Métropole - juin 2021 - sujet 1

Exercice 1 (SQL - 4 points)
Dans cet exercice, on pourra utiliser les mots clés suivants du langage SQL :
`SELECT`, `FROM`, `WHERE`, `JOIN`, `ON`, `INSERT INTO`, `VALUES`, `MIN`, `MAX`, `OR`, `AND`.
Les fonctions d'agrégation `MIN(propriete)` et `MAX(propriete)` renvoient, respectivement, la plus petite et la plus grande valeur de l'attribut `propriete` pour les enregistrements sélectionnés.

Des acteurs ayant joué dans différentes pièces de théâtre sont recensés dans une base de données `Theatre` dont le schéma relationnel est donné ci-dessous :

`Piece (idPiece, titre, langue)`

`Acteur (idActeur, nom, prenom, anneeNaiss)`

`Role (#idPiece, #idActeur, nomRole)`

Dans ce schéma, les clés primaires sont soulignées et les clés étrangères sont précédées du symbole `#`.
L'attribut `idPiece` de la relation `Role` est une clé étrangère faisant référence à l'attribut `idPiece` de la relation `Piece`.
L'attribut `idActeur` de la relation `Role` est une clé étrangère faisant référence à l'attribut `idActeur` de la relation `Acteur`.
Tous les attributs dont le nom est préfixé par `id` sont des nombres entiers ainsi que l'attribut `anneeNaiss`. Les autres attributs sont des chaînes de caractères.

1. Expliquer pourquoi il n'est pas possible d'insérer une entrée dans la relation `Role` si les relations `Piece` et `Acteur` sont vides.
2. Dans la pièce « Le Tartuffe », l'acteur Micha Lescot a joué le rôle de Tartuffe.
   L'identifiant de Micha Lescot est 389761 et celui de cette pièce est 46721.
   Ecrire une requête SQL permettant d'ajouter ce rôle dans la table (ou relation) `Role`.
3. Expliquer ce que fait la requête SQL suivante :

```
SELECT titre
FROM Livres
WHERE auteur = 'Molière'
```

4. Pour chacun des quatre items suivants, écrire une requête SQL permettant d'extraire les informations demandées.
   (a) Le nom et prénom des artistes nés après 1990.
   (b) L'année de naissance du plus jeune artiste.
   (c) Le nom des rôles joués par l'acteur Vincent Macaigne.
   (d) Le titre des pièces écrites en Russe dans lesquelles l'actrice Jeanne Balibar a joué.

Exercice 2 (Piles et POO - 4 points)

On crée une classe `Pile` qui modélise la structure d'une pile d'entiers. Le constructeur de la classe initialise une pile vide. La définition de cette classe sans l'implémentation de ses méthodes est donnée ci-dessous.

```
class Pile:
    def __init__(self):
        """Initialise la pile comme une pile vide."""

    def est_vide(self):
        """Renvoie True si la liste est vide, False sinon."""

    def empiler(self, e):
        """Ajoute l'élément e sur le sommet de la pile, ne renvoie rien."""

    def depiler(self):
        """Retire l'élément au sommet de la pile et le renvoie."""

    def nb_elements(self):
        """Renvoie le nombre d'éléments de la pile."""

    def afficher(self):
        """Affiche de gauche à droite les éléments de la pile, du fond de la pile
        vers son sommet. Le sommet est alors lélément affiché le plus à droite.
        Les éléments sont séparés par une virgule. Si la pile est vide la méthode
        affiche 'pile vide'."""
```

Seules les méthodes de la classe ci-dessus doivent être utilisées pour manipuler les objets `Pile`.

1. (a) Ecrire une suite d'instructions permettant de créer une instance de la classe `Pile` affectée à une variable `pile1` contenant les éléments 7, 5 et 2 insérés dans cet ordre.
   Ainsi, à l'issue de ces instructions, l'instruction `pile1.afficher()` produit l'affichage : `7, 5, 2`.

