TP (S2) 3 Notion de graphe

Objectifs

- Maîtriser le vocabulaire associé aux graphes
- Réviser la notion de dictionnaire
- Savoir utiliser les implémentations d'un graphe sous forme de liste d'adjacences et sous forme de matrice d'adjacences.

Fichier externe ?

OUI `TP_GraphesNotion.py` *(présent(s) dans le répertoire partagé de la classe)*

1 RETOUR SUR LA STRUCTURE DE DICTIONNAIRE

Exercice 1 **Reprise de contact !** [Sol 1] Le but de cet exercice est de manipuler à nouveau la structure de dictionnaire que nous avions vue lors du premier semestre. Cette structure sera utile pour l'implémentation des graphes en Python. On considère la fonction suivante, prévue pour être appliquée sur une liste d'entiers :

```
def occurrences(L):
    D = {} # dictionnaire vide
    for x in L:
        if x in D:
            D[x] += 1
        else:
            D[x] = 1
    return D
```

1. Écrire cette fonction en précisant sa signature et une docstring.
2. Écrire une fonction `PlusFrequent(L:list)->int` à qui on fournit une liste non vide d'entiers et qui renvoie l'entier qui apparaît le plus fréquemment dans la liste `L` en utilisant la fonction `occurrences`. Dans le cas où plusieurs entiers vérifient cette propriété, on renverra arbitrairement l'un des deux.
3. Écrire une fonction `CompareListes(L1:list,L2:list)->bool` qui détermine si deux listes contiennent les mêmes éléments, avec pour chacun le même nombre d'occurrences.

Exercice 2 **Second maximum** [Sol 2] On veut écrire une fonction à qui on fournit une liste non vide d'entiers L et qui renvoie le second maximum de L. Par exemple le second maximum de la liste $L = [2,4,1,5,3]$ est 4. Nous allons proposer pour cela un algorithme appelé algorithme du **tournoi**.

Il s'agit d'organiser une succession de « matchs » entre les éléments de la liste (le plus grand élément remportant la rencontre) selon le principe d'un tournoi à élimination directe. Le vainqueur du tournoi est bien sûr le maximum de la liste. En observant la liste des éléments que ce maximum a rencontrés et vaincus, on constate que son plus grand élément est le second maximum de la liste initiale (en effet le second maximum ne peut avoir été vaincu que par le maximum).

Pour plus de clarté, présentons le déroulement de l'algorithme du tournoi sur l'exemple de la liste :

$$L = [3,1,4,5,2,2,1,3,2,4].$$

- Dans un premier temps, on organise les rencontres des joueurs pris 2 à 2 : 3 rencontre 1 ; 4 rencontre 5 ; 2 rencontre 2 ; 1 rencontre 3 et 2 rencontre 4. Chaque vainqueur ou ex-aequo est stocké dans une nouvelle liste V (en cas d'ex-aequo, comme pour la troisième rencontre, on stocke une seule fois la valeur), on obtient ici : $V = [3,5,2,3,4]$.
- D'autre part, on construit un dictionnaire D ayant pour clés les vainqueurs de chaque rencontre et pour valeur une liste contenant l'élément qu'il a vaincu (ou

les éléments si la même valeur a vaincu dans plusieurs matchs). Attention ! en cas d'ex-aequo il n'y a ni vainqueur ni vaincu à stocker dans le dictionnaire. On obtient ici : $D = \{3 : [1,1], 5 : [4], 4 : [2]\}$.

- On recommence alors le même traitement sur la liste V qui remplace L. Si comme ici la liste a une taille impaire, le dernier élément est automatiquement rajouté à la nouvelle liste des vainqueurs sans faire de match. En outre, on ajoute les nouveaux résultats aux anciens dans le dictionnaire D. On obtient donc :
  $V = [5,3,4]$ et $D = \{3 : [1,1,2], 5 : [4,3], 4 : [2]\}$.
- On poursuit de même : $V = [5,4]$ et $D = \{3 : [1,1,2], 5 : [4,3,3], 4 : [2]\}$.
- Et enfin : $V = [5]$ et $D = \{3 : [1,1,2], 5 : [4,3,3,4], 4 : [2]\}$.
  La liste V n'ayant plus qu'un élément, 5 est le maximum de la liste L. Pour trouver le second-maximum de L, il suffit de récupérer dans le dictionnaire D la liste $[4,3,3,4]$ des perdants devant 5, et d'en chercher le maximum qui est bien 4 (on pourra pour cela construire une fonction `maximum`).

