Corrigé : bases de données (X PC 2018)

I. Implémentation des opérateurs de l'algèbre relationnelle en Python

Sélection avec test d'égalité à une constante

Question I.1.

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

Question I.2. Les comparaisons entre chaînes de caractères ainsi que la méthode append sont supposées de coût constant donc la complexité de cette fonction est proportionnelle au nombre n d'enregistrements de la table, soit un $O(n)$.

Sélection avec test d'égalité entre deux attributs

Question I.3.

```
def SelectionEgalité(table, indice1, indice2):
    sigma = []
    for e in table:
        if e[indice1] == e[indice2]:
            sigma.append(e)
    return sigma
```

Projection sur les indices

Question I.4.

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

Question I.5.

```
def Projection(table, listeIndices):
    pi = []
    for e in table:
        pi.append(ProjectionEnregistrement(e, listeIndices))
    return pi
```

Produit cartésien

Question I.6.

```
def ProduitCartesien(table1, table2):
    x = []
    for e1 in table1:
        for e2 in table2:
            x.append(e1 + e2)
    return x
```

Jointure

Question I.7. Comme le suggère l'énoncé, on définit d'abord une fonction auxiliaire :

```
def fusion(e1, e2, i1, i2):
    e = []
    for i in range(len(e1)):
        e.append(e1[i])
    for i in range(len(e2)):
        if i != i2:
            e.append(e2[i])
    return e
```

qui a pour objet de joindre deux enregistrements. Notons que le paramètre `i1` est superflu. On définit ensuite :

```
def Jointure(table1, table2, indice1, indice2):
    join = []
    for e1 in table1:
        for e2 in table2:
            if e1[indice1] == e2[indice2]:
                join.append(fusion(e1, e2, indice1, indice2))
    return join
```

Question I.8. La fonction `fusion` a une complexité en $O(k_1 + k_2)$ où k_1 et k_2 désignent les arités des deux tables ; cette fonction est utilisée au plus $n_1 n_2$ fois dans la fonction `Jointure`, n_1 et n_2 désignant les tailles des deux tables, donc la complexité totale de la fonction `Jointure` est en $O(n_1 n_2 (k_1 + k_2))$.

Distinct

Question I.9. On définit tout d'abord une fonction qui teste l'égalité entre deux enregistrements de même longueur :

```
def egalite(e1, e2):
    for i in range(len(e1)):
        if e1[i] != e2[i]:
            return False
    return True
```

puis une fonction qui détermine si un enregistrement est présent dans une table au delà d'un indice i :

```
def present(table, e, i):
    for j in range(i, len(table)):
        if table[j] == e:
            return True
    return False
```

La fonction principale ne garde que la dernière occurrence de chaque enregistrement présent une ou plusieurs fois dans la table :

```
def SupprimerDoublons(table):
    distinct = []
    for i in range(len(table)):
        if not present(table, table[i], i+1):
            distinct.append(table[i])
    return distinct
```

Question I.10. La fonction `egalite` a une complexité en $O(k)$, où k désigne la longueur commune des deux enregistrements. La fonction `present` fait appel $(n-i)$ fois à la fonction `egale` donc la fonction `SupprimerDoublon` fait appel $\sum_{i=0}^{n-1}(n-i-1) = \frac{n(n-1)}{2}$ fois à la fonction `egale` et au plus n fois à la méthode `append`, ce qui lui confère une complexité en $O(n^2 k)$.

II. Implémentation de requêtes SQL en Python

Question II.1.

```
resultat = SelectionConstante(Trajet, 1, 'Rennes')
```

Question II.2.

```
resultat = ProduitCartesien(Trajet, Vehicule)
```

Question II.3.

```
resultat = SelectionEgalite(ProduitCartesien(Trajet, Vehicule), 3, 4)
```

Dans le produit cartésien des deux tables, `Trajet.IdVehicule` a pour indice 3 et `Vehicule.IdVehicule` a pour indice 4.

Question II.4.

```
resultat = Projection(Jointure(Hotel, Chambre, 0, 1), [1, 2, 4, 5])
```

Une fois la jointure réalisée, `Date` et `Prix` ont pour indices respectifs 4 et 5 puisque `Chambre.IdHotel` a été supprimé.

Question II.5.

```
table1 = ProduitCartesien(Hotel, ProduitCartesien(Trajet, Ticket))
table2 = SelectionEgalité(table1, 2, 5)
table3 = SelectionEgalité(table2, 3, 7)
table4 = SelectionConstante(table3, 12, '50')
resultat = Projection(table4, [0])
```

La table obtenue par le produit cartésien des trois tables comporte 13 attributs, numérotés de 0 (`Hotel.IdHotel`) à 12 (`Ticket.Prix`).

