On considère dans ce TP des graphes à n sommets (numérotés de 0 à $n-1$). On peut modéliser un tel graphe par sa liste d'adjacence `graphe` de la façon suivante : pour tout sommet $k \in [\![0, n-1]\!]$, `L[k]` est la liste des sommets voisins du sommet k. *Par exemple, la liste d'adjacence du graphe de la figure 1 (en bas de page) est :*

`[[3,5], [3,4,5], [3], [0,1,2,5], [1,5,6], [0,1,3,4], [4], [8], [7]]`.

On dispose dans le fichier `TP_Graphes.py`, d'une fonction `generateur_graphe` permettant de générer des graphes non orientés. En préambule de ce fichier, on définit alors plusieurs graphes : `graphe1`, `graphe2`, `graphe3` et `graphe4`.

1 Mise en jambe

Question 1. Vérifier que le sommet 1 est le seul voisin du sommet 0 et que le sommet 4 n'a pas de voisin dans `graphe1` et vérifier que le sommet 1 a 7 voisins dans `graphe2`.

Question 2. Écrire une fonction calculant le nombre d'arêtes d'un graphe représenté par sa liste d'adjacence. *Vérifier que* `graphe1`, `graphe2`, `graphe3` *et* `graphe4` *ont respectivement 3, 261, 607 et 28 arêtes.*

2 Parcours en largeur

On rappelle le principe du ***parcours en largeur*** [1] *d'un graphe. On part d'un sommet fixé et on visite chaque sommet accessible dans un ordre précis : lorsqu'on a visité un sommet, on parcourt d'abord tous ses sommets voisins avant de parcourir les voisins de ces voisins. On visite alors tous les sommets accessibles distants d'une arrête, puis de deux arrêtes, puis de trois, etc.*

Pour implémenter le parcours en largeur, on peut utiliser une structure de **file**, modélisée par une liste dans laquelle le premier élément inséré est le premier à en partir[2].

On décrit ci-dessous l'algorithme de parcours en largeur :

- on initialise la liste des sommets visités à la liste vide et une file de sommets à traiter avec le sommet de départ ;
- tant que la file est non vide, on retire (défile) le sommet en tête de file ; si ce n'est pas déjà un sommet visité, on l'ajoute à la liste des sommets visités et on ajoute (enfile) ses voisins **non encore visités** à la file des sommets à traiter.

On illustre ci-dessous les étapes de cet algorithme sur le graphe ci-dessous, en partant du sommet 2.

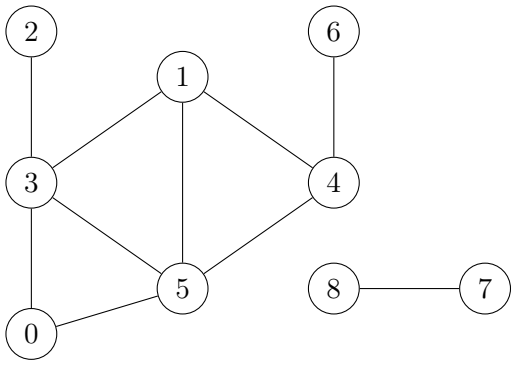

Figure 1: Exemple de graphe non orienté

Sommets visités	Sommets à traiter
`[]`	`[2]`
`[2]`	`[3]`
`[2,3]`	`[1,5,0]`
`[2,3,1]`	`[5,0,4,5]`
`[2,3,1,5]`	`[0,4,5,0,4]`
`[2,3,1,5,0]`	`[4,5,0,4]`
`[2,3,1,5,0,4]`	`[6]`
`[2,3,1,5,0,4,6]`	`[]`

La liste des sommets ordonnés dans le parcours en largeur de ce graphe à partir du sommet 2 est donc ici `[2,3,1,5,0,4,6]`. On remarquera que cette liste n'est pas unique (elle dépend de l'ordre dans lequel on enfile les voisins non visités dans la liste) : la liste `[2,3,0,1,5,4,6]` est aussi valide.

[1] On parle de *breadth first search*, ou *bfs*, en anglais.
[2] On pourra retenir l'acronyme *FIFO : "First In First Out"*.

