25NSIPE4

Exercice 1 (6 points)

Cet exercice porte sur les bases de données relationnelles et les requêtes SQL.

1. L'attribut `nom` n'est pas unique donc ne peut pas être choisi comme clé primaire, p.ex Lisa Benard et Emma Benard partagent le même nom.
2. La requête `SELECT nom, prenom FROM joueurs WHERE genre=2;` permet d'obtenir les noms et prénoms des joueuses du club.
3. La requête `INSERT INTO joueurs VALUES (6, "Gervais", "Nathan", 1) ;` permet d'ajouter dans la table le joueur dont le prénom est Nathan et le nom est Gervais, en choisissant une valeur pour l'identifiant `id` cohérente avec le reste de la base.
4. La requête `UPDATE competitions SET nom="Open de Tours" WHERE id=3 ;` permet de corriger la faute de frappe.
5. La requête `SELECT nom, annee FROM competitions ORDER BY annee ;` permet d'obtenir la liste des noms des tournois, ainsi que leur année, en triant par année croissante.
6. La requête `SELECT nom, prenom FROM joueurs JOIN participe ON joueurs.id=participe.id_joueur WHERE nb_gagnant > nb_fautes ;` permet d'obtenir le nom et le prénom des joueurs et des joueuses ayant obtenu un nombre de coups gagnants strictement supérieur au nombre de fautes lors d'une compétition, la même personne pouvant apparaître plusieurs fois si elle a rempli ces conditions lors de plusieurs compétitions.
7. La requête `SELECT joueurs.nom, prenom, competition.nom FROM joueurs JOIN participe ON joueurs.id=participe.id_joueur JOIN competitions ON participe.id_compet=competitions.id WHERE genre=2 AND annee=2023 ;` permet d'obtenir le nom, le prénom des joueuses et le nom des compétitions auxquelles elles ont participé en 2023.
8. Pour supprimer de la table `joueurs` la joueuse Emma Benard qui ne fait plus partie du club, il faut d'abord supprimer dans la table `participe` toutes les lignes concernant Emma, pour éviter des violations de contraintes de référence.
9. Pour insérer Agathe et ses résultats dans la base, en choisissant pour identifiant pour la table `joueurs` la valeur 7 et pour la table `competitions` la valeur 5, on utilise dans cet ordre les requêtes :

```
INSERT INTO joueurs VALUES (7, "Turion", "Agathe", 2) ;
DELETE FROM participe WHERE id_compet=5;
DELETE FROM competitions WHERE id=5;
INSERT INTO competitions VALUES (5, "Tournoi de Blois", 2024) ;
INSERT INTO participe VALUES (7, 5, 14, 15, 2) ;
```

On a commencé par supprimer l'Open de Nantes 2021 et toutes les références qui y sont faites dans la table `participe`.

Exercice 2 (6 points)

Cet exercice porte sur les réseaux, le routage, les graphes et la programmation.

Partie A : Réseau et adressage

1. Parmi les deux adresses IP suivantes : 137.254.128.200 et 137.254.128.210, l'adresse IP de la machine déjà connectée au sous-réseau Commerces est 137.254.128.200 à cause de la politique utilisée pour la numérotation.
2. Il n'est pas possible d'ajouter 132 machines sur le sous-réseau Commerces, puisque le masque de sous réseau est 255.255.255.0 ce qui signifie qu'il y a 254 adresses disponibles qui vont de 137.254.128.1 à 137.254.128.254 (les adresses 137.254.128.0 et 137.254.128.255 étant l'adresse du sous-réseau lui-même et l'adresse de diffusion respectivement) ; or 207 machines sont déjà connectées, on ne peut en ajouter que 47.

Partie B : Programmation d'un protocole de routage

3. Donner la liste des routeurs par lesquels transite un message envoyé depuis une machine du sous-réseau Navigation à destination d'une machine du sous- réseau Commerces est R1-R3-R6-R7 par lecture des tables de routage.
4. Lorsque qu'une machine du sous-réseau Commerces envoie des données à destination d'une machine du sous-réseau Navigation, le début du routage est R7-R4-R6-R4 et on a une boucle infinie R7-R4-R6-R4-R6-R4-R6-R4-R6-R4-...
5. Le dictionnaire correspondant au réseau de la Figure 2 est
   `{'R1': ['R2', 'R3'], 'R2': ['R1','R3','R5'], 'R3': ['R1', 'R2'], 'R5': ['R2']}`
6. Le principe d'une fonction récursive est de s'appeler elle-même.
7. On écrit une fonction `plus_court_chemin(graphe, r_depart, r_arrivee)`.

```
def plus_court_chemin(graphe, r_depart, r_arrivee):
    chemins = liste_chemins(graphe, r_depart, r_arrivee)
    meilleur_chemin=chemins[0] # on suppose qu'il y a au moins un chemin !
    for k in range(1, len(chemins)):
        if len(chemins[k])<len(meilleur_chemin):
            meilleur_chemin = chemins[k]
    return meilleur_chemin
```

8. On complète la fonction `plus_court_chemin_largeur(graphe, r_depart, r_arrivee)`.

```
def plus_court_chemin_largeur(graphe, r_depart, r_arrivee):
    dict_chemins = {}
    L = [r_depart]
    sommets_marques = [r_depart]
    dict_chemins[r_depart] = [r_depart]
    for r in L: # sérieusement, ne faites pas ça ; L est modifiée dans la boucle
        for s_r in graphe[r]:
            if not s_r in sommets_marques:
                sommets_marques.append(s_r)
                dict_chemins[s_r] = dict_chemins[r] + [s_r]
                if s_r == r_arrivee :
                    return dict_chemins[s_r]
                L.append(s_r)
```

