

ASIE – 2026 – sujet2

EXERCICE 2 (6 POINTS)

Cet exercice porte sur les bases de données et les graphes.

PARTIE A

Dans cette partie, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`), `JOIN . . . ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY`.

On considère une base de données permettant d'enregistrer des informations concernant un jeu de société. Les joueurs s'inscrivent à l'aide d'un pseudonyme et peuvent participer à plusieurs parties. Plusieurs joueurs peuvent participer à une partie de ce jeu, ils peuvent alors gagner des points grâce à cette partie.

Cette base de données est composée des deux tables suivantes :

- `personne` :
 - `id_pers`, l'identifiant de la personne ;
 - `pseudo_pers`, le pseudonyme de la personne, c'est-à-dire un nom choisi par la personne qui s'inscrit ;
 - `date_pers`, la date d'inscription de la personne, au format texte 'AAAA-MM-JJ', c'est-à-dire en spécifiant d'abord l'année puis le mois et enfin le jour.
- `participation` :
 - `id_partie`, l'identifiant d'une partie d'un jeu ;
 - `id_pers`, l'identifiant d'une des personnes qui a joué cette partie ;
 - `nb_point`, le nombre de points gagnés par cette personne à cette partie.

1. Donner la table dans laquelle l'attribut `id_pers` est une clé étrangère.
2. Justifier que la clé primaire de la table `participation` ne peut pas être `id_partie`.
3. Donner une requête permettant d'ajouter une personne dont le pseudonyme est `theorie` et dont la date d'inscription est le 14 décembre 2022. On prendra l'identifiant 42 pour cette personne.
4. Donner une requête permettant d'obtenir tous les identifiants des parties jouées par la personne dont le pseudonyme est `test` (on admettra qu'une seule personne possède ce pseudonyme).
5. Donner les requêtes permettant de supprimer la personne dont l'identifiant est 8 de la base de données.

PARTIE B

On s'intéresse au jeu suivant.

On considère un alphabet composé uniquement des voyelles a, e, i, o, u, y. Dans toute la suite, les mots considérés seront composés uniquement de ces 6 voyelles. Le jeu mélange en secret ces 6 voyelles pour obtenir un nouvel ordre alphabétique comme *ouyeai*. Ensuite, le jeu propose au joueur une liste de couples de mots distincts tels que le premier mot du couple est classé, suivant le nouvel ordre alphabétique, avant le deuxième mot. L'objectif du jeu est de proposer un ordre sur les 6 voyelles qui soit compatible avec ce classement.

On rappelle que, dans l'ordre lexicographique, la comparaison de deux mots s'effectue lettre par lettre, en commençant par la première, puis la suivante, etc. jusqu'à obtenir une différence. C'est la première lettre qui diffère qui donne l'ordre des deux mots.

Par exemple, si l'ordre alphabétique des lettres est *ouyeai*, alors :

- le mot *ouu* est classé avant le mot *aei* car *o* est avant *a* dans *ouyeai* ;
 - le mot *ouy* est classé avant le mot *oue* car les deux premières lettres sont identiques et *y* est avant *e* ;
 - le mot *ou* est classé avant le mot *oue* car *ou* est un préfixe du mot *oue*.
6. Recopier et compléter les lignes 3 et 4 du code de la fonction `indice` qui prend en paramètres deux chaînes de caractères : `lettre` qui est une des 6 voyelles et `ordre` qui est un mélange des 6 voyelles, et qui renvoie l'indice où se trouve `lettre` dans `ordre`.

```
1 def indice(lettre, ordre):
2     for i in range(len(ordre)):
3         if ...:
4             return ...
```

7. Recopier et compléter les lignes 7, 9 et 11 du code de la fonction `comparer` qui prend en paramètres deux chaînes de caractères `mot1` et `mot2` représentant deux mots distincts ainsi qu'une chaîne de caractères `ordre` représentant un ordre alphabétique composé des 6 voyelles. La fonction renvoie `True` si `mot1` est classé avant `mot2` selon l'ordre `ordre` et `False` sinon.

```
1 def comparer(mot1, mot2, ordre):
2     i = 0
3     while i < len(mot1) and i < len(mot2):
4         i1 = indice(mot1[i], ordre)
5         i2 = indice(mot2[i], ordre)
6         if i1 < i2:
7             return ...
8         elif i1 > i2:
9             return ...
10        i += 1
11    return ...
```

On s'intéresse maintenant à une stratégie pour atteindre l'objectif du jeu. Lorsque le joueur récupère la liste des couples de mots, il en déduit que certaines lettres sont placées avant d'autres. Par exemple, avec le couple de mots (*oou*, *ooai*), il peut déduire que la lettre *u* est placée avant la lettre *a*.

