INTRODUCTION

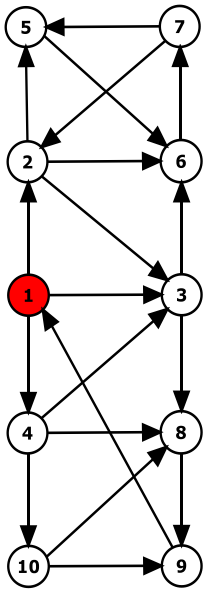
Les quelques notions de vocabulaire abordées lors du niveau 1 nous ont permis de pressentir que certains problèmes de graphes pouvaient se révéler rapidement complexes : ajouter des arcs à un graphe pour le rendre transitif, déterminer les composantes fortement connexes d'un graphe, ordonner les sommets dans une relation d'ordre…
Existe-t-il des méthodes pour résoudre ces problèmes quelle que soit la taille du graphe ?
Des algorithmes donc ?

Avant de répondre à quelques-unes de ces questions, nous allons nous intéresser aux stratégies de parcours de graphes, celles-ci seront utiles pour bien comprendre le fonctionnement de certains algorithmes, mais elles sont aussi fondamentales pour **s'adapter et résoudre de nouveaux problèmes**.

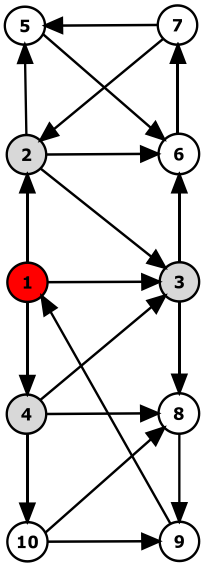
Il faut bien comprendre que l'ordinateur ne « visualise » pas le graphe comme nous le faisons, il ne « voit » pas les chemins, au mieux, à partir d'un sommet, il « voit » **ses successeurs** ou **ses prédécesseurs**.

Parcourir un graphe en partant d'un de ses sommets, consiste à visiter de façon itérative tous les sommets du graphes accessibles* à partir de ce sommet.
Accessibles : il existe un chemin à partir du sommet de départ.

PAR EXEMPLE : ON SOUHAITE PARCOURIR LE GRAPHE G CI-DESSOUS A PARTIR DU SOMMET 1.



Etape 1 : visite des successeurs du sommet 1
Le sommet 1 a trois successeurs : 2,3 et 4
Les sommets visités à la fin de cette étape sont **1,2,3,4**



Etape 2 : visite des successeurs du sommet suivant
Quel sommet prendre ?
Il y a en général deux stratégies :

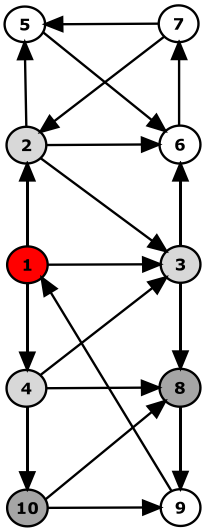
- **On reste sur le dernier sommet visité** et on regarde ses successeurs : on choisit ainsi un **parcours en profondeur**.
  Dans l'exemple, on visite les **successeurs de 4**, dernier sommet visité.
- **On repart sur le premier sommet** rencontré qui n'a pas encore été traité*: on choisit un **parcours en largeur**.
  Dans l'exemple, on visite les **successeurs de 2**, premier sommet rencontré dont on n'a pas encore regardé les successeurs.

traiter un sommet signifie, ici**, « visiter » ses successeurs.*

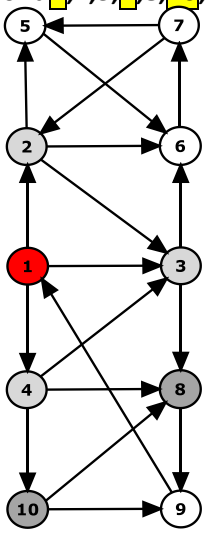
PARCOURS EN PROFONDEUR DU GRAPHE G

« ON TRAITE EN PRIORITE LES SOMMETS LES PLUS RECEMMENT VISITES »

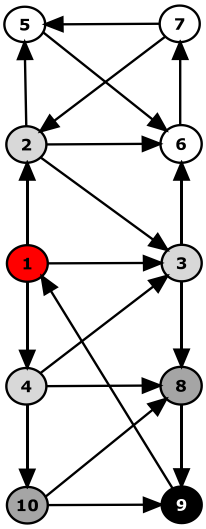
Etape 2 : successeurs de 4, le dernier sommet visité
Le sommet 4 a deux successeurs : 8 et 10
Les sommets visités à la fin de cette étape sont **1,2,3,4,8,10**
Les sommets encadrés sont ceux qui ont déjà été traités



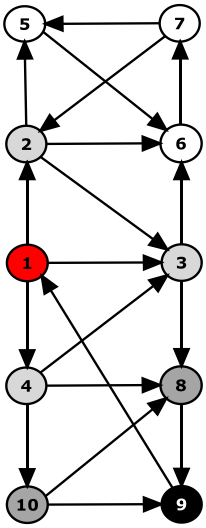
Etape 3 : successeurs de 10, le dernier sommet visité
Le sommet 10 a deux successeurs mais un seul qui n'a pas encore été vu : 9
Les sommets visités à la fin de cette étape sont **1,2,3,4,8,10,9**



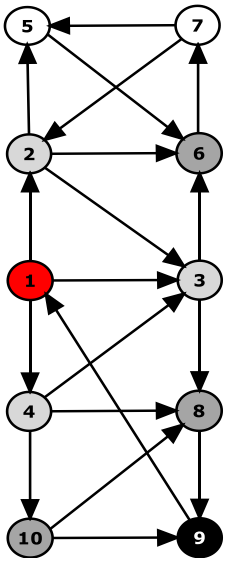
Etape 4 : successeurs de 9, le dernier sommet visité
Le sommet 9 n'a pas de successeur qui n'a pas déjà été visité. On ne s'intéresse donc pas au successeur 1.
Les sommets visités à la fin de cette étape sont **1,2,3,4,8,10,9**



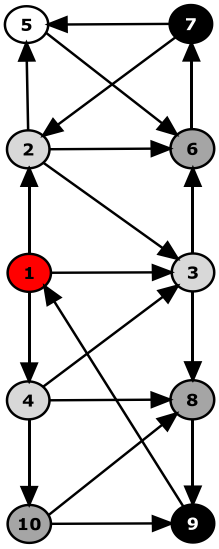
Etape 5 : successeurs de 8, le dernier sommet visité NON TRAITE
Le sommet 8 n'a pas de successeur.
Les sommets visités à la fin de cette étape sont **1,2,3,4,8,10,9**



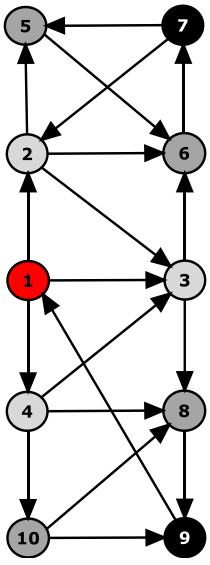
Etape 6 : successeurs de 3, le dernier sommet visité non traité
Le sommet 3 a un successeur 6.
Les sommets visités à la fin de cette étape sont **1,2,3,4,8,10,9,6**



Etape 7 : successeurs de 6, le dernier sommet visité non traité
Le sommet 6 a un successeur 7.
Les sommets visités à la fin de cette étape sont **1,2,3,4,8,10,9,6,7**



Etape 8 : successeurs de 7, le dernier sommet visité non traité
Le sommet 7 a un successeur 5.
Les sommets visités à la fin de cette étape sont **1,2,3,4,8,10,9,6,7,5**



Etape 9 & 10 : successeurs de 5 et 2, derniers sommets visités non traités
Tous les sommets ayant été visitéS, les sommets 5 et 2 restant à traiter n'ont plus de successeur non visité.

