# Amérique du sud - septembre 2024 - sujet 1

**Exercice 1 (Programmation Python, POO, bases de données et requêtes SQL - 6 points)**
*Cet exercice est composé de deux parties indépendantes A et B.*

Un éditeur de jeux produit à la fois des logiciels de jeux à installer sur un ordinateur et des applications Web de jeux accessibles en ligne via un navigateur.

**Partie A**
Dans cette partie, on se place dans le cadre d'un logiciel de jeux et on s'intéresse uniquement au code gérant les pseudonymes des joueurs et les scores associées.

On considère le programme inachevé en langage Python ci-dessous qui utilise la programmation orientée objet :

```
1  class Joueur:
2      """ Définition de la classe 'Joueur' """
3
4      def __init__(self, pseudo: str):
5          """ Méthode d'initialisation """
6          self.pseudo = pseudo
7          self.score = 0
8
9      def augmenter_score(self, nb_points: int):
10     """ Méthode augmentant le score d'un certain nombre de points """
11     ...
12
13     def retourne_score(self) -> tuple:
14     """ Méthode retournant le pseudo et le score sous forme d'un tuple """
15     ...
16
17 joueurs = []
18 joueurs.append(Joueur('THEBEST'))
19 joueurs.append(Joueur('PAS2BOL'))
20 ...
```

1. Ecrire sur la copie une instruction à placer à la ligne 11 pour que le score du joueur augmente du nombre de points passés en paramètre.
2. Ecrire sur la copie une instruction à placer à la ligne 20 permettant d'augmenter le score du joueur `'THEBEST'` de 3 points en faisant appel à la méthode `augmenter_score`.
3. Ecrire sur la copie une instruction à placer à la ligne 15 pour permettre à la méthode `retourne_score` de renvoyer sous forme d'un tuple le pseudonyme et le score d'un joueur. Par exemple, cette méthode doit renvoyer le couple `('THEBEST', 3)` pour le premier joueur.

On souhaite sauvegarder à un instant donné les pseudonymes et les scores associés. Pour cela, on utilise la fonction suivante :

```
1 def enregistrer(joueurs):
2     tab_scores = {}
3     for joueur in joueurs:
4         p, s = joueur.retourne_score()
5         tab_scores[p] = s
6     return tab_scores
7
8 tableau_scores = enregistrer(joueurs)
```

4. Indiquer quel est le type de la variable `tableau_scores` dans ce programme.

5. Donner le résultat renvoyé par cette fonction si on l'exécute sur la liste `joueurs` précédente, qui contient un premier joueur de pseudonyme `'THEBEST'` dont le score est 3 et un second joueur `'PAS2BOL'` dont le score est 0.
6. Indiquer ce qu'il se passe si la liste `joueurs` en paramètre contient plusieurs joueurs avec le même pseudonyme.
7. Ecrire une fonction `chercher_gagnant(tableau_scores)` qui prend en argument un tableau des scores obtenu par la fonction `enregistrer`, et qui renvoie le pseudonyme d'un des joueurs ayant réalisé le score le plus grand. On suppose que les scores sont toujours positifs ou nuls.

**Partie B**

Dans cette partie, on se place dans le cadre d'applications Web de jeux accessibles en ligne via un navigateur. L'éditeur commercialise plusieurs jeux et fait donc appel à un système de gestion de base de données (SGBD) qui utilise le langage SQL.

On pourra utiliser les mots-clés suivants du langage SQL : `SELECT`, `FROM`, `WHERE`, `JOIN`, `ON`, `INSERT`, `INTO`, `VALUES`, `UPDATE`, `SET`, `DELETE`, `ORDER`, `BY`, `ASC`, `DESC`, `AND` et `OR`.

Le schéma relationnel de la base `Jeux_WEB` est constitué de trois relations (ou tables) :

`joueurs(`**`id_joueur`**`, pseudo, mail, mot_de_passe)`

`jeux(`**`id_jeu`**`, nom_jeu, type_jeu)`

`scores(`**`id_jeu, id_joueur`**`, score_joueur)`

Dans ce schéma relationnel, les clés primaires sont en gras et les clés étrangères n'y sont pas représentées.

Tous les attributs dont le nom est préfixé par `id` sont des nombres entiers ainsi que l'attribut `score_joueur`. Les autres attributs sont des chaînes de caractères.

