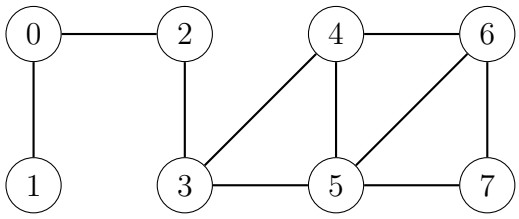
Série 11 - Corrigé

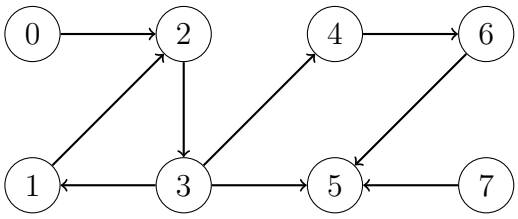
Sauf si spécifié autrement, tous les algorithmes demandés sont à écrire sous forme d'une fonction Python.

Exercice 1

(a) Pour chacun des graphes ci-dessous, donnez sa représentation par matrice d'adjacence et par listes d'adjacence.



G_1



G_2

(b) Le graphe non dirigé G_3 est représenté par la matrice d'adjacence

0	1	0	0	1
1	0	1	1	1
0	1	0	1	0
0	1	1	0	1
1	1	0	1	0

Dessinez G_3 et donnez sa représentation par listes d'adjacence.

(c) Le dictionnaire suivant représente le graphe dirigé G_4 par listes d'adjacence.
`D = {0:[1, 3], 1:[4], 2:[4,5], 3:[1], 4:[3], 5:[]}`.
Dessinez G_4 et donnez sa représentation par matrice d'adjacence.

Solution.

(a) La matrice d'adjacence de G_1 :

0	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
0	0	1	0	1	1	0	0
0	0	0	1	0	1	1	0
0	0	0	1	1	0	1	1
0	0	0	0	1	1	0	1
0	0	0	0	0	1	1	0

Un dictionnaire de listes d'adjacence de G_1 :
`G_1 ={0:[1,2], 1:[0], 2:[0,3], 3:[2,4,5], 4:[3,5,6], 5:[3,4,6,7], 6:[4,5,7], 7:[5,6]}`.
Cette représentation n'est pas unique car l'ordre des voisins d'un sommet donné peut être quelconque. Par exemple, dans cet exemple on a défini `G_1[3] = [2, 4, 5]`, mais on aurait aussi bien pu définir `G_1[3] = [4, 2, 5]` ou bien `G_1[3] = [5, 2, 4]` ou n'importe quelle autre permutation des trois voisins du sommet 3.

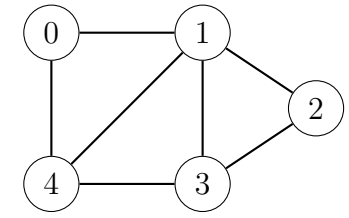
La matrice d'adjacence de G_2 :

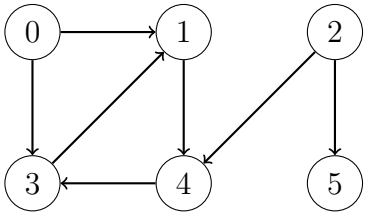
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	1	0	0	1	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0

Un dictionnaire de listes d'adjacence de G_2 :
`G_2 ={0:[2], 1:[2], 2:[3], 3:[1,4,5], 4:[6], 5:[], 6:[5], 7:[5]}`.

(b) Le graphe G_3 :
Un dictionnaire de listes d'adjacence de G_3 :
`G_3 ={0:[1,4], 1:[0,2,3,4], 2:[1,3], 3:[1,2,4], 4:[0,1,3]}`.