BILAN DU PARCOURS EN PROFONDEUR

A chaque étape :

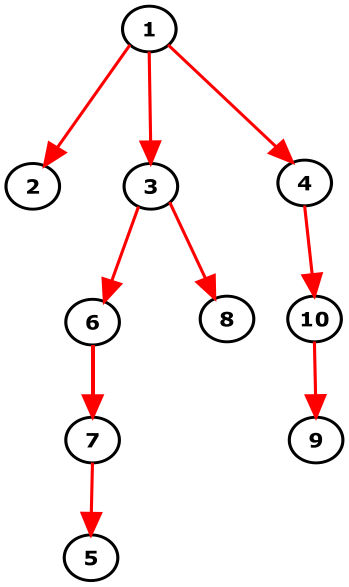
- On choisit le dernier sommet qui a été visité et qui n'a pas été traité,
- On « traite » ce sommet, c'est-à-dire que l'on regarde ses successeurs (uniquement ceux qui n'ont pas encore été visités).

Pour savoir quel est le dernier sommet traité, il faut lister, au fur et à mesure, les sommets visités et distinguer les sommets traités (ou les supprimer de la liste une fois qu'ils ont été traités).

ARBRE DE PARCOURS

Le résultat d'un parcours de graphe est l'ensemble des chemins parcourus pour visiter tous les sommets du graphe. La représentation de ces chemins peut se faire par un arbre, qui est un graphe aux propriétés particulières.

Arbre du parcours en profondeur précédent :



Un arbre* se caractérise par :

- L'existence d'un sommet **racine** sans prédécesseur : ici le sommet 1 qui est le sommet de départ du parcours,
- Le fait que tous les autres sommets ont un unique prédécesseur. Ici, le prédécesseur est le sommet à partir duquel le sommet a été vu ou visité.

**Pour plus de vocabulaire sur les arbres : Cf. p 13*

PREPARATION A LA DESCRIPTION ALGORITHMIQUE

Pour programmer un algorithme de parcours et retourner un arbre de parcours deux questions se posent :

1. Comment représenter l'information contenue dans un arbre de parcours ?
2. Comment représenter la liste des sommets visités que l'on utilise pour choisir les sommets à traiter ?

1. INFORMATION CONTENUE DANS UN ARBRE DE PARCOURS

Puisqu'un arbre est un graphe particulier, nous pourrions utiliser les mêmes structures de données pour le représenter : matrice d'adjacence, liste de successeurs…

Mais il y a plus simple : la caractéristique d'un arbre est l'unicité des prédécesseurs des sommets. Une liste (tableau, vecteur…) des prédécesseurs permet donc de reconstituer un arbre.

Par exemple, l'arbre de parcours ci-dessus peut être défini par la liste (ou le tableau) de prédécesseurs ***pred*** suivante :

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
pred	X	1	1	1	7	3	6	3	10	4

2. « LISTE » DES SOMMETS VISITES

Nous avons utilisé une liste dans laquelle nous ajoutons, à chaque étape les nouveaux sommets visité.
Evolution de cette liste dans l'exemple précédent :

- **1**
- **1,2,3,4**
- **1,2,3,4,8,10**
- **1,2,3,4,8,10,9**
- **1,2,3,4,8,10,9**
- **1,2,3,4,8,10,9**
- **1,2,3,4,8,10,9,6**
- **1,2,3,4,8,10,9,6,7**
- **1,2,3,4,8,10,9,6,7**
- **1,2,3,4,8,10,9,6,7,5**

Si on considère qu'une fois traités les sommets encadrés peuvent être enlevés de la liste :

- **1**
- ~~1~~**,2,3,4**
- **2,3,**~~4~~**,8,10**
- **2,3,8,**~~10~~**,9**
- **2,3, 8,**~~9~~
- **2,3,** ~~8~~
- **2,**~~3~~**,6**
- **2,**~~6~~**,7**
- **2,**~~7~~**,5**
- **2,**~~5~~
- ~~2~~

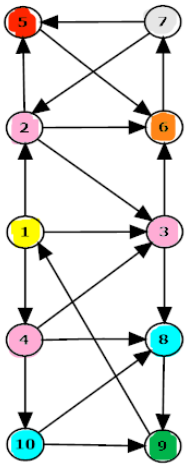
~~2~~
~~5~~
~~7~~
~~6~~
~~9~~
~~10~~
~~8~~
~~4~~
~~3~~
~~2~~
~~1~~

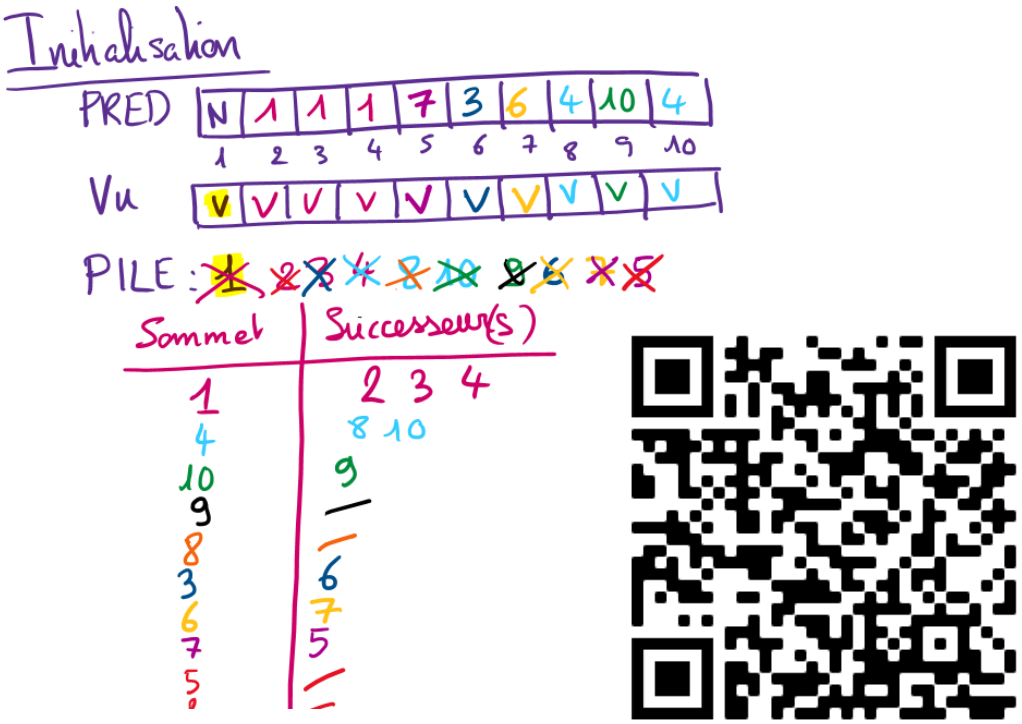
Le comportement de la liste des sommets à traiter est celui d'une pile !

