# Nouvelle-Calédonie - novembre 2025 - sujet 2 (corrigé)

**Exercice 1 (Sécurisation des communications, représentation des données et programmation Python)**
**Partie A : cryptographie symétrique**

1. Il doit saisir l'instruction suivante : `m2 = code(m1, cle)`

**Partie B : cryptographie asymétrique**

2. Il doit saisir l'instruction suivante : `m2 = code(m1, cle2_b)`
3. Eve dispose du message chiffré `m1` et de la clé publique `cle1_b` de Bob, mais pas de sa clé privée `cle2_b`. On a supposé qu'il était impossible de retrouver `m0` à partir de `m1` sans connaître `cle2_b` donc Eve ne peut pas déchiffrer le message.
4. La clé `cle1_b` est la clé publique de Bob qui est diffusée publiquement et la clé `cle2_b` est sa clé privée qu'il garde secrète.
5. On procède de même en utilisant les clés d'Alice.
   - Bob saisit l'instruction `m5 = code('Bien reçu. Rendez-vous à16h donc.', cle1_a)`
   - Alice saisit l'instruction `m6 = code(m5, cle2_a)`
6. Dans un chiffrement asymétrique, il faut bien deux clés : la clé privée pour déchiffrer et la clé publique pour chiffrer. Il est vrai qu'une fois le message chiffré tout ne repose plus que sur une seule clé (la clé privée) et que, si celle-ci est dérobée, la confidentialité est compromise, mais le protocole entier nécessite bien deux clés. La clé publique sert à fermer la porte (tout le monde peut la fermer) et la clé privée à l'ouvrir (une seule personne peut l'ouvrir).

**Partie C : signature**

7. Si `m0_s = code(m0, cle2_a)`, alors `code(m0_s, cle1_a)` est égal à `m0`. Donc si Alice a chiffré le message à l'aide de sa clé privée `cle2_a`, on a égalité entre `m0` et `code(m0_s, cle1_a)`, ce que peut vérifier Bob qui dispose de `m0`, `m0_s` et `cle1_a`. Supposons que Mallory remplace le message `m0` par un message `m3`. Il doit alors construire la signature `m3_s` telle que `code(m3_s, cle1_a)` soit égal à `m3`. Or, cela nécessite l'utilisation de `cle2_a` qu'Alice conserve secrètement. Mallory ne peut donc pas générer cette signature et se faire passer pour Alice.

**Partie D : signature par empreinte**

8. La réduction est égale à 5
9. La fonction suivante convient :

```
def reduction(m):
    res = 0
    for i in range(1, len(m)):
        res += i * abs(ord(m[i]) - ord(m[i - 1]))
    return res
```

10. Bob reçoit `m0` et `m0_s`, et connaît `cle1_a`. Il exécute les instructions suivante :

```
m0_r_1 = str(reduction(m0))
m0_r_2 = code(m0_s, cle1_a)
assert m0_r_1 == m0_r_2
```

    Autrement dit, Bob calcule d'une part la réduction du message `m0` qu'il vient de recevoir et, d'autre part, il déchiffre le message `m0_s` grâce à la clé publique d'Alice : si les deux résultats sont identiques, alors Bob est assuré que le message provient bien d'Alice.
11. En ne considérant que les dix premières lettres d'un message pour calculer la réduction, on a alors `reduction(m0)` est égal à `reduction(m3)`. Mallory peut donc changer le message tant qu'il conserve le même début de message. Ce n'est donc pas une bonne idée.
12. Alice exécute les instructions suivantes :

```
m1 = code(m0, cle1_b)
m0_r = str(reduction(m0))
m0_s = code(m0_r, cle2_a)
```

Elle transmet à Bob `m1` et `m0_s`. Bob exécute alors les instructions suivantes :

```
m2 = code(m1, cle2_b)
m2_r = str(reduction(m2))
m0_r = code(m0_s, cle1_a)
assert m2_r == m0_r
```

Autrement dit, Bob commence par déchiffrer le message `m1` à l'aide de sa clé privée et obtient un message `m2`. Pour être sûr que ce message provient d'Alice, il calcule la réduction de `m2` et déchiffre `m0_s` à l'aide de la clé publique d'Alice. Si ces deux réductions sont identiques, Bob est certain que le message provient d'Alice (ici, on n'a pas chiffré la signature avec la clé publique de Bob et on peut aussi signer le message chiffré).