8. Indiquer, pour chacun des trois enregistrements ci-dessous, la table à laquelle ils appartiennent.
   - ⋆ Enregistrement 1 :

| | | |
|---|---|---|
| 344 | Osmondia | Aventure |

   - ⋆ Enregistrement 2 :

| | | |
|---|---|---|
| 344 | 1034 | 3 |

   - ⋆ Enregistrement 3 :

| | | | |
|---|---|---|---|
| 1034 | THEBEST | thebest@meilleur.fr | the42Best |

9. Indiquer un problème que peut poser le fait de stocker les mots de passe en clair des joueurs dans une table.
10. indiquer si, avec le schéma relationnel proposé, deux joueurs peuvent ou non avoir le même pseudonyme.
11. Identifier les clés étrangères de la table `scores` et, pour chaque clé étrangère, indiquer la table et l'attribut auxquels la clé étrangère fait référence.
12. Justifier s'il est possible ou non que dans ce schéma relationnel un joueur puisse joueur à plusieurs jeux.
13. Expliquer ce que l'on obtient si on exécute successivement les deux requêtes suivantes :

```
INSERT INTO joueurs
VALUES (1035, 'PAS2BOL', 'titi@gmail.com', 'LoB2SaP');

INSERT INTO joueurs
VALUES (1035, 'SPECTRUS', 'titi@gmail.com', 'LoB2SaP');
```

14. Ecrire une requête SQL permettant d'ajouter un enregistrement à la table `scores` pour le joueur d'identifiant 1042 à un jeu d'identifiant 24 avec un score initial de 0 point.
15. Ce joueur d'identifiant 1042 venant de terminer une partie au jeu d'identifiant 24 avec un score de 8 points, écrire une requête SQL permettant de mettre à jour son score.

**Exercice 2 (Programmation Python, dictionnaires et algorithmique - 6 points)**
Lorsque l'énoncé demande la manipulation de la structure de données abstraites liste, on utilisera les `list` en Python avec la méthode `append`.

On rappelle que si `mot` est une chaîne de caractères, alors `mot[i]` renvoie le caractère de `mot` situé à la position `i+1`.
Par exemple, si `mot = 'bonjour'`, alors `mot[2]` est le caractère `'n'`.

**Partie A**
Deux mots sont des anagrammes s'ils possèdent exactement les mêmes lettres, mais dans un ordre différents. Par exemple,

- `AUBE` et `BEAU` sont des anagrammes ;
- `PIRATE`, `PARTIE`, `PATRIE` et `PARITE` sont aussi des anagrammes.

On appelle *signature* d'un mot la suite de ses lettres données dans l'ordre alphabétique. Si un mot possède plusieurs fois la même lettre, elle est répétée autant de fois qu'elle apparaît dans le mot. Par exemple, la signature du mot `AUBE` est `ABEU`.

1. Donner sans justifier la signature des mots `BEAU`, `PIRATE` et `PATRIE`.

On donne ci-dessous le code Python d'une fonction `signature` qui prend en paramètre une chaîne de caractères `mot` et qui renvoie la chaîne de caractères correspondant à la signatre de `mot`.

On suppose que l'on dispose d'une fonction `inserer_lettre` qui prend en paramètres une chaîne de caractères `mot`, une chaîne de caractères `lettre` composée d'une seul caractère et un entier `pos`. Cette fonction renvoie une nouvelle chaîne de caractères constituée des lettres de `mot` dans lesquelles la lettre `lettre` aura été insérée à la position `pos`. Par exemple,

- l'instruction `inserer_lettre('MARE', 'G', 3)` renvoie `'MARGE'` ;
- l'instruction `inserer_lettre('MER', 'A', 0)` renvoie `'AMER'`.

```
1 def signature(mot):
2     chaine = mot[0]
3     for i in range(1, len(mot)):
4         k = 0
5         while (k < len(chaine)) and (mot[i] > chaine[k]):
6             k = k + 1
7         chaine = inserer_lettre(chaine, mot[i], k)
8         print(chaine)
9     return chaine
```

2. Ecrire les affichages successifs produits par la ligne 8 lors de l'appel `signature('PLANTE')`.
3. Ecrire une fonction `sont_anagrammes` qui prend en paramètres deux chaînes de caractères `mot1` et `mot2` et qui renvoie `True` si `mot1` et `mot2` sont des anagrammes, et `False` sinon.

