IPSA 2025

DAKKAR Borhen-eddine

Lab 3: Cryptography algorithms

1. Introduction:

This laboratory aims to introduce the core principles, structures, and implementations of cryptography algorithms through practical experimentation. By implementing simplified educational algorithms—such as XOR-based encryption and toy Feistel constructions—you will gain an intuitive understanding of how cryptography transform and protect data. The lab also includes a hands-on introduction to real-world cryptographic primitives, in particular AES-GCM, which is widely deployed in secure communication protocols.

Through these exercises, you will deepen their understanding of key management, encryption modes, pseudocode design, Python implementation, and the security considerations necessary for robust cryptographic systems. This lab forms a foundational step toward mastering practical cybersecurity and modern cryptographic engineering.

2. Symetric cryptography algorithms:

This part interests to symmetric cryptography algorithms, including:

- XOR Stream Cipher (Educational Only)

- Feistel Block Cipher (Toy Example)

- AES-GCM (Real-World Secure Encryption)

2.1.XOR STREAM CIPHER

```
------------------------------------------------------------
 XOR STREAM CIPHER (EDUCATIONAL ONLY)
------------------------------------------------------------

Pseudocode:
------------
FUNCTION xor_encrypt(plaintext_bytes, key_bytes):
        ciphertext = empty list
        key_length = length(key_bytes)

        FOR i FROM 0 TO length(plaintext_bytes) - 1:
           k = key_bytes[i MOD key_length]
           c = plaintext_bytes[i] XOR k
           APPEND c TO ciphertext
        RETURN ciphertext

FUNCTION xor_decrypt(ciphertext_bytes, key_bytes):
RETURN xor_encrypt(ciphertext_bytes, key_bytes)
```

Write the python code?

```python
def xor_encrypt(plaintext_bytes: bytes, key_bytes: bytes) -> bytes:
    """
    Chiffre les données en utilisant l'opération XOR avec une clé.

    Args:
        plaintext_bytes: Les octets du texte en clair à chiffrer
        key_bytes: La clé de chiffrement (sera répétée si nécessaire)

    Returns:
        Les octets chiffrés (ciphertext)
    """
    ciphertext = []  # Liste vide pour stocker les octets chiffrés
    key_length = len(key_bytes)

    # Parcourir chaque octet du texte en clair
    for i in range(len(plaintext_bytes)):
        k = key_bytes[i % key_length]  # Clé cyclique (MOD key_length)
        c = plaintext_bytes[i] ^ k     # Opération XOR
        ciphertext.append(c)           # Ajouter à la liste

    return bytes(ciphertext)


def xor_decrypt(ciphertext_bytes: bytes, key_bytes: bytes) -> bytes:
    """
    Déchiffre les données en utilisant l'opération XOR avec une clé.

    Note: Le déchiffrement XOR est identique au chiffrement (propriété XOR).

    Args:
        ciphertext_bytes: Les octets chiffrés à déchiffrer
        key_bytes: La clé de déchiffrement (même clé que pour le chiffrement)

    Returns:
        Les octets déchiffrés (plaintext)
    """
    return xor_encrypt(ciphertext_bytes, key_bytes)
```

The code is inside cryptage_xor.py

2.2 FEISTEL BLOCK CIPHER

The Feistel Cipher is not a complete cipher itself, but a design model used to build many block ciphers, such as DES. It provides a simple way to build secure encryption and decryption algorithms. The only difference: during decryption, the round keys are applied in reverse order.

Key Features of Feistel Cipher

- Works on blocks of data (not on individual characters like substitution ciphers).
- The plaintext block is split into two halves: left (L) and right (R).
- In each round, one half is modified using a function of the other half and a round key.
- Multiple rounds make the cipher stronger.
- Same process is used for both encryption and decryption.

Step-by-Step Algorithm

Here's how a simple 2-round Feistel Cipher works:

1. **Convert Plaintext to Binary**
   - Take the plaintext characters.
   - Convert each character to its ASCII value and then into an 8-bit binary string.
2. **Divide into Two Halves**
   - Split the binary string into two equal halves:
   - Left half = L1
   - Right half = R1
3. **Generate Round Keys**
   - Create random binary keys K1 and K2.
   - Each key has a length equal to half the block size.

   Round 1 (Encryption)
   - Compute function f1:
     f1 = R1 XOR K1
   - Update halves:
     R2 = L1 XOR f1
     L2 = R1

   Round 2 (Encryption)
   - Compute function f2:
     f2 = R2 XOR K2
   - Update halves:
     R3 = L2 XOR f2
     L3 = R2

   Final Ciphertext

   Concatenate L3 and R3 to get the ciphertext.

