	**Architecture matérielle**	**NSI T^ale^**
	Chiffrement	TP

*

1. Objectifs

- Implémenter en Python un exemple d'algorithme de chiffrement symétrique (**XOR**)
- Implémenter en Python un exemple d'algorithme de chiffrement asymétrique (**KidRSA**)
- Appréhender la "sûreté" d'un algorithme de chiffrement et implémenter en Python des méthodes (en force brute ou plus habile) pour décrypter (casser) un message chiffré.

2. Consignes

- Vous devez répondre aux questions indiquées en couleur bleue.
- Vous devez déposer un fichier nommé **NOM_TP_cryptographie.py** avec NOM votre nom, qui contiendra toutes les fonctions Python demandées dans le TP.

Les fonctions Python

Les fonctions Python présentes dans le fichier sont :

```
convertit_texte_en_binaire(texte)
convertit_binaire_vers_entier_base_10(chaine_binaire)
convertit_binaire_en_texte(chaine_binaire)
chiffre_xor(chaine_binaire,clef_binaire)
convertit_binaire_vers_decimal(octet)
genere_clefs_publique_et_privee(a1,b1,a2,b2)
chiffre_message(m,clef)
bruteForceKidRSA( e, n)
egcd(a, b)
modinv(e, n)
```

Chaque fonction doit être documentée en utilisant les **docstring** et incluant plusieurs jeux de tests utilisant la bibliothèque **doctest**.

3. Un exemple de chiffrement symétrique : XOR

Le **XOR** (OU exclusif) est très utilisé dans les protocoles de chiffrement symétrique. En effet, c'est une opération qui est sa propre réciproque, ce qui n'est pas le cas du **ET** ni du **OU**.

C'est à dire que : `XOR(XOR(A,B),B) = A` (1)

Q1 : Démontrer la propriété (1) à l'aide de tables de vérité.

Nous allons programmer un protocole de chiffrement symétrique utilisant le **XOR**.

On souhaite chiffrer un message (par exemple une chaîne de caractères) à l'aide d'une clé, qui peut aussi être une chaîne de caractères.
Pour simplifier la programmation, on supposera que seuls des caractères ASCII sont utilisés (pas d'utf-8 ou plus généralement d'Unicode).

Q2 : Écrire le corps de la fonction `convertit_texte_en_binaire(texte)` qui convertit la chaîne de caractères ASCII `texte` passée en paramètre en une chaîne binaire et retourne cette chaîne binaire. Chaque caractère sera représenté par son code ASCII en binaire sur un octet.

Exemple :
`convertit_texte_en_binaire("NSI")` doit retourner la chaîne :
`'010011100101001101001001'`

En effet :

- Le code ASCII de `"N"` est `78` en décimal = `01001110` en binaire sur un octet
- Le code ASCII de `"S"` est `83` en décimal = `01010011` en binaire sur un octet
- Le code ASCII de `"I"` est `73` en décimal = `01001001` en binaire sur un octet

Et, `'01001110' + '01010011' + '01001001' = '010011100101001101001001'`

Q3 : Écrire le corps de la fonction `convertit_binaire_vers_entier_base_10(chaine_binaire)` qui convertit la chaîne binaire `chaine_binaire` passée en paramètre en le nombre décimal correspondant et retourne ce nombre décimal.

Exemple :
`convertit_binaire_vers_entier_base_10("01001110")` doit retourner l'entier `78`

En effet : `01001110` en base 2 = 0x1 + 1x2 + 1x4 + 1x8 + 0x16 + 0x32 + 1x64 + 0x128 = 2 + 4 + 8 + 64 = `78`

Q4 : Écrire le corps de la fonction `convertit_binaire_en_texte(chaine_binaire)` qui convertit la chaîne `chaine_binaire` passée en paramètre composée d'octets binaires représentant des caractères codés en ASCII en une chaîne de caractères et retourne cette chaîne de caractères.

Exemple :
`convertit_binaire_en_texte("010011100101001101001001")` doit retourner la chaîne : `'NSI'`.