Ensuite, il construit un graphe orienté de la manière suivante : les sommets sont les 6 voyelles et à chaque fois qu'il déduit qu'une lettre (par exemple *e*) est avant une autre lettre (par exemple *a*), il va ajouter un arc partant de *e* vers *a* (s'il n'existait pas déjà).

Par exemple, on suppose que le jeu fournit la liste mots_exemple :

```
mots_exemple = [("ou", "oui"), ("ye", "ieo"),
                 ("ye", "aaa"), ("a", "ieo"), ("eue", "eyy"),
                 ("ao", "au"), ("e", "a")]
```

Le joueur en déduit alors que *u* est avant *a*, *y* est avant *i*, *y* est avant *a*, *a* est avant *i*, *u* est avant *y*, *o* est avant *u* et que *e* est avant *a*. Il construit alors le graphe orienté de la figure 1.

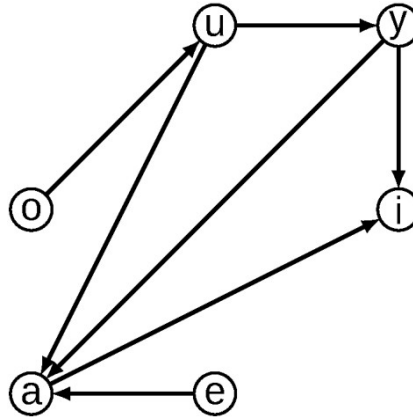


Figure 1. Graphe de l'exemple

8. Écrire le code de la fonction `premiere_diff` qui prend en paramètres deux chaînes de caractères `mot1` et `mot2` représentant deux mots distincts et qui renvoie le premier indice où les deux lettres sont différentes. Si l'un des deux mots est un préfixe de l'autre, la fonction renvoie la taille du préfixe. Exemples :

```
>>> premiere_diff("ou", "oui")
2
>>> premiere_diff("aio", "aioie")
3
```

La fonction `dico_adj`, dont le code est donné ci-dessous, prend en paramètre une liste `mots` représentant une liste des couples de mots distincts donnés par le jeu et renvoie un dictionnaire `adj` représentant les listes d'adjacence du graphe.

```

1 def dico_adj(mots):
2     adj = {}
3     for (mot1, mot2) in mots:
4         ident = premiere_diff(mot1, mot2)
5         if ident < len(mot1) and ident < len(mot2):
6             petite = mot1[ident]
7             grande = mot2[ident]
8             if petite not in adj:
9                 adj[petite] = [grande]
10            else:
11                adj[petite].append(grande)
12    return adj

```

9. Donner le dictionnaire obtenu après l'appel `dico_adj(mots_exemple)` où `mots_exemple` est la liste définie précédemment dans l'énoncé.

La fonction `parcours` réalise un parcours du graphe. Ses paramètres d'entrée sont :

- un dictionnaire `adj` représentant les listes d'adjacence d'un graphe ;
- une chaîne de caractères `s` représentant un sommet du graphe ;

- une liste `deja_vus` contenant les sommets du graphe qui ont déjà été ajoutés aux sommets à traiter ;
- une liste `tri` qui sera vide lors du premier appel puis qui va s'agrandir d'une lettre à chaque appel récursif.

```

1 def parcours(adj, s, deja_vus, tri):
2     if s in adj:
3         for v in adj[s]:
4             if v not in deja_vus:
5                 deja_vus.append(v)
6                 parcours(adj, v, deja_vus, tri)
7     tri.append(s)

```

10. Indiquer de quel type de parcours il s'agit.

11. Écrire une fonction `trier` qui prend en paramètre une liste `mots` représentant la liste des couples de mots et qui renvoie un ordre des 6 voyelles compatible avec le classement des mots donnés dans la liste. Pour cela, on suivra l'algorithme suivant :

- récupérer le dictionnaire des listes d'adjacence du graphe ;
- initialiser une liste vide appelée `tri` ;
- initialiser une liste vide appelée `deja_vus` ;
- pour chaque voyelle qui n'est pas dans `deja_vus`, l'ajouter à `deja_vus` et lancer le parcours depuis ce sommet ;
- lorsque toutes les lettres ont été parcourues, inverser les éléments de la liste `tri` et la renvoyer.

On pourra utiliser `liste.reverse()`. La méthode `reverse` ne renvoie rien mais modifie `liste` en inversant ses éléments.