TP (S2) 3 Notion de graphe

1 Généralités sur les graphes . . . . .
2 Retour sur la structure de dictionnaire . . . . . .
3 Implémentation des graphes . . .

Objectifs

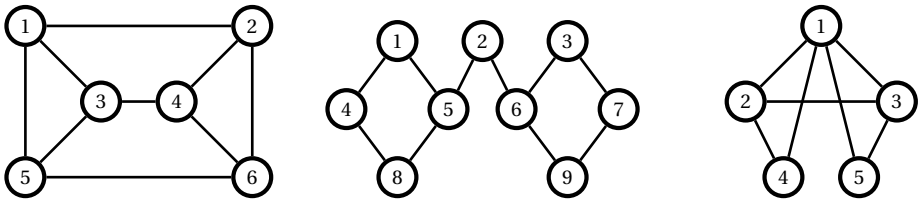
- Maîtriser le vocabulaire associé aux graphes
- Réviser la notion de dictionnaire
- Savoir utiliser les implémentations d'un graphe sous forme de liste d'adjacences et sous forme de matrice d'adjacences.

Fichier externe ?

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

1. GÉNÉRALITÉS SUR LES GRAPHES

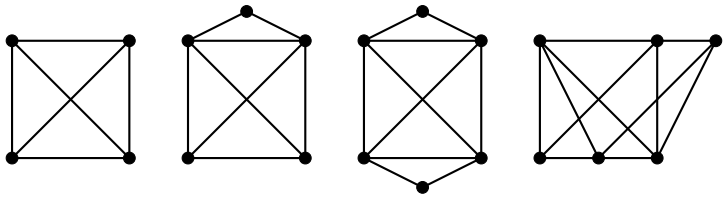
Exercice 1 **Somme des degrés dans un graphe** [Sol 1] On considère les trois graphes G_1, G_2 et G_3 ci-dessous :



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 2 **Théorème d'Euler** [Sol 2]

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.

2. RETOUR SUR LA STRUCTURE DE DICTIONNAIRE

Exercice 3 **Reprise de contact !** [Sol 3] Le but de cet exercice est de manipuler à nouveau la structure de dictionnaire que nous avions (trop ?) rapidement évoquée 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 4 Second maximum

[Sol 4] 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()`.

3. IMPLÉMENTATION DES GRAPHES

Exercice 5 **Implémentations dans le cas non pondéré** [Sol 5] Dans le répertoire partagé de la classe, vous trouverez un fichier `module_graphes_liste.py`. Sauvegardez-le dans votre répertoire de travail. Lancez l'environnement de travail `pyzo`, et créez un nouveau fichier que vous sauvegarderez lui aussi dans votre répertoire de travail, par exemple sous le nom `TP3S2.py`.

Ouvrez les deux fichiers `module_graphes_liste.py` et `TP3S2.py` dans deux fenêtres de `pyzo`.

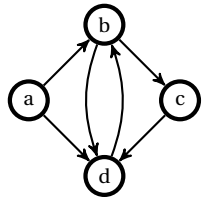
Le premier contient la définition de trois fonctions `creerGraphe`, `ajouterArc` et `afficherGraphe`, permettant de construire un graphe implémenté sous la forme d'une liste d'adjacences. Notez que ce fichier ne contient aucun corps principal, et donc son exécution ne produit aucun résultat (l'ordinateur va seulement charger dans sa mémoire la définition de ces trois fonctions).

Vous allez utiliser ces fonctions dans le fichier `TP3S2.py` pour construire et afficher un graphe. Pour cela, écrivez en première ligne du fichier `TP3S2.py` :

```
import module_graphes_liste as g
```

L'effet de cette ligne est de rendre accessible les trois fonctions du fichier `module_graphes_liste.py`, qui devront être appelées `g.creerGraphe`, `g.ajouterArc` et `g.afficherGraphe` (c'est le même principe que lorsque vous utilisez le module `numpy` par exemple). Pour que l'ordinateur sache que le fichier `module_graphes_liste.py` est à chercher dans le même répertoire où se trouve le fichier `TP3S2.py`, il est nécessaire de lancer l'exécution en choisissant « Démarrer le script » dans le menu « Exécuter » (ou le raccourci « Ctrl+Maj+E »).

1. Lisez les spécifications des fonctions `creerGraphe`, `ajouterArc` et `afficherGraphe`, ainsi que leurs codes. Utilisez ces fonctions dans le fichier `TP3S2.py` pour construire puis afficher le graphe suivant (notez que les noeuds seront ici être stockés sous forme de chaînes de caractères `'a'`, `'b'`, `'c'` et `'d'`, mais qu'avec cette implémentation il est possible de choisir des noms identifiés par des valeurs de n'importe quel type) :



2. On souhaite ajouter dans le fichier `module_graphes_liste.py` une fonction `listeVoisins(G : dict, n)-> list` renvoyant la liste des noeuds voisins du noeud `n` dans le graphe `G` (notez que la signature ne propose pas de type pour le paramètre n, puisque celui-ci peut être quelconque). Par exemple l'appel `listeVoisins(G,'a')` pour le graphe précédent renverra la liste `["b","d"]`.