METHODE ITERATIVE

Avant de passer à l'algorithme, nous pouvons décrire une méthode itérative qui nous permet de remplir le vecteur **pred** en utilisant une pile.

Voici le résultat de la méthode (ce qui est demandé en exercice !) et la vidéo d'explications

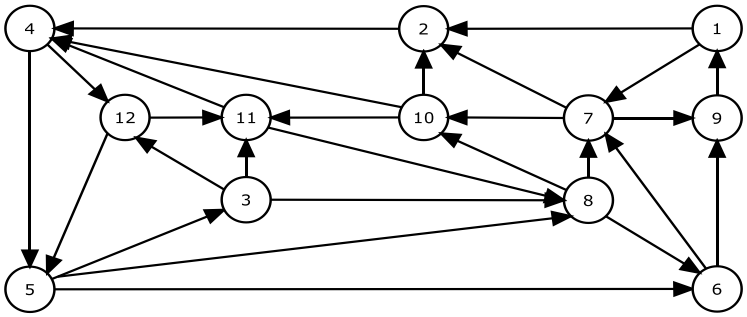




EXERCICE 1 : PARCOURS EN LARGEUR ET EN PROFONDEUR (CORRIGE P14)

Pour le graphe G ci-dessous.

1. Faire un parcours en profondeur de G à partir du sommet 1 en précisant l'arbre de parcours (liste des prédécesseurs).



2. Parcours en largeur : dans le parcours en largeur, au lieu de « rester » sur le dernier sommet visité (**pile**) nous traitons les sommets dans l'ordre dans lequel ils apparaissent. Dans la méthode itérative précédente, il suffit donc de remplacer la **pile** par une **file**.
   Appliquer la méthode pour un parcours en largeur de G à partir du sommet 1 en précisant l'arbre de parcours (liste des prédécesseurs).

ALGORITHME ET PROGRAMMATION PYTHON

ALGORITHME DE PARCOURS EN PROFONDEUR (COMPLETER – CORRIGE P15)

Données de départ : le graphe et le sommet de départ

- S ensemble des sommets (numérotés)
- $\mathbf{s_0}$ est le sommet de départ,
- Pour chaque sommet s, on connaît l'ensemble de ses successeurs ou prédécesseurs

Types abstraits utilisés :

→ Une pile **Pile** :
- `Empiler(Pile,x)` : ajouter x en fin de la pile `Pile`
- `Dépiler(Pile,x)` : enlever x en début de la pile `Pile`

→ Une liste (tableau) des prédécesseurs, **Pred**

→ Une liste (tableau) booléenne pour indiquer si un sommet a été déjà visité, **Vu**

Initialisation

```
Pour tout sommet s de S faire
      Pred[s]← null
      Vu[s]← Faux
Fin pour
s0← Sommet de départ
Vu[s0]← Vrai
Empiler(Pile,s0)
```

Itération (à compléter)

```
Tant que …
```

PYTHON

L'objectif est de programmer une fonction **parcours-profondeur(M,s)** qui :

- Prend en paramètres une **liste de liste M**, représentant la matrice d'adjacence du graphe, et **s** le sommet de départ (indice compris entre 0 et n-1),
- Renvoie la liste des prédécesseurs qui permet d'identifier l'arbre du parcours en profondeur.

IMPLEMENTATIONS :

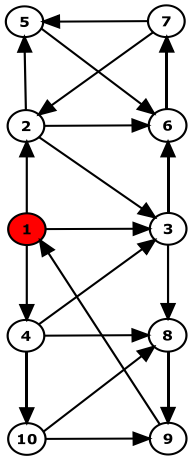
- **Pred et Vu** sont représentés par des listes
- La pile **Pile** peut aussi très facilement être représentée par une liste
  - Empiler(**Pile**,**s**) → `Pile.append(s)`
  - Depiler(**Pile,s**) → `s=Pile.pop(-1)`
- Les **successeurs** d'un sommet peuvent être visités par la fonction `lst_succ(M,s)` qui renvoie la liste des successeurs du sommet s (indice) dans le graphe représenté par la liste de liste M :

```
def lst_succ(M,s):
    lst=[]
    n=len(M)
    for j in range(n):
        if M[s][j]==1:
            lst.append(j)
    return(lst)
```

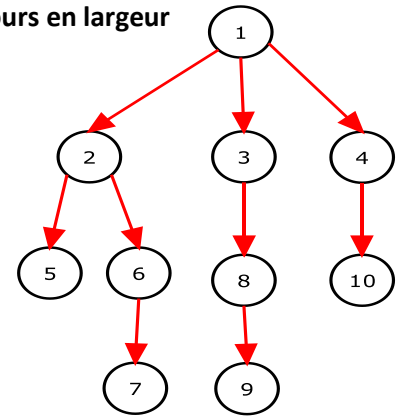
PARCOURS EN LARGEUR DU GRAPHE

« ON TRAITE LES SOMMETS DANS L'ORDRE DANS LEQUEL ON LES A VISITES »

C'est la même méthode, la différence est l'utilisation d'une **File** à la place d'une **Pile**.



Arbre du parcours en largeur



PYTHON - PARCOURS EN LARGEUR

L'objectif est le même que pour le parcours en profondeur, il s'agit de programmer une fonction **parcours-largeur(M,s)** qui :

- Prend en paramètres une **liste de liste M**, représentant la matrice d'adjacence du graphe, et **s** le sommet de départ (indice compris entre 0 et n-1),
- Renvoie la liste des prédécesseurs qui permet d'identifier l'arbre du parcours en largeur.

La différence par rapport à l'algorithme précédent est la manipulation de la file qui peut se faire également par une liste **File** :

- Enfiler(**File**,**s**) → `File.append(s)`
- Depiler(**File,s**) → `s=File.pop(0)`

DERNIERES REMARQUES

- **Le parcours en profondeur se programme aussi très naturellement de façon récursive.**
- En anglais :
  - Parcours en profondeur : **DFS**, pour *Depth-First Search*
  - Parcours en largeur : **BFS**, pour *Breadth First Search*
- Les algorithmes de parcours proposés sont, pour l'instant, « des coquilles vides » et le résultat attendu, l'arbre de parcours représenté par la liste des prédécesseurs ***pred***, peut être assez abstrait. Cette liste nous permet de vérifier que tous les sommets accessibles ont été visités. Dans une situation plus réelle, ce n'est pas forcément **pred** qui sera important, mais le traitement effectué sur chaque sommet visité.
- Ce qu'il faut retenir, c'est la méthode générale : on choisit un sommet, on regarde ses successeurs, on prend un de ces successeurs, on regarde ses successeurs… On pourra s'en inspirer face à des problèmes classiques commet non classiques.

