ICC/Programmation — Semaine 13 : Exercices

18 mai 2026

Objectif. Dans ces exercices, vous pratiquerez les fondamentaux de la cryptographie introduits lors du cours d'aujourd'hui : le chiffrement symétrique (XOR cipher), le chiffrement asymétrique (RSA), et une comparaison des deux approches.

Ces exercices sont autonomes. Aucune bibliothèque supplémentaire n'est nécessaire au-delà de la bibliothèque standard de Python.

Mise en place. Créez un nouveau fichier **`crypto.py`** et travaillez chaque exercice en ajoutant du code à la fin du fichier. Exécutez-le à tout moment avec :

```
python3 crypto.py
```

Exercice 1. Chiffrement symétrique — XOR Cipher

En cryptographie symétrique, Alice et Bob partagent une seule clé secrète utilisée à la fois pour chiffrer et déchiffrer. L'exemple le plus simple est le chiffrement XOR : chaque caractère du message est combiné avec une clé à l'aide de l'opération ou exclusif (XOR, $\oplus$). Rappel : XOR est sa propre inverse : $(m \oplus k) \oplus k = m$.

(a) **XOR encrypt/decrypt de base.** Implémentez la fonction **`xor_cipher`** qui chiffre (ou déchiffre) une chaîne de caractères en XOR-ant chaque caractère avec **`key`**. Utilisez **`ord`**, **`chr`** et l'opérateur **`^`** de Python. Vérifiez ensuite que le test s'affiche bien **`True`** :

```python
def xor_cipher(text: str, key: int) -> str:
    ...   # une ligne suffit

message    = "HELLO"
key        = 42
ciphertext = xor_cipher(message, key)
decrypted  = xor_cipher(ciphertext, key)   # même fonction, même clé
print("Round-trip OK:", decrypted == message)   # doit être True
```

Pourquoi appeler **`xor_cipher`** deux fois avec la même clé redonne-t-il le message original ? Répondez en vous appuyant sur la table de vérité XOR du cours.

(b) **Chiffrer un message plus long.** Un seul octet de clé n'offre que 256 valeurs possibles. Étendez le cipher pour qu'il parcoure une liste d'octets de clé qui se répète en boucle : la position i du plaintext doit être XOR-é avec **`key[i % len(key)]`**. Complétez la fonction, puis vérifiez le round-trip :

```python
def xor_cipher_multi(text: str, key: list[int]) -> str:
    result = []
    for i, c in enumerate(text):
        result.append(...)   # XOR ord(c) avec le bon octet de clé
    return ''.join(result)

key_multi = [42, 17, 99]
msg = "ATTACK AT DAWN"
enc = xor_cipher_multi(msg, key_multi)
dec = xor_cipher_multi(enc, key_multi)
print("Round-trip OK:", dec == msg)   # doit être True
```

(c) **Longueur de clé et sécurité.**

(i) Combien de clés possibles existe-t-il pour une clé de 2 octets ? De 4 octets ? De n octets ?

(ii) Les ciphers symétriques modernes tels qu'AES utilisent une clé de 128 bits. Combien de clés possibles cela représente-t-il ? (Exprimez votre réponse sous forme de puissance de 2 et de puissance de 10 approximative.)

Exercice 2. Chiffrement asymétrique — Toy RSA

Contexte. En cryptographie asymétrique (à clé publique), chaque participant possède deux clés mathématiquement liées : une public key que tout le monde peut utiliser pour chiffrer un message, et une private key que seul le propriétaire détient, utilisée pour déchiffrer.

RSA est l'algorithme à clé publique le plus largement utilisé. Sa sécurité repose sur le fait que multiplier deux grands nombres premiers est facile, mais que l'opération inverse — factoriser le produit en ses deux facteurs premiers — est computationnellement infaisable pour des nombres suffisamment grands.

Génération de clés RSA (simplifiée).

1. Choisir deux nombres premiers p et q (en pratique, de plusieurs centaines de chiffres). Calculer $n = p \cdot q$.
2. Calculer l'indicatrice d'Euler : $\phi(n) = (p-1)(q-1)$. Cela compte le nombre d'entiers de 1 à $n-1$ qui ne partagent aucun facteur commun avec n.
3. Choisir un exposant public e tel que $1 < e < \phi(n)$ et $\gcd(e, \phi(n)) = 1$ (c'est-à-dire que e et $\phi(n)$ ne partagent aucun facteur commun).
4. Calculer l'exposant privé d tel que $e \cdot d \equiv 1 \pmod{\phi(n)}$, c'est-à-dire que d est l'inverse modulaire de e.

