Exercice 1

```
# Exercice 1)
Sommets = ['A', 'B', 'C', 'D']

# 1)
L = [4, [[], [0], [0,1,3], [0,1,0,1]]

# 2)
from numpy import array
M = array([[0,0,0,0],
           [1,0,0,0],
           [1,1,0,1],
           [1,1,0,1]])

# 3.a)
def aretes(L, Sommets):
    aretes = [ ]
    n = L[0]
    Ad = L[1]
    for k in range(n):
        for l in Ad[k]:
            aretes.append((Sommets[k], Sommets[l]))
    return aretes

# 3.b)
def aretes(matrice,sommets):
    aretes = []
    n = len(matrice)
    for i in range(n):
        for j in range(n):
            if matrice[i,j]:
                aretes.append((sommets[i],sommets[j]))
    return aretes

# 4.a)
def nonOriente(L):
    n = L[0]
    Ad = L[1]
    for i in range(n):
        for j in Ad[i]:
            if i not in Ad[j]:
                return False
    return True
```

```
# 4.b)
def nonOriente(M):
    n = len(M)
    for i in range(n):
        for j in range(i):
            if M[i,j] != M[j,i]:
                return False
    return True

# 5.a)
def L2M(L):
    n = len(L)
    M = array([[0]*n for k in range(n)])
    Ad = L[1]
    for i in range(n):
        for j in Ad[i]:
            M[i,j] = 1
    return M

# 5.b)
def M2L(M):
    n = len(M)
    L, Ad = [n], []
    for i in range(n):
        V = []
        for j in range(n):
        if M[i,j] == 1:
            V.append(j)
        Ad.append(V)
    L.append(Ad)
    return L
```

Calculer la distance entre deux sommets dans un graphe. On utilise ici la propriété que $A^d_{i,j}$ est le nombre de chemins de longueur d de s_i à s_j, et que s'il existe un chemin de s_i à s_j, alors il en existe un de longueur $\leqslant$ au nombre de sommets.

```
from numpy import dot   # Produit matriciel

def distance(L,sommets,s1,s2):
    A = L2M(L)
    i = sommets.index(s1)
    j = sommets.index(s2)
    if i==j:
        return 0
    B = A[:,:]
    for d in range(len(A)):
```

```
        if B[i,j]:
            return d+1
        B = dot(B,A)
```

```
In [7]: distance(A,S,'D','A')
Out[7]: 1
```

Cet algorithme bien qu'élégant n'est pas efficace. En utilisant un algorithme de multiplication matriciel naïf (de complexité cubique sur la dimension de l'espace), il est en $O(n^4)$ dans le pire des cas, où n est le nombre de sommets ; avec des améliorations on peut atteindre $O(n^3)$ et même théoriquement $O(n^{2.5})$ (Coppersmith et Winograd) ; mais cela reste peu efficace. De plus l'algorithme ne fournit pas de plus court chemin.

Exercice 2.

1. Fonction `adjacent(A,i)` ; `A` est la matrice (numpy) d'adjacence et `i` l'indice d'un sommet.

```
def adjacent(A,i):
    L = []
    for j in range(len(A)):
        if A[i,j]:
            L.append(j)
    return L
```

2. Fonction calculant la composante connexe d'un sommet.

```
def fusionner(L1,L2):
    L = []
    n1, n2 = len(L1), len(L2)
    i1 = i2 = 0
    while i1<n1 and i2<n2:
        if L1[i1] < L2[i2]:
            L.append(L1[i1])
            i1 += 1
        elif L1[i1] == L2[i2]:
            L.append(L1[i1])
            i1 += 1
            i2 += 1
        else:
            L.append(L2[i2])
            i2 += 1
    L += L1[i1:]
    L += L2[i2:]
    return L


def composanteConnexe(A,i):
    L = [i]
    recherche = True
    while recherche:
        n = len(L)
        for s in L:
            L = fusionner(L,adjacent(A,s))
        if n == len(L):
            recherche = False
    return L
```

3. Fonction permettant de déterminer si un graphe non orienté est connexe.

```
def connexe(A):
    return len(A) == len(composanteConnexe(A,A[0,0]))
```