ITC – cours n°6

Tableaux à 2D

I. Problématique, principe de base

1.1. Rappels : tableaux 1D

L'informatique étant la science/technologie du traitement de l'information (en grandes quantités si possible), on stocke ces informations sous une forme exploitable par la machine. Nous avons déjà rencontré le stockage dans des tableaux à une dimension, que nous avons représenté en mémoire par des listes Python :

```
notes_Max = [11, 13, 8, 14]  # contient les notes de Max
```

Nous avons également vu comment faire des calculs sur un tel tableau : moyenne, max et min, écart-type etc...

1.2. Le tableau 2D

Il est souvent commode de stocker des informations dans un tableau à 2D :

- les tableaux à deux entrées sont classiques, par exemple pour les notes de toute une classe

Étudiant n° \ devoir n°	0	1	2	3	...
0	11	8	16	14	
1	5	4	8	9	
2	10	10	13	15	
...					

- en mathématiques, les matrices sont des objets paticulièrement importants :

$$M = \begin{pmatrix} 4 & 5 & -1 & 0 \\ -5 & -2 & 7 & 5 \\ 2 & 2 & -2 & 3 \end{pmatrix}$$

- une image bitmap est, informatiquement, un tableau 2D de pixels, chaque pixel contenant une ou plusieurs information (niveaux de gris, couleurs, transparance...).

En Python, on pourra représenter de telles informations comme une liste dont les cases contiennent des listes :

```
notes = [           \
  [11, 8, 16, 14],  \
  [5, 4, 8, 9],     \
  [10, 10, 13, 15]  \
]
```

```
M = [             \
  [4, 5, -1, 0],  \
  [-5, -2, 7, 5], \
  [2, 2, -2, 3],  \
]
```

Si on détaille l'exemple du tableau de notes :

- `notes[i]` est une liste contenant les notes de l'étudiant n°i ;
- le nombre n_e d'étudiants est donc `len(notes)` et `notes[i]` est valide si $i \in [\![0, n_e - 1]\!]$;
- pour un étudiant i **fixé**, `len(notes[i])` est le nombre n_d de devoirs[1] et `notes[i][j]` correspond à la j^{e} note pour le·a i^{e} étudiant·e.

On peut donc proposer, par exemple, un algorithme pour calculer la moyenne de l'étudiant i ou de la classe sur le devoir j :

```
def moyenne_ét(notes: list, i: int) -> float:
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........

def moyenne_devoirs(notes: list, j: int) -> float:
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
```

[1] Notons que dans un vrai tableau à deux entrées ou une matrice, il y a nécessité que le nombre de colonnes soit le même pour toutes les lignes ; mais les listes Python permettent plus de souplesse et on peut avoir une situation dans laquelle `len(notes[i1])` et `len(notes[i2])` ne sont pas égaux. Cela ne représente plus un tableau à 2D en toute rigueur, mais est possible.

Notons que ces algorithmes sont respectivement en $\mathcal{O}(n_e)$ pour le second et $\mathcal{O}(n_d)$ pour le premier. Les deux paramètres n_e et n_d sont ici pertinents pour caractériser la « taille » des données en entrée, et donc les deux peuvent intervenir dans la complexité.

II. La problématique des copies de listes de listes

Nous avons déjà vu que copier une liste ne peut pas se faire simplement avec un opérateur d'affectation, car les listes sont mutables :

```
def copie_ratée(li):
    cop = li  # créé une nouvelle étiquette sur la même liste
    return cop

def copie_ok(li):
    cop = []      # création d'une nouvelle liste
    for x in li:  # x est une étiquette vers un élément de li
        cop.append(x)
        # cop[i] contient une étiquette vers le même élément x
    return cop
--------------------------------------------------------------------------
In [1]: a = [1, 5, 7, -5, 1, -3]
In [2]: b = copie_ratée(a)  # b est la même liste que a
In [3]: c = copie_ok(a)     # c est une liste différente de a
```

La fonction `copie_ok` va bien créer une **nouvelle liste**, et la remplir en suivant les étiquettes `li[i]` . Si celles-ci contiennent des nombres, comme dans l'exemple proposé, il n'y a aucun problème.