Notons que si l'algorithme est lancé sur une liste L ne possédant qu'une seule valeur, celle-ci est bien sûr vainqueur du tournoi mais le dictionnaire est vide (toutes les rencontres ont donné des ex-aequo), et dans ce cas la fonction doit renvoyer la valeur **None**.

Écrivez le code de la fonction `second_maximum` utilisant l'algorithme du tournoi.

À retenir

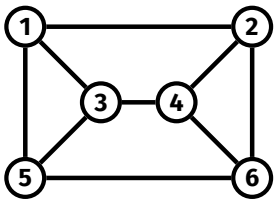
Sur la structure de dictionnaire :

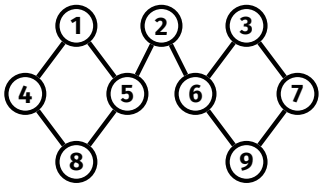
- création : `D = {clé_1 : valeur_1, clé_2 : valeur_2, ..., clé_n : valeur_n}` (où `clé_1`,...,`clé_n` sont distincts deux à deux) ;
- accès à la valeur associée une clé : `D[clé]` (permet de modifier la valeur associée à une clé existante mais aussi d'en créer une associée à une nouvelle clé) ;
- liste des clés : `D.keys()` ; liste des valeurs : `D.values()` ; liste des tuples clé, valeur : `D.items()`.

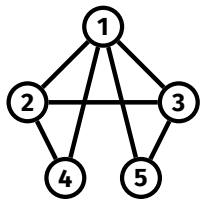
2 ASPECTS THÉORIQUES

Exercice 3 Somme des degrés dans un graphe [Sol 3]

On considère les trois graphes G_1, G_2 et G_3 ci-dessous :







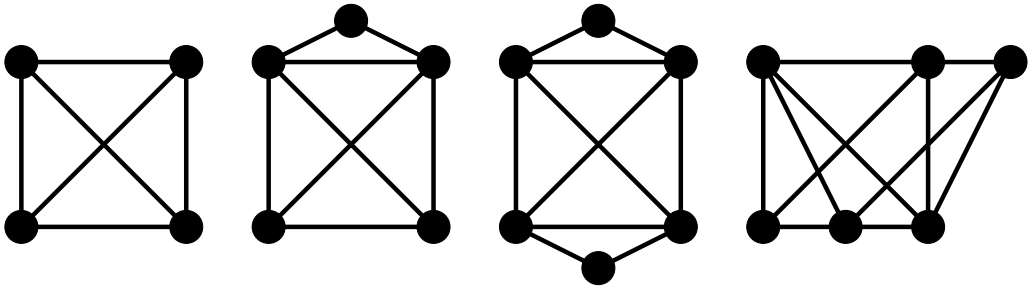
1. Pour chacun de ces trois graphes, calculer la somme des degrés de tous les sommets, ainsi que le nombre d'arêtes. Que peut-on conjecturer ?

2. Démontrer cette conjecture.

3. Que peut-on en déduire concernant la parité de la somme des degrés de tous les sommets d'un graphe ?

Exercice 4 Théorème d'Euler [Sol 4]

1. Commençons par un petit jeu ! Est-il possible de tracer les figures suivantes sans lever le crayon et sans passer deux fois sur le même trait ?



