Exercice 3 (8 points)

Cet exercice porte sur les bases de données, la programmation en Python, la récursivité et les algorithmes de parcours de 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`, `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. Écrire le résultat renvoyé par la requête SQL suivante :

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

5. Écrire 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.

   Écrire 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.

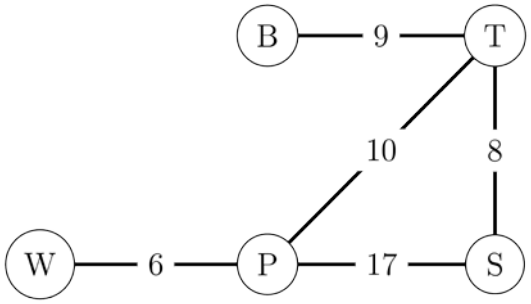

Figure 1. Graphe non orienté pondéré correspondant au réseau aérien de la compagnie *AirInfo*

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. Écrire 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. Écrire 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},
```

```
3     'P': {'W': 6, 'B': 1},
4     'B': {'W': 7, 'P': 1},
5     'T': {'S': 8},
6     'S': {'T': 8}
7 }
```

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.

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.

```
1 def parcours(graphe, visitees, ville):
2     """Parcours d'un graphe à partir d'une ville non
3     visitée, en ayant déjà visité un certain nombre de
4     villes.
5     """
6     # Marque la ville comme visitée
7     visitees.append(ville)
8     # Parcourt les voisines de la ville
9     for voisine in graphe[ville]:
10        if voisine not in visitees:
11            # Explore depuis les voisines non visitées
12            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.

```
1 visitees1 = []
2 parcours(graphe_airinfo, visitees1, 'W')
3
4 visitees2 = []
5 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.

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

17. Déterminer les affichages produits lors de l'exécution de l'appel suivant :

    ```
    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`.