25NSIPE4

Exercice 1 (6 points)

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

Dans cet exercice, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de SELECT, FROM, WHERE (avec les opérateurs logiques AND , OR), JOIN ... ON ;
- construire des requêtes d'insertion et de mise à jour à l'aide de UPDATE, INSERT, DELETE ;
- affiner les recherches à l'aide de ORDER BY.

Pour analyser les résultats et les performances de plusieurs joueurs et joueuses de tennis d'un club, on élabore une base de données relationnelle. Les données récoltées lors de plusieurs tournois, au fil des saisons, doivent ensuite permettre de fournir des statistiques. Chacune des requêtes demandées devra être écrite en langage SQL.

Voici un extrait de la table joueurs dans cette base :

joueurs			
id	nom	prenom	genre
1	Durand	Enzo	1
2	Panais	Lise	2
3	Alpin	Lucas	1
4	Benard	Lisa	2
5	Benard	Emma	2

- `id` est de type INT, cet attribut est la clé primaire de cette table ;
- `nom` est de type TEXT ;
- `prenom` est de type TEXT ;
- `genre` est de type INT (1 pour un joueur, 2 pour une joueuse).

1. Expliquer pourquoi l'attribut `nom` ne peut pas être choisi comme clé primaire.
2. Écrire une requête permettant d'obtenir les noms et prénoms des joueuses du club.
3. Écrire une requête permettant 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.

On s'intéresse maintenant à la table `competitions`, répertoriant les différents tournois auxquels ont participé les joueurs et joueuses du club.

competitions		
id	nom	annee
1	Open de Tours	2022
2	Tournoi de Blois	2023
3	Open de Toums	2023
4	Open de Nantes	2023
5	Open de Nantes	2021
6	Tournoi d'Angers	2024

- `id` est de type INT, il s'agit de la clé primaire de cette table ;
- `nom` est de type TEXT ;
- `annee` est de type INT.

4. Une faute de frappe s'est glissée dans le nom de la compétition d'identifiant 3. Écrire une requête permettant de corriger le nom en Open de Tours.
5. Écrire une requête permettant d'obtenir la liste des noms des tournois, ainsi que leur année, en triant par année croissante.

Le lien entre ces deux tables se fait à l'aide d'une table `participe`. Celle-ci contient aussi les statistiques recueillies lors de cette participation.

participe				
id_joueur	id_compet	nb_fautes	nb_gagnant	aces
1	1	16	12	1
3	1	14	8	2
1	3	7	15	2
3	3	12	8	1
2	2	15	10	3
2	4	10	17	0
4	4	7	18	4
5	4	11	15	1

- `id_joueur` est de type INT et est une clé étrangère se rattachant à la table `joueurs` ;
- `id_compet` est de type INT et est une clé étrangère se rattachant à la table `competitions` ;
- `nb_fautes` est de type INT et donne le nombre de fautes directes faites durant le tournoi ;
- `nb_gagnant` est de type INT et donne le nombre de coups gagnants réalisés durant le tournoi ;
- `aces` est de type INT et donne le nombre de services gagnants non touchés par l'adversaire durant le tournoi ;
- la clé primaire de la table est constituée du couple des attributs `id_joueur` et `id_compet`.

6. Écrire une requête permettant 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. Écrire une requête permettant d'obtenir le nom, le prénom des joueuses et le nom des compétitions auxquelles elles ont participé en 2023.
8. On souhaite supprimer de la table `joueurs` la joueuse Emma Benard qui ne fait plus partie du club. Déterminer quelle précaution on doit prendre avant de pouvoir le faire. Justifier.
9. Une joueuse du club, de prénom Agathe et de nom Turion, a participé au Tournoi de Blois en 2024, où elle a fait 14 fautes directes, réalisé 15 coups gagnants et servi 2 aces. Écrire les différentes requêtes, dans le bon ordre, permettant d'insérer cette joueuse 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.