Decryption Process

The Feistel Cipher uses the same algorithm for both encryption and decryption, with the round keys simply applied in reverse order.

Example:

Plaintext: Hello
- After encryption → Ciphertext: E1!w(
- After decryption → Retrieved Plaintext: Hello

Plaintext: Geeks
- After encryption → Ciphertext: O;Q
- After decryption → Retrieved Plaintext: Geeks

Why is Feistel Cipher Important?

- It provides a flexible framework to build strong ciphers.
- Many real-world ciphers (like DES) are based on this model.
- Ensures reversibility (you can always get back plaintext from ciphertext).
- Makes encryption/decryption more efficient and practical.

```
import random

# Function to generate a random binary key of given length
def rand_key(length):
    return "".join(str(random.randint(0, 1)) for _ in range(length))

# Function to perform XOR between two binary strings
def xor(a, b):
    return "".join("0" if a[i] == b[i] else "1" for i in range(len(a)))

# Feistel round function
def feistel_round(left, right, key):
    # Round function: f = XOR(right, key)
    f = xor(right, key)
    new_left = right
    new_right = xor(left, f)
    return new_left, new_right

# Encryption function
def feistel_encrypt(plain_text, rounds=2):
    # Convert plaintext to binary (8 bits for each character)
    pt_bin = "".join(format(ord(c), "08b") for c in plain_text)

    # Split into two halves
    n = len(pt_bin) // 2
    left, right = pt_bin[:n], pt_bin[n:]

    # Generate random round keys
    keys = [rand_key(len(right)) for _ in range(rounds)]

    # Apply Feistel rounds
    for i in range(rounds):
        left, right = feistel_round(left, right, keys[i])

    # Final ciphertext in binary
    cipher_bin = left + right
```

```
    # Convert binary to string
    cipher_text = ""
    for i in range(0, len(cipher_bin), 8):
        byte = cipher_bin[i:i+8]
        cipher_text += chr(int(byte, 2))

    return cipher_text, keys

# Decryption function
def feistel_decrypt(cipher_text, keys):
    # Convert ciphertext to binary
    ct_bin = "".join(format(ord(c), "08b") for c in cipher_text)

    # Split into two halves
    n = len(ct_bin) // 2
    left, right = ct_bin[:n], ct_bin[n:]

    # Apply Feistel rounds in reverse
    for i in reversed(range(len(keys))):
        right, left = feistel_round(right, left, keys[i])

    # Final plaintext in binary
    plain_bin = left + right

    # Convert binary back to string
    plain_text = ""
    for i in range(0, len(plain_bin), 8):
        byte = plain_bin[i:i+8]
        plain_text += chr(int(byte, 2))

    return plain_text

# Driver Code
if __name__ == "__main__":
    plaintext = "Hello"
    print("Plain Text:", plaintext)

    # Encryption
    cipher, round_keys = feistel_encrypt(plaintext)
    print("Cipher Text:", cipher)

    # Decryption
    recovered = feistel_decrypt(cipher, round_keys)
    print("Decrypted Text:", recovered)
```