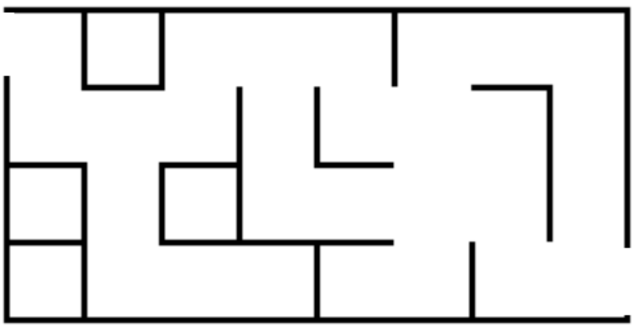
TRAVAUX PRATIQUES

RECONSTITUTION D'UN CHEMIN A PARTIR D'UNE LISTE DE PREDECESSEURS

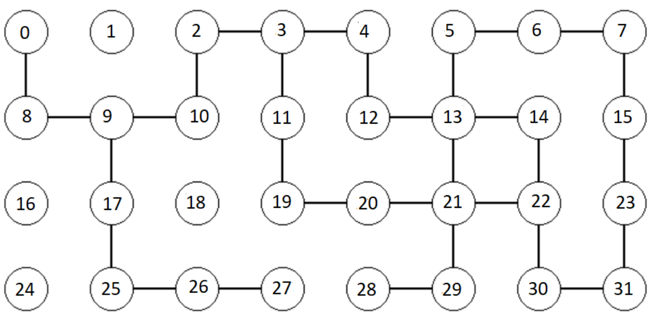
1. Ecrire une fonction **chemin_prof(depart, arrivee)** qui :
   - Prend en paramètres deux sommets **depart** et **arrivee**
   - Renvoie une liste de sommets [depart,…,arrivee] contenant le chemin parcouru pour aller de **depart** à **arrivee** en utilisant un parcours en profondeur.
2. Même question avec une fonction **chemin_larg(depart, arrivee)** et un parcours en largeur.

PARCOURS D'UN LABYRINTHE

On rentre dans le labyrinthe ci-dessous par la case située en haut à gauche et nous en sortant par la case située en bas à droite.



Ce labyrinthe peut être représenté par le graphe ci-dessous :




```
M=[[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0]]
```

Comparez les chemins obtenus pour sortir du labyrinthe en utilisant un parcours en largeur ou un parcours en profondeur (ici, il n'y a pas beaucoup de différence, le labyrinthe étant, somme toute, assez simple)

DECOMPOSITION EN NIVEAUX D'UN GRAPHE SANS CIRCUIT

Nous avons déjà vu des exemples de **graphes sans circuit** dans le niveau 1 **avec les relations d'ordre*** (relations réflexives, transitives et antisymétriques).
Nous avions vu également que le fait qu'un graphe soit **sans circuit** induisait **une hiérarchie** entre les sommets et, pour la relation d'ordre, nous avions utilisé le diagramme de Hasse pour représenter cette hiérarchie.

C'est cette hiérarchie qui nous intéresse dans cette partie : existe-t-il une méthode itérative, un algorithme, permettant d'ordonner entre eux les sommets d'un graphe sans circuit ?

Pour définir cette hiérarchie, nous utiliserons la notion de niveau et les définitions et propriétés ci-dessous :

Source & Puits
Dans un graphe, un sommet sans prédécesseur est appelé **source**, un sommet sans successeur est appelé **puits**.

Propriétés d'un graphe sans circuit
Dans un graphe sans circuit, il existe au moins un sommet **puits** et au moins un sommet **source.**

Décomposition en niveau
Si G est un graphe sans circuit, il est possible de définir **un niveau** pour chaque sommet du graphe.
Ce niveau se définit de la façon suivante :

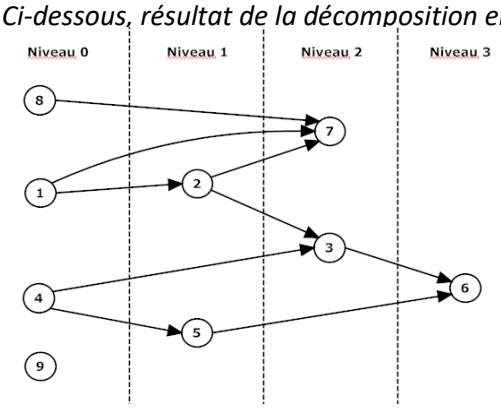
- Les sommets sans prédécesseurs sont de niveau 0,
- Tout sommet x a un niveau supérieur aux niveaux de ses prédécesseurs :

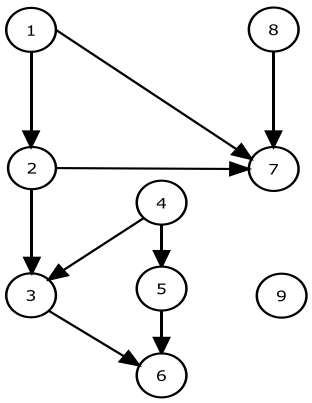
$$niveau(x) = \max_{\substack{y \text{ prédecesseur} \\ de\ x}} niveau(y) + 1$$

Pour obtenir le niveau d'un sommet, il faut ajouter 1 au plus grand niveau de ses prédécesseurs
Le niveau d'un sommet x correspond aussi à la distance maximum de x à un sommet source.

Le graphe peut être ensuite redessiné en disposant les sommets de gauche à droite dans l'ordre croissant des niveaux. Si le graphe représente une relation d'ordre, la décomposition en niveaux permet de dessiner le diagramme de Hasse (dans lequel on ne représente pas les raccourcis et les boucles)

Ci-dessous, résultat de la décomposition en niveau du graphe ci-contre :





En effet, un graphe à la fois **transitif et antisymétrique est forcément sans circuit (nous ne comptons pas les boucles).*
Démonstration par l'absurde : supposons qu'un graphe transitif et antisymétrique contienne un circuit qui n'est pas une boucle.
Prenons a et b, deux sommets distincts de ce circuit : il existe donc un chemin allant de a vers b et un chemin allant de b vers a. Le graphe étant transitif, les raccourcis (a, b) et (b, a) de ces chemins appartiennent au graphe, ce qui nous amène à une contradiction puisque le graphe est antisymétrique.

METHODE ITERATIVE POUR DETERMINER LES NIVEAUX D'UN GRAPHE SANS CIRCUIT ?

