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

Question I.1.

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

Question I.2. En notant n la taille de la table, on effectue en une boucle au plus trois opérations à coût unitaire. Ainsi au plus $3n+3$ opérations à coût unitaire sont effectués,

en ayant notée n la taille de la table, la complexité de `SelectionConstante` est donc en $\mathcal{O}(n)$ (linéaire)

Sélection avec test d'égalité entre deux attributs

Question I.3.

```
def SelectionEgalite(table, indice1, indice2):
    ntable=[]
    for e in table: # e pour "enregistrement"
        if e[indice1] == e[indice2]:
            ntable.append(e)
    return ntable
```

Projection sur des indices

Question I.4.

```
def ProjectionEnregistrement(enregistrement,listeIndices):
    ne=[] # ne pour "nouvel enregistrement"
    for l in listeIndices:
        ne.append(enregistrement[l])
    return ne
```

Question I.5.

```
def Projection(table,listeIndices):
    ntable=[]
    for e in table: # e pour "enregistrement"
        ne=ProjectionEnregistrement(e,listeIndices)
        ntable.append(ne)
    return ntable
```

Produit cartésien

Question I.6.

```
def ProduitCartesien(table1, table2):
    ntable=[] #ntable pour "nouvelle table"
    for e1 in table1:
        for e2 in table2:
            ntable.append(e1+e2)
    return ntable
```

Jointure

Question I.7. On commence donc par créer une fonction effectuant la jointure de deux enregistrements et qui renvoie la liste vide pour signifier qu'il n'y a pas de jointure possible.

```
def JointureEnregistrement(enregistrement1, enregistrement2, indice1, indice2):
    ne=[] # ne pour "nouvel enregistrement"
    for valeur in enregistrement1:
        ne.append(valeur)
# On aurait pu faire une copie autrement : ne=enregistrement1+[]
# cela coûte moins mais ne semble pas être dans l'esprit de ce qui est attendu
    k2=len(enregistrement2)
    for indice in range(k2):
        if indice !=indice2:
            valeur=enregistrement2(indice)
            ne.append(valeur)
    return ne


def Jointure(table1, table2, indice1, indice2):
    nt=[] # nt pour "nouvel table"
    for e1 in table1:
        for e2 in table2:
            if e1[indice1] ==e2[indice2]:
                ne=JointureEnregistrement(e1, e2, indice1, indice2)
                nt.append(ne)
    return(nt)
```

Question I.8. Notons a_i et n_i respectivement les arités et les tailles des `table`$_i$ pour $i \in \{1,2\}$.

Hors boucles de la fonction `JointureEnregistrement`, il y a deux opérations unitaires et dans les boucles s'effectue en au plus trois opérations unitaires. On effectue $a_1 + a_2$ tours de boucles, chaque boucle ayant au plus un coût de 3 unités.

Ainsi la fonction `JointureEnregistrement` s'effectue en au plus $3(a_1 + a_2) + 2$ opérations élémentaires.

Enfin dans la fonction `Jointure` alors il y a alors au plus $3(a_1 + a_2) + 4$ opérations élémentaires pare boucles et $n_1 n_2$ tours de boucles sont effectués.

La complexité de mon implémentation de `Jointure` est en $\mathcal{O}(n_1 n_2 (a_1 + a_2))$

Distinct

Question I.9. Commençons par une fonction qui teste l'égalité entre enregistrements d'une même table :

```
def egaliteEnregistrement(enregistrement1, enregistrement2):
    k=len(enregistrement1)
    indice=0
    while indice <k and enregistrement1[indice] == enregistrement2[indice]:
        indice+=1
    return indice == k
```

Ensuite une fonction qui teste l'absence d'un enregistrement dans une table de même arité :

```
def absentTable(enregistrement,table):
    for e in table:
        if egaliteEnregistrement(enregistrement, e):
            return False
    return True
```

Dans la fonction suivante, on crée une table vide et on rajoute un nouvel enregistrement de l'ancienne table s'il est absent dans la nouvelle.

```
def SupprimerDoublons(table):
    nt=[]
    for e in table:
        if absentTable(e,nt):
            nt.append(e)
    return nt
```

Question I.10. Le test d'égalité de deux enregistrements de la fonction `egaliteEnregistrement` est d'une complexité en $\mathcal{O}(k)$ où k est l'arité.