En effet :
`'010011100101001101001001' = '01001110' + '01010011' + '01001001'`

- L'octet binaire `01001110` correspond au nombre décimal `78` qui représente en ASCII le caractère `"N"`
- L'octet binaire `01010011` correspond au nombre décimal `83` qui représente en ASCII le caractère `"S"`
- L'octet binaire `01001001` correspond au nombre décimal `73` qui représente en ASCII le caractère `"I"`

La chaîne retournée est donc bien `"NSI"`.

Q5 : Écrire le corps de la fonction `chiffre_xor(chaine_binaire,clef_binaire)` qui chiffre la chaîne binaire `chaine_binaire` passée en paramètre avec la clef binaire `clef_binaire` passée en paramètre en effectuant l'opération XOR bit à bit entre la chaîne binaire et la clef binaire (répétée). La fonction doit retourner la chaîne ainsi chiffrée.

Exemple :
Chiffrons la chaîne `"SPECIALITE NSI"` avec la clef `"TERM"`.

```
convertit_texte_en_binaire("SPECIALITE NSI")
```

Retourne
`'010100110101000001000101010000110100100101000001010011000100100101010100010000101001000000100111001010011010010001'.`

```
convertit_texte_en_binaire("TERM")
```

Retourne `'01010100010001010101001001001101'`.

L'application du XOR entre
`'010100110101000001000101010000110100100101000001010011000100100101010100010000101001000000100111001010011010010001'`
et la clef binaire répétée `'01010100010001010101001001001101'` donne :
`01010011010100000100010101000011010010010100000101001100010010010101010001000101001000000100111001010011010010001`
XOR
`01010100010001010101001001001101010101010001000101010100100100110101010100010001010101001000100111001010101000100010100`
=
`00000111000101010001011100001110000111010000010000011110000001000000000000000000011000100000001100000111000011100`

Donc
`chiffre_xor('010100110101000001000101010000110100100101000001010011000100100101010100010000101001000000100111001010011010010001','01010100010001010101001001001101')`
doit retourner
`'00000111000101010001011100001110000111010000010000011110000001000000000000000001110010000000110000011100001100'`.

Q6 : Montrer sur un exemple (message et clef de chiffrement de votre choix), que si on chiffre le message avec la clef et qu'on chiffre le message chiffré avec la même clef, on retrouve le message de départ en clair.

Par exemple :

```
message = "SPECIALITE NSI"
clef = "TERM"
message_binaire = convertit_texte_en_binaire(message)
print(message,"en binaire",message_binaire)
clef_binaire = convertit_texte_en_binaire(clef)
print(clef,"en binaire",clef_binaire)
message_binaire_chiffre = chiffre_xor(message_binaire,clef_binaire)
print("message chiffré",message_binaire_chiffre)
message_binaire_dechiffre = chiffre_xor(message_binaire_chiffre,clef_bi-
naire)
print("message_binaire_dechiffre",message_binaire_dechiffre)
print("message dechiffré en ASCII",convertit_binaire_en_texte(message_bi-
naire_dechiffre))
```

conduit à l'affichage suivant :

```
SPECIALITE NSI en binaire
01010011010100000100010101000011010010010100000101001100010010010101010001000101001000000100111001010011010010001
TERM en binaire 01010100010001010101001001001101
message chiffré
00000111000101010001011100001110000111010000010000011110000001000000000000000000011100100000001100001110000001100
message_binaire_dechiffre
01010011010100000100010101000011010010010100000101001100010010010101010001000101001000000100111001010011010010001
message dechiffré en ASCII SPECIALITE NSI
```

4. Un exemple de chiffrement asymétrique : KidRSA

Un algorithme de chiffrement asymétrique très utilisé se nomme **RSA**.
Dans ce TP, nous allons utiliser une version plus simple nommée **KidRSA** : le **RSA** pour les "enfants".