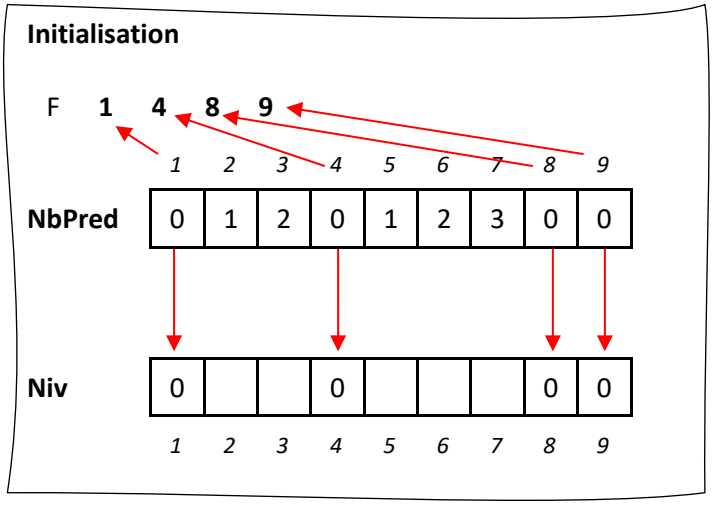
La méthode repose sur l'observation précédente :

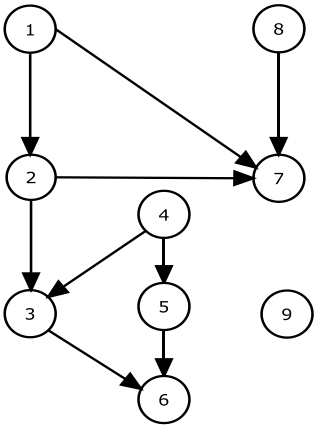
Pour obtenir le niveau d'un sommet, il faut ajouter 1 au plus grand niveau de ses prédécesseurs

Pour connaître le niveau d'un sommet, nous devons donc connaître le niveau **de tous ses prédécesseurs.**
On pourra donc utiliser un vecteur contenant initialement les degrés entrants de chaque sommet et au fil des itérations, le nombre de prédécesseurs restant à traiter.
Ce qui signifie qu'à chaque fois qu'un sommet est visité, on décrémente le nombre de prédécesseurs restant à traiter. Quand il n'y a plus de prédécesseurs à traiter, on peut donc déterminer le niveau du sommet et le traiter (visiter ses successeurs). Les sommets à traiter peuvent être mis dans une file, par exemple (nous reviendrons sur ce choix plus tard)

Nous utilisons donc :

- F : une file dans laquelle nous enfilons les sommets pour lesquels nous connaissons les niveaux. Initialement nous y enfilons la ou les sources (de niveau 0)
- NbPred : une liste (vecteur ou tableau) contenant initialement le nombre de prédécesseurs (degrés entrants) de chaque sommet.
- Niv : bien sûr la liste résultat (vecteur ou tableau) contenant les niveaux des sommets

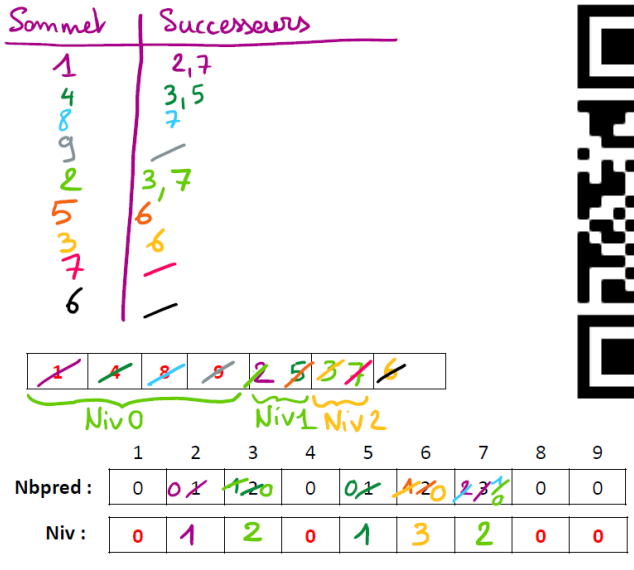




Principe de la méthode itérative

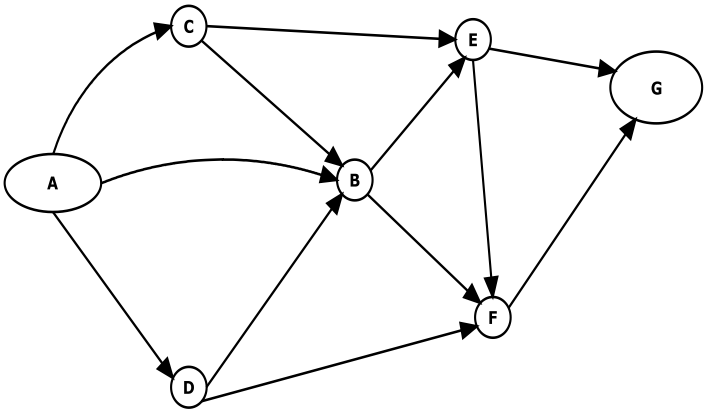
→ On défile un sommet
→ On le traite (on regarde ses successeurs)
→ On décrémente **NbPred** pour les successeurs visités
→ Si **NbPred** d'un de ces successeurs est nul : on l'enfile et on met à jour son niveau

Résultat final
et vidéo d'explication



*Remarque : les sommets sont enfilés par ordre croissant de leur niveau : le dernier prédécesseur visité (pris dans la **file**) est donc toujours celui de plus haut niveau. Ce ne serait pas forcément le cas avec une **pile**.*

REMARQUE : TRI TOPOLOGIQUE

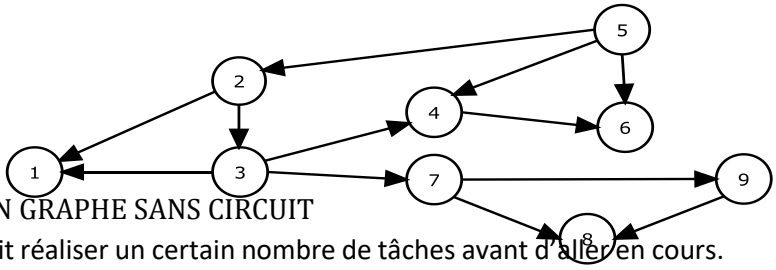


Un tri topologique d'un graphe orienté sans circuit $G = (S, A)$ est une façon d'ordonner tous les sommets du graphe de sorte que si l'arc $(x, y) \in A$ alors x apparaît avant y dans le tri topologique.
Un même graphe peut avoir plusieurs tris topologiques, ainsi (A,D,C,B,E,F,G) et (A,C,D,C,B,E,F,G) sont des tris topologiques du graphe ci-contre.
Une décomposition en niveau permet d'obtenir un tri topologique, il suffit de prendre l'ordre des sommets dans la file.

EXERCICES (CORRIGES P16)