Dans la fonction `absentTable`, il y a n tours de boucles où n est la taille de la table, ainsi la complexité de cette fonction est en $\mathcal{O}(nk)$.

Enfin pour la fonction `SupprimerDoublons`, si i est l'indice de l'enregistrement `e`, la taille de la table `nt` est au plus de i.

Ainsi la complexité est de $\mathcal{O}\left(\sum_{i=0}^{n-1} ik\right)$ or $\sum_{i=0}^{n-1} ik = \frac{n(n-1)k}{2}$

La complexité de `SupprimerDoublons` est en $\mathcal{O}(n^2k)$ où k est l'arité et n la taille de la table

II. Implémentation de requêtes SQL en Python

Question II.1.

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

Question II.2.

```
ProduitCartesien(Trajet, Vehicule)
```

Question II.3.

```
r2=ProduitCartesien(Trajet, Vehicule)
SelectionEgalite(r2,3,4) #arite_Trajet+0_Vehicule=4
```

Question II.4.

```
r4=Jointure(Hotel, Chambre, 0,1)
Projection(r4,[1,2,4,5])
```

Question II.5. Je procède de façon la plus directe mais pas la plus légère.

Je commence par le produit des trois tables puis les trois sélections et enfin la projection

```
P1=ProduitCartesien(Hotel, Trajet)
Prod=ProduitCartesien(P1, Ticket) # Hotel X Trajet X Ticket
Sel1=SelectionEgalite(Prod,2,5)  # Hotel.Ville = Trajet.VilleA
Sel2=SelectionEgalite(Sel1,3,8) #  Trajet.IdTrajet = Ticket.IdTrajet
Sel3=SelectionConstante(Sel2,12, '50')  # Ticket.Prix = '50'
res5=Projection(Sel3,[0])
res5 #pour la question suivante
```

Question II.6. Ici je ne peux pas traduire mécaniquement cette re quête avec les fonction précédente.

Je choisis de traduire la requête avec une jointure qui supprime l'attribut de la deuxième table qui est d'arité 1.

```
Sel6=SelectionConstante(Chambre, 3, '100')
r6=Jointure(Sel6, res5, 1, 0)
SupprimerDoublons(r6)
```

III. Amélioration des performances

Tables triées par rapport à un indice

Question III.1.

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

Question III.2. Cette question nous invite à la recherche dichotomique qui est logarithmique

```
def SelectionConstanteTrie(table,indice,constante):
# on crée une fonction donnant la liste des enregistrement qui conviennent
    def chercher(debut, fin): # fonction récursive
    # elle renvoie les indices qui conviennet entre debut et fin inclus
        if fin < debut:
            return [] # au cas où "c=debut ou fin" dans l'appel récursif
        elif table[debut][indice]> constante:
            return []
        elif table[fin][indice]< constante:
            return []
        else:
            c=(debut+fin)//2
            ldeb=chercher(debut, c-1)
            lfin=chercher(c+1,fin)
            if table[c][indice]== constante:
                return ldeb+[c]+lfin
            else:
                return ldeb+lfin
    presences=chercher(0, len(table)-1)
    for j in presences:
        e=table[j]
        ntable.append(e)
    return ntable
```

Question III.3. J'utilise la fonction auxiliaire `JointureEnregistrement` du **I.7.**.