4.1. Principe de l'algorithme KidRSA

- Choisir 4 entiers : a_1, b_1, a_2, b_2.
- On calcule les entiers suivants :
  - $M = a_1 \times b_1 - 1$
  - $e = a_2 \times M + a_1$
  - $d = b_2 \times M + b_1$
  - $n = (e \times d - 1) / M$
- La **clef publique** est **(e, n)**
- La **clef secrète** est **(d, n)**

- Pour chiffrer un message représenté par un entier `m` plus petit que `n`, on effectue l'opération `e x m (modulo n)`.
- Pour déchiffrer un message représenté par un entier `m` plus petit que `n`, on effectue l'opération `d x m (modulo n)`.

4.2. Un exemple (avec des petits nombres)

Prenons $a_1 = 5$, $b_1 = 3$, $a_2 = 7$ et $b_2 = 5$

Q7 :

- **Calculer M, e, d et n.**
- **En déduire la clef publique et la clef secrète.**
- **Donner le message chiffré par la clef publique représenté par le code ASCII de la lettre 'a' (en minuscule).**
- **Montrer que si on chiffre le message avec la clef secrète on retrouve bien le code ASCII de la lettre 'a'.**

4.3. Une implémentation en Python du chiffrement avec l'algorithme KidRSA

Q8 : Implémenter les fonctions Python suivantes :

- **`genere_clefs_publique_et_privee(a1,b1,a2,b2)` : génère et retourne la clef publique `(n,e)` et la clef secrète `(n,d)` à partir des 4 entiers passés en paramètre a_1, b_1, a_2, b_2**
- **`chiffre_message(m,clef)` : chiffre un message m qui est une chaîne de caractères avec la clef, en remplaçant chaque caractère par son code ASCII en décimal.**
  Le message chiffré retourné est une liste de nombres. La taille de la liste étant égale à la longueur de la chaîne de caractères `m`.
- **`dechiffre_message(m,clef)` : déchiffre un message m qui est une liste de nombres et renvoie le message déchiffré sous la forme d'une chaîne de caractères.**

4.4. Tests de cette implémentation en Python de KidRSA

On choisit comme valeurs $a_1 = 13, b_1 = 32, a_2 = 69$ et $b_2 = 35$

Q9 : Donner les informations suivantes à partir des fonctions Python codées pour l'algorithme KidRSA :

- **Donner les valeurs des clefs publique et secrète.**
- **Donner la liste des 3 nombres pour le chiffrement de la chaîne de caractères "NSI" avec la clef publique.**
- **Montrer que si on déchiffre avec la clef secrète le message NSI préalablement chiffré avec la clef publique, on retrouve bien la chaîne "NSI".**

4.5. Casser (ou décrypter) le chiffrement KidRSA

Un message a été chiffré avec **KidRSA** à l'aide de la clef publique suivante
(e,n) = (53447,5185112).

Voici le message chiffré obtenu :

```
[3580949, 2084433, 3687843, 4436101, 4489548, 1710304, 4329207, 4542995,
3901631, 1710304, 4061972, 3687843, 1710304, 3527502, 4222313, 4436101,
4436101, 1710304, 3687843, 4168866, 1710304, 4168866, 4436101, 3901631,
1710304, 3367161]
```

On souhaite "casser" le message chiffré et retrouver le message en clair. Pour cela, on a besoin de connaître la clef secrète **(n,d)**. Comme on connaît déjà **n** (**n=5185112**), il faut trouver une méthode pour calculer **d**.

Attaque de KidRSA par force brute

En étudiant la relation entre les nombres qui constituent les clefs publique et privées **e**, **d** et **n** : **n = (e × d − 1) / M**, qui peut s'écrire aussi : **e × d − 1 = n × M**.

On en déduit que **e × d − 1** est divisible par **n**.

Pour trouver l'entier **d** qui fait partie de la clef secrète, comme on connaît déjà **n** et **e**, il suffit d'étudier parmi toutes les valeurs de **d** comprises entre **1** et **n−1**, laquelle vérifie la condition "**e × d − 1** est divisible par **n**."