On donne ci-dessous le code Python d'une fonction `mystere`.

```
1 def mystere(liste):
2     dico = {}
3     for mot in liste:
4         s = signature(mot)
5         if s in dico:
6             dico[s].append(mot)
7         else:
8             dico[s] = [mot]
9     return dico
```

4. Ecrire le résultat renvoyé par l'instruction suivante :

```
mystere(['PIRATE', 'AUBE', 'PATRIE', 'ELEPHANT', 'PARTIE', 'BEAU'])
```

5. Ecrire un script Python permettant d'implémenter la fonction `inserer_lettre`.

**Partie B**
On considère un jeu de lettres constitué de jetons sur lesquels sont inscrits une lettre. Ce jeu consiste à former des mots à partir des lettres que l'on possède puis à placer ces mots sur une grille de façon à ce qu'ils s'intègrent aux mots déjà présents sur la grille.

Chaque joueur dispose de 7 jetons (donc de 7 lettres), tirés au hasard. Par exemple, avec les jetons portant les lettres `A`, `A`, `E`, `R`, `T`, `U` et `U`, les mots `RATE`, `TARE`, `RATEAU` et `TAUREAU` sont des mots possibles.

Un joueur n'a donc pas l'obligation de former un mot de 7 lettres, le mot peut être plus court. En revanche, le mot devra figurer dans une liste de mots fournie par le jeu.

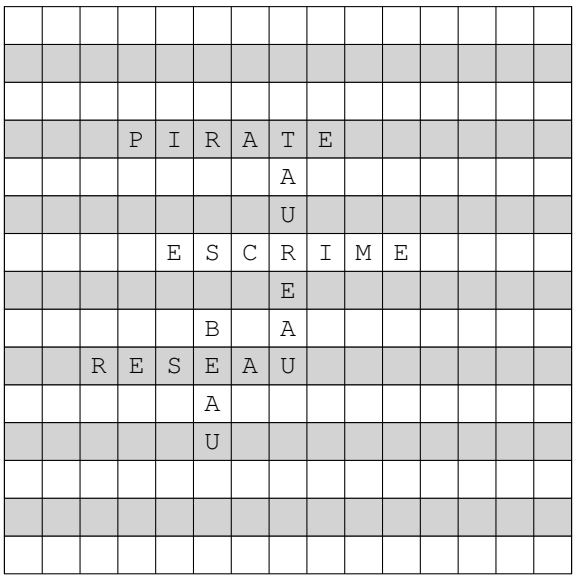


FIGURE 1 – Exemple de partie

Chaque lettre est pondérée par un score qui dépend de la difficulté à utiliser cette lettre dans un mot. Pour cela, on utilisera un dictionnaire `score_lettres` indiquant le nombre de points correspondant à chaque lettre. Ce dictionnaire est défini dans le corps de la fonction `score_mot` ci-dessous.

La fonction `score_mot` prend en paramètre une chaîne de caractères `mot` et renvoie un entier correspondant au score de `mot` d'après les informations du dictionnaire `score_lettres`. Par exemple, l'instruction `score_mot('PIRATE')` renvoie `8`.

```
1  def score_mot(mot):
2      score_lettres = {'A':1, 'B':3, 'C':3, 'D':2, 'E':1, 'F':4, 'G':2, 'H':4,
3                       'I':1, 'J':8, 'K':10, 'L':1, 'M':2, 'N':1, 'O':1, 'P':3,
4                       'Q':8, 'R':1, 'S':1, 'T':1, 'U':1, 'V':4, 'W': 10, 'X':10,
5                       'Y': 10, 'Z':10}
6      ...
```

6. Ecrire le script de la fonction `score_mot` à partir de la ligne 6.
7. Ecrire une fonction `meilleur_mot` qui prend en paramètre une liste de chaîne de caractères `liste_mots` et qui renvoie la chaîne de caractères de `liste_mots` dont le score est le plus élevé. Par exemple, avec la liste de mots

   ```
   L = ['PIRATE', 'AUBE', 'SOIGNER', 'CHAPEAU', 'DRAPEAU']
   ```

   l'instruction `meilleur_mot(L)` renvoie `'CHAPEAU'`.

Dans toute la suite de l'exercice, on appellera *chevalet* le support sur lequel le joueur dispose ses jetons.

