Exercice 3 (Bases de données, programmation python, récursivité et graphes - 8 points)
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`, `DELETE`;
- affiner les recherches à l'aide de `DISTINCT`, `ORDER BY`.

La compagnie aérienne *AirInfo* souhaite utiliser un système de gestion de bases de données relationnelles afin de l'aider à gérer les informations dont elle dispose sur les aéroports desservis, les vols proposés, les passagers et les réservations effectuées par ces passagers.

Elle crée donc la base de données `compagnie_aerienne`, constituée de quatre relations. Le schéma relationnel de cette base de données est le suivant :

- `aeroport(id_aeroport : TEXT, ville : TEXT, pays : TEXT)`
- `vol(id_vol : TEXT, aeroport_dep : TEXT, aeroport_arr : TEXT, distance : INT)`
- `passager(id_passager : INT, nom : TEXT, prenom : TEXT, ville : TEXT, d_totale : INT)`
- `reservation(id_vol : TEXT, id_passager : INT)`

Une clé primaire de la relation `vol` est l'attribut `id_vol`. Les autres clés primaires et étrangères ne sont pas précisées.

Les quatre tables ci-après constituent, en l'état, la base de données `compagnie_aerienne` complète.

Les valeurs des champs `distance` et `d_totale` intervenant respectivement dans les tables `vol` et `passager` sont exprimées en milliers de kilomètres.

aeroport		
id_aeroport	ville	pays
CDG	Paris	France
IAD	Washington	USA
QPP	Berlin	Allemagne
NRT	Tokyo	Japon
SYD	Sydney	Australie
YUL	Montréal	Canada

vol			
id_vol	aeroport_dep	aeroport_arr	distance
AI0006	CDG	IAD	6
AI0015	IAD	CDG	6
AI0256	CDG	SYD	17
AI0258	SYD	CDG	17
AI0276	CDG	NRT	10
AI0292	NRT	CDG	10
AI1280	NRT	QPP	9
AI1681	QPP	NRT	9
AI1785	NRT	SYD	8
AI1845	SYD	NRT	8

passager				
id_passager	nom	prenom	ville	d_totale
1	Dupont	Alice	Paris	6
2	Smith	John	Washington	6
3	Brown	Sarah	Berlin	9
4	Yamada	Taro	Tokyo	17
5	Williams	Emma	Sydney	10

reservation	
id_vol	id_passager
AI0006	1
AI0006	2
AI0256	4
AI0276	5
AI1681	3

1. Expliquer pourquoi l'attribut `id_vol` ne peut pas être une clé primaire de la table `reservation`.
2. Proposer une clé primaire pour la table `reservation`.
3. Expliquer le rôle d'une clé étrangère dans une relation.
4. Ecrire le résultat renvoyé par la requête SQL suivante :

```
SELECT id_vol FROM vol WHERE aeroport_arr = 'CDG';
```

5. Ecrire une requête SQL qui donne les noms des villes qui sont destination d'un vol au départ de l'aéroport CDG.

La compagnie *AirInfo* envisage de récompenser la fidélité de ses usagers en tenant compte de la distance totale qu'ils parcourent sur leurs lignes. Ainsi, à chaque nouvelle réservation, la table `passager` doit être mise à jour : si l'usager avait déjà parcouru une distance totale d_{totale}, puis réserve un vol d'une distance d_{vol}, l'attribut `d_totale` est modifié en prenant pour nouvelle valeur $d_{\text{totale}} + d_{\text{vol}}$.

6. La passagère d'identifiant 5 a déjà parcouru 10 milliers de kilomètres. Elle réserve un vol de Washington (IAD) à Paris (CDG), d'une distance de 6 milliers de kilomètres. Ecrire la requête permettant de mettre à jour la table passager.

La compagnie aérienne offre maintenant la possibilité de relier Paris CDG à Montréal YUL (Canada) par un vol de 6 milliers de kilomètres. Afin de prendre en compte ce nouveau trajet dans la base de données, on souhaite l'ajouter dans la table `vol` dont la clé primaire est `id_vol`. On propose, de manière incorrecte, la requête d'insertion ci-après :

```
INSERT INTO vol VALUES ('AI0256', 'CDG', 'YUL', 6);
```

7. Proposer, en justifiant, une correction relative à l'erreur commise.

Partie B

Dans cette partie, on considère les villes suivantes : Washington (W) ; Paris (P) ; Berlin (B) ; Tokyo (T) ; Sydney (S).