(c) Le graphe G_4 :
La matrice d'adjacence de G_4 :





0	1	0	1	0	0
0	0	0	0	1	0
0	0	0	0	1	1
0	1	0	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

Exercice 2

(a) Le Jupyter notebook `BFS.ipynb` contient l'algorithme de parcours en largeur vu en cours, avec une instruction `print` qui affiche, en plus du sommet parcouru, la liste des sommets stockés à chaque étape dans `a_parcourir`. Il contient également les représentations par listes d'adjacence de deux graphes G et H que vous pouvez utiliser pour mieux comprendre le fonctionnement de l'agorithme `BFS`.
Exécutez `BFS(G,s)` et `BFS(H,s)` avec divers choix du sommet de départ `s`. Changez l'ordre où les sommets voisins sont stockés dans les listes d'adjacence données et observez s'il y a une différence dans l'ordre des sommets affichés.

(b) Soient G_1 et G_2 les graphes donnés à l'exercice 1a, et soient `G_1` et `G_2` les représentations par listes d'adjacence que vous avez données de ces graphes.
Faites à la main un parcours en largeur (BFS) du graphe G_1 à partir du sommet 2, c'est-à-dire écrivez les sommets dans l'ordre où vous les visitez.
Vérifiez votre réponse en exécutant `BFS(G_1,2)` dans le Jupyter notebook.

(c) Même question pour un parcours en largeur de G_2 à partir de 0.

Solution.

(b) On représente G_1 par le dictionnaire de listes d'adjacence `G_1 ={0:[1,2], 1:[0], 2:[0,3], 3:[2,4,5], 4:[3,5,6], 5:[3,4,6,7], 6:[4,5,7], 7:[5,6]}`. Un parcours en largeur à partir du sommet 2 de G_1 ainsi représenté visite les sommets dans l'ordre suivant :

$$2 - 0 - 3 - 1 - 4 - 5 - 6 - 7$$

Ce qu'on peut vérifier avec un appel à `BFS(G_1,2)` dans le Jupyter notebook.

(c) On représente G_2 par le dictionnaire de listes d'adjacence `G_2 ={0:[2], 1:[2], 2:[3], 3:[1,4,5], 4:[6], 5:[], 6:[5], 7:[5]}`.

Un parcours de G_2 en largeur à partir de 0 visite les sommets dans l'ordre suivant :

$$0 - 2 - 3 - 1 - 4 - 5 - 6$$

Ce qu'on peut vérifier avec un appel à `BFS(G_2,0)` dans le Jupyter notebook.

Exercice 3

(a) Soit $G = (V, E)$ un graphe **non dirigé** à n sommets et m arêtes.
Pour $v \in V = \{0, 1, \ldots, n-1\}$, on définit le **degré** de v comme étant le nombre d'arêtes incidentes à v, c'est-à-dire le nombre de voisins de v.

Par exemple, pour le graphe G_1 donné à l'exercice 1a, la liste suivante contient les degrés des sommets (`L[u]` contient le degré du sommet `u`) :

`L = [2, 1, 2, 3, 3, 4, 3, 2].`

(i) On suppose que G est représenté par une matrice d'adjacence. Donnez un algorithme qui prend en entrée la matrice d'adjacence de G et retourne une liste `L` de taille n, telle que `L[v]` contient le degré du sommet v. Testez votre algorithme avec le graphe G_1 donné à l'exercice 1a.

Quel est le temps de parcours de votre algorithme en fonction de n et m ?

(ii) On suppose maintenant que G est représenté par des listes d'adjacence. Donnez un algorithme qui prend en entrée un dictionnaire de listes d'adjacence de G et retourne la liste `L` décrite au point (i). Testez votre algorithme avec le graphe G_1 donné à l'exercice 1a.

Quel est le temps de parcours de votre algorithme en fonction de n et m ?

(b) On considère maintenant un graphe **dirigé** $G = (V, E)$ à n sommets et m arêtes.

Pour $v \in V = \{0, 1, \ldots, n-1\}$, on définit

— le **degré sortant** de v comme le nombre d'arêtes sortant de v, c'est-à-dire le nombre d'arêtes (v, u) pour u un quelconque autre sommet de G ;

— le **degré entrant** de v comme le nombre d'arêtes entrant dans v, c'est-à-dire le nombre d'arêtes (u, v) pour u un quelconque autre sommet de G.

