Corrigé PSI

1)
```
def est_egal(enregistrement1,enregistrement2) :
    if len(enregistrement1) != len(enregistrement2) :
        return False

    for ind in range(len(enregistrement1)) :
        if enregistrement1[ind] != enregistrement2[ind] :
            return False
    return True
```

2) La complexité est en O(len(enregistrement1)).

3)
```
def Egalite_indice(enregistrement, indice, constante)  :
    return enregistrement[indice] == constante
```

4)
```
def ConditionConstante(table,indice,constante) :
    rep = []
    for v in table :
        if Egalite_indice(v,indice,constante) :
            rep.append(v)
    return rep
```

5)
```
def ConditionEgalite(table, indice1, indice2)  :
    rep = []
    for v in table :
        if Egalite_indice(v,indice2,v[indice1]) :
            rep.append(v)
    return rep
```

6) La complexité en en O(len(table)

7)
```
def SelectionEnregistrement(enregistrement,listeIndices) :
    rep = []
    for i in in listeIndices :
        rep.append(enregistrement[i])
    return rep
```

8)
```
def Selection(table,listeIndices):
    rep = []
    for v in table :
        rep.append(SelectionEnregistrement(v,listeIndices)
    return rep
```

9) La complexité de l'opération `l1 + l2` est en O(len(l1) + len(l2)). Cette opération est appelée concaténation.

10)
```
def ProduitCartesien(table1,table2) :
    rep = []
    for v1 in table1 :
        for v2 in table2 :
            rep.append(v1 + v2)
    return rep
```

11) 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.

12)
```
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)
```

13) 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.

14)
```
def SupprimerDoublons(table) :
    rep = []
    for v in table :
        present = False
        for vmis in rep :
            if est_egal(v,vmis) :
                present = True
        if not present :
            rep.append(v)
    return rep
```

15)
La complexité de `est_egal` 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.

16)
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 présences, pour savoir si on a déjà croisé 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.

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

18)
```
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"
```

19)
```
SELECT COUNT(*)
FROM Vehicule
```

20)
```
SELECT IdHotel, AVG(Prix)
FROM Chambre
GROUP BY IdHotel
HAVING AVG(Prix) < 50
```

21)
```
SELECT IdVehicule FROM Vehicule
EXCEPT
SELECT IdVehicule
FROM Trajet
WHERE VilleD = "Zurich" or VilleA = "Zurich"
```

22)
```
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")
```

23)
```
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"
```

24)
```
resultat0 = ConditionConstante(Trajet,1,"Rennes")
```

25)
```
resultat1 = ProduitCartesien(Trajet,Vehicule)
```

26)
```
resultat2 = ConditionEgalite(resultat1,3,4)
```

27)
```
resultat3 = Jointure(Hotel,Chambre,0,1)
resultat4 = Selection(resultat3,[1,2,5,6])
```

28)
```
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])
```

29)
```
resultat11 = ConditionConstante(Chambre,3,100)
resultat12 = ProduitCartesien(resultat11,resultat10)
resultat13 = ConditionEgalite(resultat12,1,4)
resultat14 = Selection(resultat13,[0,1,2,3])
```

30)
```
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
```

31)
```
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
```

32) 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.