2. Dans un graphe connexe, un chemin qui parcourt toutes les arêtes une et une seule fois est appelé chemin Eulérien. Démontrer que si un graphe connexe possède un chemin eulérien, alors, tous les sommets sont de degré pair, sauf au plus 2. La réciproque est vraie mais plus difficile à démontrer. Cette équivalence est appelée le théorème d'Euler pour les graphes, et le lecteur curieux pourra en trouver une démonstration en suivant le lien : `https://fr.wikipedia.org/wiki/Graphe_eulérien`

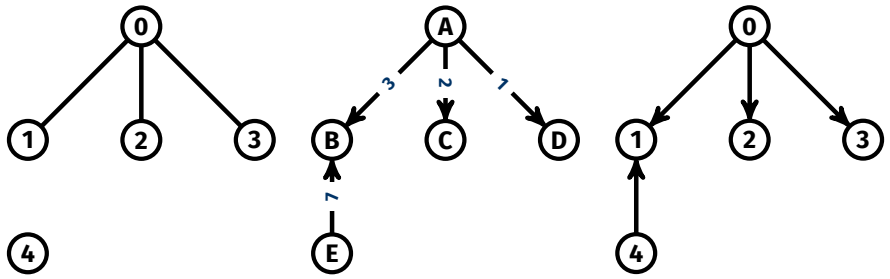
3. Réfléchir à nouveau à la question 1 en utilisant le théorème d'Euler.

3 ASPECTS INFORMATIQUES

3.1 Implémentation & Manipulations élémentaires

Exercice 5 Trois exemples [Sol 5]

1. Implémenter, directement dans votre éditeur Pyzo et sous forme de dictionnaire des voisins, les graphes ci-dessous.



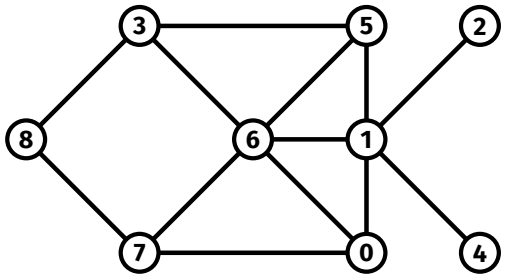
GRAPHE G_1 GRAPHE G_2 GRAPHE G_{2b}

2. À partir de plusieurs instructions à taper directement dans la console : récupérer les voisins de 2 pour `G_1`, et les voisins de `A` pour `G_2` et `G_2b` (pour le premier, on pourra utiliser une liste par compréhension).

3. Implémenter, directement dans votre éditeur Pyzo et sous forme de tableau numpy, les graphes G_1 et G_{2b}.

Exercice 6 Fonctions usuelles (implémentation par dictionnaire) [Sol 6]

1. Dans le fichier `TP_GraphesNotion.py`, vous trouverez le dictionnaire codant ce graphe :



GRAPHE G_3

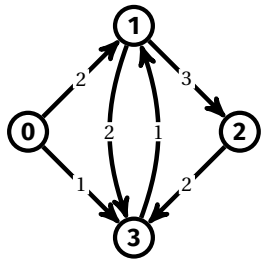
1.1) Écrire une fonction `sommets(G:dict)->list` qui prend en argument un dictionnaire codant un graphe, et renvoyant la liste des sommets.

1.2) Écrire une fonction `sans_vois(G:dict)->bool` qui renvoie la liste des sommets n'ayant aucun voisin, c'est-à-dire n'ayant aucun arc sortant partant de ce sommet.

1.3) [Non pondéré] Écrire une fonction `est_oriente(G:dict)->bool` qui renvoie `True` si le graphe est orienté, et `False` dans le cas contraire. *On supposera ici pour simplifier que le dictionnaire `G` correspond à un graphe non pondéré.*

1.4) Testez vos fonctions sur des exécutions pertinentes concernant G_1, G_2, G_3.

2. **[Pondéré]** Dans le fichier `TP_GraphesNotion.py`, vous trouverez le dictionnaire codant ce graphe :



GRAPHE G_4

Écrire une fonction `poidsMoy(G : dict)->float` renvoyant la moyenne des poids de `G`.

Exercice 7 Fonctions usuelles (implémentation par matrice) [Sol 7]

On se contente ici de quelques fonctions dans le cas non pondéré, conformément au cours (même si la notion de matrice d'adjacence peut être remplacée par celle de matrice des poids dans le cas pondéré).

1. Écrire une fonction `sommets_m(G:np.array)->int` qui prend en argument une matrice codant un graphe, et renvoyant la liste des sommets.

2. Écrire une fonction `est_oriente_m(G:np.array)->bool` qui renvoie `True` si le graphe est orienté, et `False` dans le cas contraire.

