Exercice 1 - base de données rugbystique

On rappelle tout d'abord que pour marquer des points dans un match de rugby, on peut réaliser un *drop* ou une *pénalité* (trois points dans les deux cas) ou un *essai* (cinq points), chaque essai pouvant être suivi d'une tentative de *transformation* (deux points de plus en cas de réussite). Ainsi, un score total de 23 est par exemple possible avec quatre essais non transformés et un drop ($4 \times (5+0) + 1 \times 3$), mais aussi deux essais transformés et trois drops ($2 \times (5+2) + 3 \times 3$), trois essais dont un transformé et deux drops ($1 \times (5+2) + 2 \times (5+0) + 2 \times 3$), et d'autres combinaisons.

1. Écrire une fonction qui prend en argument un entier positif, vu comme un score, et qui renvoie le nombre de façons de réaliser ce score au rugby. L'ordre dans lequel les points sont marqués n'est pas important.
2. Considérons le dictionnaire suivant, appelé `joueurs`, qui est une variable globale : les clés sont des équipes, et les valeurs sont des listes de noms de joueurs (il n'y a aucun doublon sur l'ensemble des listes).
   On a par exemple dans `joueurs["France"]` les éléments `"Melvyn Jaminet"`, `"Antoine Dupont"`, etc. En particulier, en tant que variable globale, ce dictionnaire n'a jamais besoin d'être mis en argument d'une fonction.
   Écrire une fonction qui prend en argument une chaîne de caractères dont on garantit qu'elle est présente quelque part dans joueurs et qui renvoie l'équipe dans laquelle le joueur associé est. Préciser la complexité de cette fonction.
3. Écrire une fonction sans argument qui construit un dictionnaire indexé par les noms de joueurs et dont les valeurs sont les équipes associées.
4. Considérons à présent un compte-rendu de match. La gestion chaotique fait que tous les événements ont été mélangés sans séparer les équipes. Ainsi, un glorieux essai français se situe côte à côte avec une misérable pénalité anglaise. Nous avons donc des comptes-rendus sous la forme de listes dont les éléments sont le joueur ayant marqué, ce qu'il a marqué (`"essai"`, `"transformation"`, `"drop"` ou `"penalite"`) et le temps de jeu associé, heureusement dans l'ordre chronologique. Un exemple d'élément sera alors `["Anthony Jelonch", "essai", 9]`.
   Écrire une fonction prenant en argument un compte-rendu et déterminant s'il est cohérent : une transformation ne peut avoir lieu que directement après un essai; le joueur doit être dans la même équipe; enfin il ne peut y avoir que (au plus) deux équipes concernées pour un compte-rendu.

La suite de l'épreuve se fera en SQL, avec la base de données suivante :

- table `Equipe`, avec pour attributs `equipe_id` (entier) et `equipe_nom` (chaîne de caractères);
- table `Joueur`, avec pour attributs `joueur_id` (entier), `joueur_nom` (chaîne de caractères) et `equipe_id` (entier);
- table `Match`, avec pour attributs `match_id` (entier), `match_date` (chaîne de caractères), `match_equipe1` (entier) et `match_equipe2` (entier);
- table `Deroulement`, avec pour attributs `match_id` (entier), `joueur_id` (entier), `match_evenement` (chaîne de caractères) et `match_moment` (entier);
- table `Points`, avec pour attributs `match_evenement` (chaîne de caractères) et `points_points` (entier).

Explications :

- les attributs nommés `id` préfixés par le nom de la table où ils figurent sont des clés primaires (dans les autres cas ce sont des clés étrangères y faisant référence);
- la plupart des attributs correspond aux informations utilisées précédemment en Python, notamment `match_evenement`;
- les attributs `match_equipe1` (entier) et `match_equipe2` sont aussi des clés étrangères et correspondent à `equipe_id`;
- la date d'un match est donnée sous la forme `AAAA-MM-JJ`;
- la table `Points` sert à faciliter le travail des dernières requêtes tout en économisant de la mémoire en ne mettant pas l'information dans chaque enregistrement de la table `Deroulement`.

