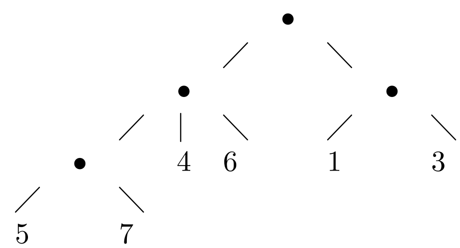
Interrogation d'informatique n°5. *Barème sur 22.5 pts*

Exercice A

[3 pts] Soit un arbre dont les feuilles sont étiquetées par des entiers :



L'arbre ci-dessus est codé par la liste $A = [\ [\ [5,7]\ ,\ 4\ ,\ 6\ ]\ ,\ [1,3]\ ]$.

Un arbre réduit à une feuille est un entier. Sinon, il est une liste d'arbres.

Écrire une fonction **récursive** `somme(A)` qui renvoie la somme des étiquettes des feuilles de l'arbre A.

On utilisera notamment la fonction booléenne `isinstance(x,int)` qui renvoie `True` ssi x est un entier.

Exercice B. Ordonnancements de tâches

On considère un ensemble $T = [\![0, n-1]\!]$ de tâches à accomplir. On suppose que l'accomplissement de chaque tâche dure une heure. On connait aussi une liste de contraintes A données sous la forme d'une liste de listes :

Pour tout $i \in T$, $A[i]$ est la liste des tâches ne pouvant être accomplies qu'une fois la tâche i accomplie.

On note $i \longrightarrow j$ pour signifier que j appartient la liste $A[i]$. On dit que (i, j) est une arête (ou contrainte) du graphe des tâches, et on note m le nombre d'arêtes.

Autrement dit, m est la somme des longueurs des listes $A[i]$, pour $i \in T$.

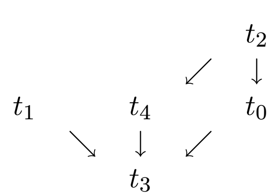
Remarque : Lorsque $i \longrightarrow j$ et $j \longrightarrow k$, alors la tâche k doit être accomplie après j, donc après i, mais l'arête $i \longrightarrow k$ n'apparaît pas nécessairement dans la liste $A[i]$ donnée initialement.

Plus généralement, on note $i \prec j$ pour signifier que i doit être accomplie avant j.

On suppose que **le graphe des tâches est acyclique** (= sans cycle) : on ne peut avoir à la fois $i \prec j$ et $j \prec i$.

Un **ordonnancement** est une application $\sigma : \{0, 1, ..., n-1\} \to T$ qui attribue un ordre d'exécution des tâches respectant les contraintes : autrement dit, les tâches peuvent être effectuées dans l'ordre $\sigma(0), \sigma(1), ..., \sigma(n-1)$.

Exemple : Considérons le graphe des tâches :



Un ordonnancement possible est $\sigma(0) = t_2$, $\sigma(1) = t_1$, $\sigma(2) = t_4$, $\sigma(3) = t_0$ et $\sigma(4) = t_3$.

Autrement dit, on peut effectuer les tâches dans l'ordre t_2, t_1, t_4, t_0, t_3.

En Python, cet ordonnancement est codé par la liste `L = [2,1,4,0,3]`.

1) [2 pts] Écrire une fonction `inverse(A)` qui étant donné A renvoie B, de sorte que $j \in A[i]$ ssi $i \in B[j]$.

Ainsi, B est la liste des listes $B[j]$, où $B[j]$ est la liste des tâches devant être accomplies avant j.

On proposera un algorithme **de complexité linéaire** $O(n+m)$, où n est le nombre de tâches (sommets du graphe) et m le nombre de contraintes (= arêtes), c'est-à-dire m est la somme des longueurs des $A[i]$.

2) On considère l'algorithme suivant, correspondant à **un étiquetage par hauteur** depuis les racines :

Étape 1. Initialement, aucune tâche n'est étiquetée : on initialise le tableau H par : `H[i] = -1`

Étape 2. À l'itération d'ordre $k = 0$, on parcourt l'ensemble des tâches et on affecte l'étiquette 0 aux tâches racines, c'est-à-dire celles qui peuvent être accomplies en premier.