3. On souhaite dans cette question écrire deux fonctions permettant de convertir une matrice d'adjacence en dictionnaire des voisins et inversement.

3.1) Écrire une fonction `mat_to_dict(G:np.array)->dict` qui renvoie le dictionnaire des voisins associé à `G`.

3.2) Écrire une fonction `dict_to_mat(G:dict)->np.array` qui renvoie le dictionnaire des voisins associé à `G`. *On commencera par vérifier, à l'aide de commandes* **`assert`**, *que les sommets bien des entiers consécutifs numérotés à partir de zéro.*

4. Testez vos fonctions sur des exécutions pertinentes concernant G_1, G_{2b} et G_3.

Exercice 8 Retour sur le théorème d'EULER [Sol 8]

Dans l'exercice, les graphes donnés en entrée seront supposés connexes.

1. Écrire une fonction `degre(G:dict, s)->int` qui prend en argument un dictionnaire codant un graphe non orienté, un sommet `s` et renvoie le degré du sommet. *On commencera par vérifier, à l'aide de commandes* **`assert`**, *le fait que le graphe n'est pas orienté (voir un exercice précédent) et que* `s` *est bien un sommet du graphe.*

2. En déduire une fonction `existe_eulerien(G:dict)->bool` qui prend en argument un dictionnaire codant un graphe non orienté et renvoie **`True`** s'il existe un chemin EULÉRIEN.

3.2 Algorithme de coloriage

Exercice 9 Un algorithme de coloration [Sol 9]

Nous revenons ici sur l'algorithme de coloration des graphes présenté en cours, que nous allons programmer. On considère le graphe des régions de France métropolitaine continentale adjacentes déjà présenté en cours (voir la Figure 2). Dans le répertoire partagé de la classe, vous trouverez dans le fichier habituel `TP_GraphesNotions.py` contenant le graphe des régions de France présenté ci-dessus.

1. On rappelle que le principe de l'algorithme de coloriage est de parcourir les sommets dans un ordre spécifié, et pour chacun lui associer la plus petite couleur non déjà utilisée par ses voisins.

On souhaite écrire une fonction `coloriage(G, S : list) -> dict`, où `G` est un graphe (donc soit un dictionnaire soit une liste suivant l'implémentation choisie) et `S` est la liste des sommets de `G` décrits dans un certain ordre, renvoyant un dictionnaire associant à chaque sommet sa couleur choisie par l'algorithme pour cet ordre de parcours.

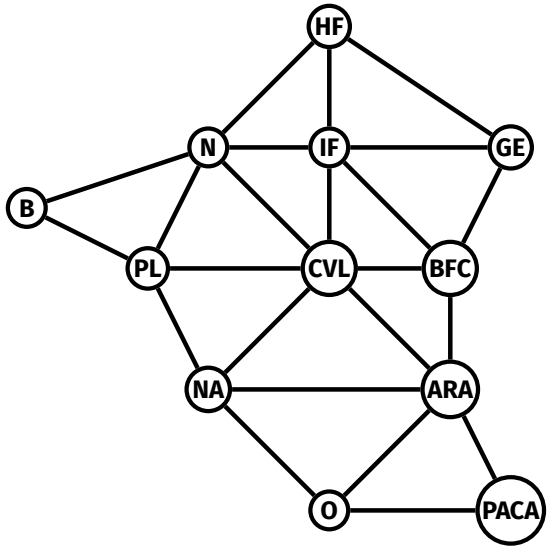


FIGURE 2 : Graphe des régions de France métropolitaine continentale adjacentes.

1.1) On commence par écrire `premierNonPresent(L : list)->int` renvoyant, pour une liste `L` formée d'entiers naturels, le plus petit entier positif non présent dans `L`. On propose le code suivant :

```
def premierNonPresent(L) :
    """
    renvoie le plus petit entier naturel non présent
    dans la liste d'entiers naturels L
    """
    i = 0
    while i in L:
        i += 1
    return i
```

Justifier que la complexité de cette fonction est en $O(n^2)$ dans le pire des cas (où n désigne la longueur de la liste `L`). On comptabilisera pour le test `i in L`, dans le pire des cas, un nombre d'opérations égal à n.

