**I.1** On parcourt tous les enregistrements de `table`, et on ajoute dans le résultat ceux dont la valeur pour l'indice `indice` vaut `constante`.

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

**I.2** Dans la fonction précédente, il n'y a pas d'opération coûteuse en dehors de la boucle `for`. Celle-ci s'exécute `len(table)` fois et on effectue deux opérations élémentaires au plus à chaque étape. Ainsi,

La complexité de la fonction `SelectionConstante` est en O(`len(table)`).

**I.3** Parcourons tous les enregistrements de la table `table` et ajoutons dans le résultat ceux dont la valeur pour l'indice `indice1` est égale à celle pour l'indice `indice2`.

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

**I.4** Il suffit de parcourir la liste `listeIndices` en créant un nouvel enregistrement auquel on ajoute les valeurs de `enregistrement` pour les indices souhaités, donnés dans `listeIndices`.

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

**I.5** Il suffit d'appliquer la fonction précédente à tous les éléments de `table`, car ceux-ci sont indépendants, et de rassembler les nouveaux enregistrements dans une nouvelle table que l'on renvoie en fin de programme.

```
def ProjectionTable(table, listeIndices):
    resultat = []
    for enr in table:
        proj = ProjectionEnregistrement(enr, listeIndices)
        resultat.append(proj)
    return resultat
```

**I.6** L'utilisation de deux boucles `for` permet de créer toutes les concaténations d'un élément de `table1` et d'un élément de `table2`, qui sont successivement ajoutés à une nouvelle table.

```
def ProduitCartesien(table1, table2):
    resultat = []
    for enr1 in table1:
        for enr2 in table2:
            resultat.append(enr1 + enr2)
    return resultat
```

**I.7** Comme dans la question précédente, on parcourt les couples d'enregistrements à l'aide de deux boucles. Si les attributs de ces deux enregistrements coïncident, on concatène au premier enregistrement une copie du second dans laquelle seul l'attribut d'indice $i_2$ n'a pas été recopié. Le résultat de cette concaténation est ensuite ajouté dans la nouvelle table.

```
def JointEnregistrements(enr1, enr2, indice2):
    return (enr1 + enr2[0:indice2] + enr2[indice2 + 1:])
```

> L'utilisation d'une fonction annexe n'est pas obligatoire.

```
def Jointure(table1, table2, indice1, indice2):
    resultat = []
    for enr1 in table1:
        for enr2 in table2:
            if enr1[indice1] == enr2[indice2]:
                jointure = JointEnregistrements(enr1, enr2, indice2)
                resultat.append(jointure)
    return resultat
```

> On peut aussi utiliser une boucle `for` pour la jointure :
>
> ```
> def JointEnregistrements(enr1, enr2, indice2):
>     resultat = enr1[:]
>     for i in range(len(enr2)):
>         if i != indice2:
>             resultat.append(enr2[i])
>     return resultat
> ```
>
> L'astuce `enr1[:]` sert à créer une copie de `enr1` ; il ne faudrait pas que les modifications apportées à `resultat` modifient également l'argument `enr1`.

**I.8** La fonction `Jointure` fait intervenir deux boucles. La première boucle `for` s'exécute `len(table1)` fois. Pour chaque étape, une nouvelle boucle `for` s'exécute `len(table2)` fois, en effectuant une lecture de liste et une copie d'un enregistrement de `table2`. Finalement,

> La complexité de la fonction `Jointure` est en $O(\texttt{len(table1)} \cdot \texttt{len(table2)} \cdot \texttt{len(table2[0])})$.

**I.9** La nouvelle table sans doublons se construit au fur et à mesure. Initialisée au premier enregistrement `table[0]`, on ajoute ensuite successivement les autres enregistrements, en vérifiant toutefois à chaque fois qu'il n'y a pas déjà un élément identique dans la nouvelle table. Cette vérification se fait de manière naïve, en comparant la nouvelle valeur à insérer à toutes les autres déjà sélectionnées.

```
def Egalite(enr1, enr2):
    # Renvoie True ssi les deux enregistrements sont égaux
    for indice in range(len(enr1)):
        if enr1[indice] != enr2[indice]:
            return False
    return True

def SupprimerDoublons(table):
    resultat = [table[0]]
    # Parcours sur la table
    for enr1 in table[1:]:
        deja_enregistre = False
        # Parcours sur les enregistrements déjà dans le resultat
        for enr0 in resultat:
            if Egalite(enr1, enr0):
                deja_enregistre = True
        # Ajout si pas déjà enregistré
        if not deja_enregistre:
            resultat.append(enr1)
    return resultat
```