Exercice 2 (6 points)

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

Un aéroport dispose d'un réseau informatique décomposé en différents sous-réseaux :

- Navigation (N) : utilisé principalement par la tour de contrôle ;
- Guichets (G) : utilisé aux guichets dans le hall ;
- Achats (A) : utilisé sur les bornes d'achat placées dans le hall ;
- Sécurité (S) : utilisé aux contrôles de sécurité ;
- Portes (P) : utilisé au niveau des portes d'accès aux avions ;
- Bagages (B) : utilisé par les services qui gèrent le transit des bagages ;
- Commerces (C) : utilisé par tous les commerces.

Le réseau possède l'architecture suivante, où R1, R2, R3, R4, R5, R6 et R7 sont des routeurs :

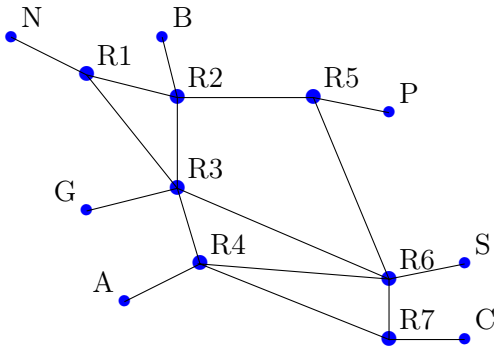


Figure 1. Schéma du réseau.

Partie A : Réseau et adressage

On souhaite ajouter des machines sur le sous-réseau Commerces sur lequel sont déjà connectées 207 machines. L'adresse du sous-réseau est 137.254.128.0 et le masque de sous-réseau utilisé est 255.255.255.0 (l'adresse IP du réseau est donc 137.254.128.0/24 en notation CIDR). On rappelle que cela signifie que les adresses IP du réseau ont toutes en commun leurs 24 premiers bits lorsque les adresses IP sont écrites en binaire.

À part le routeur, toutes les machines déjà présentes sur le sous-réseau sont numérotées dans l'ordre croissant en partant de la plus petite IP disponible.

1. Parmi les deux adresses IP suivantes : 137.254.128.200 et 137.254.128.210, donner l'adresse IP de la machine déjà connectée au sous-réseau Commerces.
2. Préciser s'il est possible ou non d'ajouter 132 machines sur le sous-réseau Commerces, en justifiant la réponse.

Partie B : Programmation d'un protocole de routage

Dans la suite de l'exercice, pour simplifier, on ne considère que les routeurs. Les tables de routage simplifiées sont données dans le tableau suivant, précisant pour chaque routeur en tête de colonne, la passerelle (c'est-à-dire le routeur à contacter) correspondant au routeur destination en début de ligne.

	Source	R1	R2	R3	R4	R5	R6	R7
Destination	R1		R1	R1	R6	R2	R4	R4
	R2	R2		R2	R3	R2	R5	R6
	R3	R3	R3		R3	R6	R3	R4
	R4	R3	R3	R4		R6	R4	R4
	R5	R2	R5	R2	R6		R5	R6
	R6	R2	R5	R6	R6	R6		R6
	R7	R3	R3	R6	R7	R6	R7	

Ainsi, selon ce tableau, si le routeur R3 reçoit des données à transmettre au routeur R5, il enverra ses données au routeur R2.

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.
4. Décrire le problème rencontré lorsque qu'une machine du sous-réseau Commerces envoie des données à destination d'une machine du sous-réseau Navigation.

Pour éviter ce problème, on veut reconfigurer les routeurs en réécrivant leurs tables de routage à l'aide d'un programme. Pour y parvenir, on modélisera le réseau par un graphe.

Dans toute la suite, les sommets du graphe, qui représenteront les routeurs du réseau, seront décrits par leur nom (type str) et un graphe sera représenté par un dictionnaire associant à chaque sommet la liste des sommets qui lui sont liés par une arête.

Pour la prochaine question uniquement, on considère le réseau obtenu en se limitant aux routeurs R1, R2, R3 et R5. On obtient alors le réseau suivant :

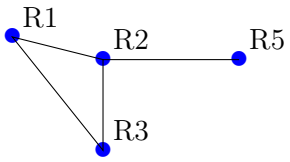


Figure 2. Schéma du réseau restreint aux routeurs R1, R2, R3 et R5.