Write the python code?

```python
import random

def text_to_binary(text):
    """Convertit un texte en chaîne binaire (8 bits par caractère)."""
    binary = ""
    for char in text:
        # Convertir chaque caractère en valeur ASCII puis en binaire sur 8 bits
        binary += format(ord(char), '08b')
    return binary

def binary_to_text(binary):
    """Convertit une chaîne binaire en texte."""
    text = ""
    # Traiter par blocs de 8 bits
    for i in range(0, len(binary), 8):
        byte = binary[i:i+8]
        if len(byte) == 8:
            text += chr(int(byte, 2))
    return text

def xor(a, b):
    """Effectue un XOR entre deux chaînes binaires de même longueur."""
    result = ""
    for i in range(len(a)):
        # XOR: 0^0=0, 0^1=1, 1^0=1, 1^1=0
        result += '1' if a[i] != b[i] else '0'
    return result

def generate_key(length):
    """Génère une clé binaire aléatoire de la longueur spécifiée."""
    key = ""
    for _ in range(length):
        key += str(random.randint(0, 1))
    return key
```

```python
def feistel_encrypt(plaintext, key1, key2):
    """
    Chiffrement Feistel à 2 tours.

    Étapes:
    1. Convertir le texte en binaire
    2. Diviser en deux moitiés (L0 et R0)
    3. Tour 1: f1 = R0 XOR K1, R1 = L0 XOR f1, L1 = R0
    4. Tour 2: f2 = R1 XOR K2, R2 = L1 XOR f2, L2 = R1
    5. Texte chiffré = L2 || R2
    """
    # Étape 1: Convertir le texte clair en binaire
    binary = text_to_binary(plaintext)

    # S'assurer que la longueur est paire (ajouter un padding si nécessaire)
    if len(binary) % 2 != 0:
        binary += '0'

    # Étape 2: Diviser en deux moitiés
    mid = len(binary) // 2
    L0 = binary[:mid]  # Moitié gauche
    R0 = binary[mid:]  # Moitié droite

    # Ajuster les clés à la taille des moitiés
    half_size = len(L0)
    K1 = (key1 * ((half_size // len(key1)) + 1))[:half_size]
    K2 = (key2 * ((half_size // len(key2)) + 1))[:half_size]

    # Tour 1 (Encryption)
    f1 = xor(R0, K1)      # Calculer f1 = R0 XOR K1
    R1 = xor(L0, f1)      # Nouvelle moitié droite: R1 = L0 XOR f1
    L1 = R0               # Nouvelle moitié gauche: L1 = R0

    # Tour 2 (Encryption)
    f2 = xor(R1, K2)      # Calculer f2 = R1 XOR K2
    R2 = xor(L1, f2)      # Nouvelle moitié droite: R2 = L1 XOR f2
    L2 = R1               # Nouvelle moitié gauche: L2 = R1

    # Texte chiffré final: concaténer L2 et R2
    ciphertext_binary = L2 + R2

    return binary_to_text(ciphertext_binary)
```

```python
def feistel_decrypt(ciphertext, key1, key2):
    """
    Déchiffrement Feistel à 2 tours.

    Le déchiffrement utilise le même algorithme que le chiffrement,
    mais avec les clés appliquées dans l'ordre inverse (K2 puis K1).
    """
    # Convertir le texte chiffré en binaire
    binary = text_to_binary(ciphertext)

    # Diviser en deux moitiés
    mid = len(binary) // 2
    L2 = binary[:mid]
    R2 = binary[mid:]

    # Ajuster les clés à la taille des moitiés
    half_size = len(L2)
    K1 = (key1 * ((half_size // len(key1)) + 1))[:half_size]
    K2 = (key2 * ((half_size // len(key2)) + 1))[:half_size]

    # Tour inverse 2 (avec K2) - inverse du Tour 2
    R1 = L2                 # R1 = L2
    f2 = xor(R1, K2)        # Recalculer f2 = R1 XOR K2
    L1 = xor(R2, f2)        # L1 = R2 XOR f2

    # Tour inverse 1 (avec K1) - inverse du Tour 1
    R0 = L1                 # R0 = L1
    f1 = xor(R0, K1)        # Recalculer f1 = R0 XOR K1
    L0 = xor(R1, f1)        # L0 = R1 XOR f1

    # Reconstituer le texte clair
    plaintext_binary = L0 + R0

    return binary_to_text(plaintext_binary)
```

The code is inside feistel_block_cypher_cryptage.py

2.3 AES-GCM (Secure real-world cryptography)

Using the **cryptography** library.

```
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
```

```
def aes_gcm_encrypt(key: bytes, plaintext: bytes, associated_data: bytes =
b""):
    """
    Encrypt with AES-GCM.
    key: 16, 24, or 32 bytes (AES-128/192/256)
    returns (nonce, ciphertext)
    """
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)  # 96-bit nonce
    ciphertext = aesgcm.encrypt(nonce, plaintext, associated_data)
    return nonce, ciphertext

def aes_gcm_decrypt(key: bytes, nonce: bytes, ciphertext: bytes,
associated_data: bytes = b""):
    """
    Decrypt with AES-GCM.
    Raises an exception if authentication fails.
    """
    aesgcm = AESGCM(key)
    plaintext = aesgcm.decrypt(nonce, ciphertext, associated_data)
    return plaintext


# Example usage
if __name__ == "__main__":
    key = AESGCM.generate_key(bit_length=256)  # 32-byte key
    plaintext = b"Symmetric cryptography example"
    aad = b"header or metadata"

    nonce, ciphertext = aes_gcm_encrypt(key, plaintext, aad)
    print("Ciphertext (hex):", ciphertext.hex())

    recovered = aes_gcm_decrypt(key, nonce, ciphertext, aad)
    print("Recovered:", recovered)
```

Test the above code.

```
Ciphertext (hex): 2d477c276841347df54671360eff62a99020b149f58aca90520519e63bf0d3eb537899b97d9160a9b006bf3ef178
Recovered: Symmetric cryptography example
```

3. Asymetric cryptography algorithms:

Asymmetric encryption, also called **public-key cryptography**, is a cryptographic system that uses **two different keys**:

- a **public key**, which can be shared openly and is used for **encryption**, and
- a **private key**, which must be kept secret and is used for **decryption**.

