

À faire individuellement ou par petits groupes de deux ou trois.

Exercice 1. Matrices: représentation et manipulations

Dans cet exercice, nous allons représenter des matrices par des `list[list[float]]` et les manipuler. L'idée de base est de se dire qu'une matrice telle que

$$M_1 = \begin{bmatrix} 1 & 2 & 3 \\ 9 & 8 & 7 \end{bmatrix}$$

à $m = 2$ lignes et $n = 3$ colonnes peut être représentée en Python par une liste de 2 éléments, chacun de ces éléments représentant une ligne de la matrice et étant lui-même une liste de 3 éléments:

```

1 # ceci est une définition de type: on dit que chaque fois qu'on
2 # écrit Matrix comme type, ça veut dire list[list[float]]. On ne
3 # fait ceci qu'une seule fois pour tout le programme. Notez que
4 # ceci ne définit pas une matrice en elle-même, mais cela dit
5 # comment une matrice est en fait représentée par des types Python
6 Matrix = list[list[float]]
7
8 # on définit une matrice
9 m1: Matrix = [[1, 2, 3], [9, 8, 7]]

```

Une liste de liste a ceci de particulier qu'on va utiliser, pour faire référence à une valeur de la matrice à une double indexation. Si on écrit `m1[0]`, on fait référence à la sous-liste `[1, 2, 3]` — qu'on peut à nouveau indexer. Pour obtenir la valeur 2 par exemple, on écrira `m1[0][1]`, ce qui pourrait se lire ainsi: «regarde la variable `m1` et va à sa case numéro 0, où l'on trouve une sous-liste, dont on va ensuite à la case numéro 1».

De la même manière, s'il s'agissait de replacer le 7 par un 14, on écrirait `m1[1][2] = 14`, ce qui se lirait par «dans la liste `m1`, prends la sous-liste qui est à la case 1, et dans cette sous-liste, va à la case 2 (qui contient actuellement 7) et écris la nouvelle valeur 14».

(a) Définissez `m1` comme ci-dessus. Définissez ensuite, de manière similaire à la dernière ligne ci-dessous, les matrices suivantes:

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, M_3 = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix}, M_4 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 5 & 7 \end{bmatrix}, M_5 = [1 \ 2 \ 3], M_6 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

(b) Écrivez une fonction `print_matrix()` qui prend une matrice en paramètre et qui l'affiche sur le terminal formatée un peu plus joliment que ce que `print()` ferait. Par exemple, `print_matrix(m1)` doit afficher ceci:

```

[
  1 2 3
  9 8 7
]

```

Pour faire ceci:

- Affichez sur une ligne le crochet ouvrant
- Utilisez une boucle pour itérer à travers les lignes de la matrice. À l'intérieur:
 - Utilisez une boucle pour itérer à travers les colonnes de la matrice pour la ligne en cours et affichez chaque composant.

Indice: la fonction `print()` a un paramètre optionnel appelé `end`. Par défaut, sa valeur est le caractère de nouvelle ligne: c'est pour ça que, normalement, `print()` affiche un retour à la ligne après avoir affiché ses arguments. Cependant, il est possible de changer sa valeur pour ne pas provoquer de retour à la ligne automatique à chaque appel de `print()` ainsi: `print(ma_variable, end='')`

- À la fin des deux boucles, affichez le crochet fermant

Créez une liste avec toutes les matrices de **m1** à **m6** et affichez-les avec cette fonction à l'aide d'une boucle **for-in**.

- (c) Écrivez une fonction **dim_m()** qui prend une matrice en paramètre et qui retourne sa dimension m , son nombre de lignes.
- (d) Écrivez une fonction **dim_n()** qui prend une matrice en paramètre et qui retourne sa dimension n , son nombre de colonnes. Attention aux matrices potentiellement vides. S'il se trouve que vous trouvez plusieurs lignes qui n'ont pas le même nombre d'éléments, retournez **0**.
- (e) En utilisant **dim_m()** et **dim_n()**, affichez la dimension de toutes vos matrices.
- (f) Écrivez une fonction **empty()** qui renvoie la matrice zéro de taille **m** par **n**, où **m** et **n** sont des paramètres de la fonction. Attention à bien recréer une nouvelle liste de zéros pour chaque ligne de la matrice et de ne pas réutiliser la même. Au besoin, utilisez **my_list.copy()** pour créer une copie indépendante de la liste **my_list**.
- (g) Écrivez une fonction **identity()** qui renvoie la matrice identité de taille **n** par **n**, où **n** est un paramètre de la fonction. Vérifiez que **identity(3) == m2**.
- (h) Écrivez une fonction **mult()** qui prend deux matrices en paramètres et qui renvoie une nouvelle matrice obtenue par la multiplication des deux matrices d'entrée, ou **[]** si la multiplication n'est pas possible.

Rappel: la multiplication de deux matrices A, B de taille $m_1 \times n_1$ et $m_2 \times n_2$, respectivement, est possible si et seulement si $n_1 = m_2$. Le résultat est une matrice C de taille $m_1 \times n_2$, dans laquelle on a pour chaque élément:

$$c_{ij} = \sum_{k=1}^{n_1} a_{ik} b_{kj}.$$

Testez votre fonction **mult()** avec quelques exemples.

Exercice 2. Recherche dichotomique

Ici, nous profitons des propriétés d'une liste triées pour chercher efficacement si elle contient un élément donné et, si c'est le cas, à quelle position.

La recherche dichotomique (*binary search* en anglais) fonctionne avec une liste triée (ici, dans l'ordre croissant) et un élément à rechercher, disons e . On regarde d'abord l'élément du milieu de la liste, disons m : s'il a la même valeur que e , on a terminé la recherche (avec, admettons-le, pas mal de chance), et on a trouvé la position de e dans la liste. Sinon, il y a deux cas de figure: soit $e < m$, et alors on continue à chercher e , mais dans la moitié inférieure de la liste seulement — donc depuis le début jusqu'à la position qui précède m . Sinon, si $e > m$, on continue aussi à chercher e , mais cette fois dans l'autre moitié de la liste: celle qui commence à l'élément qui suit m et jusqu'à la fin de la liste.

La recherche dans la moitié inférieure ou supérieure de la liste se déroule de la même façon: on prend l'élément qui se trouve au milieu de la moitié de la liste dans laquelle on recherche, et soit on trouve e , soit on procède comme ci-dessus, et on continue de chercher dans un quart de la liste initiale.

On finit forcément soit par trouver e quelque part, soit par se retrouver à chercher e dans une zone de la liste qui ne contient plus d'éléments, et alors on sait que la liste ne contient pas e . Ceci, avec ces critères précis, fonctionne uniquement, rappelons-le, si la liste d'entrée est triée dans l'ordre croissant.

L'exercice consiste à compléter la fonction **binary_search()** dans le code ci-dessous pour implémenter une recherche dichotomique selon cette description. Cette fonction doit retourner l'index de **item** si **item** est un élément de **values**, ou **-1** si ce n'est pas le cas.

Le code de départ est sur la page suivante.

```
1  # on définit une liste de nombre et on la trie
2  numbers = sorted([1, 5, 6, 2, 8, 12, 5])
3  print(numbers)
4
5  def binary_search(values: list[int], item: int) -> int:
6      ... # à compléter
7
8  # pour chaque nombre dans la liste, la fonction de recherche doit le retrouver
9  for v in numbers:
10     print(binary_search(numbers, v))
11
12 # pour des éléments non existants, la fonction de recherche doit retourner -1
13 print(binary_search(numbers, 3))
14 print(binary_search(numbers, 7))
15 print(binary_search(numbers, 42))
```