On appelle ce type d'attaque par force brute car on doit étudier (dans le pire des cas) tous les entiers **d** inférieurs à **n**. Ce qui peut être long si **n** est grand.

Q10 : Écrire le corps de la fonction `bruteForceKidRSA(e,n)` qui permet de calculer et de retourner le premier entier `d` inférieur à `n` qui vérifie la relation "`e×d−1` est divisible par `n`."

Q11 : En déduire la valeur de `d` et obtenir ainsi le déchiffrement du message chiffré donné plus haut.

Attaque de KidRSA plus subtile (à l'aide de l'algorithme d'Euclide étendu)

La réponse au message chiffré précédent est aussi chiffrée avec **KidRSA** mais avec une clef publique de longueur beaucoup plus grande.

Voici la clef publique utilisée : **(e,n) = (230884490440319,194326240259798261076)**.

Et voici le message chiffré obtenu avec la clef publique :

[16623683311702968, 19625181687427115, 16392798821262649, 16392798821262649, 20548719649188391, 7388303694090208, 17547221273464244, 15931029840382011, 19163412706546477, 7388303694090208, 15238376369061054, 18239874744785201, 18008990254344882, 19163412706546477, 7388303694090208, 19394297196986796, 19625181687427115, 20548719649188391, 15007491878620735, 19625181687427115, 20317835158748072, 7388303694090208, 7619188184530527]

Q12 : Essayer de retrouver la clef secrète à l'aide de la fonction `bruteForceKidRSA(e,n)`. Quel est le problème rencontré ?

Il faut résoudre l'équation $e \times d \equiv 1 \pmod{n}$ avec une méthode plus efficace que par force brute. Cela revient à trouver l'inverse entier de e modulo n.

Une méthode mathématique qui permet de résoudre ce problème se nomme l'**algorithme d'Euclide étendu**.

Exemple d'utilisation de l'algorithme d'Euclide étendu

L'algorithme d'Euclide étendu est basé sur une suite de divisions euclidiennes que nous ne détaillerons pas ici. Dans la partie arithmétique du programme de mathématiques expertes, il est possible que cet algorithme soit étudié.

On peut retenir que pour rechercher la clef secrète associée à une clef publique, l'algorithme d'Euclide étendu est beaucoup plus efficace que l'attaque par force brute. La complexité algorithmique par force brute est dans le pire des cas en $O(n)$ alors que celle avec l'algorithme d'Euclide étendu est en $O(\log(n))$.

Voici deux fonctions données en pseudo-code qui permettent de trouver l'inverse d'un entier modulo un autre entier.

```
fonction modinv(e,n)
    g, x, y = egcd(e, n)
    si g != 1 alors
        retourner Faux
    Sinon
        retourner x % n
```

```
fonction egcd(a,b)
    Si a = 0 alors
        retourner (b, 0, 1)
    Sinon
        g, y, x = egcd(b % a, a)
        retourner (g, x - (b//a)*y, y)
```

Q13 : Coder ces deux fonctions en Python. En déduire la clef privée secrète associée à la clef publique : (19432624025979826176, 230884490440319). En déduire le décodage du deuxième message chiffré avec cette clef publique.

5. *Une étude de la vulnérabilité de l'algorithme RSA*

Nous venons de vérifier que l'algorithme **KidRSa** pouvait être facilement "cassé" même avec des clefs assez grande à l'aide de l'algorithme d'Euclide étendu.

Heureusement, le véritable algorithme **RSA** utilisé par **HTTPS** sur internet est bien plus robuste.

Q14 : Donner la taille des clefs couramment utilisées par RSA pour sécuriser des données sur Internet. Donner aussi quelle nouvelle technologie pourrait permettre de casser RSA en quelques secondes.

En complément, un bel article de CultureMath Voyage au cœur de la cryptographie en rapport avec le programme de Mathématiques Expertes.

Auteur : Hugues Malherbe (basé sur un document de Frédéric Mandon et Romain Mallet)