X-ENS : informatique hors option
Un corrigé

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

Dans tout le sujet, on omettra les problèmes éventuels d'identité physique des listes quand on fera une "copie" de liste. Ainsi, si on dispose d'une table `table`, on écrira par exemple

```
T=[table[0],table[2]]
```

en ayant conscience qu'une modifcation de `T[0]` modifiera aussi `table[0]`. Ceci ne semble pas porter à conséquence puisque l'on ne modifie pas dynamiquement les listes.

Sélection avec test d'égalité à une constante

Q.I.1.
```
def SelectionConstante(table,indice,constante):
    T=[]
    for e in table:
        if e[indice]==constante:
            T.append(e)
    return T
```

Q.I.2. La boucle est effectuée n fois (n est la taille de la table). Chaque tours de boucle se fait en temps constant. la complexité est ainsi $O(n)$.

Sélection avec test d'égalité entre deux attributs

Q.I.3.
```
def SelectionEgalite(table,indice1,indice2):
    T=[]
    for e in table:
        if e[indice1]==e[indice2]:
            T.append(e)
    return T
```

Projection sur des indices

Q.I.4.
```
def ProjectionEnregistrement(enregistrement,listeIndices):
    l=[]
    for i in listeIndices:
        l.append(enregistrement[i])
    return l
```

Q.I.5.
```
def Projection(table,listeIndices):
    T=[]
    for e in table:
        T.append(ProjectionEnregistrement(e,listeIndices))
    return T
```

Produit cartésien

Q.I.6.
```
def ProduitCartesien(table1,table2):
    T=[]
    for e1 in table1:
        for e2 in table2:
```

```
            T.append(e1+e2)
    return T
```

Jointure

Q.I.7. Une première fonction prend en argument un enregistrement et un indice et renvoie l'enregistrement obtenu en supprimant l'atribut associé à l'indice.

```
def supprimer(e,i):
    l=[]
    for j in range(len(e)):
        if j!=i:
            l.append(e[j])
    return l
```

Il reste à agir comme pour le produit cartésien mais en n'ajoutant que les bons enregistrements.

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

Q.I.8. La suppression d'une coordonnée a un coût $O(k_2)$. On itère $n_1 \times n_2$ fois et chaque étape coûte potentiellement de l'odre de k_2 opérations. la complexité est ainsi $O(n_1 n_2 k_2)$.

Distinct

Q.I.9. On utilise une fonction `egaux` testant l'égalité de deux enregistrements.

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

Pour chaque enregistrement de la table, on teste s'il existe un enregistrement de numéro strictement plus grand qui lui est égal. Si ce n'est pas le cas, on l'ajoute à une table en constuction.

```
def SupprimerDoublons(table):
    T=[]
    for i in range(len(table)):
        j=i+1
        while j<len(table) and not(egaux(table[i],table[j])):
            j=j+1
        if j==len(table):
            T.append(table[i])
    return T
```

Q.I.10. Le test d'égalité d'enregistrements a un coût $O(k)$.
dans la fonction, pour chaque i, on fait au plus de l'ordre de $n-i$ appels à la fonction d'égalité. La complexité est alors (somme des $k(n-i)$) égale à $O(kn^2)$).

II. Implémentation de requêtes SQL en Python

Q.II.1.
```
resultat = SelectionConstante (Trajet,1,"Rennes")
```

Q.II.2.
```
resultat = ProduitCartesien(Trajet,Vehicule)
```

Q.II.3.
```
r1 = ProduitCartesien(Trajet,Vehicule)
resultat = SelectionEgalite(r1,3,0)
```

Q.II.4. Il faut faire attention au décalage d'indice pour la seconde table (puisque l'on supprime l'attribut redondant dans la jointure)

```
r1 = Jointure(Hotel,Chambre,0,1)
resultat = Projection(r1,[1,2,4,5])
```

Q.II.5. On effectue une première jointure (table `r1`) et on ne garde, par ailleurs, que les tickets de prix demandé (table `r2`).

```
r1 = Jointure(Hotel,Trajet,2,2)
r2 = SelectionConstante(Ticket,5,"50")
```

