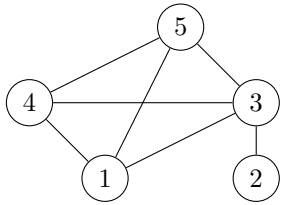
I - Généralités, représentation des graphes

Exercice 1: (Solution)

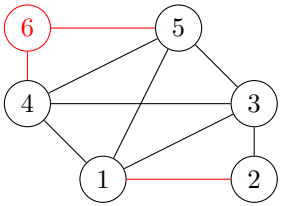
On considère le graphe suivant.



1. Compléter les instructions suivantes :

```
D={}
D[1]=[3,4,5] # les voisins de 1 sont 3,4,5
D[2]=[3] # 2 a un seul voisin : 3
D[3]=
D[4]=
D[5]=
```

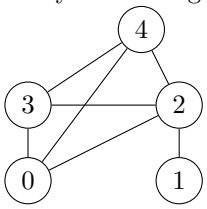
2. Modifier le dictionnaire D précédent pour qu'il représente maintenant le graphe suivant :



Les différences sont en rouge.

Exercice 2: (Solution)

Écrire une fonction `degre(G,s)` renvoyant le degré du sommet s du graphe non orienté G donné sous forme du dictionnaire des listes d'adjacence.
Essayez avec le graphe suivant



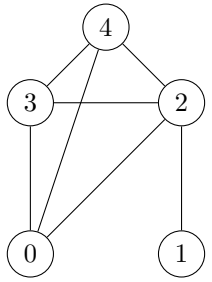
Le graphe sera implémenté avec Python à l'aide d'un dictionnaire

Exercice 3: (Solution)

Soit $G = (S, A)$ un graphe non orienté avec $S = \{0, \dots, n-1\}$.

1. Écrire une fonction `matrice(D)` d'argument le `dictionnaire` des listes d'adjacence des sommets de `G` et renvoyant la `matrice d'adjacence` de G.
2. Écrire une fonction `dico(M)` d'argument la `matrice d'adjacence` du graphe G et renvoyant le `dictionnaire` des listes d'adjacence de G.

Tester avec le graphe suivant :



Exercice 4: (Solution)

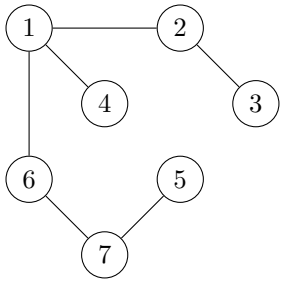
1. Écrire une fonction `comptage(L)` d'argument une liste d'entiers et renvoyant le dictionnaire des occurrences (= nombre d'apparitions) de chaque élément de `L`.
   Par exemple `comptage([4,1,5,1,1,5])` renvoie le dictionnaire :
   `{1 : 3, 4 : 1, 5 : 2}`.
2. Utiliser la fonction comptage pour créer une fonction `TriComptage(L)` qui trie dans l'ordre croissant une liste d'entiers. Par exemple, `TriComptage([4,1,5,1,1,5])` renvoie la liste `[1,1,1,4,5,5]`.

II - Parcours en largeur et en profondeur. Applications

II - A Parcours en largeur

Exercice 5: (Solution)

On considère le graphe suivant :

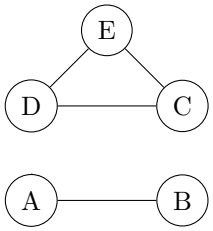


1. Le sommet de départ est 1. Appliquer, sur votre cahier, l'algorithme de parcours en largeur sur ce graphe en décrivant à chaque étape :
   — l'état de la file `L`
   — la valeur de la variable `u` défilée
   — La coloration (blanc, gris et noir) des sommets non visités, en cours de visite et déjà visités.
   — l'état du dictionnaire des prédécesseurs
2. Implémenter le parcours en **largeur** à l'aide de Python (pseudo-code dans le cours). La fonction `BFS(G,dep)` aura pour arguments un graphe `G` (implémenter par un dictionnaire) et un sommet de départ `dep`.
   La fonction doit renvoyer le dictionnaire des prédécesseurs de chaque sommet lors du parcours du graphe à partir du sommet `dep`.
3. Tester avec `BFS(G,1)` où G est le graphe précédent.