5. Donner le dictionnaire correspondant au réseau de la Figure 2.
6. Rappeler le principe d'une fonction récursive.

Pour remplir les tables de routage en évitant le problème soulevé à la question 4, on souhaite utiliser le protocole RIP, qui minimise le nombre de routeurs par lesquels les paquets transitent. Une première idée est de construire la liste de tous les chemins possibles reliant ces deux routeurs puis de choisir un chemin le plus court possible dans cette liste.

On suppose que l'on dispose d'une fonction `liste_chemins(graphe, r_depart, r_arrivee)` qui prend en paramètres un graphe, un routeur de départ et un routeur d'arrivée et qui renvoie la liste de tous les chemins liant les deux routeurs, les chemins étant représentés par les listes des routeurs par lesquels passer.

En notant `g` le graphe écrit à la question 5, on a donc :

```
>>> liste_chemins(g, 'R1', 'R5'))
[['R1', 'R2', 'R5'], ['R1', 'R3', 'R2', 'R5']]
```

On a besoin de connaître un chemin le plus court possible entre deux routeurs en utilisant le protocole RIP.

7. Écrire une fonction `plus_court_chemin(graphe, r_depart, r_arrivee)` qui renvoie une liste représentant un des plus courts chemins entre les routeurs `r_depart` et `r_arrivee` en utilisant le protocole RIP. On utilisera la fonction `liste_chemins` définie à la question précédente.

L'agent responsable du réseau consulte un informaticien au sujet de cette fonction. Il lui explique que cette fonction a un défaut : construire tous les chemins liant deux routeurs peut être long pour un réseau étendu. En effet, le nombre de chemins augmente de façon quasi exponentielle avec le nombre de routeurs. Pour remédier à ce problème et améliorer le temps d'exécution de la recherche d'un plus court chemin, l'informaticien lui propose d'utiliser une autre approche basée sur un parcours en largeur du graphe. En effet, avec un tel parcours, si un chemin est trouvé, il est forcément de longueur minimale.

8. Compléter la fonction `plus_court_chemin_largeur(graphe, r_depart, r_arrivee)` suivante qui traduit l'idée de l'informaticien, réalisant un parcours en largeur et dans laquelle le dictionnaire `dict_chemins` associe à un routeur le chemin reliant `r_depart` à ce routeur.

```
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:
        for s_r in graphe[r]:
            if not s_r in sommets_marques:
                sommets_marques.append(...)
                dict_chemins[s_r] = dict_chemins[r] + [s_r]
                if s_r == r_arrivee :
                    return ...
                L.append(s_r)
```

9. Écrire alors une fonction `table_routage(graphe, routeur)` qui renvoie la table de routage du routeur passé en paramètre sous la forme d'un dictionnaire associant à chaque routeur destination la passerelle orrespondante. On pourra utiliser les fonctions écrites dans les questions précédentes.

Partie C : Utilisation du protocole OSPF

Le réseau utilise trois types de connexion :

- Ethernet (E) : débit de 10 megabits par seconde ;
- Fast Ethernet (FE) : débit de 100 megabits par seconde ;
- Fibre (F) : débit de 500 megabits par seconde.

Les types de connexion sont reportés sur la figure du réseau suivante :

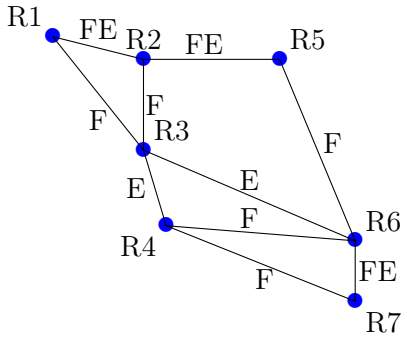


Figure 1. Types de connexions du réseau.

La qualité des liaisons entre les routeurs étant de natures différentes, on décide finalement d'opter pour un routage utilisant le protocole OSPF (Open Shortest Path First). On rappelle que le protocole OSPF configure les routeurs en privilégiant les routes dont le coût total est minimal, où le coût des connexions est donné par la formule suivante : coût $= \dfrac{10^9}{\text{débit}}$, où le débit est exprimé en bits par seconde.

10. Calculer le coût correspondant à chaque type de liaison.
11. Donner la liste des routeurs par lesquels transite un message envoyé depuis le routeur R1 à destination du routeur R7 en respectant le protocole OSPF.
12. Recopier et compléter la table de routage du routeur R2 toujours en respectant le protocole OSPF.