En Python, le contenu d'un chevalet sera représenté par une liste de caractères. Par exemple, si le joueur dispose des lettres `A`, `Z`, `T`, `R`, `N`, `E` et `P`, le chevalet sera représenté par la liste `['A', 'Z', 'T', 'R', 'N', 'E', 'P']`.

A chaque fois que le joueur dépose des lettres sur la grille, il est nécessaire de mettre à jour son chevalet. Par exemple, si le chevalet du joueur correspond à la liste `['A', 'E', 'S', 'E', 'E', 'S', 'R']` et que le joueur dépose les letttes `R`, `E`, `S`, `E` et `A`, alors son chevalet devient `['E', 'S']`. Cette mise à jour du chevalet est assurée par la fonction `chevalet_apres_depot` qu'on ne demande pas de programmer. Par exemple, l'appel

```
chevalet_apres_depot(['A', 'E', 'S', 'E', 'E', 'S', 'R'], ['R', 'E', 'S', 'E', 'A'])
```

renvoie la liste `['E', 'S']`.

En début de partie, puis à chaque fois que le joueur a déposé des lettres sur la grille, il faut compléter son chevalet pour qu'il contienne sept jetons. Pour cela, on donne ci-après le code incomplet d'une fonction `pioche` qui prend en paramètre une liste de caractères `liste_lettres` éventuellement vide. Cette fonction permet de compléter le chevalet d'un joueur en y ajoutant autant de lettres que nécessaire. Par exemple,

- en début de partie, l'instruction `pioche([])` permet de renvoyer une liste de 7 caractères aléatoires ;
- en cours de partie, si, après avoir déposé des lettres sur la grille, le joueur dispose encore des lettres `C`, `A`, `E`, `T` et `C`, alors l'appel `pioche(['C', 'A', 'E', 'T', 'C'])` peut renvoyer la liste `['C', 'A', 'E', 'T', 'C', 'S', 'U']`, c'est-à-dire que le joueur a tiré, au hasard, un jeton porant la lettre `S` et un jeton portant la lettre `U`.

```
1  from random import randint
2  def pioche(liste_lettres):
3      lettres = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
4                 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
5      m = len(liste_lettres)
6      while ... :
7          # Tirage aléatoire d'une lettre
8          nb_alea = randint(0, 25)
9          tirage = lettres(...)
10         liste_lettres.append(tirage)
11         m = m + 1
12     return liste_lettres
```

8. Recopier et compléter les lignes 6 et 9 de la fonction `pioche`.

Le fichier `mots_valides.txt` contient un peu plus de 142 000 mots, écrits en minuscules et sans accent à raison d'un mot par ligne. Chaque mot est donc séparé par un caractère *retour à la ligne* représenté par `n` en Python. On donne ci-dessous les cinq premières lignes de ce fichier.

```
abaissement
abaisser
abandon
abandonnant
abandonne
```

La méthode `strip` appliquée à une chaîne de caractères `mot` permet, entre autre, de renvoyer la même chaîne de caractères privée du caractère `n` si celui-ci est en fin de la chaîne de caractères `mot`. On donne une illustration du fonctionnement de cette méthode ci-dessous :

```
>>> mot = 'abaissement\n'
>>> mot = mot.strip()
>>> mot
'abaissement'
```

Dans le jeu, un mot constitué avec les lettres du chevalet est considéré comme valide uniquement si ce mot est un des mots de ce fichier texte.

On donne le code Python suivant qui permet de créer le dictionnaire des anagrammes des mots contenus dans le fichier `mots_valides.txt`. On suppose que ce fichier est enregistré dans le même répertoire que le script python.

```
1  def lecture(mon_fichier):
2      fichier = open(mon_fichier, 'r', encoding = 'UTF-8')
3      liste_mots = []
4      for ligne in fichier:
5          mot = ligne.strip()
6          liste_mots.append(mot)
7      fichier.close()
8      return liste_mots
9  L = lecture('mots_valides.txt')
10 dico = mystere(L)
```

9. Expliquer ce que signifie la valeur `'r'` prise par l'un des arguments de la fonction `open` à la ligne 2.

Après l'exécution de ce script, l'instruction `dico['EIMPRS']` renvoie la liste des anagrammes

`['IMPERS', 'MEPRIS', 'PERMIS', 'PRIMES', 'PRISME']`.

