X/ENS Informatique B MP-PC-PSI 2018 — Corrigé

Ce corrigé est proposé par Emma Kerinec (ENS Lyon) ; il a été relu par Cyril Ravat (professeur en CPGE) et Vincent Puyhaubert (professeur en CPGE).

Ce sujet regroupe une collection de requêtes SQL élémentaires et leur traduction en algèbre relationnelle, qu'il faut implémenter en Python. Aucune connaissance théorique spécifique n'est requise ; seule la maîtrise primaire de la syntaxe SQL est nécessaire. Le sujet est idéal pour s'entraîner sur des requêtes SQL simples, ainsi que sur les principes basiques des algorithmes et leur implémentation en Python. La première partie doit être abordée avant les deux suivantes, lesquelles peuvent en revanche être traitées indépendamment l'une de l'autre.

- La première partie introduit l'utilisation des listes et des tables afin de définir des fonctions servant de briques de base pour les parties suivantes, notamment la jointure de tables et la sélection d'éléments selon différents critères.
- La deuxième partie met en évidence le lien entre ces fonctions et les principales requêtes SQL ; il faut alors se servir de la partie précédente pour implémenter quelques requêtes SQL simples.
- La dernière partie propose d'améliorer les fonctions définies dans la première partie en exploitant des propriétés de tri concernant les tables ou grâce à l'introduction de dictionnaires.

Si ce sujet est d'une difficulté moyenne, il a sans doute déconcerté les candidats ayant fait l'impasse sur le langage SQL. Il introduit efficacement les concepts utilisés, notamment la structure hors-programme de dictionnaire, ainsi que tous les outils associés auxquels doivent se restreindre les candidats. Malgré l'évocation de tris dans la dernière partie, ce sujet est abordable en fin de première année.

Indications

Partie I

I.5 Observer que les enregistrements d'une table sont indépendants.

I.6 Parcourir les deux tables et utiliser la concaténation.

I.7 Parcourir les deux tables et comparer les bons attributs, ne pas oublier de supprimer l'attribut associé au second indice.

I.9 Comparer chaque enregistrement de la table à tous ceux déjà retenus dans le résultat.

Partie II

II.1 Utiliser la fonction `SelectionConstante` définie dans la partie précédente.

II.2 L'opération demandée est le produit cartésien de `Trajet` et `Véhicule`.

II.3 On peut enchaîner les fonctions élémentaires.

II.4 Attention aux indices des colonnes après jointure.

II.5 Il vaut mieux faire une double jointure suivie d'une unique sélection.

II.6 Penser à réutiliser le résultat précédent, sous forme de jointure.

Partie III

III.1 Parcourir la table en comparant les enregistrements consécutifs.

III.2 Quel est l'algorithme classique de recherche sur les listes triées ?

III.3 Parcourir en parallèle les deux tables et les synchroniser.

I.1 On parcourt tous les enregistrements de `table`, et on ajoute dans le résultat ceux dont la valeur pour l'indice `indice` vaut `constante`.

```
def SelectionConstante(table, indice, constante):
    resultat = []
    for enr in table:
        if enr[indice] == constante:
            resultat.append(enr)
    return resultat
```

I.2 Dans la fonction précédente, il n'y a pas d'opération coûteuse en dehors de la boucle `for`. Celle-ci s'exécute `len(table)` fois et on effectue deux opérations élémentaires au plus à chaque étape. Ainsi,

La complexité de la fonction `SelectionConstante` est en O(`len(table)`).

I.3 Parcourons tous les enregistrements de la table `table` et ajoutons dans le résultat ceux dont la valeur pour l'indice `indice1` est égale à celle pour l'indice `indice2`.

```
def SelectionEgalite(table, indice1, indice2):
    resultat = []
    for enr in table:
        if enr[indice1] == enr[indice2]:
            resultat.append(enr)
    return resultat
```

I.4 Il suffit de parcourir la liste `listeIndices` en créant un nouvel enregistrement auquel on ajoute les valeurs de `enregistrement` pour les indices souhaités, donnés dans `listeIndices`.

```
def ProjectionEnregistrement(enregistrement, listeIndices):
    resultat = []
    for indice in listeIndices:
        resultat.append(enregistrement[indice])
    return resultat
```

I.5 Il suffit d'appliquer la fonction précédente à tous les éléments de `table`, car ceux-ci sont indépendants, et de rassembler les nouveaux enregistrements dans une nouvelle table que l'on renvoie en fin de programme.

```
def ProjectionTable(table, listeIndices):
    resultat = []
    for enr in table:
        proj = ProjectionEnregistrement(enr, listeIndices)
        resultat.append(proj)
    return resultat
```

I.6 L'utilisation de deux boucles `for` permet de créer toutes les concaténations d'un élément de `table1` et d'un élément de `table2`, qui sont successivement ajoutés à une nouvelle table.

```
def ProduitCartesien(table1, table2):
    resultat = []
    for enr1 in table1:
        for enr2 in table2:
            resultat.append(enr1 + enr2)
    return resultat
```

I.7 Comme dans la question précédente, on parcourt les couples d'enregistrements à l'aide de deux boucles. Si les attributs de ces deux enregistrements coïncident, on concatène au premier enregistrement une copie du second dans laquelle seul l'attribut d'indice i_2 n'a pas été recopié. Le résultat de cette concaténation est ensuite ajouté dans la nouvelle table.

```
def JointEnregistrements(enr1, enr2, indice2):
    return (enr1 + enr2[0:indice2] + enr2[indice2 + 1:])
```

> L'utilisation d'une fonction annexe n'est pas obligatoire.

```
def Jointure(table1, table2, indice1, indice2):
    resultat = []
    for enr1 in table1:
        for enr2 in table2:
            if enr1[indice1] == enr2[indice2]:
                jointure = JointEnregistrements(enr1, enr2, indice2)
                resultat.append(jointure)
    return resultat
```

> On peut aussi utiliser une boucle `for` pour la jointure :
>
> ```
> def JointEnregistrements(enr1, enr2, indice2):
>     resultat = enr1[:]
>     for i in range(len(enr2)):
>         if i != indice2:
>             resultat.append(enr2[i])
>     return resultat
> ```
>
> L'astuce `enr1[:]` sert à créer une copie de `enr1` ; il ne faudrait pas que les modifications apportées à `resultat` modifient également l'argument `enr1`.

I.8 La fonction `Jointure` fait intervenir deux boucles. La première boucle `for` s'exécute `len(table1)` fois. Pour chaque étape, une nouvelle boucle `for` s'exécute `len(table2)` fois, en effectuant une lecture de liste et une copie d'un enregistrement de `table2`. Finalement,

> La complexité de la fonction `Jointure` est en $O(\texttt{len(table1)} \cdot \texttt{len(table2)} \cdot \texttt{len(table2[0])})$.

I.9 La nouvelle table sans doublons se construit au fur et à mesure. Initialisée au premier enregistrement `table[0]`, on ajoute ensuite successivement les autres enregistrements, en vérifiant toutefois à chaque fois qu'il n'y a pas déjà un élément identique dans la nouvelle table. Cette vérification se fait de manière naïve, en comparant la nouvelle valeur à insérer à toutes les autres déjà sélectionnées.

```
def Egalite(enr1, enr2):
    # Renvoie True ssi les deux enregistrements sont égaux
    for indice in range(len(enr1)):
        if enr1[indice] != enr2[indice]:
            return False
    return True

def SupprimerDoublons(table):
```