   (b) Donner l'affichage produit après l'exécution des instructions suivantes :

```
element1 = pile1.depiler()
pile1.empiler(5)
pile1.empiler(element1)
pile1.afficher()
```

2. On donne la fonction `mystere` suivante :

```
def mystere(pile, element):
    pile2 = Pile()
    nb_elements = pile.nb_elements()
    for i in range(nb_elements):
        elem = pile.depiler()
        pile2.empiler(elem)
        if elem == element:
            return pile2
    return pile2
```

   (a) Dans chacun des quatre cas suivants, quel est l'affichage obtenu dans la console ?

Cas 1	Cas 2

`>>> pile.afficher()`
`7, 5, 2, 3`
`>>> mystere(pile, 2).afficher()`
	`>>> pile.afficher()`
	`7, 5, 2, 3`
	`>>> mystere(pile, 9).afficher()`

Cas 3	**Cas 4**

`>>> pile.afficher()`
`7, 5, 2, 3`
`>>> mystere(pile, 3).afficher()`
	`>>> pile.est_vide()`
	`True`
	`>>> mystere(pile, 3).afficher()`

(b) Expliquer ce que permet d'obtenir la fonction `mystere`.

3. Ecrire une fonction `etendre` qui prend en arguments deux objets `Pile` appelés `pile1` et `pile2` et qui modifie `pile1` en lui ajoutant les éléments de `pile2` rangés dans l'ordre inverse. Cette fonction ne renvoie rien.
On donne ci-dessous les résultats attendus pour certaines instructions :

```
INSERT INTO Plat
VALUES(58, 'Pêche Melba', 'dessert', 'Pêches et glace vanille', 6.5)
```

4. Ecrire une fonction `supprime_toutes_occurences` qui prend en arguments un objet `Pile` appelé `pile` et un élément `element` et supprime tous les éléments `element` de `pile`.
On donne ci-dessous les résultats attendus pour certaines instructions :

```
def supprime_toutes_occurences(pile, element):
    p2 = Pile()
    while not pile.est_vide():
        x = pile.depiler()
        if x != element:
            p2.empiler(x)
    while not p2.est_vide():
        x = p2.depiler()
        pile.empiler(x)
```

Exercice 3 (Processus et routage - 4 points) *Les parties A et B sont indépendantes.*

Partie A : processus.

La commande UNIX `ps` présente un cliché instantané des processus en cours d'exécution.
Avec l'option `-eo pid,ppid,stat,command`, cette commande affiche dans l'ordre l'identifiant du processus PID (process identifier), le PPID (parent process identifier), l'état STAT et le nom de la commande à l'origine du processus.
Les valeurs du champ STAT indique l'état des processus :

- R : processus en cours d'exécution ;
- S : processus endormi.

Sur un ordinateur, on exécute la commande `ps -eo pid,ppid,stat,command` et on obtient un affichage dont on donne ci-dessous un extrait :

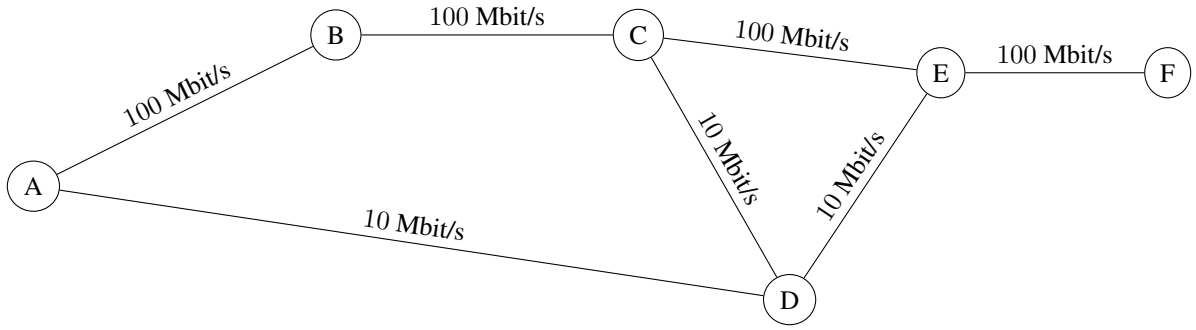
```
$ ps -eo pid,ppid,stat,command

  PID  PPID  STAT  COMMAND
  1    0     Ss    /sbin/init
  .... ....  ....  ....
  1912 1908  Ss    Bash
  2014 1912  Ss    Bash
  1920 1747  Sl    Gedit
  2013 1912  Ss    Bash
  2091 1593  Sl    /usr/lib/firefox/firefox
  5437 1912  Sl    python programme1.py
  5440 2013  R     python programme2.py
  5450 1912  R+    ps -eo pid,ppid,stat,command
```