9. On écrit une fonction `table_routage(graphe, routeur)`.

```
def table_routage(graphe, routeur):
    table = dict()
    for destination in graphe:
        if destination != routeur:
            route = plus_court_chemin_largeur(graphe, routeur, destination)
            table[destination] = route[1]
    return table
```

Partie C : Utilisation du protocole OSPF

10. Le coût correspondant à la liaison Ethernet est $\frac{10^9}{10 \times 10^6} = 100$, le coût correspondant à la liaison Fast Ethernet est $\frac{10^9}{100 \times 10^6} = 10$, le coût correspondant à la liaison Fibre est $\frac{10^9}{500 \times 10^6} = 2$.
11. Un message envoyé depuis le routeur R1 à destination du routeur R7, en respectant le protocole OSPF, suit la route R1-R3-R2-R5-R6-R4-R7.

12. On complète la table de routage du routeur R2 en respectant le protocole OSPF ; on a indiqué les coûts.

Table de routage de R2		
Destination	Suivant	Coût
R1	R3	4
R3	R3	2
R4	R5	14
R5	R5	10
R6	R5	12
R7	R5	16

Exercice 3 (8 points)

Cet exercice porte sur l'algorithmique des tableaux, la gestion de bugs, les listes, les piles et la programmation orientée objet.

Partie A

1. Le résultat que doit renvoyer `tri_dictatorial([31, 45, 41, 28, 37, 108, 127, 2, 124, 421])` est `[31, 45, 108, 127, 421]`.
2. Le tri dictatorial ne conserve pas l'ensemble des valeurs de la liste passée en paramètre donc il n'est pas un algorithme de tri.
3. On déroule l'appel `tri_dictatorial([8, 2, 9, 6, 12])` dans un tableau.

étape	i	serie[i]	serie[i-1]	serie[i]>=serie[i-1]	serie_triee
0					[8]
1	1	2	8	False	[8]
2	2	9	2	True	[8, 9]
3	3	6	9	False	[8, 9]
4	4	12	6	True	[8, 9, 12]

La valeur renvoyée est `[8, 9, 12]`.

4. L'erreur obtenue est une erreur d'index puisqu'on essaie d'accéder au premier élément d'une liste vide. On peut proposer cette modification, qui respecte davantage la description de l'algorithme :

```
def tri_dictatorial(serie):
    if serie == []:
        return []
    serie_triee = [serie[0]]
    for i in range(1, len(serie)):
        if serie[i] >= serie[i - 1]:
            serie_triee.append(serie[i])
    return serie_triee
```

5. Des tests utilisant des cas d'usage ne peuvent pas être exhaustifs, il se peut qu'un cas d'usage non testé provoque une erreur, même si en général on essaie de choisir suffisamment de cas d'usage pour avoir une bonne couverture du code.
6. La cause du problème est la comparaison `serie[i] >= serie[i - 1]` qui devrait être `serie[i] >= serie_triee[-1]`.

Partie B

7. On construit les trois maillons `m1`, `m0` et `m8` et la liste chaînée représentés ci-dessus avec les instructions suivantes.

```
m8 = Maillon(8, None)
m0 = Maillon(0, m8)
m1 = Maillon (1, m0)
ma_liste = Liste(m1)
```

8. On indique ce que renvoie chacune des instructions.

```
>>> m1.valeur == 1
True
>>> m1.suivant.valeur == 8
False
>>> m1.suivant.suivant == None
False
>>> m1.suivant.suivant.suivant == None
True
```

9. L'instruction `ma_liste.tete.suivant = m8` convient, mais le résultat **n'est pas** une liste chaînée.
10. On complète la fonction `tri_dictatorial_chaine`.

```
def tri_dictatorial_chaine(chaine):
    maillon = chaine.tete
    while maillon.suivant is not None :
        if maillon.valeur <= maillon.suivant.valeur:
            maillon = maillon.suivant
        else:
            maillon.suivant = maillon.suivant.suivant
```

Partie C

11. Une pile est une structure linéaire LIFO last-in first-out, dans laquelle on peut empiler un élément à la fois, et dépiler le dernier élément empilé si la pile n'est pas vide.
12. On a remis les lignes de pseudo-code dans l'ordre en respectant les indentation.

    ⋆ si `p` n'est pas vide :
    - on crée une pile intermédiaire `p2` vide ;
    - on dépile `p`, on stocke l'élément obtenu dans la variable `dernier_conservé` et on l'empile dans `p2` ;
    - tant que `p` n'est pas vide :
        + si `candidat` est supérieure ou égal à `dernier_conservé` :
            − `dernier_conservé` prend la valeur de `candidat` et on l'empile dans `p2`
    - tant que `p2` n'est pas vide :
        + on dépile `p2` et on empile l'élément obtenu dans `p` ;

13. On écrit en Python la fonction `tri_dictatorial_pile` qui prend en paramètre `p` une instance de `Pile` et modifie cette pile afin qu'elle ne conserve que des éléments triés selon le pseudo-tri dictatorial.

```
def tri_dictatorial_pile(p):
    if not p.est_vide():
        p2 = Pile
        dernier_conserve = p.depiler()
        p2.empiler(dernier_conserve)
        while not p.est_vide():
            candidat = p.depiler()
            if candidat>=dernier_conserve:
                dernier_conserve=candidat
                p2.empile(candidat)
        while not p2.est_vide():
            p.empile(p2.depile())
```