```
# ============================== #
# Algorithme de DIJKSTRA         #
# pour des graphes NON orientés. #
# Version du 26 décembre 2017    #
# ============================== #

## Importations
import numpy as np

## Construction de la matrice des poids
# Fonction fMatW : construction de la matrice des poids d'un graphe NON
ORIENTE
# connexe simple pondéré d'ordre n à partir de la liste (argument L) des
# arêtes pondérées.
# Dans cette version du code, on traite des graphes NON ORIENTEES. La
matrice
# construite est donc symétrique.
def fMatW(n,L):
    A = np.zeros((n,n))
    for k in range(len(L)):
        i, j = L[k][0], L[k][1]
        A[i,j], A[j,i] = L[k][2], L[k][2]
    return A

## Fonction principale
def fDijkstra(A,sd,sa):
    # Le nombre de sommets
    n = A.shape[0]

    # L'"infini"...
    infinity = float('inf')

    # Initialisation de la liste S.
    # Elle contient autant d'éléments qu'il y a de sommets dans le graphe.
    # le deuxième élément, n, ne correspond, volontairement (!), à aucun
numéro
    # de sommet valide.
    S = [[infinity,n,False]]
    S = S*n

    # Le sommet de départ.
    # Remarque : nous faisons ici le choix de ne pas fixer le sommet de
départ
    # avant la boucle principale. Il sera fixé à la première exécution de la
    # boucle principale ci-dessous.
    S[sd] = [0,sd,False]

    # BOUCLE PRINCIPALE #
    # On l'exécute tant que le sommet d'arrivée n'a pas été fixé.
    while not S[sa][2]:
        # On cherche, parmi les sommets NON FIXES de S, le sommet ayant la
distance au
        # sommet de départ la plus petite.
        dmin = infinity
        for i in range(n):
            if not S[i][2] and S[i][0] < dmin:
                imin, dmin = i, S[i][0]
        # On fixe le sommet d'indice imin
        S[imin][2] = True
        # Si le sommet fixé n'est pas le sommet d'arrivée, on met à jour les
```

```
        # sommets NON FIXES adjacents à ce sommet. Sinon... on ne fait rien
!
        if imin != sa:
            for i in range(n):
                if A[imin,i] != 0 and not S[i][2]:
                    New_d = S[imin][0] + A[imin,i]
                    if New_d < S[i][0]:
                        S[i] = [New_d,imin,False]
    # Construction d'une chaîne de caractères correspondant à une chaîne de
    # longueur minimale
    Cmin = str(sa)
    s = sa
    while s != sd:
        s = S[s][1]
        Cmin = str(s) + ' --> ' + Cmin
    # Renvoi de la chaîne de caractères et de la longueur minimale...
    return(Cmin,S[sa][0])

## Pour tester...
# DONNEES
# =======

# Description du graphe pondéré (arêtes et poids)

# Première situation vue en cours
#L = [ [3,0,3] , [3,1,12] , [0,5,35] , [0,2,5] , [1,2,9] , [1,4,15] ,
[2,5,8] , [2,4,10] , [5,6,13] , [4,6,14] ]
#n = 7

# Deuxième situation vue en cours
L = [ [0,1,3] , [0,3,5] , [0,5,21] , [1,3,4] , [1,2,5] , [2,4,2] , [2,6,1] ,
[3,5,15] , [3,4,4], [4,5,7] , [4,7,1] , [6,7,6] , [6,8,5] , [5,8,3] ,
[7,8,2] ]
n = 9

# Construction de la matrice des poids et affichage pour "visualiser" le
graphe.

# Le troisième argument est positionné à :
#    --> True si le graphe est NON orienté
#    --> False si le graphe est orienté
A = fMatW(n,L)
print(A)

# Saisie du sommet de départ et du sommet d'arrivée.
sd = int(input('Veuillez saisir le numéro du sommet de départ : '))
sa = int(input('Veuillez saisir le numéro du sommet de d\'arrivée : '))

# RECHERCHE D'UN PLUS COURT CHEMIN
# ================================
# Appel de la fonction fDijkstra
(C,L) = fDijkstra(A,sd,sa)

# Affichage des résultats
print('\nUne chaîne de longueur minimale : ')
print(C)
print('La longueur totale vaut : ',L)
```