Question 3. Écrire une fonction en Python qui prend en argument un graphe (représenté par sa liste d'adjacence) et un sommet et qui réalise un parcours en largeur à partir de ce sommet. On renverra la liste des sommets dans l'ordre induit par le parcours en largeur.

Testez votre code en réalisant un parcours en largeur sur `graphe4` *à partir du sommet 11. Réponses possibles :* `[11, 37, 49, 13, 22, 42, 29, 17, 48]` *ou* `[11, 49, 37, 13, 22, 42, 29, 48, 17]`*).*

3 Parcours en profondeur

Le principe du **parcours en profondeur** [3] d'un graphe ressemble beaucoup à celui du parcours en largeur. On part d'un sommet fixé et on visite chaque sommet accessible dans un ordre précis : lorsqu'on a visité un sommet x, on visite un sommet voisin y de x puis les voisins de y avant les autres voisins de x. Le parcours en profondeur explore donc chaque chemin partant du sommet fixé avant de revenir en arrière[4].

Pour implémenter le parcours en longueur on peut utiliser une structure de **pile**, modélisée par une liste. Une pile est une structure de données linéaire définie par le principe suivant : le dernier élément inséré d'une pile est le premier à en partir[5].

On décrit ci-dessous l'algorithme de parcours en largeur :

- on initialise la liste des sommets visités à la liste vide et une pile de sommets à traiter avec le sommet de départ ;
- tant que la pile est non vide, on retire (dépile) le sommet en tête de file ; on l'ajoute à la liste des sommets visités et on ajoute (empile) ses voisins non encore visités à la pile des sommets à traiter.

On illustre ci-dessous les étapes de l'algorithme de parcours en profondeur sur le graphe ci-dessous, en partant du sommet 9.

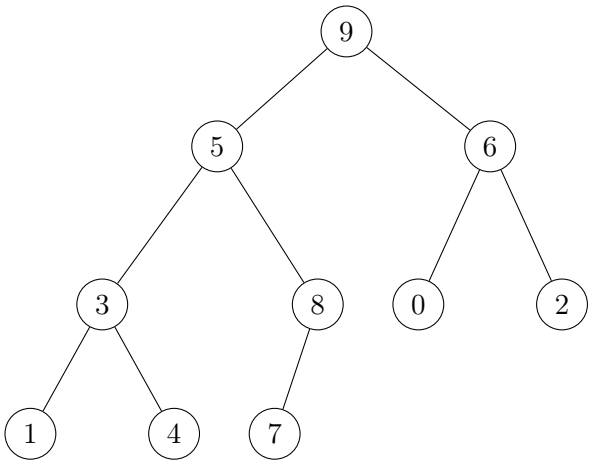

Figure 2: Exemple de graphe non orienté

Sommets visités	Sommets à traiter
`[]`	`[9]`
`[9]`	`[5,6]`
`[9,6]`	`[5,0,2]`
`[9,6,2]`	`[5,0]`
`[9,6,2,0]`	`[5]`
`[9,6,2,0,5]`	`[3,8]`
`[9,6,2,0,5,8]`	`[3,7]`
`[9,6,2,0,5,8,7]`	`[3]`
`[9,6,2,0,5,8,7,3]`	`[1,4]`
`[9,6,2,0,5,8,7,3,4]`	`[1]`
`[9,6,2,0,5,8,7,3,4,1]`	`[]`

Question 4. Écrire une fonction en Python qui prend en argument un graphe (représenté par sa liste d'adjacence) et un sommet et qui réalise un parcours en profondeur à partir de ce sommet. On renverra la liste des sommets dans l'ordre induit par le parcours en profondeur. **On s'inspirera fortement du code réalisant un parcours en largeur**.

Testez votre code en parcourant `graphe3` *à partir du sommet 60.*
Réponse : `[60, 106, 313, 674, 450, 366]` *ou* `[60, 106, 313, 366, 450, 674]`*.*

[3] On parle de *depth first search*, ou *dfs* en anglais.
[4] Le parcours en profondeur porte parfois le nom de *retour sur trace*, ou *backtracking* en anglais
[5] On pourra retenir l'acronym *LIFO : "Last In First Out"*.