Exercice 6: (Solution)

On rappelle que l'algorithme de parcours en largeur `BFS(G,dep)` permet de construire le dictionnaire des prédécesseurs de chaque sommet partant d'un sommet initial.
On rappelle également qu'un graphe non orienté est connexe si pour tout couple `(s1,s2)` de sommets de `G`, il existe un chemin reliant `s1` à `s2`.

1. Quelle est la longueur du dictionnaire des prédécesseurs renvoyé par la fonction `BFS(G,dep)` lorsque le graphe est connexe ?
2. Écrire une fonction `connexe(G)` d'argument un graphe `G` et renvoyant un booléen indiquant si le graphe `G` est connexe.
3. On rappelle que les composantes connexes d'un graphe non connexe sont les sous parties connexes de ce graphe. Par exemple :



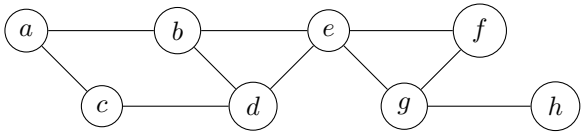
Ce graphe est non connexe.
Ses composantes connexes sont :

$$\{A, B\} \quad \{C, D, E\}.$$

Écrire une fonction `composantes(G)` renvoyant les composantes connexes d'un graphe non orienté `G`.

Exercice 7: (Solution)

On rappelle qu'un cycle d'un graphe est un chemin reliant un sommet à lui-même sans emprunter plusieurs fois une même arrête.
Dans cet exercice, on propose d'écrire une fonction de **détection** de cycle dans un graphe utilisant le principe du parcours en largeur.



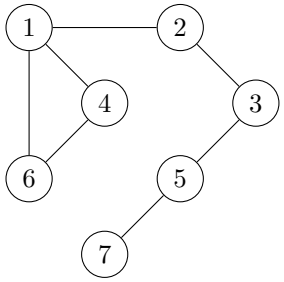
Partons du sommet `h` (sans prédécesseur) : il possède un unique voisin `g`. Le prédécesseur de `g` est le sommet `h`.
Le sommet `g` possède trois voisins `e,f,h`. Les sommets `e,f` ont le même prédécesseur `g`.
Le sommet `e` possède quatre voisins dont le sommet `f`. Ce sommet a déjà été découvert et son prédécesseur est `g`$\neq$`e`.
On a détecté un cycle !
Autrement dit, on détecte un cycle lorsqu'on retombe sur un sommet déjà découvert mais via un sommet prédécesseur différent.
Attention, dans cet exercice on ne propose pas de construire un tel cycle mais simplement de le détecter.

1. En modifiant l'algorithme de parcours en largeur, écrire une fonction `BFS_cycle(G,dep)` qui renvoie un booléen indiquant si un graphe connexe `G` contient un cycle en partant d'un sommet `dep`.
2. Si le graphe n'est pas connexe, il se peut qu'un graphe contienne un cycle qui n'appartient pas à la composante connexe du sommet `dep`.
   Écrire une fonction `cycle(G)` d'argument un graphe non orienté et non nécessairement connexe et indiquant si `G` contient un cycle.

