Les parties sont indépendantes, sauf pour les définitions.

I Définitions

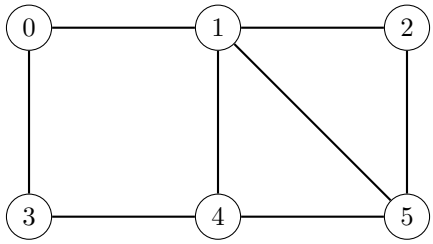
Soit $G = (V, E)$ un graphe non-orienté, où V est un ensemble de sommets et E un ensemble d'arêtes. Soit $k \in \mathbb{N}^*$. Un ***k*-coloriage** de G est une fonction $c : V \longmapsto \{0, ..., k-1\}$ telle que :

$$\{u, v\} \in E \implies c(u) \neq c(v)$$

Dit autrement, un k-coloriage donne une couleur (qu'on suppose être un entier entre 0 et $k-1$, pour simplifier) à chaque sommet, tel que deux sommets adjacents soient de couleurs différentes.

Suivant les questions, on utilisera soit une matrice d'adjacence, soit une liste d'adjacence. Attention à ne pas confondre.

Soit G_1 le graphe suivant :



1. Écrire une ou plusieurs instruction(s) Python pour définir G_1 par **liste d'adjacence**.
2. Donner une 3-coloration pour G_1. On pourra recopier G_1 en mettant, à côté de chaque sommet, une couleur (c'est-à-dire 0, 1 ou 2).
3. Justifier que G_1 ne possède pas de 2-coloration.
4. Si n est le nombre de sommets d'un graphe G, montrer que G possède une n-coloration.

Dans toute la suite, on représente une k-coloration par une liste `C` telle que `C[i]` est la couleur (entre 0 et $k-1$) du sommet `i`.

5. Écrire une fonction `valid(G, C)` déterminant si la coloration `C` est valide sur le graphe représenté par la **liste d'adjacence** `G`.
   Par exemple, si `G1` est la liste d'adjacence de G_1, `valid(G1, [0, 0, 1, 2, 3, 4])` doit renvoyer `False` (car les sommets 0 et 1 sont adjacents et sont tous les deux coloriés avec la couleur 0) mais `valid(G1, [3, 0, 1, 0, 3, 4])` doit renvoyer `True` (toutes les arêtes ont bien des extremités de couleurs différentes).

II Degré

1. Écrire une fonction `deg(G, v)` renvoyant le degré d'un sommet dans le graphe `G` représenté par **liste d'adjacence**.
2. Écrire une fonction `deg_max(G)` calculant le degré maximum d'un sommet dans le graphe `G` représenté par **liste d'adjacence**. On appelle $\Delta(G)$ ce nombre.

Il existe un algorithme simple donnant une $(\Delta(G)+1)$-coloration pour un graphe G : considérer chaque sommet v un par un (de 0 à $n-1$, où n est le nombre de sommets) et lui donner la plus petite couleur n'apparaissant pas parmi les voisins de v.

3. Écrire une fonction `delta_color(G)` renvoyant une $(\Delta(G)+1)$-coloration de `G` représenté par **matrice d'adjacence**. Le résultat sera donc une liste `C` telle que `C[v]` est la couleur donnée à `v`.

III Clique

Une **clique** d'un graphe G est un sous-graphe complet, c'est-à-dire un ensemble de sommets contenant toutes les arêtes possibles entre deux sommets. La taille d'une clique est son nombre de sommets.
Par exemple, l'ensemble de sommets $\{1, 2, 5\}$ forme une clique de taille 3 de G_1, puisque ces 3 sommets sont tous reliés par des arêtes.

1. Soit $k \in \mathbb{N}^*$. Montrer que s'il existe une clique de taille k dans G, alors G n'est pas $(k-1)$-coloriable.

2. Écrire une fonction `is_clique(G, V)` déterminant si la liste des sommets `V` forme une clique dans la **matrice d'adjacence** `G`, c'est-à-dire si tous les sommets de `V` sont reliés 2 à 2.
Par exemple, si `G1` est la matrice d'adjacence de G_1, `is_clique(G1, [1, 2, 5])` doit renvoyer `True` mais `is_clique(G1, [1, 2, 3])` doit renvoyer `False`.