Il faut alors associer chaque trajet de `r1`, au ticket correspondant, ce que l'on fait par une nouvelle jointure. On ne garde alors que l'attribut identifiant l'hôtel.

```
resultat = Projection(Jointure(r1,r2,3,1),[0])
```

Q.II.6. On commence par reprendre la requête précédente en supprimant les doublons.

```
r1 = Jointure(Hotel,Trajet,2,2)
r2 = SelectionConstante(Ticket,5,"50")
r3 = SupprimerDoublons(Projection(Jointure(r1,r2,3,1),[0]))
```

Par ailleurs, on ne garde de la table `Chambre` que les chambre ayant le bon prix.

```
r4 = SelectionConstante(Chambre,3,"100")
```

Il ne fautgarder de `r4` que les enregistrements dont l'attribut `IdHotel` est dans `r3`. Comme en question précédente, on peu modéliser cela avec une jointure.

```
resultat = Jointure(r4,r3,1,0)
```

III. Amélioration des performances

Tables triées par rapport à un indice

Q.III.1. On parcourt la table de haut en bas en regardant si un enregistrement est plus petit que son successeur. Je propose ici la version naturelle avec boucle conditionnelle et une autre version plus simple à écrire avec interruption prématurée de boucle.

```
def VerifieTrie(table,indice):
    i=0
    while i<len(table)-1 and table[i][indice]<=table[i+1][indice]:
        i=i+1
    return i==len(table)-1

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

Q.III.2. On pourrait bien sûr parcourir la table depuis le haut pour trouver le premier indice `i` tel que `table[i]==constante` et de même par le bas pour obtenir le dernier indice `j` tel que `table[j]==constante`. Il resterait à selectionner la partie de la table entre les indices `i` et `j`. Le gain serait cependant a priori minime.
Plus naturellement, cette "recherche" dans une table triée appelle une approche dichotomique. On écrit une fonction locale `explorer` qui prend en argument deux entiers `a` et `b` compris entre 0 et $n-1$ où n est le nombre d'enregistrements. Dans l'appel `explorer(a,b)` on renvoie la table (triée) contenant les enregistrements dont l'attribut d'indice `indice` vaut enregistrement et dont le numéro est plus grand que `a` et plus petit que `b`. La fonction utilise le principe de dichotomie.

```
def SelectionConstanteTrie(table,indice,constante):
    def explorer(a,b):
        if a>b:
            return []
        else:
            c=(a+b)//2
            if table[c][indice]<constante:
                return explorer(c+1,b)
            elif table[c][indice]>constante:
                return explorer(a,c-1)
            else:
                return explorer(a,c-1)+[table[c]]+explorer(c+1,b)
    return explorer(0,len(table)-1)
```

Q.III.3. Je propose de gérer deux indice i_1 et i_2 correspondant à une position dans chaque table. On va parcourir les tables du haut vers le bas.

- Quand les éléments d'indices `indice1` et `indice2` des enregistrement regardés dans chaque table sont différents, on peut progresser dans l'une des tables (celle pour laquelle l'élément est le plus petit).
- Sinon, nos deux enregistrements ont aux bons indices le même attribut. D'après l'hypothèse faite sur les valeurs distinctes dans chaque table, il suffit d'ajouter le jointure de ces deux enregistrements et de poursuivre avec les enregistrements suivants. On utilisera la fonction `supprimer` déjà utilisée pour `Jointure`.

```
def JointureTrie(table1,table2,indice1,indice2):
    n1,n2=len(table1),le,(table2)
    i1,i2=0,0
    T=[]
    while i1<n1 and i2<n2:
        if table1[i1][indice1]>table[i2][indice2]:
            i2=i2+1
        elif table1[i1][indice1]<table[i2][indice2]:
            i1=i1+1
        else:
            T.append(table1[i1]+supprimer(table2[i2],indice2))
            i1=i1+1
            i2=i2+1
    return T
```

Remarquons que l'on pourrait se passer de l'hypothèse sur la valeurs distinctes. Quand on rencontre des enregistrements avec même valeur d'attribut, il suffit alors de créer les sous-tables `T1` et `T2` associées à cette valeur commune et d'ajouter à la liste résultat la jointure de ces sous-tables.