5. Écrire une requête qui affiche le nom de tous les joueurs de l'équipe de France sans autre attribut.
6. Écrire une requête qui affiche le moment le plus tôt et le moment le plus tardif où des points ont été marqués sur l'ensemble des matchs.
7. Écrire une requête qui affiche le nom de tous les joueurs de l'équipe de France ayant marqué au moins un essai.
8. Écrire une requête qui affiche le nom des deux joueurs de l'équipe de France ayant marqué le plus d'essais.
9. Écrire une requête qui affiche le résultat du match France-Angleterre sous la forme d'une table contenant deux enregistrements et ayant deux attributs : `equipe_nom` et l'attribut que l'on nommera `score`. On admettra que les deux équipes ont marqué pour ne pas avoir à compliquer la requête.

1.

```python
def variantes(s):
    """ entrée
              s int
        sortie
              n entier correspondant au
                  nombre de façons
                  d'obtenir le score s au
                       rugby
    """
    n = 0
    dmax = s//3
    emax = s//5
    for d in range(dmax+1):
        for e in range(emax+1):
            for t in range(e+1):
                if 3*d + 5*e + 2*t == s:
                    n += 1
    return n
```

2.

```python
def equipe(j):
    """ entrée
              joueur j (str)
        sortie
              équipe (str) dans laquelle
                  joue le joueur
    """
    for team in joueurs:
        for player in joueurs[team]:
            if player == j:
                return team
```

La complexité est linéaire (O(n) où n est le nombre total de joueurs).

3.

```python
def dicoj():
    """ pas d'entrée
        sortie
              dictionnaire d de clés les
                  joueur,
                  et de valeurs les
                      équipes associées
    """
    d = {}
    for team in joueurs:
        for player in joueurs[team]:
            d[player] = team
    return d
```

4.

```python
def coherence(cr):
    n = len(cr)
    teams = {}
    if cr[0][1] == "transformation":
        return False
    teams[equipe(cr[0][0])] = True
    for i in range(1, n):
        if cr[i][1] == "transformation":
            if cr[i-1][1] != "essai" or
                equipe(cr[i][0]) != equipe(
                cr[i-1][0]):
                return False
        teams[equipe(cr[i][0])] = True
    if len(teams) > 2:
        return False
    return True
```

5.

```sql
SELECT joueur_nom
  FROM Equipe JOIN Joueur ON Equipe.
      equipe_id = Joueur.equipe_id
  WHERE equipe_nom = "France"
```

6.

```sql
SELECT MIN(match_moment), MAX(match_moment)
  FROM Deroulement
```

7.

```sql
SELECT DISTINCT Joueur.joueur_nom
  FROM Equipe JOIN Joueur ON Equipe.
      equipe_id = Joueur.equipe_id
              JOIN Deroulement ON Joueur.
                  joueur_id = Deroulement.
                  joueur_id
  WHERE equipe_nom = "France" AND
      match_evenement = "essai"
```

8.

```sql
SELECT Joueur.joueur_nom, COUNT(*) as nb
  FROM Equipe JOIN Joueur ON Equipe.
      equipe_id = Joueur.equipe_id
              JOIN Deroulement ON Joueur.
                  joueur_id = Deroulement.
                  joueur_id
  WHERE equipe_nom = "France" AND
      match_evenement = "essai"
  GROUP BY Joueur.joueur_id
  ORDER BY nb DESC
  LIMIT 2
```

9.

```sql
SELECT equipe_nom, SUM(points_points) AS
    score
  FROM Equipe JOIN Joueur ON Equipe.
      equipe_id = Joueur.equipe_id
              JOIN Deroulement ON Joueur.
                  joueur_id = Deroulement.
                  joueur_id
              JOIN Points ON Deroulement.
                  match_evenement = Points.
                  match_evenement
  WHERE match_id =
      ( SELECT match_id
          FROM Match JOIN Equipe AS E1 ON
              match_equipe1 = E1.equipe_id
                    JOIN Equipe AS E2 ON
                        match_equipe2 = E2.
                        equipe_id
          WHERE E1.equipe_nom = "France"
              AND E2.equipe_nom = "
              Angleterre"
                    OR E2.equipe_nom = "
                        France" AND E1.
                        equipe_nom = "
                        Angleterre"
      )
  GROUP BY equipe_nom
```