La **public key** est la paire $(e,\ n)$; la **private key** est d (avec n). Chiffrement : $c = m^e \bmod n$. Déchiffrement : $m = c^d \bmod n$.

Exemple jouet (utilisé tout au long de cet exercice) : $p = 3,\ q = 11,\ n = 33,\ \phi(n) = 20,\ e = 3,\ d = 7$, car $3 \times 7 = 21 \equiv 1 \pmod{20}$.

(a) **Vérifier les paramètres à la main.** Avant d'écrire du code, confirmez les points suivants sur papier (ou mentalement) :

(i) Vérifiez que $p = 3$ et $q = 11$ sont tous deux premiers.

(ii) Vérifiez que $n = p \cdot q = 33$ et $\phi(n) = (p-1)(q-1) = 20$.

(iii) Vérifiez que $e \cdot d = 3 \times 7 = 21 \equiv 1 \pmod{20}$, donc d est bien l'inverse modulaire de e.

(iv) Chiffrez le message $m = 4$ à la main : calculez $4^3 \bmod 33$. Puis déchiffrez le résultat : élevez-le à la puissance $d = 7$ modulo 33. Retrouvez-vous bien 4 ?

(b) **Exponentiation modulaire.** L'approche naïve — multiplier en boucle — est beaucoup trop lente pour de grands exposants. Implémentez **`mod_exp_naive`** en complétant la boucle ci-dessous, puis vérifiez qu'elle concorde avec la fonction native Python **`pow(base, exp, mod)`**, qui utilise en interne une exponentiation rapide (square-and-multiply) en $O(\log exp)$ multiplications :

```python
def mod_exp_naive(base: int, exp: int, mod: int) -> int:
    result = 1
    for _ in range(exp):
        ...   # mettre à jour result à chaque itération
    return result

e, d, n = 3, 7, 33
for m in range(1, 10):   # messages 1..9, tous < n = 33
    assert mod_exp_naive(m, e, n) == pow(m, e, n), f"Échec pour m={m}"
print("mod_exp_naive correct pour m = 1..9")
```

Pourquoi l'implémentation naïve devient-elle impraticable pour le vrai RSA, dont les exposants ont des milliers de chiffres ?

(c) **Chiffrer et déchiffrer un mot.** Pour chiffrer du texte avec RSA, on traite chaque caractère comme son code Unicode (obtenu avec **`ord`**) et on chiffre chaque code indépendamment avec **`pow`**. Implémentez les deux fonctions, puis chiffrez et déchiffrez le message **`"HI"`** avec les paramètres jouets :

```python
def rsa_encrypt(plaintext: str, e: int, n: int) -> list[int]:
    ...   # retourner la liste des codes chiffrés

def rsa_decrypt(ciphertext: list[int], d: int, n: int) -> str:
    ...   # retourner la chaîne déchiffrée

e, d, n = 3, 7, 33
enc = rsa_encrypt("HI", e, n)
dec = rsa_decrypt(enc, d, n)
print("Encrypted:", enc)
print("Decrypted:", dec)   # ne donne PAS "HI" -- pourquoi ?
```

Vous remarquerez que **`"HI"`** ne fait pas un round-trip correct. Expliquez pourquoi en consultant les code points Unicode de **`'H'`** et **`'I'`** et en les comparant à $n = 33$.

Pourquoi faut-il que $m < n$? Le chiffrement calcule $c = m^e \bmod n$, donc c tombe toujours dans $\{0, 1, \ldots, n-1\}$ quelle que soit la taille de m. Deux valeurs différentes de m qui diffèrent d'un multiple de n produisent exactement le même ciphertext, donc le déchiffrement ne peut pas les distinguer. Autrement dit, RSA ne « voit » que $m \bmod n$; toute information dans m au-delà de cela est définitivement perdue.

