 599*w^22 + 171*w^21 + 7*w^20 +
429*w^19 + 913*w^18 + 695*w^17 + 772*w^16 + 790*w^15 + 59*w^14 + 53*w^13 +
476*w^12 + 14*w^11 + 441*w^10 + 323*w^9 + 477*w^8 + 894*w^7 + 111*w^6 + 578*w^5
+ 455*w^4 + 156*w^3 + 794*w^2 + 602*w + 670, 98*w^255 + 280*w^254 + 29*w^253 +
984*w^252 + 489*w^251 + 70*w^250 + 71*w^249 + 508*w^248 + 821*w^247 + 79*w^246 +
369*w^245 + 227*w^244 + 85*w^243 + 737*w^242 + 405*w^241 + 107*w^240 + 853*w^239
+ 902*w^238 + 272*w^237 + 618*w^236 + 799*w^235 + 350*w^234 + 627*w^233 +
972*w^232 + 989*w^231 + 420*w^230 + 553*w^229 + 896*w^228 + 789*w^227 +
339*w^226 + 646*w^225 + 902*w^224 + 417*w^223 + 824*w^222 + 272*w^221 +
851*w^220 + 115*w^219 + 82*w^218 + 812*w^217 + 953*w^216 + 802*w^215 + 566*w^214
+ 89*w^213 + 996*w^212 + 121*w^211 + 305*w^210 + 392*w^209 + 978*w^208 +
683*w^207 + 863*w^206 + 248*w^205 + 21*w^204 + 383*w^203 + 49*w^202 + 174*w^201
+ 696*w^200 + 883*w^199 + 910*w^198 + 52*w^197 + 548*w^196 + 174*w^195 +
879*w^194 + 395*w^193 + 250*w^192 + 502*w^191 + 427*w^190 + 4*w^189 + 353*w^188
+ 270*w^187 + 724*w^186 + 140*w^185 + 366*w^184 + 1023*w^183 + 106*w^182 +
68*w^181 + 306*w^180 + 253*w^179 + 74*w^178 + 842*w^177 + 438*w^176 + 111*w^175
+ 744*w^174 + 264*w^173 + 899*w^172 + 108*w^171 + 843*w^170 + 530*w^169 +
657*w^168 + 439*w^167 + 295*w^166 + 442*w^165 + 396*w^164 + 765*w^163 +
587*w^162 + 491*w^161 + 763*w^160 + 412*w^159 + 190*w^158 + 790*w^157 +
139*w^156 + 665*w^155 + 761*w^154 + 357*w^153 + 427*w^152 + 447*w^151 +
450*w^150 + 692*w^149 + 760*w^148 + 175*w^147 + 983*w^146 + 279*w^145 +
462*w^144 + 398*w^143 + 590*w^142 + 945*w^141 + 1010*w^140 + 896*w^139 +
961*w^138 + 751*w^137 + 45*w^136 + 658*w^135 + 838*w^134 + 520*w^133 +
1023*w^132 + 274*w^131 + 131*w^130 + 255*w^129 + 570*w^128 + 133*w^127 +
1021*w^126 + 468*w^125 + 883*w^124 + 940*w^123 + 401*w^122 + 647*w^121 +
906*w^120 + 820*w^119 + 435*w^118 + 427*w^117 + 200*w^116 + 147*w^115 + 39*w^114
+ 238*w^113 + 881*w^112 + 648*w^111 + 518*w^110 + 9*w^109 + 429*w^108 +
810*w^107 + 611*w^106 + 905*w^105 + 301*w^104 + 525*w^103 + 938*w^102 +
818*w^101 + 88*w^100 + 935*w^99 + 211*w^98 + 489*w^97 + 702*w^96 + 799*w^95 +
898*w^94 + 298*w^93 + 556*w^92 + 711*w^91 + 535*w^90 + 287*w^89 + 674*w^88 +
689*w^87 + 366*w^86 + 1008*w^85 + 618*w^84 + 557*w^83 + 132*w^82 + 389*w^81 +
947*w^80 + 125*w^79 + 461*w^78 + 60*w^77 + 432*w^76 + 113*w^75 + 33*w^74 +
870*w^73 + 235*w^72 + 535*w^71 + 626*w^70 + 93*w^69 + 516*w^68 + 319*w^67 +
956*w^66 + 629*w^65 + 853*w^64 + 74*w^63 + 603*w^62 + 428*w^61 + 316*w^60 +
981*w^59 + 217*w^58 + 919*w^57 + 345*w^56 + 227*w^55 + 255*w^54 + 945*w^53 +
426*w^52 + 823*w^51 + 304*w^50 + 377*w^49 + 57*w^48 + 808*w^47 + 996*w^46 +
63*w^45 + 927*w^44 + 198*w^43 + 314*w^42 + 423*w^41 + 558*w^40 + 757*w^39 +
985*w^38 + 295*w^37 + 956*w^36 + 720*w^35 + 15*w^34 + 223*w^33 + 375*w^31 +
145*w^30 + 768*w^29 + 765*w^28 + 130*w^27 + 234*w^26 + 365*w^25 + 840*w^24 +

```
412*w^23 + 11*w^22 + 252*w^21 + 430*w^20 + 432*w^19 + 321*w^18 + 737*w^17 +