1.2) Un meilleur algorithme pour la fonction `premierNonPresent(L : list)->int` consiste à définir une liste `VU` (comme **valeurs utilisées**) initialisée d'abord avec $n+1$ fois la valeur 0 (où n désigne toujours la taille de `L`), puis parcourir une à une chaque valeur e de L, et si $e \leqslant n$, placer VU[e] à 1.

Une fois ce parcours achevé, il reste à parcourir la liste `VU` à la recherche de

la première position contenant encore 0 (on est sûr qu'il y en a au moins une puisque la liste `VU` ayant plus d'éléments que la liste `L`, toutes les positions de `VU` n'ont pas pu être mises à 1). Cette première position contenant 0 est bien la plus petite couleur non utilisée dans `L`.

Pour vous approprier le fonctionnement de cet algorithme, construisez « à la main » la liste `VU` correspondant à la liste `L = [2,0,5,2]`.

Écrivez ensuite une nouvelle version de la fonction `premierNonPresent(L : list)->int` en utilisant cet algorithme.

Justifier enfin que sa complexité est en $O(n)$ cette fois.

2. Écrire la fonction `coloriage(G, S: list)-> dict`, où `G` désigne un graphe et `S` désigne une liste contenant ses sommets dans un certain ordre, et renvoyant le dictionnaire associant à chaque sommet sa couleur. On utilisera bien sûr la fonction `premierNonPresent`.

Testez la fonction `coloriage` sur le graphe des régions, par exemple pour les sommets décrits dans l'ordre Nord-Sud, Ouest-Est, ces ordres étant stockés dans les listes `S_1` et `S_2` fournies dans le fichier externe.

SOLUTIONS DES EXERCICES

Solution 1

1.
```python
def occurrences(L:list)->dict:
    """
    Détermine, pour chaque élément de L, le nombre
    de fois où il apparait dans L
    Paramètres
    ----------
    L : list
        liste que l'on souhaite examiner
    Retour
    -------
    dict
        dictionnaire dont les clés sont les éléments de L
        et les valeurs associées sont les nombres
        d'apparitions de chacun ce des éléments
    Exemple
    -------
    >>> occurrences([5,7,3,10,3,3,7,2,3])
    {2:1, 3;4, 5:1, 7:2, 10:1}
    """
    D = {} # dictionnaire vide
    for x in L:
        if x in D:          # si x est déjà une clé de D,
            D[x] = += 1     # on augmente sa valeur
        else:
            D[x] = 1    # sinon, on crée une nouvelle clé
    return D
```

2.
```python
def PlusFrequent(L:list)->int:
    """
    Détermine l'entier qui apparaît le plus fréquemment
    dans la liste L
    """
    D = occurrences(L)
    Maxi = 0
    cle = 0
    for x in D:
        if D[x] > Maxi:
            Maxi = D[x]
            cle = x
    return cle
```

3.
```python
def CompareListes(L1:list,L2:list)->bool:
    """
    Détermine si deux listes contiennent les mêmes éléments, \
    ↪ avec pour chacun le même nombre d'occurrences.
    """
    D1 = occurrences(L1)
    D2 = occurrences(L2)
    for x in D1:
        if (x not in D2) or (D1[x] != D2[x]):
            return False
    return True
```

On aurait aussi pu envisager la solution suivante :

```python
def CompareListes2(L1:list,L2:list)->bool:
    """
    Détermine si deux listes contiennent les mêmes éléments, \
    ↪ avec pour chacun le même nombre d'occurrences.
    """
    D1 = occurrences(L1)
    D2 = occurrences(L2)
    return (D1 == D2)
```