Par exemple, pour le graphe G_2 donné à l'exercice 1a, la liste suivante contient les degrés sortants de tous les sommets :

```
L_out = [1, 1, 1, 3, 1, 0, 1, 1]
```

Et la liste suivante contient les degrés entrants de tous les sommets :

```
L_in = [0, 1, 2, 1, 1, 3, 1, 0].
```

(i) On suppose que G est représenté par une matrice d'adjacence. Donnez un algorithme qui prend en entrée la matrice d'adjacence de G et retourne deux listes `L_out` et `L_in` de taille n, telles que `L_out[v]` contient le degré sortant et `L_in[v]` le degré entrant du sommet v. Testez votre algorithme avec le graphe G_2 donné à l'exercice 1a.

Quel est l'ordre du temps de parcours de votre algorithme en fonction de n et m ?

(ii) On suppose maintenant que G est représenté par des listes d'adjacence. Donnez un algorithme qui prend en entrée un dictionnaire de listes d'adjacence de G et retourne les deux listes décrites au point (i). Testez votre algorithme avec le graphe G_2 donné à l'exercice 1a.

Votre algorithme doit avoir temps de parcours $\Theta(n + m)$.

Remarque 1 : Les listes `L_out` et `L_in` sont calculées indépendamment l'une de l'autre.

Remarque 2 : A part pour le calcul de `L_in` au point (b-ii), un algorithme naïf fera l'affaire pour toutes les listes demandées.

Solution.

(a) (i) L'algorithme ci-dessous calcule, à partir de la matrice d'adjacence de G, le degré de chaque sommet : pour calculer le degré du sommet u, l'algorithme parcourt la ligne u de la matrice d'adjacence et compte le nombre de 1 dans cette ligne (en calculant la somme de tous les éléments de la ligne).

On a inclus dans le code l'appel à `degre_matrice` avec la matrice d'adjacence de G_1.

Les deux boucle imbriquées donnent un temps de parcours de $\Theta(n^2)$.

```
def degre_matrice(G):
    '''
    Entree: G matrice d'adjacence d'un graphe
    Sortie: Liste L des degres des sommets du graphe
    '''
    n = len(G)
    L = []

    for u in range(n):
        deg = 0
        for v in range(n):
            deg += G[u][v]
        L.append(deg)
    return L


G = [[0,1,1,0,0,0,0,0],[1,0,0,0,0,0,0,0],[1,0,0,1,0,0,0,0],
    [0,0,1,0,1,1,0,0],[0,0,0,1,0,1,1,0],[0,0,0,1,1,0,1,1],
     [0,0,0,0,1,1,0,1],[0,0,0,0,0,1,1,0]]
print(degre_matrice(G))
```

(ii) L'algorithme ci-dessous parcourt les éléments du dictionnaire de listes d'adjacence, càd les sommets du graphe : pour chaque sommet, le degré est égal à la longueur de la liste d'adjacence correspondante.

On a inclus dans le code l'appel à `degre_liste` avec le dictionnaire de listes d'adjacence de G_1.

Cet algorithme a temps de parcours $\Theta(n)$ puisque l'appel à la

méthode `len` se fait en temps constant, et puisqu'on parcourt un dictionnaire de taille n en temps linéaire en n (l'accès à chaque élément du dictionnaire se faisant en temps constant).

```
def degre_liste(G):
    '''
    Entree: G dict de listes d'adjacence d'un graphe
    Sortie: Liste L des degres des sommets du graphe
    '''
    L = []
    for u in G:
        L.append(len(G[u]))
    return L


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

(b) (i) L'algorithme ci-dessous calcule le degré sortant pour chaque sommet u (c'est le nombre de 1 dans la ligne u de la matrice d'adjacence) et le degré entrant pour chaque sommet u (c'est le nombre de 1 dans la colonne u de la matrice d'adjacence).