(d) **Paramètres légèrement plus grands.** Les paramètres ci-dessous utilisent des nombres premiers plus grands (mais toujours petits) qui rendent n suffisamment grand pour chiffrer n'importe quel caractère ASCII imprimable (code points 32–126).

```
# Génération de clés (pour référence -- pas besoin de les dériver) :
#   p = 61,  q = 53          (tous deux premiers)
#   n = p * q       = 3233   (public, partie des deux clés)
#   phi(n) = (p-1)*(q-1)
#          = 60 * 52 = 3120 (gardé secret)
#   e = 17                   (exposant public, gcd(17, 3120) = 1)
#   d = 2753                 (exposant privé : 17 * 2753 mod 3120 = 1)
e2, d2, n2 = 17, 2753, 3233
```

(i) En utilisant vos fonctions **`rsa_encrypt`** et **`rsa_decrypt`**, chiffrez puis déchiffrez **`"Hello, Bob!"`** et confirmez le round-trip avec un **`assert`**.

(ii) Combien de messages entiers distincts ces paramètres peuvent-ils traiter ? (c'est-à-dire combien de valeurs vérifient $0 \leq m < n$?)

(iii) Vérifiez en Python que $e_2 \cdot d_2 \bmod \phi(n) = 1$, confirmant que d_2 est bien le bon exposant privé. *Rappel :* $\phi(n) = (p-1)(q-1) = 3120$.

(e) **Pourquoi RSA est-il difficile à casser ?** Un attaquant qui connaît la public key (e, n) doit retrouver d. La seule méthode connue pour y parvenir efficacement est de factoriser n en p et q, puis de recalculer $\phi(n)$ et résoudre pour d. Implémentez le corps du factoriseur par division d'essai (trial division) ci-dessous :

```
from time import perf_counter
import math

def factorise(n: int) -> tuple[int, int]:
    """Retourne (p, q) tel que p * q == n, ou (1, n) si n est premier."""
    for p in range(2, math.isqrt(n) + 1):
        ...   # tester si p divise n et retourner la paire si c'est le cas
    return 1, n

start   = perf_counter()
p, q    = factorise(3233)
elapsed = perf_counter() - start
print(f"Factored 3233 = {p} * {q}  in {elapsed:.6f} s")
```

(i) Combien de divisions la trial division nécessite-t-elle au pire pour factoriser n ? (Exprimez en fonction de n.)

(ii) Le vrai RSA utilise n avec 2048 bits ($n \approx 10^{617}$). Estimez le nombre de divisions nécessaires pour la trial division et expliquez pourquoi c'est totalement infaisable même sur le matériel moderne le plus rapide.

(iii) Une fois que vous avez factorisé $n = 3233$ en p et q, calculez $\phi(n)$ et retrouvez l'exposant privé d à l'aide de **`pow(e2, -1, phi_n)`**. Confirmez que le résultat vaut bien 2753 et qu'il déchiffre correctement un message chiffré avec **`e2`**.

Exercice 3. Comparaison chiffrement symétrique et asymétrique

(a) **Comparaison de vitesse.** Écrivez le code qui mesure le temps nécessaire pour chiffrer une chaîne de 1 000 caractères avec votre XOR cipher (clé = 42) et avec votre toy RSA (paramètres **`e2, n2`**). Utilisez **`time.perf_counter`** et affichez les deux durées ainsi que le rapport entre elles.

```
from time import perf_counter

long_msg = "A" * 1000

# Mesurez et affichez le temps pour xor_cipher
...

# Mesurez et affichez le temps pour rsa_encrypt
...

# Affichez combien de fois RSA est plus lent que XOR
...
```

En pratique, des systèmes comme HTTPS utilisent le chiffrement asymétrique uniquement pour échanger une courte session key aléatoire, puis passent au chiffrement symétrique pour toutes les données suivantes. Pourquoi cela est-il logique au vu de vos résultats ?

(b) **Le problème de distribution de clés.** Le chiffrement symétrique exige qu'Alice et Bob se mettent d'accord sur une clé secrète partagée avant de pouvoir communiquer de façon sécurisée — mais comment échangent-ils cette clé sans que personne ne l'intercepte ?

(i) Si n personnes souhaitent toutes communiquer en privé avec chacune des autres en utilisant un chiffrement symétrique, combien de clés partagées distinctes sont nécessaires au total ? Remplissez le tableau et donnez une formule générale :

Personnes (n)	**Clés nécessaires**
2	
5	
10	
100	

(ii) Comment le chiffrement à public key résout-il le problème de distribution de clés ? Combien de clés chaque personne doit-elle détenir ? (Répondez en 2–3 phrases.)