```
def JointureTrie(table1,table2,indice1,indice2):
    n1,n2=len(table1),le,(table2)
    i1,i2=0,0
    T=[]
    while i1<n1 and i2<n2:
        if table1[i1][indice1]>table[i2][indice2]:
            i2=i2+1
        elif table1[i1][indice1]<table[i2][indice2]:
            i1=i1+1
        else:
            const=table1[i1][indice1]
            T1=[]
            while table1[i1][indice1]==const:
                T1.append(table1[i1])
                i1=i1+1
            T2=[]
            while table2[i2][indice2]==const:
                T2.append(table2[i2])
                i2=i2+1
            T=T+Jointure(T1,T2,indice1,indice2)
    return T
```

Q.III.4. Dans la boucle, à coup sûr l'une des variables i_1 ou i_2 est incrémentée. La boucle est donc effectuée au plus $n_1 + n_2$ fois. Chaque itération se faisant en temps constant, la complexité est $O(n_1 + n_2)$. Cette complexité est la même dans tous les cas. Elle est toujours meilleurs que celle $O(n_1 n_2)$ de la fonction initiale.
Dans le cas plus général où on ne suppose plus les valeurs distinctes, il faut ajouter les jointures partielles effectuées et la mise à jour associée de la table résultat. Le côut sera de l'ordre de grandeur de la table créée qu'il est difficile d'exprimer simplement.

Utilisation d'un dictionnaire (index)

Q.III.5. On parcourt la table. Pour un enregistrement donné, on met à jour le dictionnaire soit en créant une association (si elle n'existe pas) soit en la modifiant (si elle existe).

```
def CreerDictionnaire(table,indice):
    dico={}
    for i in range(len(table)):
        e=table[i]
        if e[indice] in dico:
            dico[e[indice]].append(i)
        else:
            dico[e[indice]]=[i]
    return dico
```

Q.III.6. On regarde si la constante est une clé. Si oui, on crée la table en ne gardant que les enregistrements dont le numéro est dans la liste associée. Sinon, on renvoie une table vide.

```
def SelectionConstante(table,indice,constante,dico):
    if constante in dico:
        T=[]
        for i in dico[constante]:
            T.append(table[i])
```

```
        return T
    else:
        return []
```

Q.III.7. Toutes les opérations se font en temps constant. Leur nombre dépend du nombre des itérations. Il y en a autant que d'enregistrements à selectionner. La complexité est donc $O(m)$ où m est le nombre d'enregistrements à selectionner.
Si la table contient peu (voire pas) d'enregistrement de valeur cherchée, la fonction est plus efficace. C'est le cas s'il y a un grand nombre de valeurs pour l'attribut considéré. A l'invesre, si on ne manipule que peu de valeur d'attributs (dison un nombre constant indépendant de n, taille de la table) alors on n'y gagne rien en ordre de grandeur (et on doit en plus créer et stocker le dictionnaire).

Q.III.8. Pour chaque enregistrement de `table1`, `dico2` nous indique quels enregistrements de `table2` il faut lui associer.

```
def JointureDictionnaire(table1,table2,indice1,indice2,dico2):
    T=[]
    for e in table1:
        if e[indice1] in dico2:
            for i2 in dico2[e[indice1]]:
                T.append(e+supprimer(table2[i2],indice2))
    return T
```

Q.III.9. Pour chaque enregistrement `e` de la table `table1`, on va écrire une boucle qui s'effectue au plus k_2 fois et dont chaque tour se fait en temps constant. La complexité est ainsi $O(n_1 k_2)$.

Q.III.10. Si on choisit d'indexer la table numéro 1, on aura une complexité $O(k_1 n_2)$. Dans la mesure où l'on crée potentiellement une table de taille $k_1 k_2$, on pourrait choisir d'indexer la table contenant le plus d'éléments.