Exercice 3 (8 points)

Cet exercice porte sur les graphes, les bases de données, les tris, les algorithmes gloutons et la récursivité.

Une association s'occupe d'enfants de 0 à 18 ans. Elle souhaite pouvoir former des groupes d'enfants qui s'entendent durant les activités proposées.

Partie A : base de données

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` , `OR`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT`, `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT`, `ORDER BY`.

On considère une base de données composée des 3 tables suivantes.

- Table **parent** :
  - `nom` est le nom de famille du parent ;
  - `tel` est le numéro de téléphone du parent ;
  - `codep` est le code postal de la ville où réside le parent.
- Table **enfant** :
  - `id` est l'identifiant de l'enfant pour l'association ;
  - `prenom` est le prénom de l'enfant ;
  - `num_parent` est le téléphone du parent référent (par soucis de simplicité, on suppose qu'un enfant n'est référencé que par un seul parent) ;
  - `annee` est l'année de naissance de l'enfant.
- Table **mesentente** :
  - `enfant1` est l'identifiant d'un premier enfant ;
  - `enfant2` est l'identifiant d'un second enfant.

Ainsi, on considère que deux enfants qui se trouvent sur la même ligne dans la table `mesentente` ne peuvent pas effectuer de sortie ensemble.

Le schéma relationnel de la BDD est donné en figure 1, avec la convention que les attributs formant une clef primaire sont soulignés tandis que ceux d'une clef étrangère sont précédés d'un croisillon (symbole #) avec une flèche vers l'attribut référencé.

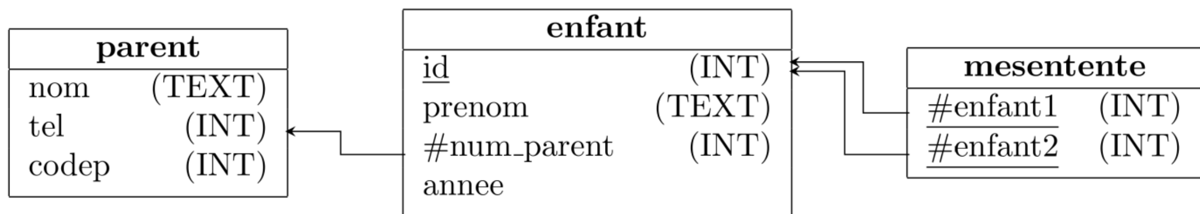

Figure 1. Schéma relationnel de la BDD

On considère la table **enfant** suivante :

enfant			
id	prenom	num_parent	annee
2	‘Hawa’	33619911212	2012
3	‘Adrien’	33619861232	2013
6	‘Kian’	33619834521	2012
8	‘Gabin’	33619847852	2014
12	‘Nakamura’	33619732453	2009
14	‘Maya’	33600782153	2017
17	‘Olivier’	33619868564	2017
21	‘Tess’	33619835876	2016
23	‘Rachelle’	33600785482	2023

1. Donner le type pour l’attribut `annee` de la table **enfant**.
2. Expliquer quelle contrainte de domaine supplémentaire serait pertinente pour cet attribut `annee`.
3. Donner un exemple d’attribut de la table **enfant** qui suit une contrainte de référence.
4. En expliquant ce choix, proposer une clef primaire pour la table **parent**.

Suite à une mauvaise saisie, le véritable téléphone d’un parent (`33619782812`) a été transformé en `33600782812`. On souhaite corriger cette anomalie avec la requête suivante, mais elle lève une erreur.

```
UPDATE parent SET tel = 33619782812 WHERE tel = 33600782812;
```

5. Expliquer pourquoi la requête proposée lève une erreur.

6. Recopier et compléter alors cette suite de commandes qui permet de changer le numéro de téléphone d'un parent du parent de nom `'Bauges'` habitant au code postal `73340` et ayant pour téléphone erroné `33600782812` au lieu de son véritable téléphone `33619782812` :

```
INSERT INTO parent VALUES ('Bauges', 33619782812, 73340);
UPDATE enfant SET num_parent = ... WHERE num_parent = ...;
DELETE FROM parent WHERE tel = ...;
```

