Métropole - septembre 2025 - sujet 2 (corrigé)

Exercice 1 (Programmation Python et cryptographie)
Partie A

1. On a le carré de chiffrement suivant :

E	P	R	U	V
D	N	S	I	A
B	C	F	G	G
J	K	L	M	O
Q	T	X	Y	Z

2. L'assertion suivante convient :

```
assert 'W' not in clef
```

3. Le code suivant convient :

```
def creer_carre(liste_clef):
    carre = [[0 for i in range(5)] for j in range(5)]
    for i in range(25):
        carre[i//5][i%5] = liste_clef[i]
    return carre
```

Partie B

4. Le code suivant convient :

```
def couper_en_digrammes(message):
    digrammes = []
    i = 0
    while i < len(message) - 1:
        if message[i] == message[i+1]:
            digrammes.append(message[i] + 'X')
            i = i + 1
        else:
            digrammes.append(message[i] + message[i+1])
            i = i + 2
    if i == len(message) - 1: # il reste une lettre isolée
        digrammes.append(message[i] + 'X')
    return digrammes
```

5. On obtient le découpage suivant : `["BO","NJ","OU","RX"]`
6. Chaque groupe de lettres devient respectivement `HV`, `QG`, `NV` et `CU`. On obtient ainsi `HVQGNVCU`
7. La fonction suivante convient :

```
def ligne_colonne(lettre, carre):
    for i in range(len(carre)):
        for j in range(len(carre)):
            if carre[i][j] == lettre:
                return i, j
```

8. La fonction suivante convient :

```
def sur_la_meme_ligne(digramme, carre):
    lettre1 = digramme[0]
    lettre2 = digramme[1]
    return ligne_colonne(lettre1, carre)[0] == ligne_colonne(lettre2, carre)[0]
```

9. Le code suivant convient :

```
def chiffrer_digramme(digramme, carre):
    lettre1 = digramme[0]
    lettre2 = digramme[1]
    i1, j1 = ligne_colonne(lettre1, carre)
    i2, j2 = ligne_colonne(lettre2, carre)
    if sur_la_meme_ligne(digramme, carre):
        digramme_chiffre = carre[i1][(j1 + 1)%5] + carre[i2][(j2 + 1)%5]
    elif sur_la_meme_colonne(digramme, carre):
        digramme_chiffre = carre[(i1 + 1)%5][j1] + carre[(i2 + 1)%5][j2]
    else:
        digramme_chiffre = carre[i1][j2] + carre[i2][j1]
    return digramme_chiffre
```

10. La fonction suivante convient :

```
def chiffrer_playfair(message, clef):
    carre = creer_carre(creer_liste_clef(clef))
    d_msg = couper_en_digrammes(message)
    msg_chiffre = ""
    for d in d_msg:
        msg_chiffre += chiffrer_digramme(d, carre)
    return msg_chiffre
```

Exercice 2 (Systèmes d'exploitation et POO)

Partie A

1. La commande `ls -l` permet d'obtenir le listing détaillé des fichiers et dossiers présents
2. Les caractères `-rwxrw-r-` signifie que le propriétaire peut lire, écrire/modifier et exécuter le fichier, que les autres membres du groupe peuvent lire et écrire/modifier et que les autres utilisateurs ne peuvent que le lire
3. Il faut saisir l'instruction `rm concat.pls`
4. La commande `chmod g-wx` indique que les permissions en écriture et en exécution sont retirées aux membres du groupe `adminsys`

Partie B

5. La chaîne `'rw-'` donne $(110)_2$ qui vaut 6
6. Les deux codes suivants conviennent :

```
def bin_to_oct(chaine):
    valeur = 0
    for i in range(len(chaine)):
        valeur = valeur + int(chaine[i]) * 2 ** (2 - i)
    return valeur
```

```
def bin_to_oct(chaine):
    valeur = 0
    poids = 1
    for i in range(len(chaine)-1, -1, -1):
        valeur = valeur + int(chaine[i]) * poids
        poids = poids * 2
    return valeur
```

