Partie 1 : 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 modification de `T[0]` *modifiera aussi* `table[0]`*. Ceci ne semble pas porter à conséquence puisque l'on ne modifie pas dynamiquement les listes.*

Partie 1.1 : Sélection - égalité constante

Q.1.

Un parcours de la table permet de stocker tous les éléments pertinents.

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

Q.2.

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

Partie 1.2 : Sélection -égalité deux attributs

Q.3.

Le programme est quasiment le même que précédemment sauf que le test ne se fait plus par rapport à une constante mais entre les deux valeurs testées.

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

Partie 1.3 : Projection sur des indices

Q.4.

On accède par le parcours de la liste des indices aux seules valeurs que l'on veut conserver et que l'on stocke dans la liste `l`.

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

Q.5.

Il suffit d'appeler la fonction précédente pour chacun des enregistrements de la table considérée.

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

Partie 1.4 : Produit cartésien

Q.6.

Pour chaque élément de `table1`, on ajoute à la table le résultat de la concaténation d'avec tous les éléments de `table2`; cela se fait immédiatement avec une double boucle.

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

Partie 1.5 : Jointure

Q.7.

Il suffit ici de parcourir l'enregistrement et de ne pas retenir la valeur correspondant à l'attribut à éliminer.

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

Q.8.

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.9.

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'ordre de k_2 opérations. la complexité est ainsi $O(n_1 n_2 k_2)$.

Partie 1.6 : Distinct

Q.10.

On utilise ici deux sous-fonctions :

- egaux teste l'égalité de deux enregistrements ;
- `pasDansTable(e,table)` teste si l'enregistrement `e` est dans la table `table` et retourne `True` s'il en est absent

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

def pasDansTable(e,table):
  for enr in table :
    if egaux(enr,e):
      return False
  return True
```

Il ne reste qu'à parcourir la table et à n'ajouter ensuite un enregistrement à la table des résultats que s'il n'y figure pas déjà.

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

Q.11.

Pour les deux sous-fonctions

- le test d'égalité d'enregistrements a un coût $O(k)$
- la fonction `egaux` est appelée au plus n_T fois, où n_T est le nombre d'enregistrement de la table `T`, soit une complexité en $O(kn_T)$

La fonction `SupprimerDoublons` appelle n fois la fonction `pasDansTable` en ayant au pire un taille n pour la table `T`. La complexité est donc en $O\left(kn^2\right)$.

On peut préciser le nombre d'opérations plus finement en disant que lors du i-ème appel de la fonction `pasDansTable`*, la taille de la table* `T` *est au plus i ; le nombre d'opérations de la fonction* `SupprimerDoublons` *est en :* $O\left(\sum_{i=0}^{n-1} ki\right)$.

Partie 2 : Implémentation de requêtes SQL en Python

Q.12.

Ici il s'agit simplement de réaliser une sélection sur la ville de départ

```
SELECT * FROM Trajet
 WHERE Trajet.VilleD = Rennes ;
```

Q.13.

Le produit cartésien s'écrit très simplement en SQL :

```
SELECT * FROM Trajet, Vehicule ;
```

Q.14.

On souhaite ne conserver dans le produit cartésien effectué que les enregistrements ayant le même identifiant `IdVehicule`, ce qui s'effectue facilement à l'aide d'une projection :

```
SELECT * FROM Trajet, Vehicule
 WHERE Trajet.IdVehicule=Vehicule.IdVehicule
```

Q.15.

La première ligne correspond à la réalisation d'une jointure selon l'attribut `IdHotel`, tandis que la seconde est une projection classique selon les attributs Classe, `Ville`, `Date` et `Prix`

```
SELECT Classe, Ville, Date, Prix
FROM Hotel
JOIN Chambre
 ON Hotel.IdHotel = Chambre.IdHotel;
```

Partie 3 : Amélioration des performances

Partie 3.1 : Tables triées par rapport à un indice

Q.16.

On parcourt la table de haut en bas en regardant si un enregistrement est plus petit que son successeur. L'utilisation du `return` permet d'arrêter la boucle à la première « erreur » rencontrée.

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

Q.17.

Parcourir la table depuis le haut pour trouver le premier indice `i` tel que `table[i]==constante` et par le bas pour obtenir le dernier indice `j` tel que `table[j]==constante` n'apporte qu'un gain minime en termes de performance.
On attend ici une approche dichotomique. Pour cela, 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 évalue la valeur `table[c][indice]` de l'enregistrement `table[c]` et on la compare à la constante d'intérêt. Trois situations se peuvent présenter :

- cette valeur est plus petite que la constante, il faut donc s'intéresser uniquement aux enregistrements strictement précédents,
- cette valeur est plus grande que la constante, il faut donc s'intéresser uniquement aux enregistrements strictement suivants,
- cette valeur est la constante, on garde donc l'enregistrement en question et l'on recherche les enregistrements précédents **et** suivants ceux qui peuvent également convenir.

Cette fonction sera de complexité logarithmique.

```
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)
```

Partie 3.2 : Utilisation d'un dictionnaire

Q.18.

On parcourt la table et 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,e in enumerate(table):
    if e[indice] in dico:
      dico[e[indice]].append(i)
    else:
      dico[e[indice]]=[i]
  return dico
```

Q.19.

On regarde si la constante est une clef. 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 SelectionConstanteDictionnaire(table,
    indice,constante,dico):
  if not(constante in dico):
    return []
  T=[]
  for i in dico[constante]:
    T.append(table[i])
  return T
```

Q.20.

Toutes les opérations sur les dictionnaires se font en temps constant et il y en a autant que d'enregistrements à sélectionner. La complexité est donc linéaire en la taille du dictionnaire.
La fonction est efficace dans le cas où peu d'enregistrements ont la valeur recherchée (imaginons par exemple les noms de famille sur une base de données des élèves de Ginette), puisque le dictionnaire évitera un parcours complet de la table.
A l'inverse, si l'attribut est tel que le nombre de valeurs possibles est faible (par exemple les années de naissance sur la même base), l'utilisation du dictionnaire est inefficace puisqu'elle ne fait pas gagner de temps d'accès et nécessite une place en mémoire supplémentaire.

Q.21.

On reprend le même programme que précédemment mais en ne bouclant cette fois que sur les valeurs de `table1` que l'on sait être pertinentes grâce à `dico2`.

```
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.22.

Pour chacun des n_1 enregistrements `e` de la table `table1`, on va écrire une boucle qui s'effectue au plus N_2 fois. Le coût de cette boucle est dû au `append` de coût constant et à l'appel à la fonction `supprimer` de coût k_2. La complexité est ainsi $O(n_1 N_2 k_2)$.

Q.23.

Si on choisit d'indexer la table numéro 1, la complexité est $O(n_2 N_1 k_1)$. Or $N_1 \leqslant n_1$ et le nombre d'enregistrements est généralement beaucoup plus grand que l'arité de la table. Il est donc plus efficace de choisir d'indexer la table contenant le plus d'éléments.