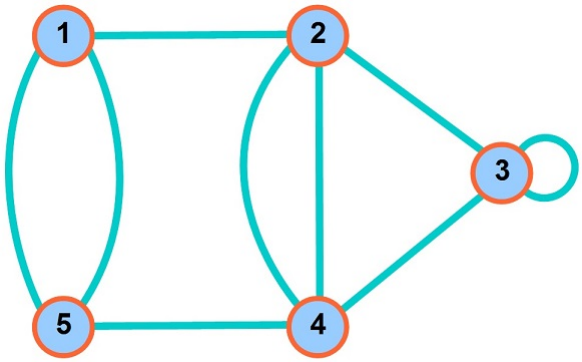
Le but de ce premier TP est de mettre en oeuvre en Python, la représentation d'un graphe par listes d'adjacences, et de commencer à utiliser cette représentation pour écrire nos premières fonctions simples de traitement des graphes.

1. Les graphes non orientés.

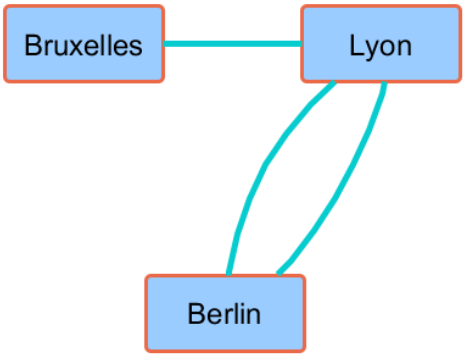
Considérons les deux multigraphes non orientés suivants et leurs représentations possibles :

Représentation sagittale

G_1



$G_{1'}$



Liste d'arêtes

[(1,2), (1,5), (1,5), (2,3), (2,4), (2,4), (3,3), (3,4), (4,5)]

[(Bruxelles,Lyon), (Lyon,Berlin), (Lyon,Berlin)]

Listes d'adjacence

sommets	successeurs
1	[2, 5, 5]
2	[1, 3, 4, 4]
3	[2, 3, 4]
4	[2, 2, 3, 5]
5	[1, 1, 4]

sommets	successeurs
Bruxelles	[Lyon]
Lyon	[Bruxelles, Berlin, Berlin]
Berlin	[Lyon, Lyon]

Matrice d'adjacence

	1	2	3	4	5
1	0	1	0	0	2
2	1	0	1	2	0
3	0	1	1	1	0
4	0	2	1	0	1
5	2	0	0	1	0

	Bruxelles	Lyon	Berlin
Bruxelles	0	1	0
Lyon	1	0	2
Berlin	0	2	0

Les différentes représentations illustrées ci-dessus sont toutes équivalentes et représentent les 2 mêmes graphes. Cependant, d'un point de vue algorithmique, toutes ne sont pas équivalentes.

La représentation sous forme de liste d'arêtes impose un parcours systématique de la liste. La représentation sous la forme d'une matrice d'adjacence est efficace mais pose un problème de correspondance entre les étiquettes des sommets et les indices en ligne et colonne de la matrice.

La représentation sous forme de listes d'adjacence se révèle être la plus pertinente. En utilisant un dictionnaire, nous bénéficions d'un accès direct à tous les successeurs d'un sommet, et aucune restriction n'est imposée sur les étiquettes des sommes (nombre non consécutifs, caractères, chaînes, etc).

En python, les 2 graphes se représenteront ainsi de la manière suivante :

```
G1 = {
    1 : [2,5,5],
    2 : [1,3,4,4],
    3 : [2,3,4],
    4 : [2,2,3,5],
    5 : [1,1,4]
}
```

```
G1b = {
    'Bruxelles' : ['Lyon'],
    'Lyon' : ['Berlin', 'Berlin'],
    'Berlin' : ['Lyon', 'Lyon']
}
```

Les différents types de parcours de cette structure sont illustrés dans l'annexe en fin de document.

Dans le reste des TP, et pour faciliter l'écriture et la manipulation des graphes, on représentera par convention les sommets par des nombres consécutifs numérotés de 1 à n.

1.1. Premières fonctions

G est un graphe ou un multigraphe non orienté, représenté par listes d'adjacences. Écrire les fonctions suivantes :

1. `nbSommets(G)` : retourne le nombre de sommets du graphe G
2. `nbArete(G)` : retourne le nombre d'arêtes du graphe G
3. `ajoutArete(G,i,j)` : ajoute l'arête (i,j) au graphe G
4. `enleveArete(G,i,j)` : enlève une arête (i,j) du graphe G (si elle existe !)
5. `deg(G,i)` : retourne le degré du sommet i
6. `degre(G)` : retourne un dictionnaire D tel que $D[i]$ est le degré du sommet i
7. `kuratowski(n)` : retourne le n-ième graphe de Kuratowski représenté par ses listes d'adjacences

N'oubliez pas de tester vos fonctions grâce à la feuille de tests unitaires fournies dans le sujet.