Écrivez son code (dans le fichier `module_graphes_liste.py`), puis testez-le (dans le fichier `TP3S2.py`). Vous écrirez avec soin la docstring de cette fonction, et utiliserez la commande **`assert`** pour vérifier que `n` est bien un noeud du graphe `G`.

3. Toujours dans le répertoire partagé de la classe, vous trouverez un fichier `module_graphes_matrice.py`. Sauvegardez-le dans votre répertoire de travail et ouvrez-le dans `pyzo`. Il contient les définitions des trois fonctions `creerGraphe`, `ajouterArc` et `afficherGraphe`, mais utilisant cette fois une implémentation par matrice d'adjacences. En parcourant son code, vous constaterez que, pour permettre de choisir des noms de noeuds non nécessairement entiers, on définit en plus de la matrice d'adjacences un dictionnaire associant à chaque nœud un numéro entier (pris consécutifs à partir de 0) : c'est ce numéro qui repère le noeud dans la matrice. Dans cette implémentation, un graphe est alors un **tuple à deux éléments**, dont le premier élément est le dictionnaire d'association nom-numéro des noeuds, et le deuxième élément est la matrice d'adjacence.

Par exemple, le graphe de la question 1 est stocké sous la forme `G = (D,M)` où `D` est le dictionnaire d'association nom-numéro des noeuds :

$$D = \{'a' : 0, 'b' : 1, 'c' : 2, 'd' : 3\}$$

et M est la matrice d'adjacence :

M =

	0	1	2	3
0	False	True	False	True
1	False	False	True	True
2	False	False	False	True
3	False	True	False	False

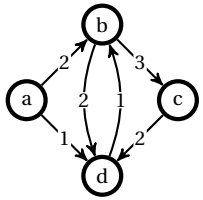
Pour utiliser ce fichier au lieu de l'autre dans le fichier `TP3S2.py`, il suffit de remplacer sa première ligne par :

```
import module_graphes_matrice as g
```

Le reste du code doit fonctionner sans modification, puisque les résultats sont indépendants de l'implémentation.

Vérifier-le en reprenant la question 1 avec `module_graphes_matrice.py` au lieu de `module_graphes_liste.py`. Puis, comme en question 2, ajouter au fichier `module_graphes_matrice.py` la fonction `listeVoisins(G : dict, n)-> list` adaptée pour une implémentation sous forme de matrice d'adjacences.

Exercice 6 **Implémentation par liste dans le cas pondéré** [Sol 6] Nous avons vu dans le cours que l'implémentation par liste d'adjacence s'étendait facilement au cas pondéré. Par exemple,




```
0 : [(1, 2), (3, 1)], 1 : [(2, 3), (3, 2))], 2 : [(3, 2)], 3 : [(1, 1)]
```

Les clefs restant identiques (au cas non pondéré), on constate que seule la fonction `ajouterArc` sera à adapter.

1. Ajouter au module `module_graphes_liste.py` une fonction `ajouterArcPond` d'arguments `G` un dictionnaire correspondant au graphe (sous forme de liste d'adjacence), `A` une liste à deux éléments qui doivent tous les deux être des noeuds du graphe, ainsi que `p` un flottant correspondant au poids de l'arrête que l'on souhaite ajouter, et qui modifie `G` en ajoutant une arête pondérée de poids `p` reliant les deux sommets de `A`.

2. Créer alors le graphe de la figure précédente à l'aide de cette nouvelle fonction.

3. Écrire une fonction `poidsMoy(G : dict)-> list` renvoyant la moyenne des poids de `G`. Par exemple l'appel `poidsMoy(G)` pour le graphe précédent renverra `1.8333333333333333`.

