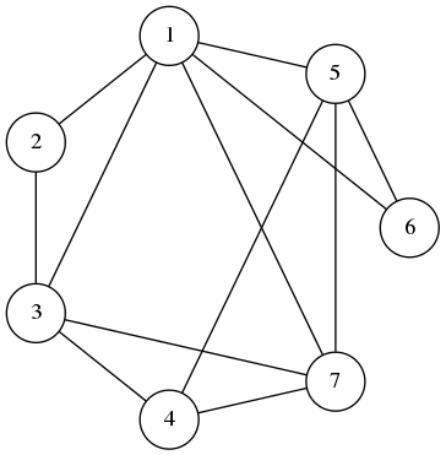
I Coloration d'un graphe par la méthode de Welsh et Powell

La **coloration** d'un graphe consiste à colorier les sommets d'un graphe de sorte que deux sommets joints par une arête n'aient jamais la même couleur ; le but étant évidemment d'utiliser au total un minimum de couleurs. On a démontré qu'un graphe planaire (que l'on peut représenter sans que les arêtes ne se croisent) peut se colorer en utilisant au plus 4 couleurs.
La méthode décrite ici permet de déterminer un moyen de colorer un graphe avec assez peu de couleurs (mais ce n'est pas la coloration optimale). Pour cela, on introduit le **degré** d'un sommet du graphe : c'est le nombre d'arêtes qui partent de ce sommet. On procède alors de la façon suivante :

- on classe les sommets dans l'ordre décroissant de leur degré (en cas de degrés égaux, on les classe par ordre croissant de leur numéro).
- en parcourant cette liste dans l'ordre, on attribue une nouvelle couleur au premier sommet non encore coloré ainsi qu'à tous les autres sommets non encore colorés et qui ne sont pas voisins d'un sommet de cette couleur.
- on recommence jusqu'à ce que tous les sommets soient colorés.

On suppose le graphe implémenté par son dictionnaire d'adjacence : les clés sont des entiers (les numéros des sommets) et les valeurs associées sont les listes des sommets voisins (accessibles directement avec une seule arrête).

1. Appliquer cet algorithme pour colorer le graphe suivant



2. Écrire une fonction `degre(s:int,G:dict)->int` qui prend en argument le numéro d'un sommet `s`, le dictionnaire d'adjacence `G` du graphe, et qui renvoie son degré.

3. On suppose disposer d'une liste `Couleurs` suffisamment longue et qui contient les couleurs utilisables (par exemple, `Couleurs=['bleu','rouge','vert','jaune',...]`), ainsi que d'une fonction `trier(G:dict)->list` qui renvoie la liste des sommets du graphe triée dans l'ordre des degrés décroissants.

Écrire une fonction `colorer(G:dict,L:list)->dict` qui prend en argument le dictionnaire d'adjacence du graphe et la liste des couleurs et qui renvoie un dictionnaire dont les clés sont les sommets et les valeurs associées la couleur du sommet.

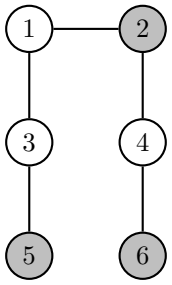
On pourra utiliser la méthode `remove(elt)` qui permet de supprimer `elt` d'une liste : si `L` est la liste `[1,2,3]` alors `L.remove(2)` modifie `L` en `[1,3]`.

II Chasse au trésor sur un graphe

On suppose disposer d'un graphe non orienté connexe (toutes les arêtes peuvent être parcourues dans les deux sens et pour tout couple de sommets, il existe au moins un chemin permettant de les joindre) représenté par son dictionnaire d'adjacence. On suppose également avoir placé des pièces sur certains sommets du graphe : un dictionnaire `pieces` possède comme clés les sommets du graphe et la valeur associé est `True` si des pièces sont placées sur le sommet, `False` sinon.
On souhaite trouver un chemin permettant de récupérer toutes les pièces du graphe, en essayant de minimiser autant que possible le chemin parcouru. Pour cela on adopte une stratégie gloutonne qui consiste à aller chercher en premier les pièces placées sur le sommet le plus proche de notre position actuelle.

1. Appliquer la stratégie décrite sur le graphe suivant, en partant du sommet 1 ; obtient-on un chemin de longueur minimale ?

Les pièces sont placées sur les sommets grisés.



2. On commence par écrire une fonction `suivant(s:int,G:dict,pieces:dict)->list` qui prend en argument le numéro d'un sommet `s`, le dictionnaire `G` du graphe et le dictionnaire `pieces` et qui renvoie la liste des sommets par lesquels il faut passer pour récupérer la pièces suivante.

a) Pourquoi le parcours en largeur est-il plus adapté que le parcours en profondeur pour cette démarche gloutonne ?

b) Écrire la fonction `suivant(s:int,G:dict,T:dict)->int`.

3. En déduire une fonction `recolte(s:int,G:dict,pieces:dict)->list` qui renvoie le trajet (représenté par une liste de sommets) à effectuer pour ramasser toutes les pièces ; on pourra commencer par déterminer le nombre de sommets sur lesquels sont posées des pièces.