Les distances des vols entre ces villes, exprimées en milliers de kilomètres, sont les suivantes :

- Washington - Paris : 6 ;
- Washington - Berlin : 7 ;
- Paris - Berlin : 1 ;
- Paris - Tokyo : 10 ;
- Paris - Sydney : 17 ;
- Berlin - Tokyo : 9 ;
- Tokyo - Sydney : 8.

La compagnie *AirInfo* propose certains de ces vols. Son réseau aérien est représenté par le graphe pondéré non orienté de la Figure 1 dans lequel :

- chaque ville est représentée par un sommet ;
- chaque vol direct entre deux villes est représenté par une arête.

Le nombre indiqué sur chaque arête est appelé poids : il représente la distance entre deux villes. Cette distance est la même dans un sens ou dans l'autre.

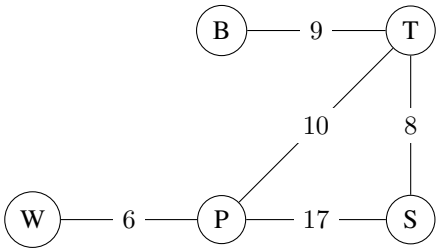

FIGURE 1 – Graphe non orienté pondéré correspondant au réseau aérien de la compagnie *AirInfo*

On choisit de représenter ce graphe en Python à l'aide d'un dictionnaire dans lequel chaque clé est un sommet du graphe et la valeur associée à cette clé est elle-même un dictionnaire, dont les clés sont les villes voisines et les valeurs sont les distances correspondantes. Le dictionnaire associé au graphe, représenté par la Figure 1, est donné ci-après.

```
graphe_airinfo = {'W': {'P': 6},
                  'P': {'W': 6, 'T': 10, 'S': 17},
                  'B': {'T': 9},
                  'T': {'B': 9, 'P': 10, 'S': 8},
                  'S': {'P': 17, 'T': 8}
                 }
```

8. Déterminer la valeur de l'expression `graphe_airinfo['T']['P']`.
9. Ecrire le code d'une fonction Python `vol_direct` permettant de déterminer s'il existe une liaison directe entre deux villes. Cette fonction prend en paramètres un dictionnaire `graphe` représentant le graphe, et deux clés du dictionnaire `ville1` et `ville2` représentant deux villes, et renvoie `True` si une telle liaison entre les villes `ville1` et `ville2` existe, c'est-à-dire si les deux sommets `ville1` et `ville2` sont reliés dans le graphe `graphe` par une arête, `False` sinon.
   Exemples :
   ```
   >>> vol_direct(graphe_airinfo, 'T', 'B')
   True
   >>> vol_direct(graphe_airinfo, 'W', 'B')
   False
   ```

Afin de limiter leurs empreintes carbone, les voyageurs demandent fréquemment aux compagnies aériennes de déterminer, à partir d'une ville donnée, la liste des villes qu'il est possible d'atteindre par un vol direct tout en ne dépassant pas une certaine distance.

10. Ecrire le code d'une fonction Python `liste_villes_proches` qui permet de répondre à la demande des voyageurs. Cette fonction prend en paramètres un dictionnaire `graphe` représentant le graphe, une clé du dictionnaire `ville` et un entier `d_max` représentant une distance et renvoie la liste des villes reliées à `ville` par une arête dans `graphe` dont le poids est au plus égal à `d_max`.
    Exemples :
    ```
    >>> liste_villes_proches(graphe_airinfo, 'T', 7)
    []
    >>> liste_villes_proches(graphe_airinfo, 'T', 9)
    ['B', 'S']
    ```

La compagnie aérienne *Droidevant*, concurrente de la compagnie *AirInfo*, assure des liaisons différentes. Son réseau aérien peut également être représenté par un graphe dont le dictionnaire correspondant en Python est donné ci-après.

```
graphe_droidevant = {'W': {'P': 6, 'B': 7},
                     'P': {'W': 6, 'B': 1},
                     'B': {'W': 7, 'P': 1},
                     'T': {'S': 8},
                     'S': {'T': 8}
                    }
```

11. Dessiner, en indiquant le poids sur chaque arête, le graphe représentant le réseau aérien de la compagnie *Droidevant*.

