Corrigé PC*

1)
```
def ConditionConstante(table,indice,constante) :
    rep = []
    for v in table :
        if v[indice] == constante :
            rep.append(v)
    return rep
```

2)
```
def ConditionEgalite(table, indice1, indice2)  :
    rep = []
    for v in table :
        if v[indice1] == v[indice2] :
            rep.append(v)
    return rep
```

3) La complexité en en O(len(table)

4)
```
def SelectionEnregistrement(enregistrement,listeIndices) :
    rep = []
    for i in in listeIndices :
        rep.append(enregistrement[i])
    return rep
```

5)
```
def Selection(table,listeIndices):
    rep = []
    for v in table :
        rep.append(SelectionEnregistrement(v,listeIndices)
    return rep
```

6)
```
def ProduitCartesien(table1,table2) :
    rep = []
    for v1 in table1 :
        for v2 in table2 :
            rep.append(v1 + v2)
    return rep
```

7)
La complexité est en O(len(table1)*len(table2)*(a1 + a2)), avec a1 et a2 l'arité de chacune des tables.
Les deux boucles sont imbriquées l'une dans l'autre. La complexité est donc le nombre d'itération de chaque boucles multipliée entre elles avec la complexité de v1 + v2, qui dépend de la taille de v1 et v2.

8)
```
def Jointure(table1, table2, indice1, indice2) :
    produit = ProduitCartesien(table1,table2)
    if len(table1) == 0 :
        return []
    indice2p = indice2 + len(table1[0])
    return ConditionEgalite(rep,indice1,indice2p)
```

9)
Les deux appels de fonctions sont ici fait de manière séquentielles, donc la complexité est la somme des deux. La complexité est donc en O(len(rep) + len(table1)*len(table2)*(a1 + a2)). Comme la longueur de rep est len(table1)*len(table2), la complexité est la même que le produit cartésien.

10)

```python
def egaliteEnregistrement(v1,v2) :
    for i in range(len(v1)) :
        if v1[i] != v2[i] :
            return False
    return True

def SupprimerDoublons(table) :
    rep = []
    for v in table :
        present = False
        for vmis in rep :
            if egaliteEnregistrement(v,vmis) :
                present = True
        if not present :
            rep.append(v)
    return rep
```

11)

La complexité de `egaliteEnregistrement` est en O(a), avec a l'arité de la table. La complexité totale est donc en $O(len(table)^2 * a)$, car il y a deux boucles imbriqués.

12)

Les tuples peuvent être utilisés comme clefs de dictionnaire, car c'est un type de données non mutables.

On peut alors utiliser un dictionnaire de présence, pour savoir si on a déjà croiser un enregistrement ou non. La recherche de présence d'une clef étant en O(1), la complexité serait alors en O(len(table)).

La boucle `for vmis in rep :` pourrait être remanié.

13)

```
SELECT *
FROM Ticket
WHERE Date = "2025-10-17"
```

14)

```
SELECT Prix, Type
FROM Vehicule
JOIN Trajet ON Vehicule.IdVehicule = Trajet.IdVehicule
JOIN Ticket ON Ticket.IdTrajet = Trajet.IdTrajet
WHERE VilleD = "Lille" AND VilleA = "Strasbourg"
```

15)

```
SELECT COUNT(*)
FROM Vehicule
```

16)

```
SELECT IdHotel, AVG(Prix)
FROM Chambre
GROUP BY IdHotel
HAVING AVG(Prix) < 50
```

17)

```
SELECT IdVehicule FROM Vehicule
EXCEPT
SELECT IdVehicule
FROM Trajet
WHERE VilleD = "Zurich" or VilleA = "Zurich"
```

18)

```
SELECT Hotel.IdHotel
FROM Hotel JOIN Chambre
ON Hotel.idHotel = Chambre.idHotel
WHERE Ville = "Bordeaux" AND
Prix =
  (SELECT min(Prix) FROM
  Hotel JOIN Chambre
  ON Hotel.idHotel = Chambre.idHotel
  WHERE Ville = "Bordeaux")
```

19)

```
SELECT DISTINCT(H1.idHotel, H2.idHotel)
FROM Hotel as H1, Hotel as H2
JOIN Chambre AS CH1 ON CH1.idHotel = H1.idHotel
JOIN Chambre AS CH2 ON CH2.idHotel = H2.idHotel
WHERE H1.Classe <> H2.Classe
AND CH1.Date <> CH2.Date
AND H1.Ville = H2.Ville
AND H1.Ville = "Paris"
```