Solution 2

```python
def second_maximum(L : list) -> int or None :
    """ renvoie le second maximum de la liste non vide
    d'entiers L s'il existe, et None sinon """
    D = {}
    while len(L) > 1 :
        V=[]
        for i in range(0,len(L)-1,2) :  # on parcourt les
            if L[i] > L[i+1] :        # indices de 2 en 2
                g, p = L[i],L[i+1]     # g est le gagnant
            else :                   # et p le perdant
                g, p = L[i+1], L[i]     # g=p si ex-eaquo
            V.append(g)
            if g != p :    # ne pas considérer les ex-eaquo
```

```
                if g not in D.keys() :# comme perdants
                    D[g] = [p]          # dans le dictionnaire
                else :
                    D[g].append(p)
        if len(L)%2 != 0 :    # si la taille est impaire
            V.append(L[len(L)-1])  # le dernier élément est \
            ↪ gagnant
        L = V  # actualisation de la liste des adversaires
    if D == {} :
        return None  # si aucun vainqueur pas de second meilleur
    else :
        return maximum(D[L[0]]) # sinon, c'est le meilleur
                                # perdant devant le vainqueur
```

Solution 3

1. Pour G_1, la somme des degrés vaut 18 et le nombre d'arêtes 9. Pour G_2, on trouve respectivement 20 et 10, et pour G_3, on trouve 14 et 7. On peut donc conjecturer que la somme des degrés est égale au double du nombre d'arêtes.

2. Chaque arête a deux extrémités. Elle est donc comptée deux fois quand on somme les degrés de tous les sommets. On peut formaliser un peu plus. Notons :
$$\delta_{i\to j} = \begin{cases} 1 & \text{si } i \text{ relié à } j, \\ 0 & \text{sinon.} \end{cases}$$
pour tout couple de sommets $(i,j) \in S$. On suppose sans restriction que $S = [\![1, n]\!]$ avec n le nombre de sommets du graphe. Alors le degré peut s'écrire :
$$d(i) = \sum_{j=1}^{n} \delta_{i\to j},$$
donc la somme des degrés est :
$$\begin{aligned} \sum_{i=1}^{n} d(i) &= \sum_{i=1}^{n}\sum_{j=1}^{n} \delta_{i\to j} = \sum_{(i,j)\in S} \delta_{i\to j} \\ &= 2 \sum_{\substack{(i,j)\in S \\ i<j}} \delta_{i\to j} \\ &= 2N \end{aligned}$$
pour des graphes non orientés
où N désigne le nombre d'arêtes (ici, dans la nouvelle somme, on ne compte qu'une seule fois chaque arête puisque $i \neq j$).

3. La somme des degrés de tous les sommets d'un graphe est un nombre pair.

Solution 4

1. Ce ne semble pas possible pour le premier mais possible pour les trois autres.

2. Empruntons le chemin eulérien, et imaginons que l'on supprime les arêtes qui ont été parcourues. A chaque passage sur un sommet (sauf au début et à la fin), on supprime l'arête qui arrive sur ce sommet et l'arête qui en part. Ainsi, sauf pour le sommet de départ ou d'arrivée, la parité du degré reste inchangée. À la fin du parcours, toutes les arêtes sont supprimées, ce qui permet de conclure sur la parité des sommets.

3. Dans le premier graphe, les quatre sommets sont de degré 3, donc, d'après le théorème d'Euler, il n'existe pas de chemin eulérien. Dans les autres graphes, les degrés des sommets sont respectivement 2,3,3,4,4 (pour le deuxième), 2,2,4,4,4,4 (pour le troisième) et 2,2,3,3,4,4 (pour le quatrième). D'après le théorème d'Euler, il existe donc un chemin eulérien.

Solution 5

```
>>> # Dictionnaires
>>> G_1 = {0 : [1, 2, 3], 1:[0], 2:[0], 3:[0], 4:[]}
>>> G_2 = {"A" : [(3, "B"), (2, "C"), (1, "D")], "B" : [], "C" : \
↪ [], "D" : [], "E" : [(7, "B")]}
>>> G_2b = {"A" : ["B", "C", "D"], "B" : [], "C" : [], "D" : [], \
↪ "E" : ["B"]}
>>> G_1[2]
[0]
>>> [X[1] for X in G_2["A"]]
['B', 'C', 'D']
>>> G_2b["A"]
['B', 'C', 'D']
>>> # Matrices
>>> import numpy as np
>>> G_1m = np.array([
...     [False, True, True, True, False], \
...     [True, False, False, False, False], \
...     [True, False, False, False, False], \
...     [True, False, False, False, False], \
...     [False, False, False, False, False]])
>>> G_2bm = np.array([
...     [False, True, True, True, False], \
...     [False, False, False, False, False], \
```

```
...     [False, False, False, False, False], \
...     [False, False, False, False, False], \
...     [False, True, False, False, False]])
```