```python
def degre_in_out_matrice(G):
    '''
    Entree: G matrice d'adjacence d'un graphe dirige
    Sortie: Listes Lin, Lout des degres entrant/sortant
    des sommets du graphe
    '''
    n = len(G)
    Lin = []
    Lout = []

    for u in range(n):
        deg = 0
        for v in range(n):
            deg += G[u][v]
        Lout.append(deg)

    for u in range(n):
        deg = 0
        for v in range(n):
            deg += G[v][u]
        Lin.append(deg)

    return Lin, Lout


G = [[0,0,1,0,0,0,0,0],[0,0,1,0,0,0,0,0],[0,0,0,1,0,0,0,0],
    [0,1,0,0,1,1,0,0],[0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0],
     [0,0,0,0,0,1,0,0],[0,0,0,0,0,1,0,0]]
print(degre_in_out_matrice(G))
```

Les deux boucles `for` imbriquées donnent un temps de parcours de $\Theta(n^2)$ pour le calcul de chacune des deux listes, et on a donc un temps de parcours total de $\Theta(n^2)$.
On aurait pu écrire le code de manière plus concise comme ci-dessous (avec toujours un temps de parcours de $\Theta(n^2)$) :

```python
def degre_in_out_matrice(G):
    '''
    Entree: G matrice d'adjacence d'un graphe dirige
    Sortie: Listes Lin, Lout des degres entrant/sortant
    des sommets du graphe
    '''
    n = len(G)
    Lin = []
    Lout = []

    for u in range(n):
        deg_in = 0
        deg_out = 0
        for v in range(n):
            deg_out += G[u][v]
            deg_in += G[v][u]
        Lin.append(deg_in)
        Lout.append(deg_out)

    return Lin, Lout


G = [[0,0,1,0,0,0,0,0],[0,0,1,0,0,0,0,0],[0,0,0,1,0,0,0,0],
    [0,1,0,0,1,1,0,0],[0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0],
     [0,0,0,0,0,1,0,0],[0,0,0,0,0,1,0,0]]
print(degre_in_out_matrice(G))
```

(ii) L'algorithme ci-dessous calcule le degré sortant pour chaque u de manière similaire au calcul du degré de chaque sommet pour un graphe non dirigé.

Le degré entrant d'un sommet u est le nombre de fois où u apparaît dans les listes d'adjacences du dictionnaire. Pour calculer le degré entrant de n'importe quel sommet, il suffit donc de parcourir toutes les listes d'adjacence et de compter le nombre de fois que u apparaît. Le parcours de toutes les listes d'adjacence prend un temps $\Theta(n+m)$: il faut parcourir tous les n éléments du dictionnaire, et il faut parcourir toutes les listes, qui ont une taille combinée de m éléments. Le calcul du degré entrant d'un seul sommet prend donc temps $\Theta(n+m)$; s'il fallait répéter ce calcul n fois, on aurait un temps de parcours total de $\Theta(n^2+mn)$.

L'astuce est de calculer tous les degrés entrants avec un seul parcours : on maintient la liste `Lin`, et on parcourt chaque élément de chaque liste d'adjacence, en augmentant à chaque fois l'index

correspondant de `Lin`.

```
def degre_in_out_liste(G):
    '''
    Entree: G dict de listes d'adjacence d'un graphe dirige
    Sortie: Listes Lin, Lout des degres entrant/sortant
    des sommets du graphe
    '''
    Lout = []
    for u in G:
        Lout.append(len(G[u]))

    n = len(G)
    Lin = [0 for u in range(n)]
    for u in G:
        for v in G[u]:
            Lin[v] += 1

    return Lin, Lout


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

Exercice 4

(a) Soit $G = (V, E)$ un graphe dirigé à n sommets et m arêtes. Prouver que

$$0 \leq m \leq n(n-1).$$

(b) Soit $G = (V, E)$ un graphe non dirigé à n sommets et m arêtes. Prouver que

$$0 \leq m \leq \frac{n(n-1)}{2}.$$

Solution.