Because the keys are mathematically related but not identical, a message encrypted with the public key **can only be decrypted** with the corresponding private key.

3.1. Simple Shift Cipher (Caesar Cipher)

```
FUNCTION encrypt(text, key):
   result = empty string

   FOR each character c in text:
      IF c is a letter:
         shift = (ASCII(c) - ASCII('a') + key) MOD 26
         new_char = ASCII('a') + shift
         APPEND new_char to result
      ELSE:
         APPEND c to result   # keep non-letters the same

   RETURN result


FUNCTION decrypt(text, key):
   RETURN encrypt(text, -key)
```

Write the code on python?

```python
import random

def cesar_encrypt(plaintext, shift):
    """
    Chiffre un texte avec le chiffrement de César.

    Le chiffrement de César décale chaque lettre de l'alphabet
    d'un nombre fixe de positions (la clé/shift).

    Paramètres:
        plaintext (str): Le texte clair à chiffrer
        shift (int): Le décalage (clé de chiffrement), entre 0 et 25

    Retourne:
        str: Le texte chiffré
    """
    ciphertext = ""

    for char in plaintext:
        if char.isalpha():
            # Déterminer si c'est une majuscule ou minuscule
            if char.isupper():
                # Décaler dans l'alphabet majuscule (A=65 à Z=90)
                # Formule: (position + shift) mod 26
                new_char = chr((ord(char) - ord('A') + shift) % 26 + ord('A'))
            else:
                # Décaler dans l'alphabet minuscule (a=97 à z=122)
                new_char = chr((ord(char) - ord('a') + shift) % 26 + ord('a'))
            ciphertext += new_char
        else:
            # Garder les caractères non alphabétiques inchangés
            ciphertext += char

    return ciphertext
def cesar_decrypt(ciphertext, shift):
    """
    Déchiffre un texte chiffré avec le chiffrement de César.

    Le déchiffrement est simplement le chiffrement avec un décalage
    négatif (ou équivalent: 26 - shift).

    Paramètres:
        ciphertext (str): Le texte chiffré à déchiffrer
        shift (int): Le décalage utilisé lors du chiffrement

    Retourne:
        str: Le texte clair retrouvé
    """
    # Déchiffrer = chiffrer avec le décalage inverse
    return cesar_encrypt(ciphertext, -shift)
```

The code is inside cesar_cypher.py

3.2. Substitution Cipher (Simple Mapping Cipher)

Instead of shifting letters, this algorithm **maps each letter to another letter** using a substitution table.

a → q
b → w
c → e
d → r
...

```
FUNCTION encrypt(text, mapping):
        result = empty string

        FOR each character c in text:
            IF c in mapping:
                APPEND mapping[c] to result
            ELSE:
                APPEND c to result   # keep spaces and punctuation

        RETURN result


FUNCTION decrypt(text, reverse_mapping):
        result = empty string

        FOR each character c in text:
            IF c in reverse_mapping:
                APPEND reverse_mapping[c] to result
            ELSE:
                APPEND c to result

          RETURN result
```