> Il ne faut pas hésiter à utiliser `for i in liste` mais ce n'est cependant pas toujours possible, comme au sein de la fonction `Egalite` dans laquelle il est nécessaire de connaître l'indice de l'élément considéré.
>
> La boucle `for` est plus intéressante que la boucle `while` quand on connaît à l'avance le nombre d'itérations, donc en particulier lors du parcours itératif d'une structure de taille fixe comme c'est le cas ici. Néanmoins l'utilisation de la boucle `while` est possible.
>
> ```
> def SupprimerDoublons(table):
>     resultat = []
>     # Parcours sur la table
>     i1 = 0
>     while i1 < len(table):
>         # Parcours sur les enregistrements
>         # déjà dans le résultat
>         i0 = 0
>         while i0 < len(resultat) and not \
>                 Egalite(table[i1], resultat[i0]):
>             i0 = i0 + 1
>         if i0 == len(resultat):
>             resultat.append(table[i1])
>         i1 = i1 + 1
>     return resultat
> ```

**I.10** La fonction `SupprimerDoublons` utilise deux boucles `for` imbriquées. La première s'exécute exactement `len(table)` fois, la suivante au plus `len(table)` fois. À l'intérieur de ces boucles, on trouve un appel à la fonction `Egalite`, de complexité O(`len`(`table`[0])). Il en découle que

> La fonction `SupprimeDoublons` est dans le pire cas de complexité $O(\texttt{len}(\texttt{table})^2 \times \texttt{len}(\texttt{table}[0]))$.

> L'algorithme demandé est plutôt mauvais. En pratique il vaut mieux trier, par exemple avec l'algorithme de tri-fusion en $O(n\ln(n))$, puis comparer les éléments consécutifs en complexité linéaire.

**II.1** Il est demandé l'intégralité des valeurs pour les enregistrements de `Trajet` avec `Trajet.VilleD` avec pour valeur `"Rennes"`. L'indice de `VilleD` est 1 dans la liste `Trajet`. On obtient ainsi la table souhaitée par l'instruction

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

**II.2** On demande le produit cartésien entre `Trajet` et `Vehicule`. Il suffit d'écrire

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

> La syntaxe `FROM table1,table2` est celle du produit cartésien, valable aussi pour une jointure si elle est suivie de `ON condition`. Cette syntaxe est cependant hors programme dans le cas des jointures, qui doivent être écrites avec le mot-clef `JOIN`. Ceci a dû gêner bon nombre de candidats.

**II.3** La requête est une sélection au sein du produit cartésien des tables `Trajet` et `Vehicule`. La sélection consiste à conserver les enregistrements présentant une même valeur sur les colonnes `IdVehicule` de chaque table. Le résultat s'obtient donc par l'instruction :

```
SelectionEgalite(ProduitCartesien(Trajet, Vehicule), 3, 4)
```

> Le résultat de cette requête est identique à celui d'une jointure de la forme `FROM Trajet JOIN Vehicule` avec la condition `ON Trajet.IdVehicule = Vehicule.IdVehicule`. Cette requête est donc une très mauvaise façon d'écrire une jointure, car elle demande au logiciel de réaliser réellement le produit cartésien, qui peut être très volumineux. Au contraire, une jointure correctement écrite bénéficie d'un processus optimisé, à la fois en temps de calcul et en occupation de la mémoire.
>
> En remarquant la possibilité de jointure, on peut écrire de façon équivalente : `Jointure(Trajet, Vehicule, 3, 0)`.

**II.4** On souhaite ici récupérer les valeurs des attributs `Classe`, `Ville`, `Date` et `Prix` pour les enregistrements de la jointure des tables `Hotel` et `Chambre` selon la contrainte `Hotel.IdHotel = Chambre.IdHotel`. L'instruction suivante renvoie ce résultat.

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

> De même que pour la question précédente on peut remplacer la jointure par un produit cartésien auquel on applique ensuite une sélection.
>
> Attention à la numérotation des indices dans la jointure `Hotel + Chambre`, la colonne `Chambre.IdHotel` a disparu.