On dispose également d'une fonction `signatures_depuis_tirage` qui, à partir d'un tirage de lettres donné sous la forme d'une liste de caractères, renvoie la liste des signatures de tous les mots construits à partir de ce tirage. Par exemple, l'instruction suivante, avec une liste de quatre lettres en argument, `signatures_depuis_tirage(['A', 'B', 'C', 'D'])` renvoie la liste `['A', 'B', 'AB', 'C', 'AC', 'BC', 'ABC', 'D', 'AD', 'BD', 'ABD', 'CD', 'ACD', 'BCD', 'ABCD']`.

On donne ci-dessous le début d'un script python :

```
1 chevalet = pioche([])
2 mots_possibles = []
3 liste_signatures = signatures_depuis_tirage(chevalet)
4 ...
```

10. Recopier et compléter ce script afin de répondre aux trois points suivants :
    - ⋆ déterminer tous les mots du fichier `mots_valides.txt` qu'il est possible d'écrire à partir de la liste de lettres contenues dans la variable `chevalet` ;
    - ⋆ afficher le mot dont le score est le plus important ;
    - ⋆ le programme devra compléter le chevalet pour qu'il contienne à nouveau sept jetons, en supposant que le joueur a déposé sur la grille les lettres permettant de constituer le mot dont le score est le plus élevé.

**Exercice 3 (Graphes - 8 points)**

Lorsque l'énoncé demande la manipulation de la structure de données abstraites liste, on utilisera les `list` en Python avec la méthode `append`.

**Partie A**

On rappelle qu'un graphe non orienté est composé

- ⋆ d'un ensemble de sommets $S$ ;
- ⋆ d'un ensemble d'arêtes $A$ de la forme $\{X ; Y\}$, où $X$ et $Y$ sont deux sommets de l'ensemble $S$.

On considère le graphe non orienté suivant :

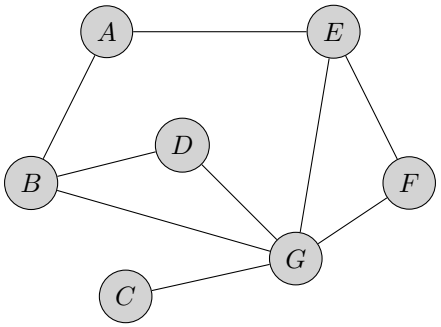


FIGURE 1 – Graphe non orienté `graphe_1`

1. Déterminer le nombre de sommets et le nombre d'arêtes de ce graphe.
2. Donner la matrice d'adjacence de ce graphe en utilisant l'ordre alphabétique pour ordonner les sommets du graphe.
3. Pour chaque sommet, donner la liste de ses voisins.

Dans toute la suite de l'exercice, on implémente un graphe par un dictionnaire (`dict` en Python) dont les clés sont les sommets au format `str` et les valeurs sont des listes contenant les sommets successeurs.

4. Tracer sur votre copie le graphe implémenté par le dictionnaire suivant :

```
graphe_2 = {'A': ['B', 'C'],
            'B': ['A', 'C'],
            'C': ['A', 'B'],
            'D': ['E'],
            'E': ['D'],
            'F': []}
```

5. On donne le code Python suivant :

```
1 def fonction_mystere(graphe):
2     """ entree : (dictionnaire)
3         sortie : (booleen) """
4     for sommet in graphe:
5         liste_voisins = graphe[sommet]
6         for voisin in liste_voisins:
7             if sommet not in graphe[voisin]:
8                 return False
9     return True
```

Déterminer ce que permet de tester cette fonction sur un graphe.

6. On considère que le parcours des voisins d'un sommet se fait dans l'ordre alphabétique. On donne ci-dessous le code Python permettant de réaliser cette opération :

```
1 def parcours_profondeur_depuis(sommet, graphe, marquage):
2     """ Paramètres : - sommet est une chaîne de caractères
3                      - graphe et marquage sont des dictionnaires """
4     marquage[sommet] = True
5     for voisin in graphe[sommet]:
6         if marquage[voisin] == False:
7             parcours_profondeur_depuis(voisin, graphe, marquage)
```

Déterminer l'ordre de parcours du graphe `graphe_1` obtenu par un parcours en profondeur en partant du sommet $A$.

