

DM 2 - CORRECTION

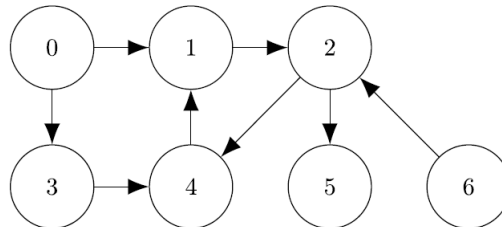
Durée : 1 heure 30 min

Calculatrice interdite

Exercice 0.1. Soit $G := (S, A)$ un graphe où $S = \llbracket 0; n - 1 \rrbracket$.

On appelle liste d'adjacence le tableau g de longueur n tel que pour tout $i \in \llbracket 0; n - 1 \rrbracket$, g_i est la liste des successeurs $j \in \llbracket 0; n - 1 \rrbracket$ de i .

On considère le graphe suivant :



1. Donner la liste d'adjacence correspondante (sous forme de liste de listes avec la syntaxe Python).

Par exemple :

```
>>> liste_adj[0]
[1, 3]
>>> liste_adj[2]
[4, 5]
```

La liste d'adjacence de ce graphe est :

```
· liste_adj=[[1, 3], [2], [4, 5], [4], [1], [], [2]]
```

2. On rédige une fonction **parcourt**(g, s) qui reçoit la liste d'adjacence d'un graphe ainsi que l'indice $s \in \llbracket 0; n - 1 \rrbracket$ d'un sommet et qui renseigne les sommets accessibles depuis le sommet s sous forme d'une liste de booléens.

```
def parcourt(g, s):
    n=len(g)
    visite=[False for i in range(n)]
    pile=[s]
    while len(pile)!=0:
        x=pile.pop()
        if not visite[x]:
            visite[x]=True
            for y in g[x]:
                pile.append(y)
    return(visite)
```

Par exemple :

```
>>> parcourt(liste_adj,0)
[True, True, True, True, True, True, False]
```

- (a) De quel type d'algorithme de parcourt de graphe s'agit-il ? Connaissez-vous une autre méthode de parcourt ? Expliquer la différence entre les deux méthodes.

Il s'agit d'un parcours en profondeur, ce dernier utilise une structure de pile pour lister les sommets rencontrés. C'est le dernier sommet rencontré qui est visité le premier.

Il existe également un algorithme de parcours en largeur pour lequel on utilise une structure de file : c'est le premier sommet rencontré qui est visité le premier.

- (b) Expliquer le fonctionnement de cet algorithme en quelques lignes. On initialise une liste **visite** de booléens ne contenant que des False car au début aucun sommet n'a été visité.

On crée une liste **pile**, gérée comme une structure de pile : on y mettra au sommet le dernier sommet rencontré, lequel sera aussi le premier visité. Elle est initialisée avec le sommet reçu en argument (c'est le premier rencontré!).

Tant que la pile n'est pas vide (il reste des sommets à visiter) :

- on extrait le sommet de la pile
- si ce dernier n'a pas été visité, on le marque comme visité puis on ajoute à la pile tous les sommets vers lesquels il pointe.

On renvoie la liste *visite* telle que $visite[i]=True$ si le sommet i est accessible depuis le sommet s .

3. Rédiger une fonction **sommets(g,s)** qui renvoie la liste des sommets accessibles depuis le sommet s du graphe g (représenté par sa liste d'adjacence).

Par exemple :

```
>>> sommets(liste_adj,6)
[1, 2, 4, 5, 6]
```

```
def sommets(g,s):
    · visite=parcourt(g,s)
    · sommets=[]
    · for i in range(len(visite)):
    ·     if visite[i]:
    ·         sommets.append(i)
    · return(sommets)
```

4. Rédiger une fonction `relies(g,s1,s2)` qui renvoie `True` si le sommet `s2` est accessible depuis `s1` en parcourant le graphe. Par exemple :

```
>>> relies(liste_adj,0,3)
True
>>> relies(liste_adj,0,6)
False

def relies(g,s1,s2):
    listes_sommets=sommets(g,s1)
    for i in listes_sommets:
        if i==s2:
            return(True)
    return(False)
```

5. Un graphe orienté est dit fortement **connexe** s'il existe un chemin orienté depuis tout sommet `s1` vers tout sommet `s2`.

- (a) Le graphe cité en exemple est-il fortement connexe ?
Non car les sommets 0 et 6 ne sont pas reliés par une chaîne.
- (b) Rédiger une fonction `estconnexe(g)` qui renvoie `True` si le graphe `g` représenté par sa liste d'adjacence est connexe, `False` sinon.

```
def estconnexe(g):
    n=len(g)
    for i in range(n):
        for j in range(i+1,n):
            if not relies(g,i,j):
                return(False)
    return(True)
```

Exercice 0.2. On s'intéresse à n articles périssables représentés par une liste $[0, 1, \dots, n - 1]$.

Pour chacun de ces articles, on dispose :

- de son poids
- de son prix en euros
- du nombre de jours avant péremption

Chacune de ses données est représentée sous forme de liste :

```
Articles=[0,1,2]
Poids=[2.3,4,1.5]
Prix=[2.5,5,8]
Jours=[6,5,4]
```

Par exemple, l'article 1 pèse 4 kg, coûte 5 euros et sera périmé dans 5 jours.

1. Écrire une fonction **rentable**(**Poids**, **Prix**) qui reçoit en paramètres :

- une liste **Poids** des poids en kilos de n articles périssables
- une liste **Prix** des prix de ces articles

et renvoie :

- l'indice de l'article le moins cher au kilo.
- le prix au kilo de cet article

```
def rentable(Poids,Prix):
    n=len(Poids)
    rapport=10**6
    article=0
    for i in range(1,n):
        if Poids[i]!=0:
            r=Prix[i]/Poids[i]
            if r<rapport:
                article=i
                rapport=r
    return(article,rapport)
```

2. On rappelle le tri par insertion pour trier dans l'ordre croissant une liste T :

```
def tri_insertion(T):
    for i in range(1,len(T)):
        j = i
        x=T[i]
        while j>0 and T[j-1]>x:
            T[j] = T[j-1]
            j = j-1
        T[j] = x
    return T
```

Écrire une fonction **tri_articles**(**Poids**) qui renvoie la liste des poids triés dans l'ordre croissant avec pour chaque poids, l'article correspondant.

Par exemple :

```
>>> tri_articles(Poids)
[[2, 1.5], [0, 2.3], [1, 4]]
```

```
def tri_articles(Poids):  
    n=len(Poids)  
    T=[]  
    for i in range(n):  
        T.append([i,Poids[i]])  
    for i in range(1,len(T)):  
        j = i  
        article,x=T[i]  
        while j>0 and T[j-1][1]>x:  
            T[j] = T[j-1]  
            j = j-1  
        T[j] =[article,x]  
    return T
```