2024 métropole jour 2

Exercice 1

Partie A

1. Non, car les valeurs de ce champ ne sont pas uniques : un artiste peut avoir plusieurs albums et donc apparaître plusieurs fois dans la table `CD`.

2.

Nom_artiste
Nightwish
The Rasmus

3.

Annee
1986
2001
1986

4.
```
UPDATE CD SET Annee = 2000 WHERE id_album = 4;
```

5.

```
SELECT Titre_album
FROM CD
JOIN Artiste ON CD.Nom_artiste = Artiste.Nom_artiste
JOIN Rangement ON CD.id_album = Rangement.id_album
WHERE Style = 'Metal' AND Numero_etagere = 1;
```

6. Il doit d'abord supprimer la ligne avec `id_album = 5` de la table `Rangement` pour supprimer la clé étrangère faisant référence à la clé primaire de la table `CD`.
Ensuite il peut supprimer la ligne avec `id_album = 5` de la table `CD` pour supprimer la clé étrangère faisant référence à la clé primaire de la table `Artiste`.
Enfin il peut supprimer la ligne avec `Nom_artiste = 'The Rasmus'` de la table `Artiste`.
La requête de la suppression de l'album est :

```
DELETE FROM CD WHERE id_album = 5;
```

Partie B

7. Un algorithme de chiffrement symétrique utilise la même clé pour chiffrer et pour déchiffrer un message.

8. Un algorithme de chiffrement asymétrique utilise la clé publique du destinataire pour chiffrer le message et le destinataire utiliser sa clé privée pour le déchiffrer.

9. Le serveur doit chiffrer la clé C avec la clé publique de Bob. Il lui envoie alors la clé C chiffrée. Puis Bob déchiffre cette clé avec sa clé privée.

Exercice 2

Partie A

1.

```
class Marchandise:
    def __init__(self, p: int, v: int) -> 'Marchandise':
        assert v > 0
        self.prix = p
        self.volume = v
```

2. `m1 = Marchandise(20, 7)`

3.

```
def ratio(self) -> float:
    return self.prix / self.volume
```

4.

```
def prixListe(tab: list) -> int:
    somme = 0
    for m in tab:
        somme = somme + m.prix
    return somme
```

Partie B

5.

- m1 -> prix : 40, volume : 20
- m2 -> prix : 210, volume : 70
- m3 -> prix : 160, volume : 40
- m4 -> prix : 50, volume : 50
- m1, m2 -> prix : 250, volume : 90
- m1, m3 -> prix : 200, volume : 60
- m1, m4 -> prix : 90, volume : 70
- m3, m4 -> prix : 210, volume : 90

La meilleure combinaison est m1, m2 avec 250 €.

6. glouton

7.

```
def tri(tab: list) -> None:
    n = len(tab)
    for i in range(1, n):
        marchandise = tab[i]
        j = i-1
        while j >= 0 and marchandise.ratio() > tab[j].ratio() :
            tab[j+1] = tab[j]
            j = j-1
        tab[j+1] = marchandise
```

8. C'est un tri par insertion, son coût est quadratique.

9.

```
def charge(tab: list, volume: int) -> list:
    tri(tab)
    chargement = []
    n = len(tab)
    for i in range (n):
        if tab[i].volume <= volume:
            chargement.append(tab[i])
            volume = volume - tab[i].volume
    return chargement
```

Partie C

10.
[Attention, `n` n'est pas défini, il faut donc utiliser `len(tab)`. Et il y a une interversion entre l'option 1 et l'option 2 de l'énoncé.]

```
def chargeOptimale(tab: list, v_restant: int, i: int) -> list:
    if i >= len(tab):
        return []
    else:
        if tab[i].volume > v_restant:
            return chargeOptimale(tab, v_restant, i+1)
        else:
            option1 = chargeOptimale(tab, v_restant, i+1)
            option2 = [tab[i]] + chargeOptimale(tab, v_restant - tab[i].volume, i+1)
            if prixListe(option1) > prixListe(option2):
                return option1
            else:
                return option2
```

Exercice 3

Partie A

1.

