```python
import numpy as np
from collections import deque


M1=np.array([[0,0,0,0,0,0,1],[0,0,1,1,1,0,0],[0,1,0,0,1,1,0],[0,1,0,0,1,0,0],
             [0,1,1,1,0,1,0],[0,0,1,0,1,0,0],[1,0,0,0,0,0,0]])

L1=[[6],[2,3,4],[1,4,5],[1,4],[1,2,3,5],[2,4],[0]]

M2=np.array([[0,0,0,0,1],[1,0,1,0,0],[0,1,0,0,1],[0,1,1,0,0],[0,0,0,1,0]])

L2=[[4],[0,2],[1,4],[1,2],[3]]

##Exercice 1: Conversion matrices/listes d'adjacences
def MatriceToListe(M):
    n=len(M)
    L=[]
    for i in range(n):
        liste_voisins=[]
        for j in range(n):
            if M[i,j]==1:
                liste_voisins.append(j)
        L.append(liste_voisins)
    return L

def ListeToMatrice(L):
    n=len(L)
    M=np.zeros((n,n))
    for i in range(n):
        for j in L[i]: #la liste des voisins du sommet i
            M[i,j]=1
    return M

##Exercice 2: degré et voisins d'un sommet
def degré1(M,s):
    '''M est la matrice d'adjacence du graphe non orienté'''
    n=len(M)
    deg=0
    for j in range(n):
        deg+=M[s,j]
    if M[s,s]==1:#cas d'une boucle sur le sommet s
        deg+=1
    return deg


def degré2(L,s):
    '''L est la liste d'ajacence du graphe non orienté'''
    if s in L[s]: #cas d'une boucle sur le sommet s
        return len(L[s])+1
    else:
        return len(L[s])

def voisins1(M,s,t):
    '''M est la matrice d'ajacence du graphe'''
    return M[s,t]==1

def voisins2(L,s,t):
    '''L est la liste d'ajacence du graphe'''
    return t in L[s]

##Exercice 3: parcours en profondeur

def parcours_profondeur(L,d):
    '''L est la liste d'adjacence d'un graphe, d est le noeud de départ'''
    visités=[]
    pile=[d]
    while len(pile)!=0:
        s=pile.pop()
        if s not in visités:
            visités.append(s)
            for x in L[s]: #on parcourt les voisins de s
                if x not in visités:
```

```
                    pile.append(x)
    return visités


#amélioration de la complexité avec le marquage des sommets visités
def parcours_profondeurv2(L,d):
    '''L est la liste d'adjacence d'un graphe, d est le noeud de départ'''
    visités=[]
    pile=[d]
    n=len(L) #le nb de sommets
    marquage=[False for i in range(n)]
    while len(pile)!=0:
        s=pile.pop()
        if marquage[s]==False: #si on n'a pas encore visité ce sommet
            visités.append(s)
            marquage[s]=True
            for x in L[s]: #on parcourt les voisins de s
                if marquage[x]==False:
                    pile.append(x)
    return visités


##Exercice 5: Vérification de la connexité d'un graphe
def connexe(L):
    visités=parcours_profondeur(L,0) #peu importe le sommet de départ
    return len(L)==len(visités)

##Exercice 6: Parcours en largeur
def parcours_largeur(L,d):
    '''L est la liste d'adjacence d'un graphe, d est le noeud de départ'''
    n=len(L)
    visités=[]
    file=deque([d])
    marquage=[False for i in range(n)]

    while len(file)!=0:

        s=file.popleft()
        if marquage[s]==False: #si on n'a pas encore visité ce sommet
            visités.append(s)
            marquage[s]=True
            for x in L[s]: #on parcourt les voisins de s
                if marquage[x]==False:
                    file.append(x)

    return visités



##Exercice 7: Plus courts chemins deouis un sommet dans un graphe non pondéré
def distances(L, d):
    '''L est la liste d'adjacence d'un graphe, d est le nœud de départ'''
    n = len(L)

    # On initialise toutes les distances à l'infini
    dist = [float('inf') for i in range(n)]

    # Le sommet de départ est à une distance 0 de lui-même
    dist[d] = 0

    file = deque([d])

    while len(file) != 0:
        s = file.popleft()

        # On parcourt les voisins de s
        for x in L[s]:
            # Si le sommet n'a pas encore été découvert
            if dist[x] == float('inf'):
                dist[x] = dist[s] + 1
                file.append(x)