Les indices `j1` et `j2` servent à parcourir `table1` et `table2`. En remarquant que les attributs à considérer sont classés par ordre strictement croissant en fonction des indices ; en cas d'égalité d'attributs, on incrémente les deux indices et sinon, on en incrémente un seul.

```
def JointureTrie(table1, table2, indice1, indice2):
    ntable=[] # ntable pour "nouvel table"
    n1=len(table1)
    n2=len(table2)
    j1=0
    j2=0
    while j1 <n1 and j2 < n2:
        if table1[j1][indice1] == table1[j2][indice2]:
            ne=JointureEnregistrement(table1[j1][indice1],table1[j2][indice2],indice1,indice2)
            ntable.append(ne) # ne pour nouvel enregistrement
```

```
            j1+=1
            j2+=1
        elif table1[j1][indice1] < table1[j2][indice2]:
            j1+=1
        else:
            j2+=1
    return(nt)
```

Question III.4. En notant n_1 et n_2 les tailles respectives des tables `table1` et `table2`, on effectue au plus n_1+n_2 tours de boucle.
En notant a_1 et a_2 les arités, chaque tour de boucle s'effectue en au plus $3(a_1+a_2)+2+6$ selon **I.7.**.
La complexité de la fonction `JointureTrie` est donc en $(a_1+a_2)(n_1+n_2)$
Si $n_1 \geqslant 6$ et $n_2 \geqslant 6$, cette nouvelle approche est plus performante.
Si $n_1 = 1$ et que `table1[0][indice1] > table1[j][indice2]` pour tout $j \in [\![0, n_2-2]\!]$, alors les complexité sont comparables.

Application à la selection

Question III.5.

```
def CreerDictionnaire(table, indice):
    dico={}
    n=len(table)
    for j in range(n):
        c=  table[j][indice] # c pour clé
        if c in dico:
            dico[c].append(j)
        else:
            dico[c]=[indice]
    return dico
```

Question III.6.

```
def SelectionConstanteDictionnaire(table, indice, constante, dico):
    nt=[]
    if constante in dico:
        liste= dico[constante]
        for j in liste:
            e=table[j]
            nt.append(e)
    return nt
```

Question III.7. Chaque opération de cette fonction `SelectionConstanteDictionnaire` est réputée unitaire.

La fonction `SelectionConstanteDictionnaire` est d'une complexité linéaire par rapport au nombre d'occurrences de la constante `constante` dans l'attribut de la table `table` associé à l'indice `indice`.
La complexité de la fonction `SelectionConstanteDictionnaire` est donc toujours plus performante celle de la fonction `SelectionConstante` car le nombre d'occurrences est inférieur à la taille.
Si la constante `constante` n'apparaît jamais dans l'attribut de la table `table` associé à l'indice `indice`, alors l'exécution de `SelectionConstanteDictionnaire` sera de l'ordre de trois opérations unitaires ce qui est inférieur à `SelectionConstante` pour une table de taille quatre (au moins).

Application à la jointure

Question III.8. Je réutilise la fonction auxiliaire de la question **I.7**.

On parcourt `table1` et à l'aide du dictionnaire, puis en fonction de l'enregistrement de cette table, on ne travaille qu'avec les enregistrements qui correspondent dans `table2` en se servant de `dico2`.

```
def JointureDictionnaire(table1, table2, indice1, indice2, dico2):
    nt=[] # nt pour "nouvel table"
    for e1 in table1:
        cle=e1[indice1]
        l_indice=dico2[cle]
        for j2 in l_indice:
            ne=JointureEnregistrement(e1, table2[j2], indice1, indice2)
            nt.append(ne)
    return(nt)
```

Question III.9. Je réutilise les notations de la question **I.7**.

La boucle indexée par j_2 s'effectue en au plus k_2 fois, le coût de cette instruction `for j2 in...` est donc en $\mathcal{O}(k_2(a_1+a_2))$.

Dans `for e1 in...`, il y a deux opérations élémentaires en plus et celle-ci est effectuée n_1 fois.

La complexité de votre implémentation par rapport aux tailles et arités respectives des tables `table1` et `table2` ainsi que par rapport à la longueur maximale d'une liste renvoyée par le dictionnaire `dico2` est donc en$\mathcal{O}(n_1k_2(a_1+a_2))$.

Question III.10. Il est conseillé d'indexer la table ayant la plus grosse taille (nombre d'enregistrement).