**II.5** Dans le produit des tables `Hotel`, `Trajet` et `Ticket` sous les contraintes `Hotel.Ville = Trajet.VilleA` ainsi que `Trajet.IdTrajet = Ticket.IdTrajet`, on souhaite les identifiants des hôtels pour lesquels l'attribut `Ticket.Prix` (d'indice 10) vaut 50. Cela se récupère de la manière suivante.

```
table = Jointure(Hotel,Jointure(Trajet, Ticket, 0, 1), 2, 2)
Projection(SelectionConstante(table, 10, 50), [0])
```

**II.6** Il est demandé toutes les valeurs pour les enregistrements de la table `Chambre` dont la valeur pour `IdHotel` (indice 1) correspond à une valeur `IdHotel` parmi ceux obtenus à la question précédente et où `Chambre.prix` (indice 3) vaut 100. Cela peut s'obtenir ainsi.

```
table = Jointure(Hotel, Jointure(Trajet, Ticket, 0, 1), 2, 2)
table2 = Projection(SelectionConstante(table, 10, 50), [0])
SelectionConstante(Jointure(Chambre, table2, 1, 0), 3, 100)
```

> La syntaxe proposée par l'énoncé est appelée sous-requête. Elle est évitable puisque l'on peut utiliser une jointure comme on le propose ici.
>
> La jointure entre les tables `Chambre` et `table2` n'ajoute pas de colonne car `table2` n'en a qu'une, éliminée à la jointure. Si ce n'était pas le cas, il faudrait ajouter une projection pour éliminer les colonnes superflues.

> Trier les tables pour améliorer les performances est une solution réellement pertinente, à tel point qu'elle a été retenue dans toutes les implémentations actuelles de bases de données. On crée en général un index sur les colonnes de recherche fréquente. Il s'agit d'un fichier séparé stockant les valeurs ordonnées des colonnes considérées. Les optimisations proposées dans la suite du sujet sont alors techniquement possibles.

**III.1** On parcourt les enregistrements de la table `table` en comparant l'attribut d'indice `indice` de chaque enregistrement à celui de l'enregistrement précédent. Si une incohérence est repérée, on interrompt la boucle et on renvoie aussitôt `False`. Sinon, on renvoie `True` à la fin de la boucle car l'ordre est nécessairement correct si la boucle s'exécute jusqu'au bout.

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

**III.2** L'hypothèse selon laquelle le tableau est trié permet d'effectuer une recherche dichotomique pour trouver un premier enregistrement d'attribut `constante`. Une fois cet élément trouvé, on se déplace vers la gauche et vers la droite dans la table pour trouver les éléments suivants à ajouter. En effet, l'hypothèse de tri assure que les éléments ayant pour attribut `constante` sont **consécutifs** dans la table.

```
def SelectionConstanteTrie(table, indice, constante):
    debut = 0 # inclus
    fin = len(table) # exclu
    while debut < fin:
        milieu = (debut + fin)//2
        valeur = table[milieu][indice]
        # La valeur recherchée est à gauche du milieu
        if constante < valeur:
            fin = milieu
```

```
        # La valeur recherchée est à droite du milieu
        elif constante > valeur:
            debut = milieu + 1
        else:
            # La valeur est trouvée, on la cherche dans
            # les enregistrements adjacents
            resultat = []
            i = milieu
            # Recherche à gauche
            while i >= 0 and table[i][indice] == constante:
                resultat.append(table[i])
                i = i - 1
            i = milieu + 1
            # Recherche à droite
            while i < len(table) and table[i][indice] == constante:
                resultat.append(table[i])
                i = i + 1
            return resultat
    return [] # Constante n'est pas trouvée
```

> L'usage de `append` ne préserve pas l'ordre des enregistrements qui se trouvent à gauche de la première position obtenue par dichotomie. Si on souhaite le conserver, il faudrait remplacer le premier append par l'instruction `resultat = table[i] + resultat` (ce n'était pas demandé).
>
> La question peut être également résolue par une fonction récursive, c'est-à-dire une fonction qui s'appelle elle-même. Ici on peut donc observer si `constante` est plus grande ou plus petite que la valeur associée à l'indice `milieu` de `table` et appeler de nouveau la fonction sur la première ou la seconde moitié de la table. Cette technique donne un code plus compact, mais rend plus difficile l'assurance de la terminaison de l'algorithme.