IV 2-coloration par parcours en profondeur

On veut écrire un algorithme pour obtenir une 2-coloration d'un graphe connexe G. Pour cela, on exécute un parcours en profondeur depuis un sommet v quelconque (par exemple le sommet 0) de G, que l'on colorie avec la couleur 0, puis on colorie les voisins de v avec la couleur 1, puis les voisins des voisins avec la couleur 0...
On pourra utiliser le fait que si $c \in \{0, 1\}$ est une couleur, alors $1 - c$ est l'autre couleur ($1 - c = 1$ si $c = 0$ et $1 - c = 0$ si $c = 1$).
Si, à un moment de l'algorithme, on doit colorier un sommet avec une couleur alors qu'il a déjà été colorié d'une couleur différente, G n'est pas 2-coloriable.
On stockera le coloriage dans une liste `C` (comme pour la partie I) qui sera aussi utilisée pour savoir si un sommet a déjà été visité.

1. Compléter le code suivant pour renvoyer un 2-coloriage dans le graphe G représenté par liste d'adjacence. Si G n'a pas de 2-coloriage, on renverra `False`.

```
def 2_color(G):
    # définir une liste C donnant un coloriage pour chaque sommet, avec initialement que des -1
    def aux(v, c): # parcours en profondeur sur v, en lui donnant la couleur c
        # Si v a déjà la couleur 1 - c, renvoyer False
        # Si v a déjà la couleur c, renvoyer True
        # Mettre la couleur c dans C[v]
        # Appeler récursivement aux(w, 1 - c) pour chaque w voisin de v.
        # Si un de ces appels renvoie False, renvoyer False aussi. Sinon, renvoyer True.
    # Appeler aux(0, 0). Si cela renvoie False, renvoyer False. Sinon, renvoyer C
```

2. Si le graphe n'est pas connexe, il faut appliquer l'algorithme précédent sur chaque composante connexe. Modifier la fonction précédente pour le faire.

V Comptage du nombre de couleurs

Étant donnée une liste d'entiers (des couleurs), non forcément consécutifs, on veut savoir quel est le nombre d'entiers différents (le nombre de couleurs). Par exemple, le nombre de valeurs différentes de `[1, 4, 0, 4, 1]` est 3 (il y a 3 entiers différents : $0, 1, 4$). Pour cela, on étudie trois méthodes différentes (et indépendantes).

1. Écrire une fonction `ncolor1(C)` renvoyant le nombre d'entiers différents dans une liste `C`, en utilisant 2 boucles `for`. On pourra traduire le pseudo-code suivant en Python :

```
def ncolor1(C):
    L = [] # L va contenir les différentes valeurs de C
    # Pour tout élément c de C
        # Si c n'appartient pas à L
            # Alors ajouter c à L
    # Renvoyer la taille de L
```

2. Quelle est la complexité de `ncolor1(C)`, en fonction de la taille n de `C` ?

3. Si `L` est une liste, `L.sort()` permet de trier les éléments de `L` (par ordre croissant). `L.sort()` modifie `L` (mais ne renvoie pas de valeur). On admet que `L.sort()` est en complexité $O(n \log(n))$, où n est le nombre d'éléments de `L`.
Écrire une fonction `ncolor2(C)` renvoyant le nombre d'entiers différents dans une liste `C`, en triant `C`. Cette fonction doit être en complexité $O(n \log(n))$ où n est la taille de `C`, et on demande de justifier cette complexité.

4. Une 3ème méthode consiste à utiliser une liste `B` de booléens de taille p, où p est le maximum de `C`, telle que `B[i]` vaut `True` si et seulement si `i` est dans `C`.
Écrire une fonction `ncolor3(C)` renvoyant le nombre d'entiers différents dans une liste `C`, en créant et utilisant une telle liste `B`. `ncolor3(C)` doit être en complexité $O(n + p)$ et on justifiera cette complexité.