On dit qu'un graphe est connexe s'il existe un chemin entre chaque paire de sommets, autrement dit, si pour tout sommet `s1` et pour tout sommet `s2`, il existe un chemin entre `s1` et `s2` dans le graphe.

12. Indiquer, parmi les deux propositions suivantes, celle qui est correcte :
    - Proposition A : le graphe de la compagnie *AirInfo* est connexe ;
    - Proposition B : le graphe de la compagnie *Droidevant* est connexe.

On peut adapter un algorithme de parcours de graphe pour déterminer si un graphe est connexe. Pour cela, on considère la fonction `parcours` qui permet d'explorer les villes d'un graphe accessibles à partir d'une ville donnée et qui prend en paramètres :

- `graphe` : un dictionnaire représentant le graphe ;
- `visitees` : une liste des villes déjà visitées ;
- `ville` : la ville actuelle à partir de laquelle on effectue l'exploration.

```
def parcours(graphe, visitees, ville):
    """Parcours d'un graphe à partir d'une ville non
    visitée, en ayant déjà visité un certain nombre de
    villes.
    """
    # Marque la ville comme visitée
    visitees.append(ville)
    # Parcourt les voisines de la ville
    for voisine in graphe[ville]:
        if voisine not in visitees:
            # Explore depuis les voisines non visitées
            parcours(graphe, visitees, voisine)
```

13. Expliquer en quoi la fonction `parcours` est une fonction récursive.
14. Déterminer le contenu des variables `visitees1` et `visitees2` après l'exécution des lignes de code données ci-après.

```
visitees1 = []
parcours(graphe_airinfo, visitees1, 'W')
visitees2 = []
parcours(graphe_droidevant, visitees2, 'W')
```

15. Indiquer, parmi les propositions suivantes, laquelle correspond au type de parcours effectué par la fonction `parcours` :
    - Proposition A : un parcours en largeur ;
    - Proposition B : un parcours en grandeur ;
    - Proposition C : un parcours en profondeur.

On cherche à écrire une fonction `est_connexe` qui prend en paramètre un graphe, représenté à l'aide d'un dictionnaire de dictionnaires de voisins, et qui renvoie `True` si le graphe est connexe, `False` sinon.

On admet disposer d'une fonction `ville_arbitraire` qui renvoie un sommet arbitraire d'un graphe.
Par exemple `ville_arbitraire(graphe_airinfo)` pourrait renvoyer `'T'` ou n'importe quel autre sommet.

On considère le code de la fonction Python incomplet suivant :

```
1 def est_connexe(graphe):
2     """Vérifie si un graphe est connexe."""
3     depart = ville_arbitraire(graphe)
4     visitees = ...
5     parcours(graphe, visitees, depart)
6     return ...
```

16. Recopier et compléter les lignes 4 et 6 du code de la fonction `est_connexe`. On pourra, si nécessaire, ajouter de nouvelles lignes.

On considère maintenant le code de la fonction, donné ci-après, `mystere`, qui prend en paramètres :

- `graphe`, un dictionnaire représentant un graphe ;
- `ville`, une clé du dictionnaire `graphe` représentant une ville de départ ;
- `chemin`, une liste de clés représentant les villes ;
- `cout`, un entier représentant une distance ;
- `arrivee`, une clé du dictionnaire `graphe` représentant une ville d'arrivée.

```
def mystere(graphe, ville, chemin, cout, arrivee):
    # Ajoute la ville actuelle au chemin
    chemin = chemin + [ville]
    if ville == arrivee:
        # Affiche le chemin et son coût total
        print(chemin, cout)
    # Parcourt les villes voisines et leurs poids
    for voisine, poids in graphe[ville].items():
        # Vérifie que la ville n'est pas déjà visitée
        if voisine not in chemin:
            mystere(graphe, voisine, chemin, cout + poids, arrivee)
```

17. Déterminer les affichages produits lors de l'exécution de l'appel `mystere(graphe_airinfo, 'W', [], 0, 'B')`
18. Expliquer, de manière générale, ce que réalise l'appel `mystere(graphe, ville, [], 0, arrivee)` pour un graphe `graphe` et deux villes `ville` et `arrivee`.