Exercice 7 **Algorithme de coloration** [Sol 7] Nous revenons ici sur l'algorithme de coloration des graphes présenté en cours, que nous allons programmer. Pour cela, nous utiliserons les modules `module_graphes_liste.py` et `module_graphes_matrices.py` de l'exercice précédent. En fait, un seul des deux de ces deux modules suffit : vous pouvez faire le choix d'implémenter ici votre graphe

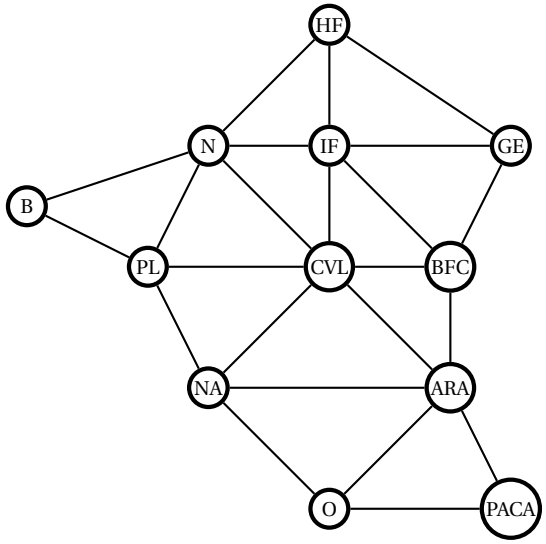


FIGURE 2. – Graphe des régions de France métropolitaine continentale adjacentes.

par une liste d'adjacence ou bien par une matrice d'adjacence, ceci ne doit pas avoir d'influence sur le programme écrit (à part la ligne `import` correspondante bien sûr).

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 un fichier `regions.py` contenant le graphe (ici le choix a été fait d'utiliser le module `module_graphes_liste.py` (dictionnaire des voisins), mais vous pourrez modifier la première ligne pour utiliser l'autre, à votre convenance). Notez bien que le graphe n'étant pas orienté, chaque arête doit être représentée par deux arcs de sens opposés. On importera le contenu de ce fichier à l'aide de la commande :

```
import module_graphes_liste as g
from regions import *
```

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.

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`, mais aussi la fonction `listeVoisins` écrite dans les modules `module_graphes_liste.py` et `module_graphes_matrices.py`.

Testez la fonction `coloriage` sur le graphe des régions, par exemple pour les sommets décrits dans l'ordre Nord-Sud, Ouest-Est.

SOLUTIONS DES EXERCICES

Solution 1

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 2

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 3

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.

```
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 :

```
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 4

```
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 5

1.

```
>>> import module_graphes_liste as g
>>> G = g.creerGraphe(['a','b','c','d'])
>>> G
{'a': [], 'b': [], 'c': [], 'd': []}
>>> g.ajouterArc(G,['a','b'])
>>> g.ajouterArc(G,['a','d'])
>>> g.ajouterArc(G,['b','c'])
>>> g.ajouterArc(G,['b','d'])
>>> g.ajouterArc(G,['c','d'])
>>> g.ajouterArc(G,['d','b'])
>>> G
{'a': ['b', 'd'], 'b': ['c', 'd'], 'c': ['d'], 'd': ['b']}
>>> g.afficherGraphe(G)
a -> b  d
b -> c  d
c -> d
d -> b
```

2.
```
def listeVoisins(G : dict,n) -> list :
    """
    Renvoie la liste des voisins du noeud n
    dans le graphe G.
    Entrée :
    --------
        G : dictionnaire d'adjacence d'un graphe.
        n : un noeud du graphe G
    Sortie :
    --------
        Liste des voisins de n dans G
    Exemple :
    ---------
    >>> G = {'a':['b', 'd'],'b':['c', 'd'],'c': ['d'],'d': \
    ↪ ['b']}
    >>> listeVoisins(G,'a')
    ['b', 'd']
    """
    # Vérification que n est un noeud de G
    assert n in G
    # Création de la liste des voisins de n
    L = G[n]
    # Retour du résultat
    return L
```

```
>>> listeVoisins(G, "b")
['c', 'd']
>>> listeVoisins(G, "d")
['b']
```

3. Commençons par redéfinir le graphe en question.

```
>>> import module_graphes_matrice as g
>>> G = g.creerGraphe(['a','b','c','d'])
>>> G
({'a': 0, 'b': 1, 'c': 2, 'd': 3}, array([[False, False, \
↪ False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False, False]]))
>>> g.ajouterArc(G,['a','b'])
>>> g.ajouterArc(G,['a','d'])
>>> g.ajouterArc(G,['b','c'])
>>> g.ajouterArc(G,['b','d'])
>>> g.ajouterArc(G,['c','d'])
>>> g.ajouterArc(G,['d','b'])
>>> G
({'a': 0, 'b': 1, 'c': 2, 'd': 3}, array([[False,  True, \
↪ False,  True],
       [False, False,  True,  True],
       [False, False, False,  True],
       [False,  True, False, False]]))
>>> g.afficherGraphe(G)
a -> b  d
b -> c  d
c -> d
d -> b
```

Passons maintenant à la fonction `listeVoisins`.