Solution 6

1. **1.1)** Les sommets correspondent simplement aux clefs du dictionnaire.

```
def sommets(G:dict)->list:
    L_s = []
    for s in G:
        L_s.append(s)
    return L_s
```

1.2) On parcourt les sommets et on note ceux sans voisin.

```
def sans_vois(G:dict)->bool:
    L_sans_vois = []
    for s in G:
        if len(G[s]) == 0:
            L_sans_vois.append(s)
    return L_sans_vois
```

1.3) Il s'agit ici de vérifier si les arcs $i \to j$ et $j \to i$ sont présents.

```
def est_oriente(G:dict)->bool:
    for s in G:
        vois_s = G[s]
        for v in vois_s:
            # on vérifie si s est présent dans les valeurs \
            ↪ de v
            if s not in G[v]:
                return True
    return False
```

2.

```
>>> G_3 = {0 : [1, 6, 7], 1 : [0, 2, 4, 5, 6], 2 : [1], 3 : \
↪ [5, 6, 8], 4 : [1], 5 : [1, 3, 6], 6 : [0, 1, 3, 5, 7], 7 : \
↪ [0, 6, 8], 8 : [3, 7]}
>>> sommets(G_3)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> est_oriente(G_1)
False
>>> est_oriente(G_2b)
True
>>> est_oriente(G_3)
False
>>> sans_vois(G_1)
[4]
>>> sans_vois(G_3)
[]
```

3.

```
def poidsMoy(G : dict)-> list:
    S = 0
    N = 0 # le nombre total de poids présents
    for s in G:
        for (p, v) in G[s]:
            S += p
            N += 1
    return S/N
```

```
>>> poidsMoy(G_4)
1.8333333333333333
```

Solution 7

1.

```
def sommets_m(G:np.array)->int:
    N = len(G) # la taille de la matrice
    return list(range(N))
```

1.1) Il s'agit ici de vérifier si les arcs $i \to j$ et $j \to i$ sont présents.

```
def est_oriente_m(G:np.array)->bool:
    N = len(G)
    for i in range(N):
        for j in range(i):
            if G[i, j] != G[j, i]:
                return False
    return True
```

2. **2.1)**

```
def mat_to_dict(G:np.array)->dict:
    N = len(G)
    D = {i:[] for i in range(N)}
    for i in range(N):
        # recherche des voisins de i
        for j in range(N):
            if G[i, j]:
                # arc i -> j présent
                D[i].append(j)
    return D
```

2.2)
```
def dict_to_mat(G:dict)->np.array:
    N = len(G)
    for s in G:
        assert 0 <= s <= N-1
    M = np.zeros((N, N), dtype=bool) # tableau de faux
    for s in G:
        for t in G[s]:
            M[t, s] = True
    return M
```

3.
```
>>> G_3m = dict_to_mat(G_3)
>>> G_3m
array([[False,  True, False, False, False, False,  True,  \
↪ True, False],
       [ True, False,  True, False,  True,  True,  True, \
       ↪ False, False],
       [False,  True, False, False, False, False, False, \
       ↪ False, False],
       [False, False, False, False, False,  True,  True, \
       ↪ False,  True],
       [False,  True, False, False, False, False, False, \
       ↪ False, False],
       [False,  True, False,  True, False, False,  True, \
       ↪ False, False],
       [ True,  True, False,  True, False,  True, False,  \
       ↪ True, False],
       [ True, False, False, False, False, False,  True, \
       ↪ False,  True],
       [False, False, False,  True, False, False, False,  \
       ↪ True, False]])
>>> mat_to_dict(G_3m)
{0: [1, 6, 7], 1: [0, 2, 4, 5, 6], 2: [1], 3: [5, 6, 8], 4: \
↪ [1], 5: [1, 3, 6], 6: [0, 1, 3, 5, 7], 7: [0, 6, 8], 8: [3, \
↪ 7]}
>>> G_3 # on retrouve bien G_3
{0: [1, 6, 7], 1: [0, 2, 4, 5, 6], 2: [1], 3: [5, 6, 8], 4: \
↪ [1], 5: [1, 3, 6], 6: [0, 1, 3, 5, 7], 7: [0, 6, 8], 8: [3, \
↪ 7]}
>>> sommets_m(G_3m)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> est_oriente(G_1m)
False
>>> est_oriente(G_2bm)
True
>>> est_oriente(G_3m)
False
```