4 Composantes connexes d'un graphe

On dit que deux sommets d'un graphe non orienté sont dans la même **composante connexe** s'il existe un chemin qui les relie. Une composante connexe d'un graphe non orienté est donc l'ensemble des sommets du graphe "reliés" à un même sommet.
Par exemple, le graphe de la figure 1 possède deux composantes connexes ($\{0,1,2,3,4,5,6\}$ et $\{7,8\}$), tandis que celui de la figure 2 n'en possède qu'une.

Question 5. En utilisant le parcours en largeur ou en profondeur, écrire une fonction en Python calculant le nombre de composantes connexes d'un graphe non orienté (représenté par sa liste d'adjacence).
Tester votre code en vérifiant que `graphe1`, `graphe2`, `graphe3` *et* `graphe4` *possèdent respectivement 7, 1, 394 et 22 composantes connexes.*

5 Chasse au trésor

On ne considère désormais que des **graphes connexes**, i.e. n'ayant qu'une seule composante connexe. On dispose dans le fichier pour cela de cinq graphes connexes (définis en préambule) : `graphe_fig1graphe_tresor1`, `graphe_tresor2`, `graphe_tresor3` et `graphe_tresor4`.
On définit la **longueur** d'un chemin entre deux arêtes comme le nombre d'arêtes le composant. La **distance** entre deux sommets d'un graphe connexe non orienté est alors définie comme le minimum des longueurs des chemins reliant ces deux sommets.
On peut adapter l'algorithme de parcours en largeur afin de calculer la distance d'un sommet `x` fixé à tous les autres sommets du graphe. Au début du parcours, on définit un tableau `distance` de telle sorte que `distance[i]` soit la distance de `x` à `i`. Tous les sommets sont initialement à une distance infinie (`numpy.inf`) du sommet x, sauf x qui est à distance 0 de lui-même. Dès qu'un sommet `y` est visité (défilé de la file des sommets à traiter), on peut mettre-à-jour la distance de ses voisins au sommet `x` : le chemin de `x` à un voisin de `y` nécessite une arête de plus que celui de `x` à `y`.

Question 6. Écrire une fonction `distance_a_x` prenant en argument un graphe connexe non orienté et un sommet `x` et renvoyant le tableau des distances de `x` à tous les sommets du graphe.
Tester votre code sur le graphe de la figure 1, en partant du sommet 2. Réponse : `[2,2,0,1,3,2,4,inf,inf]`.

On suppose qu'on dépose une pièce d'or sur certains sommets d'un graphe connexe non orienté, et qu'on dispose d'un tableau `pieces` de booléens tel que `pieces[i]` indique si une pièce d'or a été déposée sur le sommet `i`.

Question 7. Écrire une fonction `piece_plus_proche` prenant en argument un graphe connexe non orienté, un sommet fixé `x` et le tableau `pieces` et qui renvoie le sommet contenant la pièce d'or la plus proche de `x`, ainsi que la distance à ce sommet (sous la forme d'un tuple).
Testez votre code sur `graphe_tresor1` *en partant du sommet 0 dans le cas où les seules pièces sont situées sur les sommets 98 et 99. Réponse : la pièce la plus proche est sur le sommet 98, à une distance de 80 arêtes.*

On dépose désormais k pièces d'or sur les k derniers sommets numérotés d'un graphe connexe non orienté. Un voyageur de commerce se déplace sur le graphe et décide de récupérer toutes les pièces d'or en minimisant la distance totale parcourue. Pour cela, il choisit une stratégie gloutonne : à chaque instant, il se dirige vers la pièce la plus proche. Dans le cas où plusieurs pièces seraient équidistantes de sa position, il décide de choisir celle située sur le sommet de plus petit indice.

Question 8. Écrire une fonction `nb_deplacements` qui prend en argument un graphe connexe non orienté (représenté par sa liste d'adjacence) et un entier `k`, et qui renvoie la distance totale parcourue par le voyageur de commerce en appliquant la stratégie gloutonne. *On supposera que le voyageur se trouve initialement en 0.*
Testez votre code sur `graphe_tresor2` *avec* $k = 10$, `graphe_tresor3` *avec* $k = 15$ *et* `graphe_tresor4` *avec* $k = 20$. *Réponse : on trouve respectivement 27, 35 et 26 déplacements.*

Question 9. Donner un exemple de graphe simple pour lequel la stratégie gloutonne n'est pas optimale.