7. En considérant la table `enfant` fournie, donner le résultat de cette requête `SQL`.

```
SELECT prenom
FROM enfant
WHERE annee < 2014
ORDER BY annee;
```

8. Proposer une requête qui renvoie les prénoms, par ordre alphabétique, des enfants inscrits pour le parent dont le numéro de téléphone est `3619861122`.

9. Proposer une requête qui liste les identifiants et prénoms des enfants dont le parent habite dans la ville de code postal `38520`.

Partie B : graphes et algorithmique

Afin de faciliter en amont les préparations des sorties, on souhaite construire le graphe non orienté des mésententes entre les enfants de l'association. Le graphe est représenté par un dictionnaire dont les clés sont les sommets du graphe, et qui associe à chaque sommet le tableau (type `List` en Python) de ses voisins.

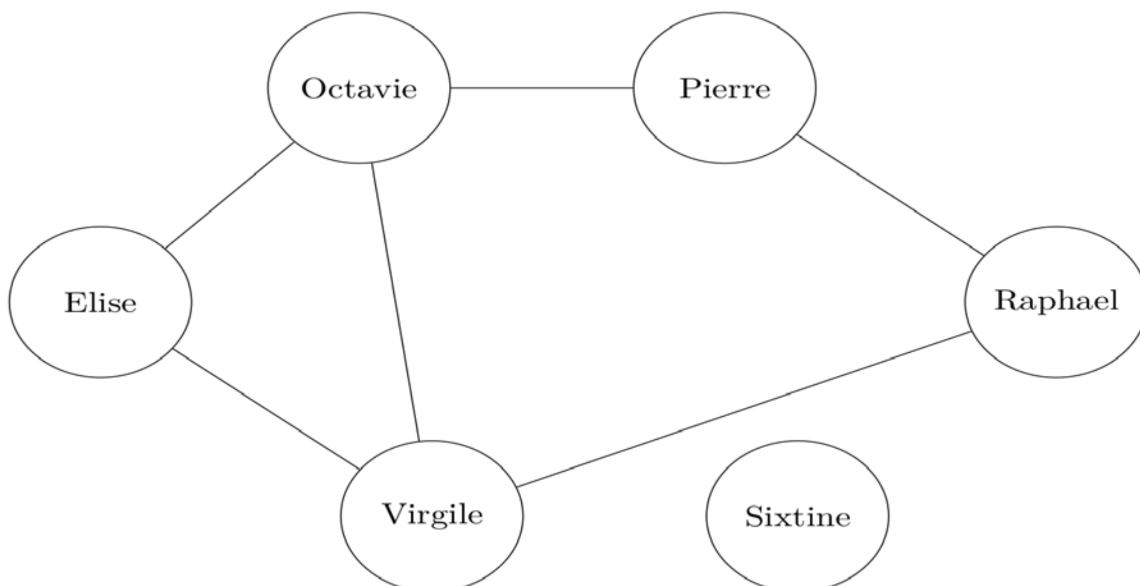

Figure 2. Graphe `g1`

Par exemple, le graphe `g1` de la figure 2 est représenté par le dictionnaire suivant :

```
1 g1 = {'Elise': ['Octavie', 'Virgile'],
2        'Octavie': ['Elise', 'Pierre', 'Virgile'],
3        'Pierre': ['Octavie', 'Raphael'],
4        'Raphael': ['Pierre', 'Virgile'],
5        'Sixtine': [],
6        'Virgile': ['Elise', 'Octavie', 'Raphael']
7       }
```