Destination	Suivant
R1	
R2	
R3	
R4	
R5	
R6	
R7	

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.

Le but de cet exercice est d'implémenter un algorithme de pseudo-tri, appelé le tri dictatorial.

L'exercice est constitué de trois parties indépendantes.

Pour chaque question, on peut considérer acquis les résultats et les fonctions demandés dans les questions précédentes, même sans les avoir traitées.

Le pseudo-tri dictatorial d'une série d'entiers suit le principe suivant :

- s'il n'y a aucun ou un seul élément, la série est considérée comme triée et n'est donc pas modifiée ;
- sinon :
  - on conserve le premier élément de la série ;
  - pour chaque élément de la série à partir du deuxième :
    - si l'élément est plus petit que le dernier élément conservé alors on l'élimine ;
    - sinon on le conserve.

Par exemple, si on considère la série 2, 3, 1, 8 :

- on conserve le 2 qui est le premier élément ;
- le 3 n'est pas plus petit que le dernier conservé (qui est 2) donc on le conserve ;
- le 1 est plus petit que le dernier conservé (qui est 3) donc on l'élimine ;
- le 8 n'est pas plus petit que le dernier conservé (qui est toujours 3) donc on le conserve.

La série triée obtenue après cet algorithme est donc 2, 3, 8.

Partie A

Dans cette partie, on implémente le tri dictatorial en utilisant le type list de Python pour représenter une série d'entiers.

On souhaite coder une fonction `tri_dictatorial` qui :

- prend en paramètre une liste d'entiers `serie` de type list ;
- renvoie une nouvelle liste d'entiers obtenue en suivant l'algorithme présenté en introduction, c'est-à-dire une liste triée, éventuellement vide, ne contenant que les éléments de `serie` à conserver ;
- ne modifie pas `serie`.

Par exemple, si `s = [5, 2, 6, 8, 3, 7]`, l'appel `tri_dictatorial(s)` devrait renvoyer la liste `[5, 6, 8]` sans modifier `s`.

On remarque que la liste obtenue est en effet triée.

1. Donner le résultat que doit renvoyer l'appel : `tri_dictatorial([31, 45, 41, 28, 37, 108, 127, 2, 124, 4`
2. Expliquer pourquoi le tri dictatorial n'est pas un algorithme de tri.

Edgar a écrit le programme suivant, qui prétend implémenter le tri dictatorial :

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

Edgar souhaite tester si sa fonction fait bien ce qu'elle est censée faire.

3. Edgar réalise l'appel `tri_dictatorial([8, 2, 9, 6, 12])`.
   Expliquer pas à pas comment la liste `serie_triee` se construit après cet appel.
4. Edgar réalise maintenant l'appel `tri_dictatorial([])` et obtient l'erreur suivante :

```
Traceback (most recent call last):
  File "tri_edgar.py", line 8, in <module>
    tri_dictatorial([])
  File "tri_edgar.py", line 2, in tri_dictatorial
    result = [serie[0]]
IndexError: list index out of range
```

Expliquer précisément l'erreur obtenue et proposer une modification du code d'Edgar afin que cet appel soit conforme à l'algorithme du tri dictatorial décrit en introduction.

Dijkstra lors de la réception de son prix Turing en 1972, a notamment déclaré :

« Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. »

que l'on peut traduire par

« Tester les programmes peut être un moyen très efficace d'y trouver des bugs, mais c'est un moyen désespérément inadéquat pour prouver leur absence. »

5. Expliquer pourquoi des tests ne peuvent pas prouver de façon certaine l'absence de bugs d'un programme en général.

Edgar décide de procéder à un test supplémentaire et réalise l'appel `tri_dictatorial([8, 2, 3, 5, 12])`. La fonction renvoie alors `[8, 3, 5, 12]` qui n'est pas une liste triée.

6. Expliquer la cause du problème et proposer une modification du code d'Edgar afin de la corriger.

Partie B

Dans cette partie, on implémente le tri dictatorial sur des listes chaînées. Cette fois-ci on va modifier la liste chaînée initiale au lieu de construire une nouvelle liste.

