```
# Correction_info10

#EX1

#Attention, il ne s'agit pas juste de vérifier s'il existe une arête enre i
et j, mais bien s'il existe un chemin constitué d'une ou plusieurs arêtes
permettant de passer de i à j... c'est pour cela qu'on va parcourir l'arbre
en profondeur.

def accessibilite(L,i,j):
    """accessibilite(L:list,i:int,j:int)->bool"""
    p=len(L)
    assert 0<=i<p and 0<=j<p
    couleur=[0 for k in range(0,p)]
    La=[i]
    while len(La)!=0:
        s=La.pop()
        couleur[s]=2
        if s==j:
            return True
        else:
            for v in L[s]:#on parcourt les voisins du sommet s
                if couleur[v]==0:
                    La.append(v)
                    couleur[v]=1
                else:
                    pass
    return False

def acces(L):
    """acces(L:list)->array"""
    p=len(L)
    G=zeros((p,p)) #on construit un tableau qu'on va compléter avec les
accessibilités entre i et j
    for i in range(0,p):
        for j in range(0,p):
            G[i,j]=accessibilite(L,i,j)
    return G

#Le graphe G étant connexe, quand on applique le programme précédent à la
liste d'adjacence de G, on obtient des 1 (donc True) partout !
#On en déduit alors qu'un graphe est connexe s'il y a des 1 partout et donc,
aucun 0 !

def connexe(L:list):
    """connexe(L:list)->bool"""
    return False not in acces(L)

#EX2
from numpy import *
M=array([[0,3,0,0,1,0],[3,0,2,0,1,0],[0,2,0,1,3,3],[0,0,1,0,5,1],
[1,1,3,5,0,0],[0,0,3,1,0,0]])
#on rappelle que la semaine dernière, on a vu comment obtenir la liste des
voisins à partir de la matrice d'adjacence du graphe. C'est le prorgramme
qui suit :
def matricetoliste(M):
    """matricetoliste(M:array)->list"""
    p,q=shape(M)
    L=[[] for k in range(0,p)]
    for i in range(0,p):
```



```
        for j in range(0,q):
            if M[i,j]!=0: #si le poids est non nul, c'est un voisin
                L[i].append(j) #on le stocke alors dans la liste des voisins
du sommet i
            else:
                pass
    return L

#on parcourt les sommets situés dans L et on compare les distances associées
à chacun de ces sommets.
def sommet(L,D):
    """sommet(L:list,D:list)->int
    """
    i0=L[0]
    dmin=D[i0]
    for x in L:
        if D[x]<dmin:
            i0,dmin=x,D[x]#on stocke le nouveau sommet et dmin
        else:
            pass
    return i0

#on va stocker les prédecesseurs de j dans une liste L en veillant à la
compléter par devant et en convenant que -1 signifie qu'il n'y a en fait
aucun prédecesseur
def chemin(P,j):
    """chemin(P:list,j:int)->list
    """
    n=len(P)
    L=[j]  #on stocke le sommet d'où on part
    k=P[j] #on regarde alors le prédecesseur éventuel de j
    while k!=-1:
        L=[k]+L #attention, on place les prédécesseurs devant !
        k=P[k]
    return L

def dijkstra(M,i,j):
    """dijkstra(M:array,i:int,j:int)->bool,list,int
    """
    (p,q)=shape(M)
    couleur=[0 for k in range(0,p)]
    La=[i]
    L=matricetoliste(M)  #on construit la liste d'adjacence contenant les
sommets voisins
    D=[float('inf') for k in range(0,p)] #la liste des distances cumulées
    D[i]=0
    P=[-1 for k in range(0,p)] #la liste des prédecesseurs
    while len(La)!=0:
        #on extrait le sommet associé à la distance minimale et on le
supprime de la liste à colorier
        s=sommet(La,D)
        del(La[La.index(s)])
        #puis, on le colorie au noir
        couleur[s]=2
        if s==j:
            return True,chemin(P,j),D[s]
        else:
            for v in L[s]:#on parcourt les voisins du sommet s
                if couleur[v]==0:
                    La.insert(0,v) #on insère le sommet v dans la file (à
```

```
l'avant)
                    couleur[v]=1
                    D[v]=D[s]+M[s,v] #on ajuste la nouvelle distance
                    P[v]=s #on stocke le prédecesseur
                elif couleur[v]==1:
                    dnew=D[s]+M[s,v]
                    if dnew<D[v]:
                        D[v]=dnew #on ajuste la nouvelle distance
                        P[v]=s #on stocke le prédecesseur
                    else:
                        pass
    return False
```