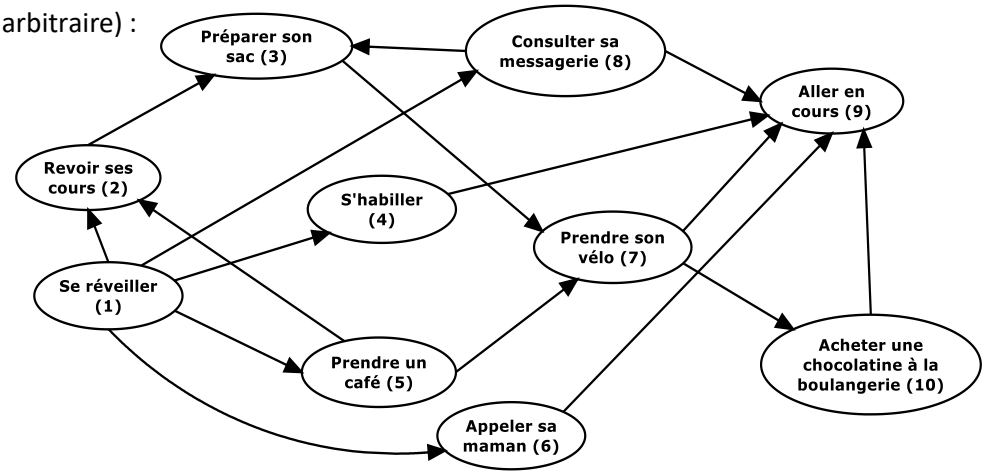
EXERCICE 2 : DECOMPOSITION EN NIVEAU D'UN GRAPHE SANS CIRCUIT

Décomposer en niveaux le graphe ci-contre



EXERCICE 3 : DECOMPOSITION EN NIVEAU D'UN GRAPHE SANS CIRCUIT

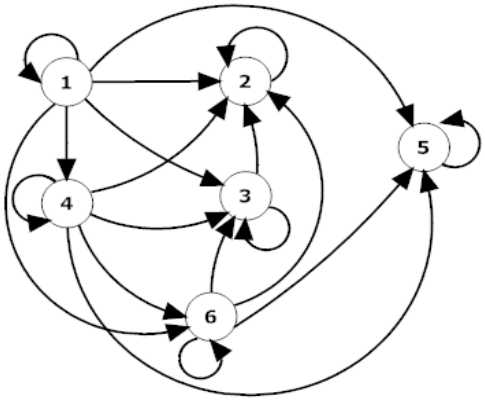
Jules est un étudiant de l'IUT. Tous les matins, il doit réaliser un certain nombre de tâches avant d'aller en cours. Celles-ci sont représentées dans le graphe ci-dessous (les sommets, entre parenthèses, ont été numérotés de façon arbitraire) :



Pour aider Jules à y voir plus clair, donner une décomposition en niveau du graphe ci-dessus (on donnera les éléments : tableaux, pile/file,… ayant permis d'obtenir la décomposition par une méthode itérative).

EXERCICE 4 : RELATION D'ORDRE ET DIAGRAMME DE HASSE

Utiliser une décomposition en niveau pour représenter un diagramme de Hasse de la relation d'ordre ci-contre :



TEST ELEARN

ALGORITHME ET PROGRAMMATION PYTHON (CORRIGE P18)

PSEUDO-CODE (COMPLETEZ)

```
Pour tout sommet s de S faire
      Niv[s]← 0
      NPred[s]←nombre de prédécesseur(s) de s
      si NPred[s]==0 alors Ajouter_file(File,s)
Fin pour
```

Initialisation

```
Tant que …
```

PYTHON

L'objectif est de programmer une fonction **decomp_niv(M)** qui :

- Prend en paramètres une **liste de liste M**, représentant la matrice d'adjacence du graphe, et **s** le sommet de départ (indice compris entre 0 et n-1),
- Renvoie la liste des niveaux des sommets (Niv)

IMPLEMENTATIONS :

- **NbPred et Niv** sont représentés par des listes
- La file **File** peut être représentée par une liste
  - Empiler(**file**,**s**) → `file.append(s)`
  - Depiler(**file**,**s**) → `s=Pile.pop(0)`
- Le remplissage initial de la liste **NbPred** peut se faire par la fonction **nb_pred(M,s)** vue dans le niveau précédent :

```
def nb_pred(M,s):
    nb=0
    for i in range(len(M)):
        if M[i][s]==1 :
            nb=nb+1
    return nb
```

- Les **successeurs** d'un sommet peuvent être visités par la fonction `lst_succ(M,s)` qui renvoie la liste des successeurs du sommet s (indice) dans le graphe représenté par la liste de liste M :

```
def lst_succ(M,s):
      lst=[]
      n=len(M)
      for j in range(n):
            if M[s][j]==1:
                  lst.append(j)
      return(lst)
```

COMPLEMENT : ARBRE - VOCABULAIRE

ARBRE

Un **arbre** est un graphe orienté $G = (S, A)$ tel que :

- G est connexe
- Il existe un unique sommet de G n'ayant pas de prédécesseur.
- Tous les autres sommets ont exactement un prédécesseur.

Dans la définition ci-dessus, on peut remplacer la condition « G est connexe » par « G est sans circuit »

VOCABULAIRE

Racine de l'arbre : le seul sommet de G qui n'a pas de prédécesseur.
Feuilles de l'arbre : les sommets qui n'ont pas de successeur.
Nœuds internes de l'arbre : tous les autres sommets (autres que la racine et les feuilles).
Branche de l'arbre : tout chemin de la racine vers le sommet.
Descendant(s) de x : le(s) successeur(s) de x.
Ascendant(s) de x : le(s) prédécesseur(s) de x.
Arbre binaire : arbre dans lequel chaque sommet a au plus deux successeurs.
Hauteur : longueur du trajet le plus long entre la racine et une feuille (compté en nombre d'arcs).

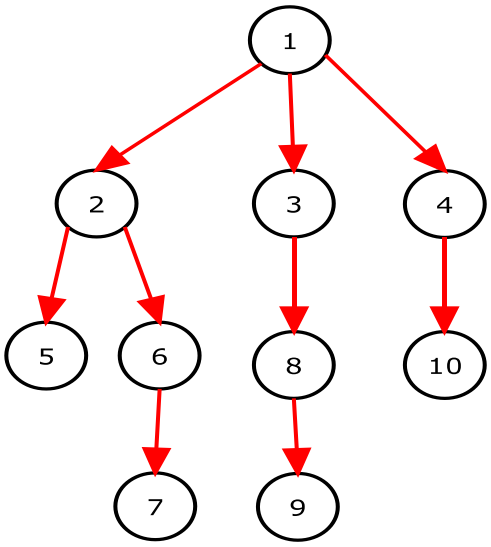
La structure d'un arbre peut être entièrement reconstruite par la liste des prédécesseurs définie par :

$$\forall y \in S, \qquad pred(y) = \begin{cases} \emptyset \; si \; y \; est \; la \; racine \; de \; G \\ x \; tel \; que \; (x, y) \in A \;\; sinon \end{cases}$$

Propriété : Soit $G = (S, A)$ un arbre de racine r. Pour tout sommet x, il existe un unique chemin de r vers x.

EXEMPLE

Racine : sommet 1
Feuilles : sommets 5,7, 9 et 10
Nœuds internes : sommets 2,3,4,6,8
Vecteur des prédécesseurs : pred=(∅,1,1,1,2,2,6,3,8,4)

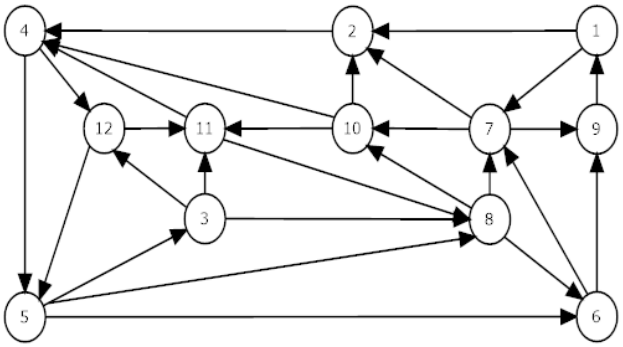


CORRIGES

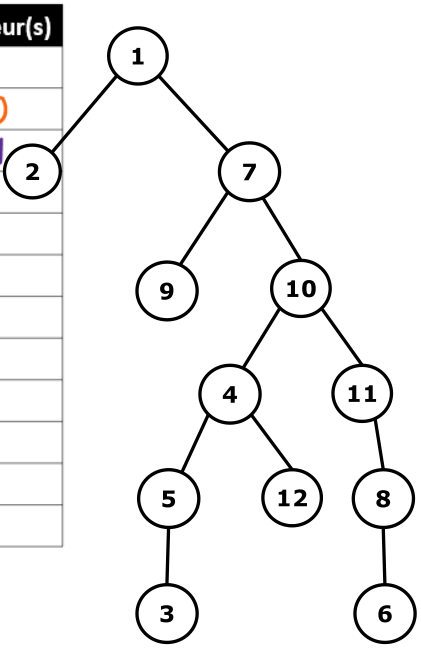
EXERCICE 1 : PARCOURS EN LARGEUR ET EN PROFONDEUR

Pour le graphe G ci-dessous.