À l'aide de cet affichage, répondre aux questions ci-dessous.

1. Quel est le nom de la première commande exécutée par le système d'exploitation lors du démarrage ?
2. Quels sont les identifiants des processus actifs sur cet ordinateur au moment de l'appel de la commande `ps` ? Justifier la réponse.
3. Depuis quelle application a-t-on exécuté la commande `ps` ? Donner les autres commandes qui ont été exécutées à partir de cette application.
4. Expliquer l'ordre dans lequel les deux commandes `python programme1.py` et `python programme2.py` ont été exécutées.
5. Peut-on prédire que l'une des deux commandes `python programme1.py` et `python programme2.py` finira avant l'autre ?

Partie B : routage.

On considère le réseau modélisé par le schéma ci-dessous. Les routeurs sont identifiés par les lettres de A à F ; les débits des liaisons entre les routeurs sont indiqués sur le schéma.



1. Dans cette question, tous les routeurs utilisent le protocole RIP (distance en nombre de sauts).
   On s'intéresse aux routes utilisées pour rejoindre F une fois les tables stabilisées.
   Recopier et compléter sur la copie la table suivante :

Machine	Prochain saut	Distance
A		
B		
C		
D		
E		

2. Dans cette question tous les routeurs utilisent le protocole OSPF (distance en coût des routes). Le coût d'une liaison est modélisé par la formule

$$\frac{10^8}{d},$$

où d est le débit de cette liaison exprimé en bit par seconde.
On s'intéresse aux routes utilisées pour rejoindre F une fois les tables stabilisées.
Recopier et compléter sur la copie la table suivante :

Machine	Prochain saut	Coût
A		
B		
C		
D		
E		

3. Des protocoles RIP et OSPF, lequel fournit le routage entre A et F le plus performant en terme de débit ? Justifier la réponse.

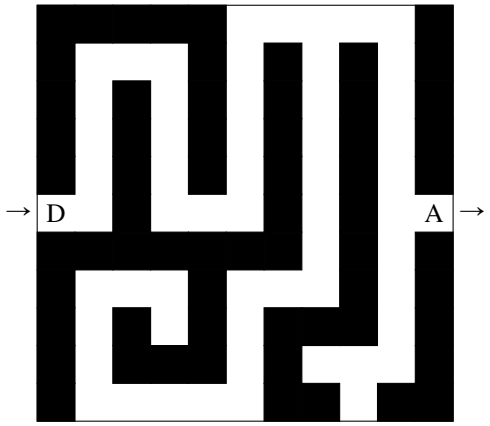
Exercice 4 (Tableaux - 4 points)

Partie A : représentation d'un labyrinthe.

On modélise un labyrinthe par un tableau à deux dimensions à n lignes et m colonnes avec n et m des entiers strictement positifs.
Les lignes sont numérotées de 0 à $n-1$ et les colonnes de 0 à $m-1$.
La case en haut à gauche est repérée par $(0,0)$ et la case en bas à droite par$(n-1,m-1)$.
Dans ce tableau :

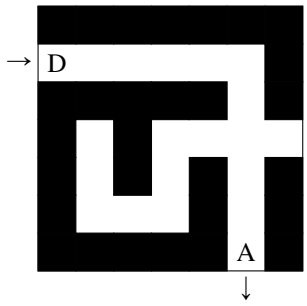
- ⋆ 0 représente une case vide, hors case de départ et arrivée,
- ⋆ 1 représente un mur,
- ⋆ 2 représente le départ du labyrinthe,
- ⋆ 3 représente l'arrivée du labyrinthe.