7. Ecrire ce qu'affiche l'exécution du code ci-dessous.

```
marquage_2 = {sommet: False for sommet in graphe_2}
parcours_profondeur_depuis('A', graphe_2, marquage_2)
print(marquage_2)
```

8. Recopier et compléter la fonction Python ci-après `sommets_accessibles` qui prend en paramètre un `sommet` et un `graphe`. Cette fonction renvoie une liste de tous les sommets du `graphe` qui sont accessibles depuis le `sommet` en utilisant un parcours en profondeur.

```
def sommets_accessibles(sommet, graphe):
    marquage = {sommet: False for sommet in graphe}
    parcours_profondeur_depuis(sommet, graphe, marquage)
    sommets_accessibles = []
    for ... :
        if ... :
            ...
    return sommets_accessibles
```

9. On souhaite parcourir tous les sommets d'un graphe. Après avoir expliqué pourquoi la fonction de la question précédente n'affiche pas tous les sommets (on pourra s'inspirer du `graphe_2`), recopier et compléter avec plusieurs lignes le code Python ci-dessous pour implémenter un tel parcours.

```
def parcours_profondeur(graphe):
    marquage = {sommet: False for sommet in graphe}
    ...
```

**Partie B**

On dit qu'un graphe non orienté `G` est connexe s'il est d'un seul tenant, c'est-à-dire si quels que soient les sommets `s1` et `s2` du graphe `G`, il existe un chemin allant de `s1` à `s2`.

Comme dans la partie précédente, on implémente un graphe par un dictionnaire (`dict` en python) dont les clés sont les sommets au format `str` et le valeurs sont des listes contenant les sommets successeurs.

Dans cette partie, on ne considère que des graphes connexes. Les sommets sont des noms de personnes et les arêtes représentent des liens d'amitiés sur un réseau social.

Exemple :

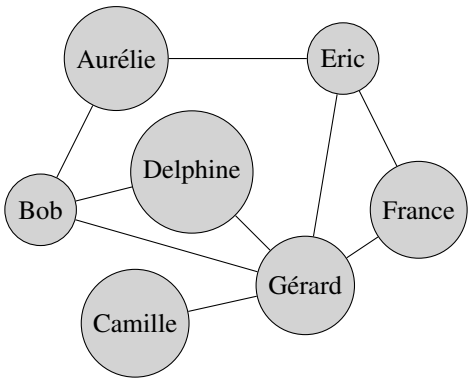


FIGURE 2 – Graphe non orienté `graphe_3`

On pourra dans cette partie utiliser la structure de données abstraite `File` munie des opérations suivantes :

- `creer_file_vide()` : renvoie une file vide ;
- `est_vide(F)` : renvoie `True` si la file `F` est vide, et `False` sinon ;
- `enfiler(F, element)` : ajoute `element` à l'entrée de la file `F` ;
- `defiler(F)` : renvoie l'élément en sortie de la file `F` en la retirant de la file `F`.

On souhaite calculer la « distance d'amitié » entre deux personnes, c'est-à-dire déterminer, en nombre d'arêtes, la chaîne de longueur minimale qui sépare deux personnes.

10. Donner la « distance d'amitié » entre deu amis et la « distance d'amitié » entre `Aurélie` et `Camille`.

On considère le code Python incomplet suivant permettant d'obtenir, pour une personne du graphe `G` et sous la forme d'un dictionnaire, les distances qui la séparent de toutes les autres personnes du graphe.

```
1  def distance_amitie_depuis(personne, G):
2      distance = {sommet: -1 for sommet in G}
3      F = creer_file_vide()
4      distance[personne] = 0
5      enfiler(F, personne)
6      while not(est_vide(F)):
7          prenom = ...
8          for ami in G[prenom]:
9              if ... :
10                 distance[ami] = ...
11                 enfiler(F, ami)
12     return distance
```

11. Déterminer le type de parcours de graphe utilisé dans la fonction `distance_amitie_depuis`.
12. Recopier et compléter le code de la fonction `distance_amitie_depuis`.
13. Déterminer la ou les instructions python permettant d'obtenir la distance entre `Aurélie` et `Camille`.
14. En s'inspirant du code précédent, écrire en Python une fonction `chaine_entre`. Cette fonction prend en paramètres deux prénoms `personne_1` et `personne_2`. Cette fonction renvoie la liste des prénoms à suivre pour relier `personne_1` et `personne_2`.
    Par exemple, `chaine_entre('Aurélie', 'France')` renvoie la liste `['Aurélie', 'Eric', 'France']`.