1. Faire un parcours en profondeur de G à partir du sommet 1 en précisant l'arbre de parcours (liste des prédécesseurs).



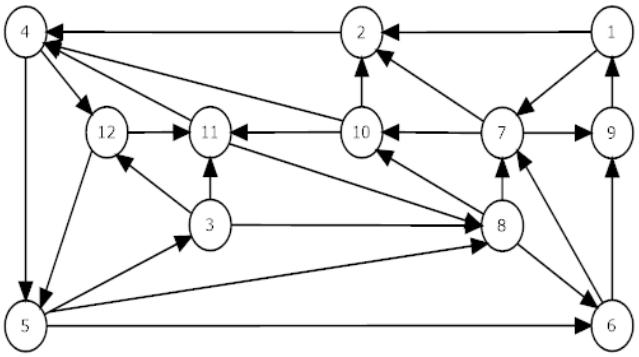
Sommet	Successeur(s)
1	2,7
7	9,10
10	4,11
11	8
8	6
6	—
4	5,12
12	—
5	3
9	—
7	—
2	—



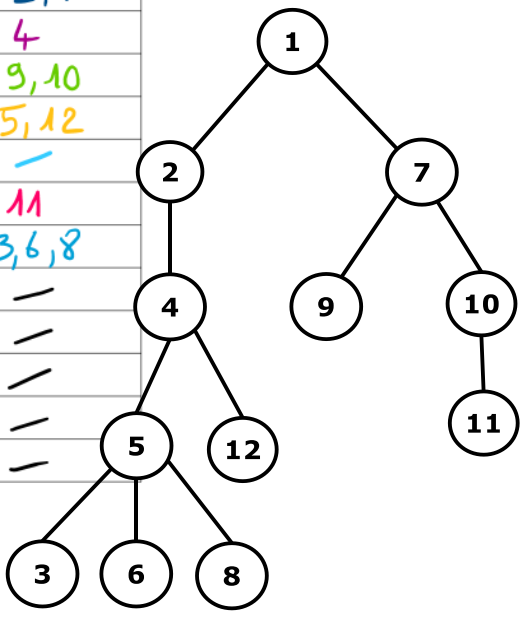
Pile : ~~1~~ ~~2~~ ~~7~~ ~~9~~ ~~10~~ ~~4~~ ~~11~~ ~~8~~ ~~6~~ ~~5~~ ~~12~~

	1	2	3	4	5	6	7	8	9	10	11	12
Pred :	X	1	5	10	4	8	1	11	7	7	10	4
Vu :	V	V	V	V	V	V	V	V	V	V	V	V

2. Parcours en largeur



Sommet	Successeur(s)
1	2,7
2	4
7	9,10
4	5,12
9	—
10	11
5	3,6,8
12	—
11	—
3	—
6	—
8	—



File : ~~1~~ ~~2~~ ~~7~~ ~~4~~ ~~9~~ ~~10~~ ~~5~~ ~~12~~ ~~11~~ 3 6 8

	1	2	3	4	5	6	7	8	9	10	11	12
Pred :	X	1	5	2	4	5	1	5	7	7	10	4
Vu :	V	V	V	V	V	V	V	V	V	V	V	V

ALGORITHME EN PSEUDO-CODE – PARCOURS EN PROFONDEUR (NON RECURSIF)

```
Pour tout sommet s de S faire
      Pred[s]← null
      Vu[s]← Faux
Fin pour
s0← Sommet de départ
Vu[s0]← Vrai
Empiler(Pile,s0)

Tant que la file Pile est non vide faire
      Dépiler(Pile,s)
      Pour tout successeur succ de s faire
            si Vu[succ]==Faux alors
                  Vu[succ] ← Vrai
                  Pred[succ] ← s
                  Empiler(Pile,succ)
            fin si
      fin Pour
Fin Tant que
```

Initialisation

```python
def parcours_profondeur(Mat,s0):
    n=len(Mat)
    pred=[None]*n
    vu=[False]*n

    vu[s0]=True
    pile=[s0]

    while pile!=[]:
        s=pile.pop(len(pile)-1)
        for succ in lst_succ(Mat,s):
            if vu[succ]==False:
                vu[succ]=True
                pred[succ]=s
                pile.append(succ)
    return(pred)

print(parcours_profondeur(M,0))
```

ALGORITHME EN PSEUDO-CODE – PARCOURS EN LARGEUR

```
Pour tout sommet s de S faire
      Pred[s]← null
      Vu[s]← Faux
Fin pour
s0← Sommet de départ
Vu[s0]← Vrai
Enfiler(File,s0)
```

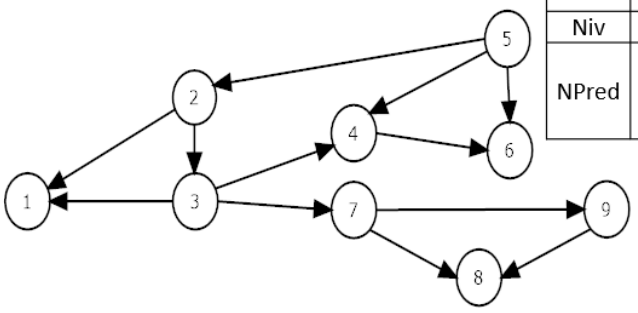
Initialisation

```
Tant que la file File est non vide faire
      Défiler(File,s)
      Pour tout successeur succ de s faire
            si Vu[succ]==Faux alors
                  Vu[succ] ← Vrai
                  Pred[succ] ← s
                  Enfiler(File,succ)
            fin si
      fin Pour
Fin Tant que
```

```python
def parcours_largeur(Mat,s0):
    n=len(Mat)
    pred=[None]*n
    vu=[False]*n

    vu[s0]=True
    file=[s0]

    while file!=[]:
        s=file.pop(0)
        for succ in lst_succ(Mat,s):
            if vu[succ]==False:
                vu[succ]=True
                pred[succ]=s
                file.append(succ)
    return(pred)

print(parcours_largeur(M,0))
```