7. On obtient en sortie la chaîne `'110101100'`
8. Le code suivant convient :

```
def symb_to_oct(chaine):
    repr_bin = symb_to_bin(chaine)
    # les 3 premiers bits correspondent au propriétaire
    user = repr_bin[0] + repr_bin[1] + repr_bin[2]
    # les 3 suivants au groupe
    group = repr_bin[3] + repr_bin[4] + repr_bin[5]
    # et les 3 derniers aux autres
    other = repr_bin[6] + repr_bin[7] + repr_bin[8]
    return str(bin_to_oct(user)) + str(bin_to_oct(group)) + str(bin_to_oct(other))
```

Partie C

9. L'instruction suivante convient :

```
mon_fichier = Fichier('concat.pls', 1257, 'root', 'wwwdata', 'rw-r--r--')
```

10. La fonction suivante convient :

```
def chown(self, nouveau_proprietaire):
    self.proprietaire = nouveau_proprietaire
```

11. La fonction suivante convient :

```
def get_executable(fichiers, proprietaire):
    executables = []
    for fichier in fichiers:
        if fichier.proprietaire == proprietaire and fichier.permission[2] == 'x':
            executables.append(fichier)
    return executables
```

Exercice 3 (Langage SQL, graphes et programmation générale)

1. La requête suivante convient :

```
INSERT INTO Produits
VALUES (10,'Croissant', 'Alimentaire', 0.90, 4)
```

2. La requête suivante convient :

```
UPDATE Fournisseurs
SET adresse = '78 Rue des Jeux', ville = 'Elbeuf',
WHERE nom_fournisseur = 'Livres en Folie'
```

3. Cette requête SQL sélectionne et affiche les noms de tous les produits de la catégorie `Alimentaire`. Avec les extraits précédents, on obtient donc `Yaourts blanc x 4`, `Lait` et `Pain`
4. La requête suivante convient :

```
SELECT * FROM Commandes WHERE total >= 1000
```

5. La requête suivante convient :

```
SELECT nom FROM Fournisseurs WHERE pays ='Espagne' OR pays='Italie'
```

6. La requête suivante convient :

```
SELECT Fournisseurs.nom FROM Fournisseurs
JOIN Produits ON Fournisseurs.id_fournisseur = Produits.id_fournisseur
WHERE catégorie = 'Alimentaire'
```

7. La requête suivante convient :

```
SELECT id_commande, date, Fournisseurs.nom FROM Fournisseurs
JOIN Commandes ON Fournisseurs.id_fournisseur = Commandes.id_fournisseur
JOIN Details ON Commandes.id_commande = Details.id_commande
JOIN Produits ON Details.id_produit = Produits.id_produit
WHERE categorie = 'Vêtement'
```

8. Le plus court chemin est : Toulouse - Bordeaux - Nantes - Calais
9. On a les sommets suivants : Calais - Nantes - Bordeaux - Toulouse - Lyon - Marseille - Paris - Strasbourg
10. On a les sommets suivants : Calais - Nantes - Paris - Strasbourg - Bordeaux - Lyon - Toulouse - Marseille
11. Les instructions suivantes conviennent :

```
graphe['Lyon']['Marseille'] = 315
graphe['Marseille']['Lyon'] = 315
```

12. La fonction suivante convient :

```
def distance(graphe, ville1, ville2):
    if ville1 in graphe and ville2 in graphe[ville1]:
        return graphe[ville1][ville2]
    else:
        return None
```

13. La fonction suivante convient :

```
def distance_totale(graphe, ville_depart, ville_destination):
    chemin = trouver_chemin(graphe, ville_depart, ville_destination)
    if chemin is None:
        return None
    distance_totale = 0
    for i in range(len(chemin) - 1):
        ville_courante = chemin[i]
        prochaine_ville = chemin[i + 1]
        distance_totale += graphe[ville_courante][prochaine_ville]
    return distance_totale
```

14. La valeur de `graphe['Paris']['Nantes'][1]` est `3.47`
15. Le code suivant convient :

```
def ratio_duree_distance(graphe):
    for ville, connexions in graphe.items():
        for destination, valeurs in connexions.items():
            distance, duree = valeurs
            ratio = duree / distance
            graphe[ville][destination].append(ratio)
    return graphe
```

16. Il s'agit d'un algorithme glouton
17. On obtient le chemin suivant :

```
['Toulouse', 'Bordeaux', 'Nantes', 'Paris', 'Calais']
```