20)

```python
resultat0 = ConditionConstante(Trajet,1,"Rennes")
```

21)

```python
resultat1 = ProduitCartesien(Trajet,Vehicule)
```

22)

```python
resultat2 = ConditionEgalite(resultat1,3,4)
```

23)

```python
resultat3 = Jointure(Hotel,Chambre,0,1)
resultat4 = Selection(resultat3,[1,2,5,6])
```

24)

```python
resultat5 = ProduitCartesien(Hotel,Trajet)
resultat6 = ProduitCartesien(resultat5,Ticket)
resultat7 = ConditionEgalite(resultat6,2,5)
resultat8 = ConditonEgalite(resultat7,3,8)
resultat9 = ConditionConstante(resultat8,12,50)
resultat10 = Selection(resutltat9,[0])
```

25)

```python
resultat11 = ConditionConstante(Chambre,3,100)
resultat12 = ProduitCartesien(resultat11,resultat10)
resultat13 = ConditionEgalite(resultat12,1,4)
resultat14 = Selection(resultat13,[0,1,2,3])
```

26)
```
def VerifieTrie(table,indice) :
    for i in range (len(table) - 1) :
        v1, v2 = table[i],table[i+1]
        if v1[indice] > v2[indice] :
            return False
    return True
```

27)
```
def ConditionConstanteTrie(table,indice,constante) :
    a = 0
    b = len(table)
    m = (a+b)//2
    while a<b+1 :
        m = (a+b)//2
        if table[m][indice] > constante :
            b = m
        elif table[m][indice] < constante :
            a = m
        else :
            inf = trouve_borne_inf(table,indice,constante,a,m)
            sup = trouve_borne_sup(table,indice,constante,m,b)
            return table[inf:sup]
    if table[a][indice] == constante :
        return [table[a]]
    return []

def trouve_borne_inf(table,indice,constante,a,m) :
    if m == a+1 :
        if table[a][indice] == constante :
            return a
        else :
            return m
    else :
        if table[(a+m)//2][indice] == constante :
            return trouve_borne_inf(table,indice,constante,a,(a+m)//2)
        else :
            return trouve_borne_inf(table,indice,constante,(a+m)//2,m)

def trouve_borne_sup(table,indice,constante,m,b) :
    if m == b-1 :
        return b
    else :
        if table[(b+m)//2][indice] == constante :
            return trouve_borne_sup(table,indice,constante,(b+m)//2,b)
        else :
            return trouve_borne_sup(table,indice,constante,m,(b+m)//2)
```

28) La complexité de la recherche est toujours en log(len(table)). Par contre, si l'intervalle à ajouter est de taille comparable à la table, alors l'opération `table[inf:sup]` est en O(len(table)).

29)
```
def CreerDictionnaire(table,indice) :
    d = {}
    for i in range(len(table)) :
        v = table[i]
        if not v[indice] in d :
            d[v[indice]] = []
        d[v[indice]].append(i)
    return d
```

30)
```
def ConditionConstanteDictionnaire(table,indice,constante,dico) :
    rep = []
    if not constante in dico :
        return rep
    for i in dico[constante] :
        rep.append(table[i])
    return d
```

31) La complexité est en O(len(dico[constante])).
Si la taille du dico est proche de la la longueur de la table, on a une complexité en O(len(table)). Dans ce cas, ça ne serait pas meilleur.

32)
```
def JointureDictionnaire(table1,table2,indice1,indice2,dico2) :
    rep = []
    for v in table1 :
        if v[indice1] in dico2 :
            for i2 in dico2[v[indice1]] :
                rep.append(v + table2[i2])
    return rep
```

33) Les opérations dans la boucle `for v in table1 :` sont en O(1 + ld), avec ld la longueur du dictionnaire v[indice1]. La somme des ld dans le dictionnaire est égal à len(table2).
D'où une complexité de O(len(table1) + len(table2)*(a1+a2).

34) La meilleure table pour table2 est la table qui est la clef primaire. En effet, dans ce cas là, la taille du dico est de 1 car chaque élément de la table1 est relié à un élément de la table2 par la contrainte de clef étrangère.