Ainsi, en `Python`, le labyrinthe ci-dessous est représentée par le tableau de tableaux `lab1`.




```
lab1 = [[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [2, 0, 1, 0, 0, 0, 1, 0, 1, 0, 3],
        [1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1]]
```

1. Le labyrinthe ci-dessous est censé être représenté par le tableau de tableaux `lab2`. Cependant, dans ce tableau, un mur se trouve à la place du départ du labyrinthe. Donner une instruction permettant de placer le départ au bon endroit dans `lab2`.




```
lab2 = [[1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 0, 1],
        [1, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 1],
        [1, 1, 1, 1, 1, 3, 1]]
```

2. Ecrire une fonction `est_valide` qui prend en argument quatre entiers `i`, `j`, `n` et `m`, et qui renvoie `True` si le couple `(i,j)` correspond à des coordonnées valides pour un labyrinthe de taille `(n,m)`, et `False` sinon. On donne ci-dessous des exemples d'appels :

```
def est_valide(i, j, n, m):
    return i >= 0 and j >= 0 and i < n and j < m
```

3. On suppose que le départ d'un labyrinthe est toujours indiqué, mais on ne fait aucune supposition sur son emplacement. Compléter la fonction `depart` ci-dessous de sorte qu'elle renvoie, sous la forme d'un tuple, les coordonnées du départ d'un labyrinthe (représenté par le paramètre `lab`). Par exemple, l'appel `depart(lab1)` doit renvoyer le tuple `(5, 0)`.

```
def depart(lab) :
    n = len(lab)
    m = len(lab[0])
    for i in range(n):
        for i in range(m):
            if lab[i][j] == 2:
                return (i,j)
```

4. Ecrire une fonction `nb_cases_vides` qui prend en argument un tableau `lab` représentant un labyrinthe et qui renvoie le nombre de cases vides de `lab` (comprenant donc l'arrivée et le départ). Par exemple, l'appel `nb_cases_vides(lab2)` doit renvoyer la valeur `19` (une fois que la case correspond au départ a été remplacée par la valeur `2`).

Partie B : recherche d'une solution dans un labyrinthe.

On suppose dans cette partie que les labyrinthes possèdent un unique chemin allant du départ à l'arrivée sans repasser par la même case. Dans la suite, c'est ce chemin que l'on appellera solution du labyrinthe.

Pour déterminer la solution d'un labyrinthe, on parcourt les cases vides de proche en proche. Lors d'un tel parcours, afin d'éviter de tourner en rond, on choisit de marquer les cases visitées. Pour cela, on remplace la valeur d'une case visitée dans le tableau représentant le labyrinthe par la valeur 4.

1. On dit que deux cases d'un labyrinthe sont voisines si elles ont un côté commun. On considère une fonction `voisines` qui prend en arguments deux entiers `i` et `j` représentant les coordonnées d'une case et un tableau `lab` qui représente un labyrinthe. Cette fonction renvoie la liste des coordonnées des cases voisines de la case de coordonnées (i,j) qui sont valides, non visitées et qui ne sont pas des murs. L'ordre des éléments de cette liste n'importe pas.
   Ainsi, l'appel `voisines(1, 1, [[1, 1, 1], [4, 0, 0], [1, 0, 1]])` renvoie la liste `[(2, 1), (1, 2)]`.
   Que renvoie l'appel `voisines(1, 2, [[1, 1, 4], [0, 0, 0], [1, 1, 0]])` ?
2. On souhaite stocker la solution dans une liste `chemin`. Cette liste contiendra les coordonnées des cases de la solution, dans l'ordre. Pour cela, on procède de la façon suivante :
   - ⋆ Initialement :
     - déterminer les coordonnées du départ : c'est la première case à visiter ;
     - ajouter les coordonnées de la case départ à la liste `chemin`.
   - ⋆ Tant que l'arrivée n'a pas été atteinte :
     - on marque la case visitée avec la valeur 4 ;
     - si la case visitée possède une case voisine libre, la première case de la liste renvoyée par la fonction voisines devient la prochaine case à visiter et on ajoute à la liste `chemin` ;
     - sinon, il s'agit d'une impasse. On supprime alors la dernière case dans la liste `chemin`. La prochaine case à visiter est celle qui est désormais en dernière position de la liste `chemin`.