```

```python
    return dist


##Exercice 8: Déclaration d'un graphe pondéré
L3=[[(1,10),(2,30)],[(0,10),(2,10),(3,20)],[(0,30),(1,10),(4,40)],
    [(1,20),(4,20),(5,60)],[(2,40),(3,20),(5,20)],[(3,60),(4,20)]]

##Exercice 10: Algorihtme de Dijkstra pour un graphe pondéré

##Version avec marquage des sommets visités,
## et fonction auxiliaire pour la recherche de la distance minimum
def sommet_min(dist,visites):
    '''on cherche l'indice du sommet de distance minimum,
    uniquement parmi ceux qui n'ont pas encore été marqués'''
    n=len(dist)
    i_min=0
    d_min=float('inf')  #la distance minimum est initialisé à l'infini au début
                        #ainsi elle est actualisée dès le premier tour de boucle
    for i in range(n):
        if visites[i]==False:
            if dist[i]<=d_min: #d_min et i_min vont être actualisés dès le 1er tour
                d_min=dist[i]
                i_min=i
    return i_min



def Dijkstra(L,d):
    n=len(L)
    visites=[False for i in range(n)]
    dist=[np.inf for i in range(n)]
    dist[d]=0
    while False in visites:

        u=sommet_min(dist,visites)

        if dist[u]==np.inf: #plus aucun sommet n'est accessible
                            #cas d'un graphe non connexe
            return dist

        visites[u]=True

        for v,poids in L[u]: #les voisins de s avec leur poids
            if visites[v]==False and dist[u]+poids<dist[v]:
                dist[v]=dist[u]+poids
    return dist

##Exercice 10: Algorithme de Dijkstra avec prédecesseur pour un graphe pondéré

def DijkstraPredecesseurs(L,d):
    n=len(L)
    visites=[False for _ in range(n)]
    dist=[np.inf for i in range(n)]
    dist[d]=0
    pred=[None for i in range(n)] #la liste du prédécesseur de chaque sommet
    while False in visites:

        #recherche de la distance minimum parmi les sommets de non marqués
        u=sommet_min(dist,visites)

        if dist[u]==np.inf:
            return dist

        visites[u]=True

        for v,poids in L[u]: #les voisins de s avec leur poids
            if visites[v]==False and dist[u]+poids<dist[v]:
                dist[v]=dist[u]+poids
                pred[v]=u

    return dist,pred

```

```python

##Plus court chemin entre deux sommets

def PlusCourtChemin(G,d,a):

    dist,pred=DijkstraPredecesseur(G,d)

    if dist[a]==np.inf:  #cas où d et a ne sont pas reliés
        return False
    chemin_envers=[a]
    sommet=a
    while sommet!=d:
        sommet=pred[sommet]
        chemin_envers.append(sommet)
    return dist[a],chemin_envers[::-1]

##Exercice 11: Dimamètre d'un graphe

##diamètre d'un graphe non pondéré
def diametre1(G):
    n=len(G)
    return max([max(distance(G,d)) for d in range(n)])


##diamètre d'un graphe pondéré
def diametre2(G):
    n=len(G)
    return max([max(Dijkstra(G,d)) for d in range(n)])



##Exercice 12:(difficile) Version récursive du parcours en profondeur

def prof_rec(L,s,marquage=None): #version récursive du parcours en profondeur

    if marquage==None:  #au premier appel la liste marquage est initialisé
        marquage=[False for i in range(len(L))]

    if len(L[s])==0: #cul de sac
        marquage[s]=True
        return [s]

    else:
      visites=[s]
      marquage[s]=True
      for v in L[s]:
          if not marquage[v]:
              marquage[v]=True
              visites=visites+prof_rec(L,v,marquage) #on transmet le marquage
      return visites


```