Write the code on python?

```python
import random
import string

def generate_random_mapping():
    """
    Génère une table de substitution aléatoire.
    Chaque lettre est mappée vers une autre lettre unique.

    Retourne:
        dict: Dictionnaire de mapping pour le chiffrement
    """
    alphabet = list(string.ascii_lowercase)
    shuffled = alphabet.copy()
    random.shuffle(shuffled)

    mapping = {}
    for i, letter in enumerate(alphabet):
        mapping[letter] = shuffled[i]
        # Ajouter aussi les majuscules
        mapping[letter.upper()] = shuffled[i].upper()

    return mapping

def get_reverse_mapping(mapping):
    """
    Crée le mapping inverse pour le déchiffrement.

    Paramètres:
        mapping (dict): Le mapping utilisé pour le chiffrement

    Retourne:
        dict: Le mapping inverse
    """
    reverse_mapping = {}
    for key, value in mapping.items():
        reverse_mapping[value] = key
    return reverse_mapping
```

```python
def encrypt(text, mapping):
    """
    Chiffre un texte avec le chiffre de substitution.

    Paramètres:
        text (str): Le texte clair à chiffrer
        mapping (dict): La table de substitution

    Retourne:
        str: Le texte chiffré
    """
    result = ""

    for c in text:
        if c in mapping:
            # Substituer le caractère par son équivalent dans le mapping
            result += mapping[c]
        else:
            # Garder les espaces et la ponctuation inchangés
            result += c

    return result

def decrypt(text, reverse_mapping):
    """
    Déchiffre un texte avec le chiffre de substitution.

    Paramètres:
        text (str): Le texte chiffré à déchiffrer
        reverse_mapping (dict): La table de substitution inverse

    Retourne:
        str: Le texte clair retrouvé
    """
    result = ""

    for c in text:
        if c in reverse_mapping:
            # Substituer le caractère par son équivalent dans le mapping inverse
            result += reverse_mapping[c]
        else:
            # Garder les espaces et la ponctuation inchangés
            result += c

    return result
```

The code is inside substitution_cypher.py

RSA (Rivest–Shamir–Adleman): https://en.wikipedia.org/wiki/RSA_cryptosystem

```
FUNCTION RSA_KeyGen(keysize):
   1. Choose two large distinct prime numbers:
      p ← random_prime(keysize / 2)
      q ← random_prime(keysize / 2)

   2. Compute modulus:
      n ← p * q

   3. Compute Euler's totient:
      ϕ ← (p - 1) * (q - 1)

   4. Choose public exponent e:
      Choose e such that 1 < e < ϕ  AND  gcd(e, ϕ) = 1
      (Common choice: e = 65537 if gcd(65537, ϕ) = 1)

   5. Compute private exponent d:
      d ← modular_inverse(e, ϕ)
      (i.e., d such that (d * e) mod ϕ = 1)

   6. Public key:
      PK ← (n, e)

   7. Private key:
      SK ← (n, d)

   8. RETURN (PK, SK)
END FUNCTION
```

- **RSA Encryption**

```
FUNCTION RSA_Encrypt(m, PK):
   INPUT:  message m   (as integer, 0 ≤ m < n)
         public key PK = (n, e)

   1. Compute ciphertext:
      c ← m^e mod n

   2. RETURN c
END FUNCTION
```