   (a) Le tableau de tableaux `lab3` ci-dessous représente un labyrinthe :

```
lab3 = [[1, 1, 1, 1, 1, 1],
        [2, 0, 0, 0, 0, 3],
        [1, 0, 1, 0, 1, 1],
        [1, 1, 1, 0, 0, 1]]
```

   La suite d'instructions ci-dessous simule le début des modifications subies par la liste `chemin` lorsque l'on applique la méthode présentée :

```
chemin.append((3, 3))
chemin.append((3, 4))
chemin.pop()
chemin.pop()
chemin.pop()
chemin.append((1, 4))
chemin.append((1, 5))
```

   Compléter cette suite d'instructions jusqu'à ce que la liste `chemin` représente la solution.
   Rappel : la méthode `pop` supprime le dernier élément d'une liste et renvoie cet élément.

   (b) Recopier et compléter la fonction `solution` donnée ci-dessous de sorte qu'elle renvoie le chemin solution du labyrinthe représenté par le paramètre `lab`. On pourra pour cela utiliser la fonction `voisines`.

```
def solution(lab):
    chemin = [depart(lab)]
    case = chemin[0]
    i = case[0]
    j = case[1]
    while lab[i][j] != 3: # tant que ce n'est pas l'arrivée
        lab[i][j] = 4 # on marque la case visitée
        voisins = voisines(i, j, lab)
        if voisins == [] : # cas de l'impasse
            chemin.pop()
            i,j = chemin[len(chemin)-1] # retour en arrière
        else:
            i, j = voisins[0]
            chemin.append((i,j))
    return chemin
```

Par exemple, l'appel `solution(lab2)` doit renvoyer `[(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5)]`.

Exercice 5 (Tableaux et récursivité - 4 points)

Dans un tableau `Python` d'entiers `tab`, on dit que le couple d'indices `(i,j)` forme une inversion lorsque `i < j` et `tab[i] > tab[j]`. On donne ci-dessous quelques exemples :

- Dans le tableau `[1, 5, 3, 7]`, le couple d'indices `(1,2)` forme une inversion car `5 > 3`. Par contre, le couple `(1,3)` ne forme pas d'inversion car `5 < 7`. Il n'y a qu'une inversion dans ce tableau.
- Il y a trois inversions dans le tableau `[1, 6, 2, 7, 3]`, à savoir les couples d'indices `(1, 2)`, `(1, 4)` et `(3, 4)`.
- On peut compter six inversions dans le tableau `[7, 6, 5, 3]` : les couples d'indices `(0, 1)`, `(0, 2)`, `(0, 3)`, `(1, 2)`, `(1, 3)` et `(2, 3)`.

On se propose dans cet exercice de déterminer le nombre d'inversions dans un tableau quelconque.

Questions préliminaires.

1. Expliquer pourquoi le couple `(1, 3)` est une inversion dans le tableau `[4, 8, 3, 7]`.
2. Justifier que le couple `(2, 3)` n'en est pas une.

Partie A : méthode itérative.

Le but de cette partie est d'écrire une fonction itérative `nombre_inversion` qui renvoie le nombre d'inversions dans un tableau. Pour cela, on commence par écrire une fonction `fonction1` qui sera ensuite utilisée pour écrire la fonction `nombre_inversion`.