Solution 8

1.
```
def degre(G:dict, s)->int:
    assert est_oriente(G) == False
    assert s in G
    return len(G[s])
>>> degre(G_1, 0)
3
>>> degre(G_1, 4)
0
>>> degre(G_3, 6)
5
def existe_eulerien(G:dict)->bool:
    N = 0 # compteur des sommets de degré pair
    for s in G:
        if degre(G, s) % 2 == 0:
            N += 1
    return N <= 2
>>> existe_eulerien(G_3)
True
```

Solution 9

1. **1.1)** On compte dans le pire des cas :

- en ligne #6 : 1 opération ;
- en lignes #7 et #8, on boucle au maximum pour n itérations, avec pour chacune de ces itérations : n opérations pour le test `i in L` et 2 opérations pour `i += 1`, puis une dernière fois n opérations pour dernier test `i in L` qui fait quitter la boucle ;

donc : $C_n = 1 + n(n+2) + n = n^2 + 3n + 1 = \boxed{O(n^2)}$.

1.2) Pour `L = [2,0,5,2]` on trouve `VU = [1,0,1,0,0]`, et la première position contenant 0 est $\boxed{1}$, qui est bien la plus petite valeur non présente dans `L`.

```python
def premierNonPresent(L : list)->int :
    """
    renvoie le plus petit entier naturel non présent dans \
    ↪ la liste d'entiers naturels L
    """
    n = len(L)
    VU = [0]*(n+1)
    for e in L:
        if e <= n :
            VU[e] = 1
    i = 0
    while VU[i] == 1 :
        i += 1
    return i
```

On compte dans le pire des cas :

- en lignes #6 et #7 : $n+2$ opérations ;
- en lignes #8, #9 et #10, on boucle pour n itérations, avec pour chacune de ces itérations au pire 2 opérations ;
- en lignes #11 : 1 opérations ;
- en lignes #12 et #13, on boucle pour au pire n itérations, avec pour chacune de ces itérations 2 opérations, et 1 dernière opération pour quitter la boucle ;

donc $C_n = n+2+2n+1+2n+1 = 5n+4 = \boxed{O(n)}$.

2.

```python
def coloriage(G, S : list) -> dict :
    """
    Renvoie le dictionnaire correspondant au coloriage des \
    ↪ sommets du graphe G pour le sens de parcours précisé \
    ↪ dans S
    """
    C = {}
    # On parcourt les sommets dans l'ordre de S.
    for s in S :
        # Construction de la liste U des couleurs
        # déjà utilisées par les voisins de s.
        U = []
        for v in G[s]:  # pour chaque voisin de s
            if v in C:  # s'il a déjà une couleur
                U.append(C[v])  # on ajoute cette couleur à la \
                ↪ liste
        # Recherche de la plus petite couleur non utilisée par \
        ↪ les voisins et association de cette couleur à s dans \
        ↪ le dictionnaire
        C[s] = premierNonPresent(U)
    return C
```

```
>>> coloriage(G_reg, S_1)
{'HF': 0, 'N': 1, 'IF': 2, 'GE': 1, 'B': 0, 'PL': 2, 'CVL': 0, \
↪ 'BFC': 3, 'NA': 1, 'ARA': 2, 'O': 0, 'PACA': 1}
>>> coloriage(G_reg, S_2)
{'ARA': 0, 'BFC': 1, 'B': 0, 'CVL': 2, 'GE': 0, 'HF': 1, 'IF': \
↪ 3, 'N': 4, 'NA': 1, 'O': 2, 'PL': 3, 'PACA': 1}
```