**Exercice 2 (Bases de données, langage SQL, programmation Python et gestion de bugs)**

1. Une clé primaire est un ou un groupe d'attributs qui permet l'authentification d'un enregistrement de la table, sa valeur doit donc être unique
2. On repère Emile Pasteur par sa référence client qui vaut `25145`. Aiinsi, le 18 mars 2025, il a acheté 5 sacs de sable et a bénéficié d'une remise de 30%
3. La requéte suivante convient :

```
INSERT INTO client
VALUES (25345, 'Bertaut', 'Gilles', 'gbertaut@fmail.fr', '0641424344')
```

4. La requéte suivante convient :

```
UPDATE client SET email='shars@fmail.fr' WHERE ref_client=25322
```

5. La requéte suivante convient :

```
SELECT DISTINCT ref_client FROM vente WHERE date >= '2025/01/01'
```

6. La requéte suivante convient :

```
SELECT DISTINCT client.nom, client.telephone FROM vente
JOIN client ON vente.ref_client=client.ref_client
WHERE date >= '2024/09/15' AND vente.ref_produit = 90222
```

7. Le code suivant convient :

```
def select_tel(client, ref_client):
    return client[ref_client][3]
```

8. Cette erreur est déclenchée car la clé `1234` n'appartient pas au dictionnaire `client`. Cela correspond à la situation où aucun client de la base de données du magasin n'a comme référence `1234`
9. La modification suivante convient :

```
def select_tel(client, ref_client):
    if ref_client not in client :
        return None
    return client[ref_client][3]
```

10. La fonction suivante convient :

```
def nb_produits (vente, ref_produit) :
    nombre = 0
    for ref_vente in vente :
        if vente[ref_vente][1] == ref_produit :
            nombre += vente[ref_vente][3]
    return nombre
```

11. Cette requête essaie de supprimer un enregistrement de la table `produit` alors qu'il est utilisé comme référence de clé étrangère d'un enregistrement dans la table `vente`. L'exécution de cette requête est refusée pour garder la base de données cohérente
12. Les modifications suivantes conviennent :

```
def contient_prod(vente, ref_produit):
    for liste in vente.values():
        if ref_produit == liste[1] :
            return True
    return False

def delete_prod(produit, vente, ref_produit):
    assert not contient_prod(vente, ref_produit), "foreign key constraint failed"
    del produit[ref_produit]
```

**Exercice 3 (Structures de données, programmation Python et graphes)**
**Partie A**

1. On peut lancer un 2, un 4 ou un 5
2. On peut uniquement lancer 0
3. Le code suivant convient :

```
def lancer_possible(etat, lancer):
    if lancer >= len(etat) or lancer < 0:
        return False
    if lancer == 0 and etat[0] == 1:
        return False
    if lancer > 0:
        if etat[0] == 0 or etat[lancer] != 0:
            return False
    return True
```

4. On a l'état suivant : `[1, 0, 1, 0, 1, 0]`
5. Le code suivant convient :

```
def lancer_balle(etat, lancer):
    # copie de l'état pour ne pas le modifier
    nouvel_etat = [balle for balle in etat]
    if lancer != 0:
        nouvel_etat[lancer] = 1
    for i in range(len(nouvel_etat) - 1):
        nouvel_etat[i] = nouvel_etat[i + 1]
    nouvel_etat[len(nouvel_etat) - 1] = 0
    return nouvel_etat
```

**Partie B**

6. La fonction suivante convient :

```
def liste_lancers_possibles(etat):
    if etat[0] == 0:
        return [0]
    else:
        lancers = []
        for i in range(len(etat)):
            if etat[i] == 0:
                lancers.append(i)
        return lancers

# Autre possibilités :
def liste_lancers_possibles_v2(etat):
    lancers = []
    for i in range(len(etat)):
        if lancer_possible(etat, i):
            lancers.append(i)
    return lancers
```

7. Elle est récursive car elle s'appelle elle-même (ligne 13)
8. Elle se termine car il y a un cas de base à `n = 0` et l'appel récursif se fait sur `n -1` qui est plus petit que `n`
9. Le code suivant convient :

```
def calcule_sequences(etat, n):
    if n == 0:
        return [[]]
    else:
        s_possibles = []
        l_lancers = liste_lancers_possibles(etat)
        for lancer in l_lancers:
            etat2 = lancer_balle_v2(etat, lancer)
            s_etat2 = calcule_sequences(etat2, n-1)
            for seq in s_etat2:
                s_possibles.append( [lancer] + seq )
        return s_possibles
```

**Partie C**

10. On a le code suivant :

```
automate = {'11000': [(3, '10100'), (2, '11000'), (4, '10010')],
            '01010': [(0, '10100')],
            '10100': [(3, '01100'), (4, '01010'), (1, '11000')],
            '01100': [(0, '11000')],
            '10010': [(1, '10100'), (2, '01100'), (4, '00110')],
            '00110': [(0, '01100')]}
```

11. La fonction suivante convient :

```
def lancer_balle_automate(automate, etat, lancer):
    for (l, e_suite) in automate[etat]:
        if l == lancer:
            return e_suite
    return ''
```

12. La fonction suivante convient :

```
def parcours_sequence_depart(automate, depart, sequence):
    """ automate est un automate, sequence une liste d'entiers
    (lancers) et depart un état de départ.
    Renvoie l'état atteint à partir de l'état de
    si la séquence est valide,
    ou bien None si un lancer est impossible."""
    etat = depart
    for lancer in sequence:
        etat = lancer_balle_automate(automate, etat, lancer)
        if etat == "":
            return None
    return etat
```

13. La fonction suivante convient :

```
def departs_siteswap(automate, sequence):
    departs = []
    for dep in automate:
        if parcours_sequence_depart(automate, dep, sequence) == dep:
            departs.append(dep)
    return departs
```