**III.3** À partir du moment où les enregistrements ont des valeurs d'attributs deux à deux distinctes dans les deux tables, il ne peut correspondre pour chaque enregistrement de `table1` qu'au plus un enregistrement dans `table2` pour lequel les attributs coïncident. On parcourt donc les deux tables simultanément à l'aide de deux compteurs `i1` et `i2`. Lorsque l'on tombe sur deux enregistrements ayant la même valeur d'attribut, on concatène ces deux enregistrements (en éliminant l'occurrence de l'attribut commun dans le second enregistrement) et on ajoute le tout dans la nouvelle table. Sinon, on incrémente le compteur de l'enregistrement ayant le plus petit attribut.

```
def JointureTrie(table1, table2, indice1, indice2):
    # Initialisation
    n1 = len(table1)
    n2 = len(table2)
    resultat = []
    i1 = 0
    i2 = 0
    # Parcours simultané des deux tables
    while i1 != n1 and i2 != n2:
        if table1[i1][indice1] < table2[i2][indice2]:
```

```
i1 = i1 + 1
```

```
        elif table1[i1][indice1] > table2[i2][indice2]:
            i2 = i2 + 1
        else :
            joint = JointEnregistrements( \
                table1[i1], table2[i2], indice2)
            resultat.append(joint)
            i2 = i2 + 1
            i1 = i1 + 1
    return resultat
```

**III.4** Dans la fonction de la question précédente, la quasi-totalité des opérations sont de coût constant, à l'exception de celle qui consiste à concaténer les deux champs lorsque leurs attributs coïncident. Cette opération est de coût $O(\texttt{len}(\texttt{table2}[0]))$. Il reste donc à savoir combien d'exécutions de boucle sont effectuées. Or, à chaque étape, on incrémente soit le compteur `i1`, soit le compteur `i2`, soit les deux et la fonction termine lorsque `i1` (respectivement `i2`) atteint la taille `n1` de `table1` (respectivement `n2` de `table2`). On a donc au plus `n1 + n2` pas d'itération, ce qui permet de conclure que

> La complexité de la fonction `JointureTrie` est en $O((\texttt{n1} + \texttt{n2}) \cdot \texttt{len}(\texttt{table2}[0]))$.

Pour comparer l'efficacité de cet algorithme avec celui de la question I.7, on peut considérer l'exemple suivant :

```
table1 = [[8,4,5,6], [17,15,7,8], [4,7,12,9], [1,15,17,19]]
table2 = [[3, 7, 8],[4, 9, 1],[17, 13, 6]]
```

que l'on veut joindre pour les indices 2 et 0. Le premier algorithme parcourt `table2` (de taille 3) pour chaque enregistrement de `table1` (taille 4) soit $3 \times 4 = 12$ pas d'itération. Pour le second algorithme, on parcourt en même temps les deux tables, il n'y a que $3 + 4 = 7$ pas d'itérations. Chacun d'entre eux effectue le même nombre de concaténations et d'ajouts, et de la même façon. Même si le second algorithme contient un peu plus d'opérations unitaires en début que le premier, il est plus efficace.

Dans tous les cas, réaliser une jointure selon une contrainte nécessite de consulter chaque donnée des tables au moins une fois. Le coût de la concaténation des deux enregistrements est indépendant de l'algorithme utilisé (que ce soit celui de la question I.8 ou III.3). Par conséquent,

> La fonction `JointureTrie` est dans tous les cas plus performante que `Jointure` lorsque les conditions de tri sont réunies.

**III.5** Il suffit d'effectuer un parcours des différents enregistrements de la table `table` en consultant la valeur de l'attribut d'indice `indice`. On vérifie si cette valeur apparaît déjà dans le dictionnaire :

- si c'est le cas, on ajoute la position de l'enregistrement courant dans la liste associée à la valeur ;
- sinon, on crée une nouvelle clef en lui associant la liste contenant (pour l'instant) uniquement la position de l'enregistrement courant.