```
def listeVoisins(G : list,n) -> list :
    """
    Renvoie la liste des voisins du noeud n
    dans le graphe G.
    Entrée :
    --------
        G : graphe sous la forme d'une tuple contenant le
        dictionnaire d'association nom-numéro des noeuds
        du graphe et la matrice d'adjacence.
        n : un noeud du graphe G
    Sortie :
    --------
        Liste des voisins de n dans G
    Exemple :
    ---------
    >>> G = ({'a': 0, 'b': 1, 'c': 2, 'd': 3}, array([[False, \
    ↪ True, False,  True],[False, False,  True,  \
    ↪ True],[False, False, False,  True],[False,  True, \
    ↪ False, False]]))
    >>> listeVoisins(G,'a')
    ['b', 'd']
    """
```

```
    D, M = G
    # Vérification que n est un noeud de G
    assert n in D
    # Création de la liste des voisins de n
    L = []
    for v in D :
        if M[D[n], D[v]]:
            L.append(v)
    # Retour du résultat
    return L
```

```
>>> listeVoisins(G, "b")
['c', 'd']
>>> listeVoisins(G, "d")
['b']
```

Solution 6

1.
```
def ajouterArcPond(G : dict, A : list, p:float) -> None :
    """
    Ajoute au graphe G un arc dont les extrémités
    sont respectivement le premier et le deuxième
    élément de la liste A, et de poids p

    Entrée :
    --------
        G : dictionnaire d'adjacence d'un graphe.
        A : liste à deux éléments qui doivent tous
        les deux être des noeuds du graphe
        p : le poids de l'arête
    Sortie :
    --------
        Aucune : le graphe G passé en paramètre
        est modifié

    Exemple :
    ---------
    >>>G = creerGraphe([1,'toto'])
    {1: [],'toto' : []}
    >>>ajouterArcPond(G,[1,'toto'], 1)
    None
    >>>print(G)
    {1: [('toto', 1)],'toto' : []}
    """
    # Vérification que A est une liste à deux éléments qui \
    ↪ sont des noeuds de G
    assert type(A)==list and len(A)==2,"A n'est pas une liste \
    ↪ à deux éléments"
    assert A[0] in G and A[1] in G ,"Un des deux éléments de A \
    ↪ n'est pas un noeud du graphe G"
    # Ajout de l'arc correspondant au dictionnaire G
    G[A[0]].append((A[1], p))
```

```
>>> G = g.creerGraphe(['a','b','c','d'])
>>> G
{'a': [], 'b': [], 'c': [], 'd': []}
>>> g.ajouterArcPond(G,['a','b'],2)
>>> g.ajouterArcPond(G,['a','d'],1)
>>> g.ajouterArcPond(G,['b','c'],3)
>>> g.ajouterArcPond(G,['b','d'],2)
>>> g.ajouterArcPond(G,['c','d'],2)
>>> g.ajouterArcPond(G,['d','b'],1)
>>> G
{'a': [('b', 2), ('d', 1)], 'b': [('c', 3), ('d', 2)], 'c': \
↪ [('d', 2)], 'd': [('b', 1)]}
>>> g.afficherGraphe(G)
a -> ('b', 2)  ('d', 1)
b -> ('c', 3)  ('d', 2)
c -> ('d', 2)
d -> ('b', 1)
```

```
def poidsMoy(G : dict)-> list:
    """
    Ajoute au graphe G un arc dont les extrémités
    sont respectivement le premier et le deuxième
    élément de la liste A, et de poids p

    Entrée :
    --------
        G : dictionnaire d'adjacence d'un graphe.
```

```
    Sortie :
    --------
        Moyenne des poids

    Exemple :
    ---------
    >>>G = creerGraphe([1,'toto'])
    {1: [],'toto' : []}
    >>>poidsMoy(G)
    1.0
    """
    S = 0
    N = 0 # le nombre de poids
    for sommet in G:
        for vois in G[sommet]:
            p = vois[1]
            S += p
            N += 1
    return S/N
```

```
>>> poidsMoy(G)
1.8333333333333333
```

Solution 7 On peut vérifier le graphe crée en l'affichant.

```
>>> g.afficherGraphe(G)
HF -> N  GE  IF
N -> HF  IF  B  PL  CVL
IF -> HF  N  GE  CVL  BFC
GE -> HF  IF  BFC
B -> N  PL
PL -> N  B  NA
CVL -> N  IF  PL  BFC  NA  ARA
BFC -> IF  GE  CVL  ARA
NA -> PL  CVL  ARA  O
ARA -> CVL  BFC  NA  O  PACA
O -> NA  ARA  PACA
PACA -> O  ARA
```

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`.

```
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.

```
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.listeVoisins(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
S = \
↪ ['HF','N','IF','GE','B','PL','CVL','BFC','NA','ARA','O','PACA']
```

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