(a) Pour chaque paire **ordonnée** de sommets $u, v \in V$, l'arête correspondante (u, v) peut appartenir à E ou pas. Or le nombre de paires ordonnées parmi n éléments est $n(n-1)$: pour former une paire ordonnée, on a n choix pour le premier élément et $n-1$ pour le second. m peut donc prendre n'importe quelle valeur entre 0 et $n(n-1)$. Par exemple, un graphe dirigé à deux sommets peut avoir 0, 1 ou 2 arêtes, et un graphe dirigé à 3 sommets peut avoir entre 0 et 6 arêtes.

(b) Pour chaque paire **non ordonnée** de sommets $u, v \in V$, l'arête correspondante $(u, v) = (v, u)$ peut appartenir à E ou pas. Or le nombre de paires non ordonnées parmi n éléments est égal au nombre de paires ordonnées divisées par 2 : (u, v) et (v, u) correspondent à la même arête. m peut donc prendre n'importe quelle valeur entre 0 et $n(n-1)/2$. Par exemple, un graphe non dirigé à deux sommets peut avoir zéro ou une arête, et un graphe non dirigé à 3 sommets peut avoir entre 0 et 3 arêtes.

Exercice 5 (révision)

Pour chacune des boucles ci-dessous, soit $F(n)$ le nombre de fois que l'instruction `print` est exécutée.

(a) Pour $n = 5$, que vaut $F(n)$ pour chacune des boucles ? Vérifiez votre réponse en exécutant la boucle avec $n = 5$.

(b) Donner, pour chacune des boucles, l'ordre de croissance de $F(n)$ en notation $\Theta(\cdot)$.

```python
#boucle 1
for i in range(n):
    for j in range(n):
        print(i, j)

#boucle 2
for i in range(n):
    for j in range(i+1, n):
        print(i, j)

#boucle 3
for i in range(n):
    for j in range(i, i+2):
        print(i, j)

#boucle 4
for i in range(n):
    for j in range(i+1, n):
        for k in range(j+1, n):
            print(i, j, k)

#boucle 5
for i in range(n):
    for j in range(i+1, n):
        for k in range(j, j+1):
            print(i, j, k)
```

Solution.

(a) — La boucle 1 exécute l'instruction `print` $5 \times 5 = 25$ fois.
- La boucle 2 exécute l'instruction `print` $4 + 3 + 2 + 1 = 10$ fois.
- Dans la boucle 3, la boucle intérieure itère 2 fois quelle que soit la valeur de i. La boucle 3 exécute donc l'instruction `print` $5 \times 2 = 10$ fois.
- Dans la boucle 4, pour `i=0`, les deux boucles intérieures sont équivalentes à

```
for j in range(1, n):
    for k in range(j+1, n):
        print(0, j, k)
```

  elles exécutent l'instruction `print` $3 + 2 + 1 = 6$ fois. De même pour `i=1`, les deux boucles intérieures exécutent l'instruction `print` $2 + 1 = 3$ fois ; pour `i=2`, les deux boucles intérieures exécutent l'instruction `print` une seule fois (pour `j = 3` et `k = 4`) ; et pour `i=3` et `i=4` les boucle intérieures n'itèrent pas du tout. On a donc un total de $6 + 3 + 1 = 10$ exécutions du `print`.
- Dans la boucle 5, la boucle qui itère sur `k` itère une seule fois quelle que soit la valeur de `j`. On a donc un total de 10 exécutions comme pour la boucle 2.

(b) — Les boucles 1 et 2 ont temps de parcours $\Theta(n^2)$.
- Dans la boucle 3, la boucle intérieure itère deux fois quelle que soit la valeur de `i`, elle s'exécute donc en temps constant. La boucle 3 a donc temps de parcours $\Theta(n)$.
- La boucle 4 a temps de parcours $\Theta(n^3)$
- Dans la boucle 5, la boucle qui itère sur `k` itère une seule fois quelle que soit la valeur de `j`, elle s'exécute donc en temps constant. La boucle 5 a donc temps de parcours $\Theta(n^2)$.