1. On donne la fonction suivante :

```
def fonction1(tab, i):
    nb_elem = len(tab)
    cpt = 0
    for j in range(i+1, nb_elem):
        if tab[j] < tab[i]:
            cpt += 1
    return cpt
```

   (a) Indiquer ce que renvoie l'appel `fonction1(tab, i)` dans les cas suivants :
   - Cas 1 : `tab = [1, 5, 3, 7]` et `i = 0`.
   - Cas 2 : `tab = [1, 5, 3, 7]` et `i = 1`.
   - Cas 3 : `tab = [1, 5, 2, 6, 4]` et `i = 1`.

   (b) Expliquer ce que permet de déterminer cette fonction.

2. En utilisant la fonction précédente, écrire une fonction `nombre_inversion` qui prend en argument un tableau `tab` et renvoie le nombre d'inversions dans ce tableau.
   On donne ci-dessous les résultats attendus pour certains appels :

```
def nombre_inversions(tab):
    nb_inv = 0
    n = len(tab)
    for i in range(n-1):
        nb_inv = nb_inv + fonction1(tab, i)
    return nb_inv
```

3. Quelle est l'ordre de grandeur de la complexité en temps de l'algorithme obtenu ? Aucune justification n'est attendue.

Partie B : méthode récursive.

Le but de cette partie est de concevoir une version récursive de la fonction `nombre_inversion`. On définit pour cela des fonctions auxiliaires.

1. Donner le nom d'un algorithme de tri ayant une complexité meilleure que quadratique.

Dans la suite de cet exercice, on suppose qu'on dispose d'une fonction `tri` qui prend en argument un tableau `tab` et renvoie un tableau contenant les mêmes éléments rangés dans l'ordre croissant.

2. Ecrire une fonction `moitie_gauche` qui prend en argument un tableau `tab` et renvoie un nouveau tableau contenant la moitié gauche de `tab`. Si le nombre d'éléments de `tab` est impair, l'élément du centre se trouve dans cette partie gauche.
   On donne ci-dessous les résultats attendus pour certains appels :

```
def moitie_gauche(tab):
    n = len(tab)
    nvx_tab = []
    if n == 0:
        return []
    mil = n//2
    if n%2 == 0:
        lim = mil
    else :
        lim = mil+1
    for i in range(lim):
        nvx_tab.append(tab[i])
    return nvx_tab
```

Dans la suite, on suppose qu'on dispose de la fonction `moitie_droite` qui prend en argument un tableau `tab` et qui renvoie la moitié droite sans l'élément du milieu.

3. On suppose qu'une fonction `nb_inv_tab` prenant en argument deux tableaux `tab1` et `tab2` a été écrite. Cette fonction renvoie le nombre d'inversions du tableau obtenu en mettant bout à bout les tableaux `tab1` et `tab2`, à condition que `tab1` et `tab2` soient triés dans l'ordre croissant.
   On donne ci-dessous deux exemples d'appel de cette fonction :

```
def nb_inversions_rec(tab):
    if len(tab) > 1:
        g = moitie_gauche(tab)
        d = moitie_droite(tab)
        return nb_inv_tab(tri(g),tri(d))+nb_inversions_rec(g)+nb_inversions_rec(d)
    else:
        return 0
```

   En utilisant la fonction `nb_inv_tab` et les questions précédentes, écrire une fonction récursive `nb_inversions_rec(tab)` qui permet de calculer le nombre d'inversions dans un tableau. Cette fonction renverra le même nombre que `nombre_inversions(tab)` de la partie A. On procédera de la façon suivante :

   - Séparer le tableau en deux tableaux de tailles égales (à une unité près).
   - Appeler récursivement la fonction `nb_inversions_rec` pour compter le nombre d'inversions dans chacun des deux tableaux.
   - Trier les deux tableaux (on rappelle qu'une fonction de tri est déjà définie).
   - Ajouter au nombre d'inversions précédemment comptées le nombre renvoyé par la fonction `nb_inv_tab` avec pour arguments les deux tableaux triés.