- **RSA Decryption**

```
FUNCTION RSA_Decrypt(c, SK):
   INPUT:  ciphertext c (integer)
         private key SK = (n, d)

   1. Compute plaintext:
      m ← c^d mod n

   2. RETURN m
END FUNCTION
```

```python
import random
import math


def is_prime(n: int, k: int = 10) -> bool:
    """
    Teste si un nombre est premier en utilisant le test de Miller-Rabin.

    Args:
        n: Le nombre à tester
        k: Le nombre d'itérations (plus k est grand, plus le test est précis)

    Returns:
        True si n est probablement premier, False sinon
    """
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False

    # Écrire n-1 comme 2^r * d
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Test de Miller-Rabin
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, d, n)

        if x == 1 or x == n - 1:
            continue

        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False

    return True
```

```python
def random_prime(bits: int) -> int:
    """
    Génère un nombre premier aléatoire de la taille spécifiée en bits.

    Args:
        bits: Le nombre de bits souhaité pour le nombre premier

    Returns:
        Un nombre premier aléatoire
    """
    while True:
        # Générer un nombre aléatoire de 'bits' bits
        n = random.getrandbits(bits)
        # S'assurer que le bit de poids fort est à 1 (pour avoir exactement 'bits' bits)
        n |= (1 << (bits - 1))
        # S'assurer que le nombre est impair
        n |= 1

        if is_prime(n):
            return n


def gcd(a: int, b: int) -> int:
    """
    Calcule le PGCD (Plus Grand Commun Diviseur) de deux nombres.
    Utilise l'algorithme d'Euclide.

    Args:
        a: Premier nombre
        b: Deuxième nombre

    Returns:
        Le PGCD de a et b
    """
    while b:
        a, b = b, a % b
    return a
```

```python
def modular_inverse(e: int, phi: int) -> int:
    """
    Calcule l'inverse modulaire de e modulo phi.
    Utilise l'algorithme d'Euclide étendu.
    Trouve d tel que (d * e) mod phi = 1

    Args:
        e: Le nombre dont on cherche l'inverse
        phi: Le module

    Returns:
        L'inverse modulaire de e modulo phi

    Raises:
        ValueError: Si l'inverse modulaire n'existe pas
    """
    # Algorithme d'Euclide étendu
    original_phi = phi
    x0, x1 = 0, 1

    if phi == 1:
        return 0

    while e > 1:
        q = e // phi
        phi, e = e % phi, phi
        x0, x1 = x1 - q * x0, x0

    if x1 < 0:
        x1 += original_phi

    return x1
```

```python
def rsa_keygen(keysize: int = 1024) -> tuple:
    4. Choisir e tel que 1 < e < ϕ(n) et pgcd(e, ϕ(n)) = 1
    5. Calculer d = inverse modulaire de e modulo ϕ(n)
    6. Clé publique = (n, e), Clé privée = (n, d)

    Args:
        keysize: La taille de la clé en bits (par défaut 1024)

    Returns:
        Un tuple (clé_publique, clé_privée) où:
        - clé_publique = (n, e)
        - clé_privée = (n, d)
    """
    # 1. Choisir deux grands nombres premiers distincts
    p = random_prime(keysize // 2)
    q = random_prime(keysize // 2)

    # S'assurer que p et q sont différents
    while p == q:
        q = random_prime(keysize // 2)

    # 2. Calculer le module n
    n = p * q

    # 3. Calculer l'indicatrice d'Euler ϕ(n) = (p-1) * (q-1)
    phi = (p - 1) * (q - 1)

    # 4. Choisir l'exposant public e
    # Valeur commune : 65537 (0x10001) si pgcd(e, ϕ) = 1
    e = 65537
    if gcd(e, phi) != 1:
        # Si 65537 ne convient pas, chercher un autre e
        e = 3
        while gcd(e, phi) != 1:
            e += 2

    # 5. Calculer l'exposant privé d (inverse modulaire de e modulo ϕ)
    d = modular_inverse(e, phi)

    # 6. Construire les clés
    public_key = (n, e)   # PK = (n, e)
    private_key = (n, d)  # SK = (n, d)

    return public_key, private_key
```

```python
def rsa_encrypt(m: int, public_key: tuple) -> int:
    """
    Chiffre un message avec la clé publique RSA.

    Formule: c = m^e mod n

    Args:
        m: Le message à chiffrer (entier, 0 ≤ m < n)
        public_key: La clé publique (n, e)

    Returns:
        Le texte chiffré c (entier)
    """
    n, e = public_key

    # Vérifier que le message est valide (0 ≤ m < n)
    if m < 0 or m >= n:
        raise ValueError(f"Le message doit être un entier entre 0 et {n-1}")

    # Calculer c = m^e mod n
    c = pow(m, e, n)

    return c


def rsa_decrypt(c: int, private_key: tuple) -> int:
    """
    Déchiffre un texte chiffré avec la clé privée RSA.

    Formule: m = c^d mod n

    Args:
        c: Le texte chiffré (entier)
        private_key: La clé privée (n, d)

    Returns:
        Le message déchiffré m (entier)
    """
    n, d = private_key

    # Calculer m = c^d mod n
    m = pow(c, d, n)

    return m
```

```python
def text_to_int(text: str) -> int:
    """
        text: La chaîne à convertir

    Returns:
        L'entier correspondant
    """
    return int.from_bytes(text.encode('utf-8'), 'big')


def int_to_text(number: int) -> str:
    """
    Convertit un entier en chaîne de caractères.

    Args:
        number: L'entier à convertir

    Returns:
        La chaîne de caractères correspondante
    """
    # Calculer le nombre d'octets nécessaires
    byte_length = (number.bit_length() + 7) // 8
    if byte_length == 0:
        byte_length = 1
    return number.to_bytes(byte_length, 'big').decode('utf-8')


def rsa_encrypt_text(text: str, public_key: tuple) -> int:
    """
    Chiffre un message texte avec RSA.

    Args:
        text: Le message texte à chiffrer
        public_key: La clé publique (n, e)

    Returns:
        Le texte chiffré (entier)
    """
    m = text_to_int(text)
    n, _ = public_key

    if m >= n:
        raise ValueError("Le message est trop long pour cette clé. Utilisez une clé plus grande.")

    return rsa_encrypt(m, public_key)
```

Code is inside RSA.py

Code TP3 : https://github.com/ArthurTouati/IN517_TP3-ArthurTouati