Exercice 3 (Graphes, bases de données, tris, algorithmes gloutons et récursivité - 8 points)
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` et `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 trois 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.

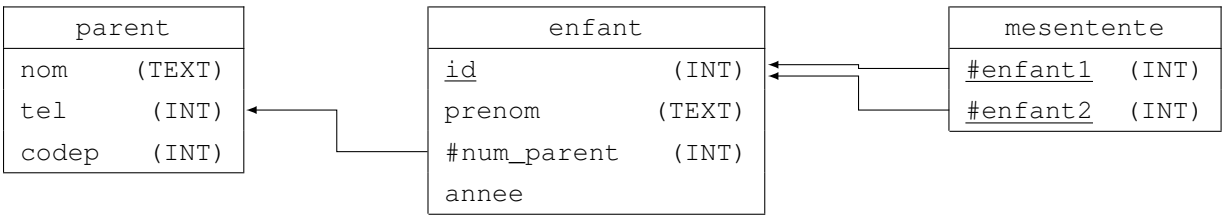

FIGURE 1 – Schéma relationnel de la BDD

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.

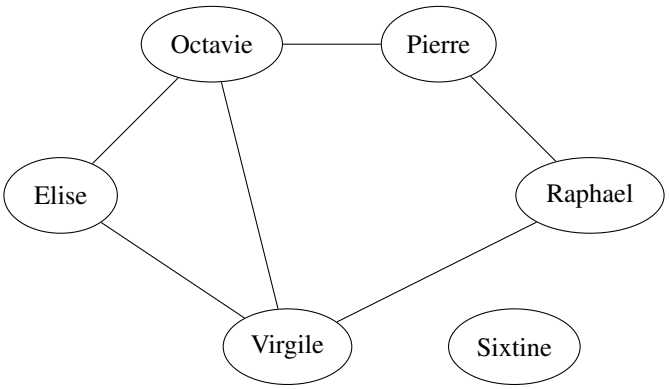

FIGURE 2 – Graphe `g1`

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

```
g1 = {'Elise': ['Octavie', 'Virgile'],
      'Octavie': ['Elise', 'Pierre', 'Virgile'],
      'Pierre': ['Octavie', 'Raphael'],
      'Raphael': ['Pierre', 'Virgile'],
      'Sixtine': [],
      'Virgile': ['Elise', 'Octavie', 'Raphael']
     }
```

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

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

```
g2 = {'Adrien': ['Elisabeth', 'Lea'],
      'Elisabeth': ['Adrien', 'Ian', 'Luca'],
      'Ian': ['Elisabeth', 'Joseph', 'Luca'],
      'Joseph': ['Ian'],
      'Lea': ['Adrien'],
      'Luca': ['Elisabeth', 'Ian']
     }
```

12. Ecrire 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, etc.). 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.

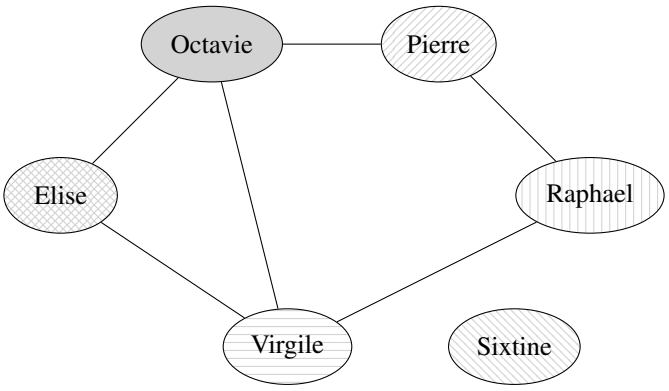

FIGURE 3 – Graphe `g1` coloré

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 pour cela 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).

```
def couleurs_voisins(g, dc, s):
    return [dc[v] for v in g[s]]

def plus_petite_couleur_hors_voisins(g, dc, s):
    couleur = 0
    n = len(g)
    cvoisins = couleurs_voisins(g, dc, s)
    while couleur < n:
        if couleur not in cvoisins:
            return couleur
        couleur = couleur + 1
    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
```