737*w^16 + 852*w^15 + 11*w^14 + 773*w^13 + 937*w^12 + 965*w^11 + 690*w^10 +
739*w^9 + 979*w^8 + 75*w^7 + 937*w^6 + 58*w^5 + 714*w^4 + 743*w^3 + 831*w^2 +
542*w + 929], 13*w^254 + 15*w^253 + 13*w^252 + 12*w^251 + 2*w^250 + 11*w^249 +
5*w^248 + 3*w^247 + 8*w^246 + 10*w^245 + 3*w^244 + 2*w^243 + 8*w^242 + 13*w^241
+ 9*w^240 + 15*w^239 + 10*w^238 + 2*w^237 + 4*w^236 + 12*w^235 + 6*w^234 +
11*w^233 + 11*w^232 + 14*w^231 + 15*w^230 + 12*w^229 + 4*w^228 + 9*w^227 +
8*w^226 + 9*w^225 + 6*w^224 + 6*w^223 + 3*w^222 + 2*w^221 + 10*w^220 + 8*w^218 +
w^217 + 7*w^216 + 8*w^215 + 15*w^213 + 7*w^212 + 3*w^211 + 14*w^210 + 4*w^209 +
12*w^207 + 15*w^206 + 2*w^205 + 2*w^204 + 11*w^203 + 2*w^202 + 6*w^201 + 7*w^200
+ 10*w^199 + 13*w^198 + 8*w^197 + 15*w^196 + 12*w^195 + 13*w^194 + 7*w^193 +
14*w^192 + w^191 + 12*w^190 + 8*w^189 + 9*w^188 + 3*w^187 + 15*w^186 + 3*w^185 +
4*w^184 + 7*w^182 + w^181 + 8*w^180 + 9*w^179 + 13*w^178 + 14*w^177 + 8*w^176 +
3*w^175 + w^174 + 2*w^173 + 11*w^172 + 4*w^171 + 14*w^170 + 6*w^169 + 15*w^168 +
10*w^167 + 9*w^166 + 5*w^165 + 4*w^164 + 12*w^163 + 10*w^162 + 2*w^160 + 6*w^159
+ 10*w^158 + 11*w^157 + 3*w^156 + 11*w^155 + 2*w^154 + 5*w^152 + 11*w^151 +
5*w^150 + 3*w^149 + 10*w^147 + 8*w^146 + 10*w^145 + 4*w^144 + 9*w^143 + 4*w^142
+ w^140 + 10*w^139 + 9*w^138 + 4*w^137 + w^136 + 8*w^134 + 9*w^133 + 4*w^132 +
13*w^131 + 12*w^130 + 3*w^129 + 11*w^128 + 12*w^127 + 6*w^126 + 10*w^125 +
5*w^124 + 14*w^123 + 7*w^122 + 7*w^121 + 7*w^120 + 3*w^119 + 3*w^118 + 6*w^117 +
7*w^116 + 4*w^115 + 11*w^114 + 9*w^113 + 8*w^112 + 4*w^111 + 2*w^110 + 11*w^109
+ 3*w^108 + w^107 + 15*w^106 + 8*w^105 + 15*w^104 + 5*w^103 + 10*w^102 +
12*w^101 + w^100 + 13*w^99 + 6*w^98 + 2*w^96 + 14*w^95 + w^94 + 2*w^93 + 10*w^92
+ 8*w^91 + 4*w^90 + 6*w^89 + 11*w^88 + 7*w^87 + 6*w^86 + 9*w^85 + 4*w^84 +
4*w^83 + 2*w^82 + 12*w^81 + 11*w^80 + 6*w^79 + 14*w^78 + 10*w^77 + 2*w^76 +
4*w^75 + 9*w^74 + 5*w^73 + 12*w^72 + 10*w^71 + 2*w^70 + 10*w^69 + 15*w^68 +
4*w^67 + 6*w^66 + 8*w^65 + 13*w^64 + 7*w^63 + 8*w^62 + 8*w^61 + 13*w^60 + 4*w^59
+ 8*w^58 + 3*w^57 + 10*w^56 + 4*w^55 + 11*w^54 + 8*w^53 + 3*w^52 + 8*w^51 +
9*w^50 + 6*w^48 + 12*w^47 + 10*w^46 + 14*w^45 + 4*w^44 + 9*w^43 + 15*w^42 + w^41
+ 11*w^40 + w^39 + 12*w^38 + 2*w^37 + 4*w^36 + 3*w^35 + 4*w^34 + 2*w^33 + 2*w^32
+ 15*w^31 + 8*w^30 + 14*w^28 + w^27 + w^26 + 5*w^25 + 12*w^23 + 9*w^22 + 12*w^21
+ 15*w^20 + 2*w^19 + 8*w^18 + 11*w^17 + 14*w^16 + 14*w^15 + 7*w^14 + 12*w^13 +
w^12 + 5*w^11 + 10*w^10 + 8*w^9 + 13*w^8 + 6*w^7 + 8*w^6 + 9*w^5 + 13*w^4 +
7*w^3 + 15*w^2 + 14*w + 2)

Chave secreta (partilhada/encapsulada):

b'\x1c+\xd9-
ZCp\xfa\x94\x05\xb5\xc3\xb5\xab\x9c\xecFT\xf5\xb4\xee\x19u8\x1d\x19\xf5Vo\r\\\xc
c'

A desencapsular a chave secreta...

A chave partilhada é:

b'\x1c+\xd9-
ZCp\xfa\x94\x05\xb5\xc3\xb5\xab\x9c\xecFT\xf5\xb4\xee\x19u8\x1d\x19\xf5Vo\r\\\xc
c'
```

```
A chave foi transmitida com sucesso!  b'\x1c+\xd9-
ZCp\xfa\x94\x05\xb5\xc3\xb5\xab\x9c\xecFT\xf5\xb4\xee\x19u8\x1d\x19\xf5Vo\r\\\xc
c'
```