10. Expliquer pourquoi la situation décrite ne nécessite qu'un graphe non orienté.

11. Dessiner le graphe `g2` défini ci-dessous.

```
1 g2 = {'Adrien': ['Elisabeth', 'Lea'],
2        'Elisabeth': ['Adrien', 'Ian', 'Luca'],
3        'Ian': ['Elisabeth', 'Joseph', 'Luca'],
4        'Joseph': ['Ian'],
5        'Lea': ['Adrien'],
6        'Luca': ['Elisabeth', 'Ian']
7       }
```

12. Écrire une fonction `degre`, qui prend en arguments un dictionnaire `g` représentant un graphe et une chaîne de caractères `s` représentant un sommet du graphe, et qui renvoie le degré du sommet `s` dans `g`. On rappelle que le degré d'un sommet est le nombre d'arêtes issues de ce sommet.

13. Recopier et compléter les lignes 7 à 10 de la fonction `sommets_tries`, qui prend en paramètre un dictionnaire `g` représentant un graphe, et qui renvoie la liste des sommets du graphe triés dans l'ordre décroissant de leur degré.

```
 1 def sommets_tries(g):
 2     sommets = [sommet for sommet in g]
 3     n = len(sommets)
 4     for i in range(1, n):
 5         sommet_courant = sommets[i]
 6         j = i-1
 7         while ... and ...:
 8             sommets[...] = sommets[...]
 9             j = j - 1
10         ...
11     return sommets
```

14. Préciser le tri utilisé dans la question précédente, ainsi que son coût d'exécution en temps dans le pire des cas selon le nombre n de sommets (constant, logarithmique soit en $\log_2(n)$, linéaire soit en n, quasi-linéaire soit en $n \log_2(n)$, quadratique soit en n^2, cubique soit en n^3, exponentiel soit en 2^n, ...). On fait l'hypothèse pour cette question que la fonction `degre` est de coût constant.

Pour faire des groupes de personnes qui peuvent s'entendre, une méthode consiste à *colorer* le graphe, c'est-à-dire attribuer une couleur à chacun de ses sommets, en prenant garde qu'aucune arête ne relie deux sommets de même *couleur*. Ainsi les

sommets d'une même couleur forment un groupe de personnes qui peuvent s'entendre. Par la suite, les couleurs sont représentées par des nombres entiers positifs, et −1 représente l'absence de couleur.