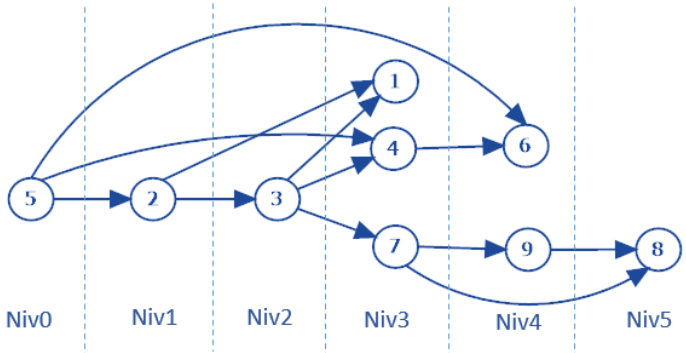
EXERCICE 2 : DECOMPOSITION EN NIVEAU D'UN GRAPHE SANS CIRCUIT

Décomposer en niveaux le graphe ci-contre



	1	2	3	4	5	6	7	8	9
Niv	3	1	2	3	0	4	3	5	4
NPred	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	~~2~~ ~~1~~ 0	0	~~2~~ ~~1~~ 0	~~1~~ 0	~~2~~ ~~1~~ 0	~~1~~ 0

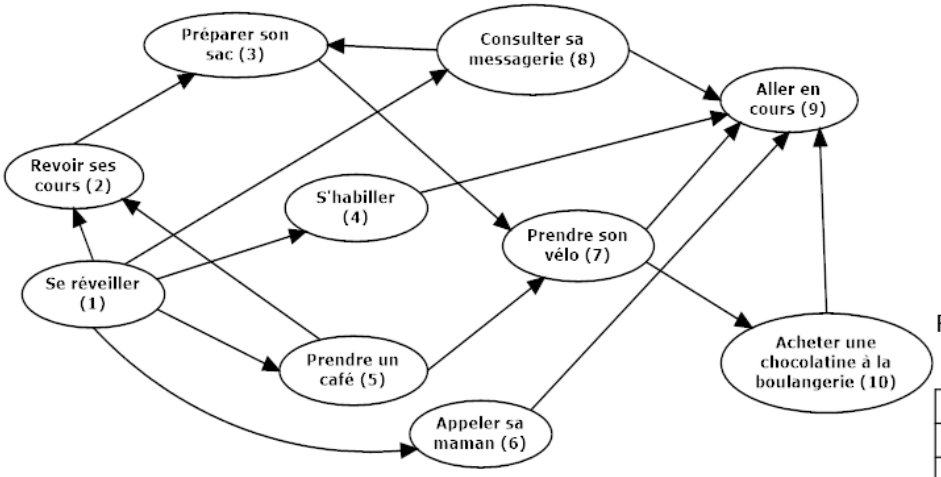
File : 5, 2, 3, 1, 4, 7, 9, 8



Sommet	Successeur(s)
5	2, 4, 6
2	1,3
3	1, 4, 7
1	-
4	6
7	8,9
6	-
9	8
8	-

EXERCICE 3 : DECOMPOSITION EN NIVEAU D'UN GRAPHE SANS CIRCUIT

Jules est un étudiant de l'IUT. Tous les matins, il doit réaliser un certain nombre de tâches avant d'aller en cours. Celles-ci sont représentées dans le graphe ci-dessous (les sommets, entre parenthèses, ont été numérotés de façon arbitraire) :

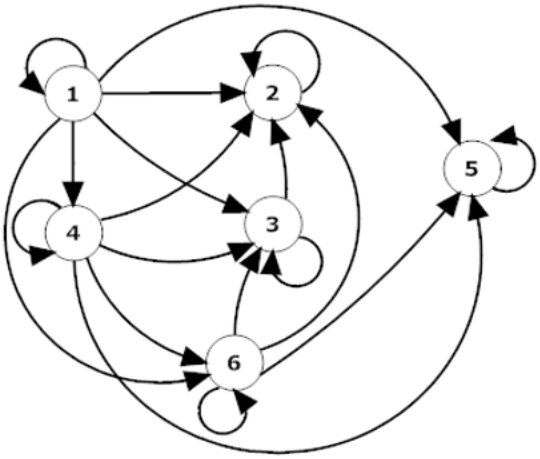


File : 1, 4, 5, 6, 8, 2, 3, 7, 10,9

	1	2	3	4	5	6	7	8	9	10
Niv	0	2	3	1	1	1	4	1	6	5
NPred	0	~~2~~ ~~1~~ 0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~2~~ ~~1~~ 0	~~1~~ 0	~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0	~~1~~ 0

Sommet	Successeur(s)
1	2, 4, 5, 6, 8
4	9
5	2,7
6	9
8	3, 9
2	3
3	7
7	9, 10
10	9
9	-

EXERCICE 4 : RELATION D'ORDRE ET DIAGRAMME DE HASSE



Si on ne tient pas compte des boucles, le graphe est sans circuit. On applique l'algorithme de détermination des niveaux.

	1	2	3	4	5	6
Niv	0	4	3	1	3	2
NPred	0	~~4~~ ~~3~~ ~~2~~ ~~1~~ 0	~~3~~ ~~2~~ ~~1~~ 0	~~1~~ 0	~~3~~ ~~2~~ ~~1~~ 0	~~2~~ ~~1~~ 0

File : 1, 4, 6, 3, 5, 2

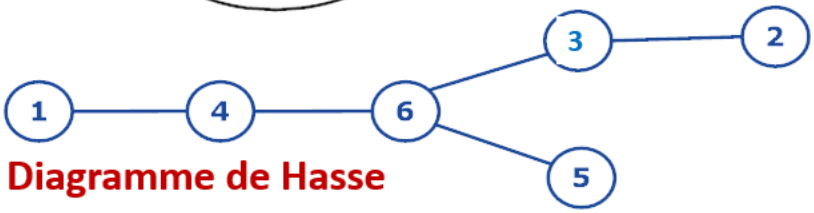


Diagramme de Hasse

Sommet	Successeur(s)
1	2, 3, 4, 5, 6
4	2, 3, 5, 6
6	2, 3, 5
3	2
5	-
2	-

PROGRAMMATION DECOMPOSITION EN NIVEAUX

PSEUDO-CODE

```
Pour tout sommet s de S faire
      Niv[s]← 0
      NPred[s]←nombre de prédécesseur(s) de s
      si NPred[s]==0 alors Ajouter_file(File,s)
Fin pour

Tant que …  la File File est non vide faire
      Defiler(File,s)
      Pour tout successeur s' de s faire
            NPred[s']← NPred[s']-1
            Si Npred[s']==0 alors
                  Enfiler(File,s')
                  Niv[s']=Niv[s]+1
            fin si
      Fin Pour
Fin Tant que
```

Initialisation

PYTHON

```
def decomp_niv(M):
    File=[]
    Niv=[0]*len(M)
    NbPred=[0]*len(M)

    for i in range(len(M)):
        NbPred[i]=nb_pred(M,i)
        if NbPred[i]==0 :
            File.append(i)

    while File != []:
        s=File.pop(0)
        for succ in lst_succ(M,s):
            NbPred[succ]=NbPred[succ]-1
            if NbPred[succ]==0 :
                File.append(succ)
                Niv[succ]=Niv[s]+1
    return Niv
```

```
def nb_pred(M,s):
    nb=0
    for i in range(len(M)):
        if M[i][s]==1 :
            nb=nb+1
    return nb
```

```
def lst_succ(M,s):
      lst=[]
      n=len(M)
      for j in range(n):
            if M[s][j]==1:
                  lst.append(j)
      return(lst)
```