Exercice 2 - sur l'algorithme des k plus proches voisins

On redonne ci-dessous une version de l'algorithme des k plus proches voisins dans un cas assez général où la fonction de distance sur $\mathbb{R}^d$ est en argument.

```
def kNN(les_points, point_sup, k, dist):
    """ entrées
                les_points : liste des couples de la forme (point, étiquette)
                    où  point est une liste de flottants de longueur d
                        et étiquette est une chaîne de caractère
                 point_sup : point (liste de flottants de longueur d)
                 k : entier
                 dist : fonction distance entre deux éléments (points) de R^d
        sortie
                rep : chaîne de caractère
                        correspondant à l'étiquette de point_sup
    """
    distances = [(dist(P[0], point_sup), P[1]) for P in les_points]
    pass
    etiquettes = dict()
    for p, etiq in distances[:k]:
        if etiq in etiquettes:
            etiquettes[etiq] += 1
        else:
            etiquettes[etiq] = 1
    rep = etiq
    for e in etiquettes:
        if etiquettes[e] > etiquettes[rep]:
            rep = e
    return rep
```

1. À la ligne 14, l'instruction `pass` a remplacé une ligne de code indispensable. Quel était l'effet de cette ligne? (on ne demande pas de la programmer).
2. Expliquer le rôle des lignes 21 à 24. Préciser ce que vaut précisément `etiq` à la ligne 21 (par rapport aux arguments de la fonction).
3. Écrire une fonction `distance` de calcul de la distance euclidienne entre deux vecteurs de $\mathbb{R}^d$ qui puisse être compatible avec la fonction `kNN`.
4. On souhaite modifier la fonction de sorte que, plus un des k plus proches voisins est proche, plus il va contribuer pour le choix de son étiquette.

   La formule suggérée est de le pondérer par l'inverse de la distance au point supplémentaire (en traitant le cas particulier d'une distance nulle : dans ce cas l'étiquette est forcément retenue).

   On admettra que les points fournis sont distincts deux à deux.

   Écrire une nouvelle version de la fonction `kNN` (on pourra se contenter de ne rédiger que les modifications à apporter à la fonction de l'énoncé).

1. Il s'agit de trier la liste `distance` en fonction du premier paramètre (par ordre croissant). Donc une instruction de la forme :

```
distance = tri(distance) # tri de
    distance selon le premier paramètre
```

2. Le dictionnaire `etiquettes` répertorie les étiquettes (*labels*) rencontrées parmi les k plus proches voisins de `point_sup` et détermine la plus fréquente.

La ligne 21 correspond à l'initialisation de la variable `rep` avec la valeur `etiq` qui correspond à l'étiquette rencontrée en dernier (donc celle du k-ième plus proche voisin).

Les lignes 22 à 24 correspondent à l'examen de toutes les étiquettes rencontrées et considère leur nombre d'occurrences; on modifie `rep` lorsque l'on trouve une étiquette plus fréquente que la précédente retenue.

3.

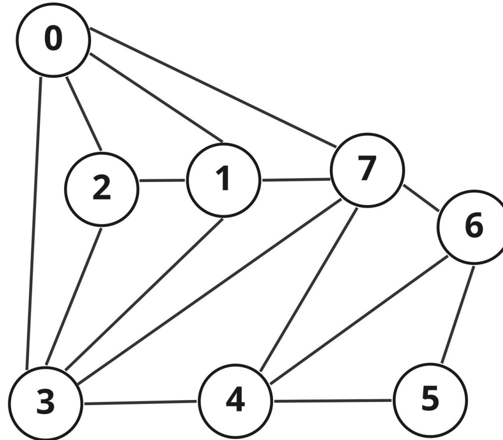
```
from math import sqrt

def distance(A, B):
    """ entrée
                A, B listes de flottants de
                    même longueur
        sortie
                flottant correspond à la
                    distance euclidienne
                    entre les vecteurs
                    représentés par A et B
    """
    d = len(A)
    s = 0
    for k in range(d):
        s += (A[k] - B[k])**2
    return sqrt(s)
```

4.

```
def kNN_modif(les_points, point_sup, k,
    dist):
    distances = [(dist(P[0], point_sup), P
        [1]) for P in les_points]
    distance = tri(distance) # tri de
        distance selon le premier paramètre
    if distance[0][0] == 0:  # si point_sup
         est dans l'échantillon...
        return distance[0][1]  # ... alors
            on renvoie son étiquette
    etiquettes = dict()
    for i in range(k):
        etiq = distance[i][1]
        if etiq in etiquettes:
            etiquettes[etiq] += 1/distance[
                i][0]
        else:
            etiquettes[etiq] = 1/distance[i
                ][0]
    rep = etiq
    for e in etiquettes:
        if etiquettes[e] > etiquettes[rep]:
            rep = e
    return rep
```

Exercice 3 - coloration d’un graphe

On considère le graphe non orienté suivant :