On dispose d'une classe `Maillon` :

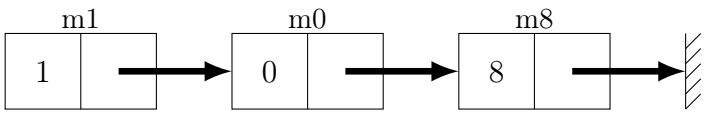
```
1  class Maillon:
2      def __init__(self, val, suiv):
3          self.valeur = val
4          self.suivant = suiv
```

L'attribut `suivant` doit correspondre à un `Maillon` (le suivant de `self`), ou à `None` si `self` est le dernier.

On dispose également d'une classe `Liste` qui implémente une liste chaînée avec pour unique attribut `tete` qui est le maillon de tête de la liste chaînée, une instance de `Maillon` :

```
class Liste:
    def __init__(self, tete):
        self.tete = tete
```

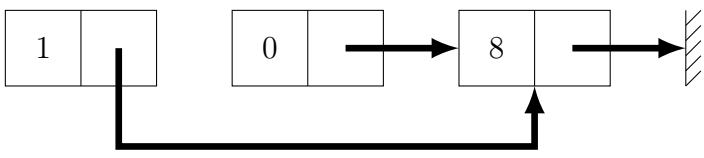
On peut représenter graphiquement une liste chaînée de la manière suivante, avec la barre à hachure symbolisant la valeur `None` :



7. Donner des instructions permettant de construire les trois maillons `m1`, `m0` et `m8` et la liste chaînée représentés ci-dessus. On nommera la liste chaînée `ma_liste`.
8. Indiquer ce que renvoie chacune des instructions ci-dessous :

```
m1.valeur == 1
m1.suivant.valeur == 8
m1.suivant.suivant == None
m1.suivant.suivant.suivant == None
```

9. Donner une instruction permettant de transformer `ma_liste` en la liste chaînée représentée ci-dessous :



On souhaite à présent une fonction `tri_dictatorial_chaine` qui prend en paramètre une instance de liste chaînée `chaine` et qui modifie cette liste chaînée démarrant en suivant l'algorithme du pseudo-tri dictatorial. La fonction ne renvoie rien.

10. Recopier et compléter la fonction `tri_dictatorial_chaine` ci-dessous.

```
def tri_dictatorial_chaine(chaine):
    maillon = chaine.tete
    while maillon.suivant ... :
        if maillon.valeur ...
            maillon = ...
        else:
            maillon.suivant = ...
```

Partie C

Une pile `p`, éventuellement vide, stocke des éléments entiers qu'on souhaite trier selon le pseudo-tri dictatorial. À l'issue du tri, on veut que cette pile soit modifiée et ne contienne plus que des éléments triés.

11. Rappeler le principe du fonctionnement d'une pile.
12. Remettre dans l'ordre les lignes ci-dessous afin d'obtenir l'algorithme attendu, en respectant une tabulation lorsque la ligne est à l'intérieur d'un bloc si ou tant que.

— si `p` n'est pas vide :
- tant que `p` n'est pas vide :
- tant que `p2` n'est pas vide :
- on dépile `p`, on stocke l'élément obtenu dans la variable `dernier_conservé` et on l'empile dans `p2` ;
- on crée une pile intermédiaire `p2` vide ;

— on dépile `p` et on stocke l'élément obtenu dans la variable `candidat` ;
— si `candidat` est supérieure ou égal à `dernier_conservé` :
— on dépile `p2` et on empile l'élément obtenu dans `p` ;
- `dernier_conservé` prend la valeur de `candidat` et on l'empile dans `p2`

On suppose maintenant que l'on dispose d'une classe `Pile` implémentant une structure de pile. L'appel `help(Pile)` entraîne l'affichage suivant :

```
Help on class Pile in module __main__:
class Pile(builtins.object)
 | Methods defined here:
 |
 | __init__(self)
 |
Initialize self. See help(type(self)) for accurate
signature.
 |
 | __str__(self)
 |
Return str(self).
 |
 | depiler(self)
 |
 | empiler(self, elt)
 |
 | est_vide(self)
```

13. Écrire 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.