TP 4

PROGRAMMATION DYNAMIQUE

Exercice 1

```
## Exercice 1

def chaine(a, t, e, x):
    """ entrées : a liste de deux listes d'entiers
                  t liste de deux listes d'entiers (un de moins que pour a)
                  e, x listes de deux entiers
        sortie : délai optimal d'après le protocole de la chaîne de montage
    """
    n = len(a[0])
    f = [[0 for k in range(n)] for i in range(2)]
    for i in range(2):
        f[i][0] = e[i] + a[i][0]
    for j in range(1, n):
        f[0][j] = min( f[0][j-1] + a[0][j], f[1][j-1] + t[1][j-1] + a[0][j])
        f[1][j] = min( f[1][j-1] + a[1][j], f[0][j-1] + t[0][j-1] + a[1][j])
    return min(f[0][n-1] + x[0], f[1][n-1] + x[1])


# exemple
ex_e = [3, 5]
ex_a = [[6, 8, 2, 4, 9], [8, 4, 7, 7, 3]]
ex_t = [[2, 1, 5, 3], [3, 2, 1, 2]]
ex_x = [2, 3]

assert chaine(ex_a, ex_t, ex_e, ex_x) == 32
```

Exercice 2

1. Si l'on part de la ligne $n-1$ alors on se contente de lire l'entier dans `t` :

$$\forall j \in [\![0, n-1]\!],\ S_{n-1,j} = t[n-1][j].$$

Considérons deux entiers i et j avec $0 \leqslant j \leqslant i \leqslant n-2$.
Si l'on part de la case à la i-ème ligne et j-ème colonne alors il y a déjà la valeur de $t[i][j]$ plus la plus grande des deux valeurs correspondant à un départ sur la case en dessous à gauche (ligne $i+1$, colonne j) ou à un départ sur la case en dessous à droite (ligne $i+1$, colonne $j+1$) donc :

$$S_{i,j} = t[i][j] + \max\left(S_{i+1,j}, S_{i+1,j+1}\right).$$

2.

```
def somme(t):
    n = len(t)
    S = [[0 for j in range(n)] for i in range(n)]
    for j in range(n):
        S[n-1][j] = t[n-1][j]
    i = n-2
    while i >= 0:
        for j in range(i+1):
            S[i][j] = t[i][j] + max(S[i+1][j], S[i+1][j+1])
        i -= 1
    return S[0][0]
```

3. Une première version en déterminant d'abord le tableau S puis le chemin optimal :

```
def chemin(t):
    n = len(t)
    S = [[0 for j in range(n)] for i in range(n)]
    for j in range(n):
        S[n-1][j] = t[n-1][j]
    i = n-2
    while i >= 0:
        for j in range(i+1):
            S[i][j] = t[i][j] + max(S[i+1][j], S[i+1][j+1])
        i -= 1
    C = [0]
    j = 0
    for i in range(n-1):
        if S[i+1][j] < S[i+1][j+1]:
            j += 1
        C.append(j)
    return S[0][0], C
```

Une autre version en calculant, en même temps que S, un tableau des chemins optimaux en partant de chaque emplacement.

```
def chemin(t):
    n = len(t)
    S = [[0 for j in range(n)] for i in range(n)]   # tableau de valeurs de S
    C = [[[] for j in range(n)] for i in range(n)]  # tableau des chemins correspondants
    for j in range(n):          # initialisation de S et C
        S[n-1][j] = t[n-1][j]
        C[n-1][j] = [j]
    i = n-2
    while i >= 0:               # on "remonte" les lignes
        for j in range(i+1):
            a = S[i+1][j] + t[i][j]
            b = S[i+1][j+1] + t[i][j]
            if a > b:
                S[i][j] = a       # si S[i+1][j] > S[i+1][j+1] alors...
                ch = [c for c in C[i+1][j]]   # on "récupère" le chemin partant de (i+1,j)
            else:
                S[i][j] = b       # sinon...
                ch = [c for c in C[i+1][j+1]] # on "récupère" le chemin partant de (i+1,j+1)
            ch.append(j)          # et on ajoute j à la fin
            C[i][j] = ch          # puis on met à jour le tableau C
        i -= 1                    # on remonte d'une ligne
    return S[0][0], list(reversed(C[0][0]))   # on "retourne" le chemin pour le donner de
        haut en bas
```

Exercice 3

```
#1
""" W^0 = D
    W^(k+1)[ni+j] = min { W^k(ni+j) , W^k(ni+k) + W^k(nk+j) }
"""


#2
from math import sqrt

def FW(tab):
    n = int(sqrt(len(tab)))  # tab est de taille n**2
    W = [[0 for _ in range(n**2)] for _ in range(n+1)]  # tableau de n+1 lignes et n**2
        colonnes
    for r in range(n**2):    # on initialise en remplissant la première ligne avec tab
        W[0][r] = tab[r]
    for k in range(n):       # on remplit les lignes en descendant (indice k)
        for i in range(n):
            for j in range(n):
                W[k+1][n*i+j] = min(W[k][n*i+j] , W[k][n*i+k] + W[k][n*k+j])
    return W[n]             # on renvoit la dernière ligne


# exemple
ex_D = [0, 80, 130, 500, 18, 0, 25, 500, 20, 15, 0, 80, 15, 40, 75, 0]
assert FW(ex_D) == [0, 80, 105, 185, 18, 0, 25, 105, 20, 15, 0, 80, 15, 40, 65, 0]


#3
def CircuitOpt(tab):
    n = int(sqrt(len(tab)))  # tab est de taille n**2
    W = [[0 for _ in range(n**2)] for _ in range(n+1)]  # tableau de n+1 lignes et n**2
        colonnes
    C = []
    for r in range(n**2):
        W[0][r] = tab[r]
    for i in range(n):
        for j in range(n):
            C.append([i, j])
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if W[k][n*i+j] <= W[k][n*i+k] + W[k][n*k+j]:
                    W[k+1][n*i+j] = W[k][n*i+j]
                else:
                    W[k+1][n*i+j] = W[k][n*i+k] + W[k][n*k+j]
                    C[n*i+j] = C[n*i+k][:-1] + C[n*k+j]  # concaténation des chemins
    return W[n], C


#4
''' Considérons un graphe orienté pondéré:
        - les sommets sont les villes
        - l'arc de i vers j est le prix du vol de la ville i à la ville j
            (on peut aussi utiliser +infini s'il n'y a pas de vol)
    L'algorithme donne, pour tout couple (i,j), le chemin de longueur
        minimale (au sens du points minimal) entre le sommet i et le sommet j.
    La complexité est en O(n^3) où n est le nombre de sommets.

    L'algorithme de Dijkstra vu en première année prend en argument un sommet i
        et donne le chemin de longueur minimale vers chacun des autres sommmets.
        La complexité est en O((n+m)log(n)) où n est le nombre de sommets
        et m est le nombre d'arcs. Donc c'est en O(n^2 log(n)).
        Si l'on veut un résultat comparable, pour chaque i, on a donc
        une complexité en O(n^3 log(n)).
'''
```