— nom : string
— denivele : int
— longueur : float
— couleur : string
— ouverte : booléen

2.

```
def set_couleur(self):
    if self.denivele >= 100:
        self.couleur = 'noire'
    elif self.denivele >= 70:
        self.couleur = 'rouge'
    elif self.denivele >= 40:
        self.couleur = 'bleue'
    else:
        self.couleur = 'verte'
```

3. D

4.

```
for piste in lievre_blanc.get_pistes():
    if piste.get_couleur() == 'verte':
        piste.ouverte = False
```

5.
[Légère incohérence entre l'ordre des paramètres de l'énoncé et celui de l'exemple.]

```
def pistes_de_couleur(couleur, lst):
    liste_noms = []
    for piste in lst:
        if piste.get_couleur() == couleur:
            liste_noms.append(piste.get_nom())
    return liste_noms
```

6.

```
def semi_marathon(L):
    distance = 0
    liste_pistes = lievre_blanc.get_pistes()
    for nom in L:
        for piste in liste_pistes:
            if piste.get_nom() == nom:
                distance = distance + piste.get_longueur()
    return distance > 21.1
```

Partie B

7. `domaine['E']['F']`

8.

```
def voisins(G, s):
    liste_voisins = []
    for v in G[s].keys():
        liste_voisins.append(v)
    return liste_voisins
```

9.

```
def longueur_chemin(G, chemin):
    precedent = chemin[0]
    longueur = 0
    for i in range(1, len(chemin)):
        longueur = longueur + G[precedent][chemin[i]]
        precedent = chemin[i]
    return longueur
```

10. La fonction parcours est récursive car elle s'appelle elle-même à la ligne 7.
[La fonction `parcours` est très mal écrite. Si on l'appelle deux fois de suite, il n'y aura pas le même résultat car l'initialisation par défaut d'un objet mutable (liste) n'est faite qu'au premier appel de la fonction. Ainsi, `lst_chemins` ne sera vide qu'au premier appel de la fonction ensuite elle contiendra les chemins déjà trouvés. L'appel récursif à la ligne 7 utilise cette propriété pour fonctionner mais c'est très dangereux! Il est préférable de ne pas utiliser de valeurs par défaut pour les objets mutables pour éviter des comportements étranges. Voilà comment il aurait fallu écrire la fonction pour éviter des bugs :]

```
def parcours(G, depart, chemin, lst_chemins):
    if chemin == []:
        chemin = [depart]
    for sommet in voisins(G, depart):
        if sommet not in chemin:
            lst_chemins.append(chemin + [sommet])
            parcours(G, sommet, chemin + [sommet], lst_chemins)
    return lst_chemins
```

11.
[Erreur dans le sujet : il n'y a pas de doublons dans la liste renvoyée par `parcours` ! Des doublons apparaissent à cause de l'écriture dangereuse de `parcours` et de son appel sans donner des listes vides dans la fonction `parcours_dep_arr`. Le corrigé ci-dessous empêche le mauvais comportement même avec l'écriture risquée de `parcours` (il n'y a donc pas besoin d'éliminer les doublons car il n'y en a pas) :]

```
def parcours_dep_arr(G, depart, arrivee):
    liste = parcours(G, depart, [], []) # Force une nouvelle initialisation
    lst_chemins = []
    for chemin in liste:
        if chemin[-1] == arrivee:
            lst_chemins.append(chemin)
    return lst_chemins
```

[Voilà la fonction attendue qui enlève les doublons créés par l'écriture risquée de `parcours` :]

```python
def parcours_dep_arr(G, depart, arrivee):
    liste = parcours(G, depart)
    lst_chemins = []
    for chemin in liste:
        if chemin[-1] == arrivee:
            if not chemin in lst_chemins: # Inutile en théorie
                lst_chemins.append(chemin)
    return lst_chemins
```

12.

```python
def plus_court(G, depart, arrivee):
    liste_chemins = parcours_dep_arr(G, depart, arrivee)
    chemin_plus_court = liste_chemins[0]
    minimum = longueur_chemin(G, chemin_plus_court)
    for chemin in liste_chemins:
        longueur = longueur_chemin(G, chemin)
        if longueur < minimum:
            minimum = longueur_chemin(G, chemin)
            chemin_plus_court = chemin
    return chemin_plus_court
```

13. Le dénivelé peut jouer un rôle important dans le temps de parcours d'une piste. Il faudrait donc mesurer le temps effectif de parcours des pistes et calculer le temps minimum plutôt que la distance minimum.