1.2. Conversions entre représentations

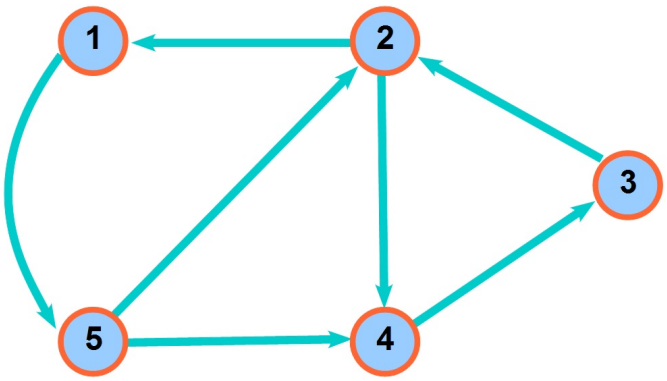
Écrire les fonctions de conversion suivantes qui permettent de passer d'une représentation à l'autre.

8. `areteToListe(n,L)` : retourne sous forme de listes d'adjacences le graphe donné par sa liste L d'arêtes (i.e. liste de tuples)
9. `listeToMatrice(G)` : retourne la matrice d'adjacences représentant G donné par listes d'adjacences. **Attention aux décalages nécessaires !** Dans la matrice, le sommet 1 se trouve à l'indice 0, et ainsi de suite.
10. `nonOriente(M)` : vérifie qu'une matrice d'adjacences est bien symétrique (et peut donc représenter un graphe non orienté)
11. `matToListe(M)` : retourne sous forme de listes d'adjacences un graphe donné par sa matrice d'adjacences

Tester vos fonctions.

2. Graphes orientés

Considérons le graphe G_2 suivant :



Le graphe G_2 est représenté par
- sa liste d'arêtes : `L = [(1,5), (2,1), (2,4), (3,2), (4,3), (5,2), (5,4)]`
- ses listes d'adjacences : `G = {1 :[5], 2 :[1,4], 3 :[2], 4 :[3], 5 :[2,4]}`
- et sa matrice d'adjacences :

```
[   [0,0,0,0,1],
    [1,0,0,1,0],
    [0,1,0,0,0],
    [0,0,1,0,0],
    [0,1,0,1,0] ]
```

2.1. Premières fonctions

En vous inspirant du paragraphe précédent, écrire les fonctions suivantes (G désigne cette fois un graphe simple orienté) :

12. `nbSommets(G)` : retourne le nombre de sommets du graphe G
13. `nbArcs(G)` : retourne le nombre d'arcs du graphe
14. `ajoutArc(G,i,j)` : ajoute l'arc (i,j) au graphe G
15. `enleveArc(G,i,j)` : enlève l'arc (i,j) du graphe G (s'il existe !)

16. `degS(G,i)` : retourne le degré sortant du sommet i
17. `degreS(G)` : retourne un dictionnaire D tel que $D[i]$ soit le degré sortant du sommet i
18. `degE(G,i)` : retourne le degré entrant du sommet i
19. `degreE(G)` : retourne un dictionnaire D tel que $D[i]$ soit le degré entrant du sommet i

Tester vos fonctions.

2.2. Conversions entre représentations

G désigne toujours un graphe simple orienté. En vous inspirant du paragraphe §1.2, écrivez les fonctions suivantes :

20. `listeToMatrice(G)` : retourne la matrice d'adjacences représentant G donné par listes d'adjacences
21. `arcsToListe(n,L)` : retourne sous forme de listes d'adjacences le graphe donné par sa liste L d'arcs
22. `matToListe(M)` : retourne sous forme de listes d'adjacences le graphe donné par sa matrice d'adjacence M

Annexe : parcourir un dictionnaire

Soit le dictionnaire `dico` suivant :

```
dico = {
  'a' : ['b','c'],
  'c' : [],
  'd' : ['a','c'],
  'b' : ['a','b','c']
}
```

Lorsqu'on souhaite parcourir les clés du dictionnaire :

```
1 for k in dico :
2 ␣␣print(k, dico[k])
```

```
a
c
d
b
```

Lorsqu'on souhaite parcourir les couples clé-valeur :

```
1 for k, v in dico.items() :
2 ␣␣print(k, '->', v)
```

```
a -> [b,c]
c -> []
d -> [a,c]
b -> [a,b,c]
```

Dans le cas où on a besoin d'énumérer les couples :

```
1 for i, (k, v) in enumerate(dico.items()) :
2 ␣␣print(i, '->', (k,v))
```

```
0 -> (a, [b,c])
1 -> (c, [])
2 -> (d, [a,c])
3 -> (b, [a,b,c])
```

(à savoir : les couples sont ordonnés dans l'ordre de leur insertion depuis python 3)

Dans le cas où on souhaite parcourir les clés triées :

```
1 for k in sorted(dico) :
2 ␣␣print(k)
```

```
a
b
c
d
```