1. Représenter le graphe de l’énoncé à l’aide d’un dictionnaire `G` dont les clés sont les noms (entiers de 0 à 7) des sommets et les valeurs associées sont les listes des voisins. Par exemple :

```
>>> G[4]
[3, 5, 6, 7]
```

2. Écrire une fonction `voisins(gr, i, j)` où `gr` est un dictionnaire représentant un graphe comme ci-dessus, `i` et `j` sont des numéros de sommets différents, et qui renvoie un booléen indiquant si ces deux sommets sont voisins ou non. Par exemple :

```
>>> voisins(G, 0, 1)
True
>>> voisins(G, 0, 5)
False
```

3. Écrire une fonction `aretes(gr)` où `gr` est un dictionnaire représentant un graphe comme ci-dessus et qui renvoie le nombre d’arêtes de ce graphe. Par exemple :

```
>>> aretes(G)
15
```

Le but de ce qui suit est de colorer ce graphe (c’est-à-dire d’attribuer une couleur pour chaque sommet) de sorte que deux sommets adjacents n’aient pas la même couleur et en cherchant à minimiser le nombre de couleurs. Les couleurs seront représentées par des numéros : 0, 1, etc.

4. La fonction `colorerSommet(gr,i,couleurs)` ci-dessous prend en arguments un dictionnaire `gr` représentant un graphe (comme ci-dessus), un entier `i` correspondant à un sommet et une liste `couleurs` (dont la longueur est le nombre de sommets) et qui indique le numéro de la couleur attribuée au sommet ou -1 si le sommet n’est pas coloré.

   Cette fonction ne renvoie rien mais modifie la liste `couleurs` pour attribuer le plus petit numéro possible au sommet `i`. Par exemple :

```
>>> C = [0, 1, 2, -1, -1, -1, -1, -1]
>>> colorerSommet(G, 3, C)
>>> C
[0, 1, 2, 3, -1, -1, -1, -1]
>>> colorerSommet(G, 4, C)
>>> C
[0, 1, 2, 3, 0, -1, -1, -1]
```

Compléter les lignes 17, 18, 21 et 22 en remplaçant les tirets :

```
def colorerSommet(gr, i, couleurs):
    """ entrée
                gr dictionnaire représentant un graphe
                i (int) nom d’un sommet
                couleurs : liste d’entiers telle que couleurs[j]
                    soit la couleur attribuée au sommet j;
                    couleurs[j] = -1 si j n’est pas encore coloré

        La fonction modifie couleurs en attribuant la couleur la plus
            petite possible (plus petit numéro) au sommet i

        pas de sortie
    """
    n = len(couleurs)
    col = [] # liste des voisins de i déjà colorés
    for j in gr[i]:
        if ------
            ------
    # on recherche maintenant la plus petite couleur possible
    new = 0
    while ------:
        ------
    couleurs[i] = new
```

5. Écrire une fonction `colorer(gr)` d’argument un dictionnaire gr représentant un graphe et qui renvoie la liste des couleurs attribuées aux sommets. On doit donc avoir :

```
>>> colorer(G)
[0, 1, 2, 3, 0, 1, 2, 4]
```

1.

```
G = {0: [1, 2, 3, 7],
     1: [0, 2, 3, 7],
     2: [0, 1, 3],
     3: [0, 1, 2, 4, 7],
     4: [3, 5, 6, 7],
     5: [4, 6],
     6: [4, 5, 7],
     7: [0, 1, 3, 4, 6]}
```

2.

```
def voisins(gr, i, j):
    """ entrée
                gr dictionnaire
                    représentant un graphe
                i et j (int) deux sommets
                    différents
        sortie
                booléen indiquant si i et j
                     sont voisins
    """
    return i in gr[j]
```

3.

```
def aretes(gr):
    """ entrée
                gr dictionnaire
                    représentant un graphe
        sortie
                nombre (int) d’arêtes de ce
                     graphe
    """
    a = 0
    for s in gr:
        a += len(gr[s])
    return a//2
```

4.

```
    for j in gr[i]:
        if couleurs[j] != -1:
            col.append(couleurs[j])
```

```
    new = 0
    while new in col:
        new += 1
```

5.

```
def colorer(gr):
    """ entrée
                gr dictionnaire
                    représentant un graphe
        sortie
                liste des couleurs
                    attribuées à chaque
                    sommet
    """
    n = len(gr)
    coul = [-1 for k in range(n)]
    for k in range(n):
        colorerSommet(gr, k, coul)
    return coul
```