II - B Parcours en profondeur

Exercice 8: (Solution)

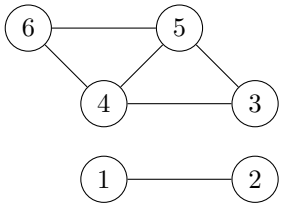
On considère le graphe suivant :



1. Le sommet de départ est 1 et on parcourt le graphe dans l'ordre croissant. Appliquer, sur votre cahier, l'algorithme de parcours en **profondeur** sur ce graphe en décrivant à chaque étape :
   — l'état de la pile `L`
   — la valeur de la variable `u` et de ses voisins `R`.
   — La coloration (blanc, gris et noir) des sommets non visités, en cours de visite (découverts) et déjà visités.
   — l'état du dictionnaire des prédécesseurs.
2. Implémenter le parcours en profondeur à l'aide de Python (pseudo-code dans le cours) et tester sur ce graphe. La fonction `DFS(G,dep)` aura pour arguments un graphe `G` (implémenter par un dictionnaire) et un sommet de départ `dep`.
   La fonction doit renvoyer le dictionnaire des prédécesseurs de chaque sommet lors du parcours du graphe à partir du sommet `dep`.
3. Tester avec `DFS(G,1)` où G est le graphe précédent.

Exercice 9: (Solution)

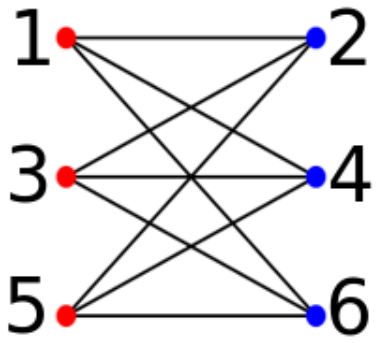
1. A l'aide de la fonction `DFS(G,dep)`, coder une fonction `chemin(G,s1,s2)` d'arguments un graphe `G` et deux sommets `s1,s2` et indiquant s'il existe un chemin reliant `s1` à `s2`.
2. On considère le graphe G suivant :



Tester la fonction codée à la question 1. avec `chemin(G,6,3)` et `chemin(G,1,6)`

Exercice 10: (Solution)

Un graphe non orienté est dit biparti si on peut partager son ensemble de sommets en deux parties U et V tels qu'il n'y ait aucune arête entre éléments de U et aucune arête entre éléments de V. Autrement dit, chaque arrête relie des sommets issus des deux sous-ensembles distincts U et V.



1. Copier-Coller l'algorithme de parcours en profondeur et le modifier de la manière suivante :
   — On part d'un sommet quelconque que l'on colore en Rouge.
   — Le premier voisin découvert de ce sommet est coloré en Bleu.
   — On poursuit le parcours du graphe en colorant chaque sommet de la couleur opposée à celle de son prédécesseur.
   Il sera nécessaire d'introduire un nouveau dictionnaire que l'on nommera `couleurs` (on conservera le dictionnaire `visites` contenant les couleurs Blanc, Gris, Noir).
2. En utilisant les modifications précédentes, écrire une fonction `biparti(G)` d'argument un graphe connexe et renvoyant un booléen indiquant si le graphe `G` est biparti. Tester avec le graphe ci-dessus.

Exercice 11: (Solution)

Le principe du parcours en profondeur est particulièrement adapté à une implémentation récursive. En effet à chaque sommet découvert, la parcours à partir du sommet successeur suit exactement le même principe que le parcours à partir du sommet prédécesseur. Il suffit de maintenir à jour le dictionnaire des prédécesseurs.
Écrire une fonction `DFS_rec(G,dep)` parcourant en profondeur le graphe `G` de manière récusrive.
On pourra implémenter comme suit :

- Le dictionnaire des prédécesseurs est initialisé à `P={dep:None}`.
- On écrit une sous-fonction `rec(G,courant)` qui parcourt les voisins `v` du sommet `courant`. Celle-ci traite les sommets `v` qui ne sont pas encore des clés du dictionnaire `P`. On pose alors `P[v]=courant` et produit l'appel récursif `rec(G,v)`.

Exercice 12: (Solution)

On rappelle qu'un cycle d'un graphe est un chemin reliant un sommet à lui-même sans emprunter plusieurs fois une même arrête.
En modifiant l'algorithme de parcours en **profondeur**, écrire une fonction `DFS_cycle(G,dep)` qui détecte la présence d'un cycle **ET** renvoie un cycle sous forme d'une liste de sommets composant ce cycle.