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

**III.6** Le dictionnaire permet d'obtenir en temps constant la liste des positions des enregistrements dont l'attribut d'indice `indice` est égal à `constante`. Il n'y a plus qu'à récupérer les enregistrements correspondants.

```
def SelectionConstanteDictionnaire(table, indice, constante, dico):
    resultat = []
    if constante in dico:
        for i in dico[constante]:
            resultat.append(table[i])
    return resultat
```

**III.7** La fonction `SelectionConstante` compare pour chaque enregistrement de la table `table` sa valeur pour `indice` à `constante`. Sa complexité est en O(`len`(`table`)). La fonction `SelectionConstanteDictionnaire` réalise de son coté une recherche `constante in dico` de complexité constante, puis effectue un parcours de la liste `dico[constante]` en O(`len`(`dico`[`constante`])). Cette deuxième méthode est plus efficace si chaque valeur n'a qu'un faible nombre d'occurrences, typiquement en O(1) après application de `SupprimerDoublons`. À l'inverse, lorsqu'une valeur apparaît trop souvent, et notamment si tous les enregistrements de `table` valent `constante` pour l'indice `indice`, les deux fonctions sont de complexité linéaire.

> Il faut noter que la création du dictionnaire faite en amont de l'appel de la fonction `SelectionConstanteDictionnaire` a un coût, en O(`len`(`table`)). Si la fonction est appelée plusieurs fois, le dictionnaire sera réutilisé et son coût amorti, distribué entre les différents appels de la fonction de sélection. Au contraire, pour une utilisation unique, le gain de complexité est nul.

**III.8** Parcourons les enregistrements de la table `table1`. Pour chacun d'entre eux, le dictionnaire `dico2` permet de savoir si l'attribut pour l'indice `indice1` correspond à des enregistrements de la table `table2`, et donne même leurs indices. Dès lors, on concatène l'enregistrement de la table `table1` à chacun de ces enregistrements, auxquels on retire l'attribut d'indice `indice2`, avant d'ajouter le tout dans la table de résultat.

```
def JointureDictionnaire(table1, table2, indice1, indice2, dico2):
    resultat = []
    # Parcours de table1
    for enr1 in table1:
        valeur = enr1[indice1]
```

```
        # Récupération si valeur présente dans dico2
        if valeur in dico2:
            for i in dico2[valeur]:
                joint = JointEnregistrements( \
                    enr1, table2[i], indice2)
                resultat.append(joint)
    return resultat
```

**III.9** La fonction précédente contient essentiellement une boucle `for` itérant sur les enregistrements de la table `table1`. À chaque itération, outre quelques opérations élémentaires comme la récupération de la valeur ou la consultation du dictionnaire, on effectue une nouvelle boucle `for` sur les éléments associés à une clef dans `dico2` qui s'effectue au plus $k_2$ fois. Pour chaque enregistrement à concaténer, l'opération s'effectue en complexité linéaire en `len(table2[0])`. Finalement,

> La complexité de la fonction `JointureDictionnaire` est en $\mathrm{O}(\texttt{len(table1)} \cdot \texttt{len(table2[0])} \cdot k_2)$.

**III.10** La complexité précédente dépend de trois quantités qu'il convient de minimiser pour réduire le temps de calcul :

- l'arité de la table sur laquelle on construit le dictionnaire (ici `len(table2[0])`). On peut raisonnablement la considérer faible par rapport à la taille des tables : en pratique, un enregistrement contient rarement plus d'une dizaine d'attributs. Choisir la table d'arité minimale ne permettrait qu'un gain anecdotique.
- la longueur de la table sur laquelle n'est pas construit le dictionnaire (c'est-à-dire `len(table1)` dans le code ci-dessus). Ce facteur incite naturellement à choisir pour cette table celle qui contient le moins d'éléments.
- la longueur maximale d'une liste dans le dictionnaire (c'est-à-dire la quantité $k_2$ donnée par l'énoncé). Ce paramètre dépend de la nature de l'attribut traité, en particulier des différentes valeurs qu'il peut prendre.
  - Si l'attribut ne peut prendre qu'un nombre $p$, assez faible, de valeurs, par exemple un mois ou une année, et que le nombre d'enregistrements est grand, on peut envisager que toutes les listes du dictionnaire auront un nombre d'éléments $k \approx n/p$ où $n$ est la taille de la table. Quelle que soit la table choisie pour le dictionnaire, le coût de la jointure sera proportionnel au produit des longueurs des deux tables.
  - Si l'attribut peut prendre un nombre très important de valeurs, comme c'est le cas pour des numéros de série, des noms de famille par exemple, on aura en revanche une valeur relativement faible pour ce paramètre $k$, ce qui rendra le coût de la jointure cette fois de l'ordre de la longueur de la table sur laquelle on ne construit pas de dictionnaire.

Ces considérations essentiellement empiriques permettent de supposer que

> Pour obtenir les meilleurs performances, la table à indexer est celle contenant le plus d'éléments.