En revanche, supposons l'application suivante : en suppoosant la matrice `M` précédente définie, on utilise `copie_ok` et on raisonne sur la suite des instructions :

```
In [1]: M2 = copie_ok(M)
```

- à l'entrée dans la fonction, `li` est une étiquette locale qui pointe vers la même liste que `M` ;
- lors du premier tour de boucle, `x` est une étiquette vers première ligne de la matrice, qui est une liste, donc mutable ;
- après l'appel à append, la case `cop[0]` contient donc une étiquette vers la même liste que `li[0]`, et donc `M[0]` ;
- et ainsi de suite pour les autres éléments.

donc finalement, la copie de `M` est superficielle : certes `M2` n'est pas `M`, mais **leurs éléments** sont les mêmes :

```
In [2]: M is M2
Out[1]: False
In [3]: M[0] is M2[0]
Out[2]: True
```

et toute modification de l'une est une modification de l'autre :

```
In [4]: M[0], M2[0]
Out[3]: [4, 5, -1, 0], [4, 5, -1, 0]
In [5]: M[0][0] = 0
In [6]: M[0], M2[0]
Out[4]: [0, 5, -1, 0], [0, 5, -1, 0]
```

Une fonction de copie qui donne des matrices réellement indépendantes doit donc répéter pour **chaque ligne** l'opération de copie élément par élément

```
def copie_matrice(mat):
    copie = []
    for ligne in mat:  # ligne est une liste de nombres
        copie_ligne = []
        for x in ligne:  # x est un nombre de la matrice
            copie_ligne.append(x)
        copie.append(copie_ligne)
    return copie
# autre proposition
def copie_matrice(mat):
    copie = [copie_ok(ligne) for ligne in mat]
    return copie
```

III. Généralisation

On peut généraliser le principe à un tableau de dimension N, si cela se prête à la forme des données.

Par exemple, le cas des notes en début de chapitre peut se complexifier : si on veut stocker les notes de chaque étudiant·e, à chaque semestre, dans chaque matière, à chaque devoir, on peut le faire dans un tableau de dimension 4 :

- `notes[i]` contient toutes les notes du i^{e} individu ;
- `notes[i][j]` contient toutes les notes du i^{e} individu au semestre j ;
- `notes[i][j][k]` contient toutes les notes du i^{e} individu au semestre j dans la matière k ;
- `notes[i][j][k][l]` contient toutes les notes du i^{e} individu au semestre j dans la matière k au devoir n°ℓ : ceci est enfin un nombre.

On peut également étudier une **suite de matrices**

$$M_n = \begin{pmatrix} M_n^{0,0} & M_n^{0,1} & \dots \\ M_n^{1,0} & M_n^{1,1} & \dots \\ \dots & \dots & \dots \end{pmatrix}$$

que l'on pourra alors stocker dans un tableau à 3 dimensions : chaque `M[n]` est la matrice au rang n de la suite, donc tableau à deux dimensions.

Dernier exemple, une image en couleur peut se représenter comme un tableau à 3 dimensions `im` :

- le premier indice correspond à la ligne du pixel considéré : `im[i]` est une liste ;
- le deuxième indice correspond à la colonne du pixel : `im[i][j]` est encore une liste, qui contient des informations sur les couleurs du pixel en ligne i, colonne j ;
- le troisième indice correspond à la couleur représentée parmi rouge, vert, bleu.

Par exemple, si `im[24][58][1]` vaut 142, cela signifie que le pixel de la 24^{e} ligne, 58^{e} colonne contient une quantité 142/255 de bleu.

On peut encore transformer cela en l'organisation d'une vidéo : il s'agit d'une ensemble d'images, donc d'un ensemble de tableaux de dimension 3, donc d'un tableau de dimension 4 : `vid[51]` est une image représentant la 51^{e} frame. Et on peut encore ajouter le son...

Il est clair que pour un tableau à N dimension, les difficultés soulevées précedemment sur les copies se retrouvent à chaque nouvelle dimension. Nous verrons dans quelques temps une manière élégante et efficace de gérer les copies de listes pour n'importe quelle structure.

ITC – cours n°6

Exercices

1 – Écrire une liste de listes (tableaux 2D) représentant les scores de n joueur·se·s sur n_c compétitions de Tetris : `score[i][j]` représentera donc le score du/de la candidat·e i sur la compétition j.

Dans toute la suite, la liste de scores proposée servira pour les tests.

2 – Écrire une fonction qui prend en paramètre le tableau des scores et renvoie une liste de taille n contenant le meilleur score pour chaque participant·e.

3 – Écrire une fonction qui prend en paramètre le tableau des scores et renvoie une liste de taille n contenant le score moyen pour chaque participant·e.

4 – Écrire une fonction qui renvoie une liste de taille n_c contenant le numéro de la personne ayant le meilleur score à chaque compétition.