Question II.6. La sous-requête correspond au résultat de la question précédente :

```
table1 = ProduitCartesien(Hotel, ProduitCartesien(Trajet, Ticket))
table2 = SelectionEgalité(table1, 2, 5)
table3 = SelectionEgalité(table2, 3, 7)
table4 = SelectionConstante(table3, 12, '50')
sousrequete = Projection(table4, [0])
```

```
resultat = SelectionConstante(Jointure(Chambre, sousrequete), 3, '100')
```

Puisque la table `sousrequete` ne comporte qu'un attribut, la jointure réalisée comporte les mêmes attributs que la table `chambre`.

III. Amélioration des performances

Tables triées par rapport à un indice

Question III.1.

```
def VerifieTrie(table, indice):
    for i in range(1, len(table)):
        if table[i][indice] < table[i-1][indice]:
            return False
    return True
```

Question III.2. On utilise bien entendu une recherche dichotomique, mais la difficulté est qu'il s'agit ici de déterminer *tous* les enregistrements qui satisfont la condition demandée. J'ai choisi pour cela d'implémenter une fonction de recherche récursive.

```
def dicho(table, indice, constante, i, j):
    if i >= j:
        return []
    k = (i + j) // 2
    if table[k][indice] < constante:
        return dicho(table, indice, constante, k + 1, j)
    elif table[k][indice] > constante:
        return dicho(table, indice, constante, i, k)
    else:
        return (
            dicho(table, indice, constante, i, k)
            + [table[k]]
            + dicho(table, indice, constante, k + 1, j)
        )
```

```
def SelectionConstanteTrie(table, indice, constante):
    return dicho(table, indice, constante, 0, len(table))
```

Question III.3. Ma solution s'inspire de l'étape de fusion de deux listes triées dans l'algorithme mergesort.

```
def JointureTrie(table1, table2, indice1, indice2):
    join = []
    i, j = 0, 0
    while i < len(table1) and j < len(table2):
        if table1[i][indice1] == table2[j][indice2]:
            join.append(fusion(table1[i], table2[j], indice1, indice2))
            i, j = i + 1, j + 1
        elif table1[i][indice1] < table2[j][indice2]:
            i = i + 1
        else:
            j = j + 1
    return join
```

La fonction `fusion` a été définie à la question I.7.

Question III.4. Avec les notations de la question I.8, cette fonction fait appel au plus $n_1 + n_2$ fois à la fonction `fusion` dont le coût est un $O(k_1 + k_2)$. La complexité totale est donc un $O((n_1 + n_2)(k_1 + k_2))$, à comparer à la complexité en $O(n_1 n_2 (k_1 + k_2))$ de la fonction `Jointure`.
Lorsque n_1 et n_2 sont de même ordre de grandeur, la fonction `Jointure` a une complexité quadratique, contre une complexité linéaire pour `JointureTrie`. En revanche, lorsque l'un de ces deux entiers est petit devant l'autre, les deux algorithmes sont de complexité linéaire vis-à-vis du plus grand des deux entiers, avec certe une plus grande constante pour le premier des deux algorithmes.

Utilisation d'un dictionnaire (index)

Question III.5.

```
def CreerDictionnaire(table, indice):
    dico = {}
    for i in range(len(table)):
        if table[i][indice] not in dico:
            dico[table[i][indice]] = [i]
        else:
            dico[table[i][indice]].append(i)
    return dico
```

Question III.6.

```
def SelectionConstanteDictionnaire(table, indice, constante, dico):
    sigma = []
    if constante in dico:
        for i in dico[constante]:
            sigma.append(table[i]):
    return sigma
```

Question III.7. Toutes les manipulation des dictionnaires étant de coût constant, la complexité de cette fonction est un $O(p)$ où p est le nombre d'enregistrements de la table à sélectionner, soit un $O(n)$ dans le pire des cas. Cette complexité est meilleure lorsque le nombre d'éléments à sélectionner p est petit devant la taille n de la table, mais identique dans le cas où ces deux entiers p et n sont du même ordre de grandeur.

Question III.8. Au lieu de parcourir tous les enregistrements de la table `table2`, on se contente de parcourir ceux donnés par le dictionnaire :

```
def JointureDictionnaire(table1, table2, indice1, indice2, dico2):
    join = []
    for e1 in table1:
        if e1[indice1] in dico2:
            for j in dico2[e1[indice1]]:
                join.append(fusion(e1, table2[j], indice1, indice2))
    return join
```

La fonction `fusion` a été définie à la question I.7.

Question III.9. Ma fonction fait appel n_1 fois à la fonction `in dico2` et au plus $n_1\ell_2$ fois à la fonction `fusion` donc sa complexité est en $O(n_1\ell_2(k_1+k_2))$.

Question III.10. Compte tenu de cette complexité, il est judicieux de choisir d'utiliser la fonction écrite lorsque $n_1\ell_2 < n_2\ell_1$, et de permuter les deux tables dans le cas contraire.