Étape 3. À l'itération $k > 0$, on parcourt **l'ensemble des tâches** en repérant toutes les tâches i qui n'ont pas encore été étiquetées mais dont toutes les tâches qui doivent être effectuées avant i sont déjà étiquetées. On affecte ensuite à chacune de ces tâches l'étiquette k.

Étape 4. L'algorithme termine quand toutes les tâches sont étiquetées (et on a nécessairement $k < n$).

On donne la fonction suivante :

```
def hauteur(A) :
    n = len(A) ; H = [-1]*n ; B = inverse(A)
    test = False
    for i in range(n) :
        if B[i] == [] :  H[i] = 0
        else :  test = True
    k = 1
    while test :
        test = False
        for i in range(n) :
            if H[i] == -1 :
                flag = True
                for j in B[i] :
                    if H[j] == -1 or H[j] == k :      ### à compléter ###
                if flag :  H[i] = k
                else :  test = True
        k = k+1
    return H
```

Remarque : $H[i]$ représente la hauteur de i, c'est-à-dire la longueur maximale k des séquences telles que

$$j_0 \to j_1 \to \ldots \to j_k \quad \text{avec } j_k = i$$

a) [3 pts] *Questions indépendantes* concernant le programme PYTHON ci-dessus :

(i) Indiquer pour chacune des quatre étapes les lignes du programme qui leur correspondent

(ii) Indiquer ce que représente le booléen `test` après l'exécution de la ligne 16

(iii) Compléter la ligne 14 et expliquer les conditions du test.

b) [1.5 pt] Donner la complexité de `hauteur(A)` en fonction du nombre n de tâches et du nombre m d'arêtes.

c) [1.5 pt] Indiquer comment modifier `hauteur(A)` de sorte à obtenir une fonction qui renvoie un ordonnancement des tâches sous la forme d'une liste $\sigma = [\sigma(0), ..., \sigma(n-1)]$ indiquant un ordre possible dans lequel les tâches peuvent être exécutées.

3) Algorithme de Kahn

Pour améliorer la complexité de la fonction définie au 2), on considère l'algorithme suivant :

Étape 1. Initialisation :

- On construit le tableau d donnant pour chaque sommet i le nombre de **sommets entrants**, c'est-à-dire le nombre de j tels que $j \to i$.

- On construit la liste P (pile) des sommets de degré entrant nul (c'est-à-dire sans prédécesseur)

Étape 2. Tant que la pile P n'est pas vide, on effectue les instructions suivantes :

- on extrait un élément i de la pile et on considère **le graphe obtenu en supprimant ce sommet** (et ses arêtes)
- on met à jour le tableau d et la pile P de sorte à tenir compte de la suppression du sommet i : pour tout successeur j de i, on décrémente $d[j]$ et on ajoute j à la pile si la nouvelle valeur de $d[j]$ vaut 0.

On obtient un ordonnancement en considérant les sommets **dans l'ordre où ils sont dépilés** de P.

a) [3.5 pts] Écrire une fonction `kahn(A)` qui renvoie un ordonnancement selon cet algorithme, c'est-à-dire la liste L des tâches dans l'ordre dans lequel elles peuvent être exécutées.

b) [1.5 pt] Déterminer la complexité de cet algorithme en fonction de n et m.

4) Utilisation d'un parcours en profondeur

On considère la fonction suivante, de complexité linéaire $O(n+m)$:

```
def f(A) :
    n = len(A) ; K = [-1]*n
    def traite(i) :
        if K[i] == -1 :
            m = 0
            for j in A[i] :
                traite(j) ; m = max(m,1+K[j])
            K[i] = m
    for i in range(n) :  traite(i)
    return K
```

a) [2 pts] Décrire le tableau K retourné à la fin (c'est-à-dire donner la signification de $K[i]$ à la sortie).

b) [1.5 pt] Expliquer comment modifier `f(A)` de sorte à obtenir une fonction qui renvoie un ordonnancement des tâches sous la forme d'une liste $\sigma = [\sigma(0), ..., \sigma(n-1)]$ indiquant un ordre possible d'exécution des tâches.

c) [1 pt] Expliquer comment on peut utiliser `f` pour définir très simplement une fonction `g(A)` qui renvoie le tableau des hauteurs H défini à la question 2).

d) [2.5 pts] (★) En utilisant une pile au lieu d'une fonction auxiliaire récursive, écrire une nouvelle version de la fonction `f` qui renvoie le tableau K.