Exercice 1 – programmation dynamique (fin du TP 4)

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

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

2. 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 :

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

3.
```
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]
```

4.
```
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 2 – complexité, diviser pour régner et utilisation de numpy

1. La formule donnant le coefficient (i, j) du produit AB de deux matrices A et B de $\mathscr{M}_n(\mathbb{R})$ est :

$$(\mathrm{AB})[i,j] = \sum_{k=1}^{n} \mathrm{A}[i,k]\mathrm{B}[k,i]$$

ce qui nécessite pour le calcul n produits et n additions (ou $n-1$ selon comment on initialise le calcul de la somme). Cela étant nécessaire pour chacun des n^2 coefficients, il faut donc n^3 multiplications et n^3 additions.

2. Dans tous les cas, il y a $2n^2$ coefficients à «consulter» (ceux des matrices A et B) et auxquels il faut bien faire subir au moins une opération. On peut aussi remarquer qu'il y a n^2 coefficients à calculer avec au moins une opération pour chacun.

Il est donc impossible d'effectuer le produit de deux matrices de $\mathscr{M}_n(\mathbb{R})$ avec une complexité de la forme $\mathrm{O}(n^\alpha)$ avec $\alpha < 2$.

3. Cette question repose sur du slicing à partir d'un tableau numpy.

```
def blocs(M):
    """ entrée : un tableau np.array M représentant un matrice carrée
                 de taille une puissance de 2
        sortie : tuple de 4 np.array de même taille correspond à la décomposition
                 de M en 4 blocs
    """
    n, _ = M.shape
    k = n//2
    return M[:k,:k], M[:k, k:], M[k:, :k], M[k:, k:]
```

4. Voici plusieurs versions avec plus ou moins de fonctions «toutes faites» :

```
def assemble(A, B, C, D):
    """ entrée : tuple (A, B, C, D) de 4 np.array représentant des matrices carrées
                    de même taille (une puissance de 2)
        sortie : np.array obtenu par blocs A B
                                         C D
    """
    return np.block([[A, B], [C, D]])
```

```
def assemble_bis(A, B, C, D):
    return np.concatenate((np.concatenate((A, B),axis=1), np.concatenate((C, D),axis=1)),
        axis=0)
```

```
def assemble_ter(A, B, C, D):
    n, _ = A.shape
    M = np.zeros((2*n, 2*n))
    for i in range(n):
        for j in range(n):
            M[i, j] = A[i, j]
            M[i+n, j] = C[i, j]
            M[i, j+n] = B[i, j]
            M[i+n, j+n] = D[i, j]
    return M
```

5.

```
def produit(M, N):
    """ entrée : deux tableaux np.array M et N représentant
                 des matrices carrées de même taille
                 (une puissance de 2)
        sortie : produit MN calculé récursivement
                 par l'algorithme de Strassen
    """
    n, _ = M.shape
    if n == 1:
        return np.array([M[0,0]*N[0,0]])
    k = n//2
    A, B, C, D = blocs(M)
    E, F, G, H = blocs(N)
    X1 = produit(B-D, G+H)
    X2 = produit(A+D, E+H)
    X3 = produit(A-C, E+F)
    X4 = produit(A+B, H)
    X5 = produit(A, F-H)
    X6 = produit(D, G-E)
    X7 = produit(C+D, E)
    Y1 = X1 + X2 - X4 + X6
    Y2 = X4 + X5
    Y3 = X6 + X7
    Y4 = X2 - X3 + X5 - X7
    return assemble(Y1, Y2, Y3, Y4)
```

6. Notons u_n le nombre de multiplications pour des matrices de taille 2^n.

Pour $N = 2^0$, il y a une multiplication : $u_0 = 1$.

De façon générale dans le cas d'une matrice de taille $N = 2^n$, pour le calcul des blocs X_1, X_2, etc. il y a 7 produits de blocs de taille $N/2$ *i.e.* 2^{n-1} donc : $u_n = 7u_{n-1}$.

La suite u étant géométrique de premier terme égal à 1, on a pour tout $n \in \mathbb{N}$, $u_n = 7^n$.

Enfin, puisque $N = 2^n$, on a $n = \log_2(N)$ donc le nombre de multiplications cherché est :

$$7^{\log_2(N)} = \exp\left(\frac{\ln(N)}{\ln(2)}\ln(7)\right) = N^{\log_2(7)}.$$

7. Quand on calcule un produit de deux matrices de taille N, il y a le calcul des produits de façon récursive et il y a des sommes et différences de matrices de taille $N/2$.

Une telle somme ou différence nécessite $(N/2)^2$ opérations (correspondant au nombre de coefficients à calculer). Le nombre total d'additions et soustractions est donc un $O(N^2)$ ce qui est négligeable devant le nombre de multiplications (puisque $\log_2(7) \approx 2,8$).

On a donc :

$$C(N) = O\left(N^{\log_2(7)}\right).$$