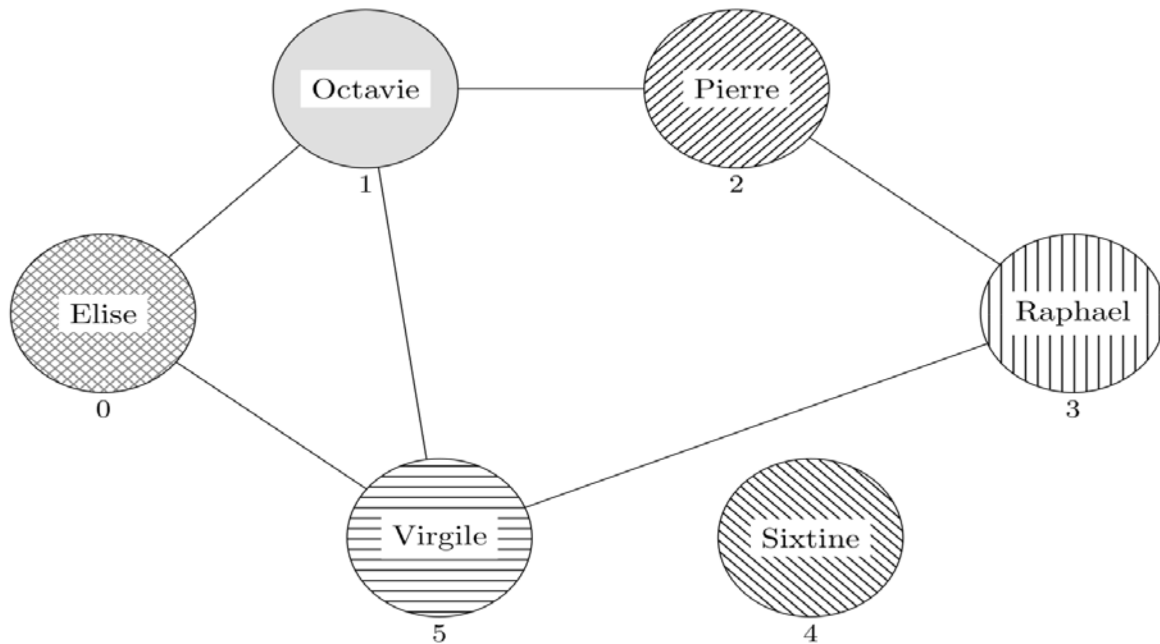

Figure 3. Graphe g1 coloré

En notant les couleurs par différents nombres précisés sous les sommets, le graphe `g1` ci-dessus est associé au dictionnaire des couleurs `dc1` suivant : `dc1 = {'Elise': 0, 'Octavie': 1, 'Pierre': 2, 'Raphael': 3, 'Sixtine': 4, 'Virgile': 5}`

On peut utiliser moins de couleurs dans cet exemple.

15. Recopier et colorer le graphe `g1` en n'utilisant que trois couleurs (0, 1 et 2).

Une méthode simple pour colorer le graphe consiste à parcourir les sommets et numéroter (colorer) chaque sommet `s` par le plus petit numéro non utilisé par ses voisins.

On dispose de la fonction `plus_petite_couleur_hors_voisins`, qui prend en paramètres

- un dictionnaire `g` représentant un graphe,
- un dictionnaire de couleurs `dc` dont les clés sont des sommets de `g`,
- une chaîne de caractères `s` correspondant à un sommet de `g`, et qui renvoie le plus petit numéro non utilisé dans `dc` par les voisins du sommet `s`.

On remarque que dans l'implémentation utilisée, les couleurs du graphe sont nécessairement numérotées entre 0 et $n-1$ (n étant le nombre de sommets).

```
 1 def couleurs_voisins(g, dc, s):
 2     return [dc[v] for v in g[s]]
 3
 4 def plus_petite_couleur_hors_voisins(g, dc, s):
 5     couleur = 0
 6     n = len(g)
 7     cvoisins = couleurs_voisins(g, dc, s)
 8     while couleur < n:
 9         if couleur not in cvoisins:
10             return couleur
11         couleur = couleur + 1
12     return couleur # au cas où len(dc) = 0
```

16. Recopier et compléter la procédure qui permet de colorer le graphe en modifiant le dictionnaire `dc` pour qu'il associe finalement à chaque sommet de `g` sa couleur.

```
1 def colorer_graphe(g, dc):
2     # Pré-condition : les clés de dc sont les sommets
3     # de g, et les valeurs de dc sont toutes à -1
4     for s in dc:
5         couleur = ...
6         ... = couleur
```

On remarque que la procédure précédente colore les sommets du graphe dans l'ordre donné par les clefs du dictionnaire `dc`. L'algorithme de Welsh-Powell consiste à colorer le graphe dans l'ordre des sommets par degré décroissant.

17. Recopier et compléter le code de la fonction `welsh_powell` donné ci-après. Cette fonction prend en paramètre un dictionnaire `g` correspondant à un graphe, et le colore selon l'algorithme de Welsh-Powell (c'est-à-dire qu'elle renvoie le dictionnaire des couleurs associé).
On pourra s'inspirer de la fonction `colorer_graphe` donnée ci-dessus et utiliser la fonction `sommets_tries`.

```
1 def welsh_powell(g):
2     # initialisation à -1 pour tous les sommets dans le
dictionnaire dc
3     dc = ... # possiblement plusieurs lignes
4     # coloration en suivant l'approche de Welsh-Powell
5     for ...
6         ...
7         ...
8     return dc
```