Algorithmique avancée - activité 1

1/15/23

Table of contents

Correction détaillée de l'activité 1 d'algorithmique avancée.

1. Nombres aléatoires

On veut écrire une fonction :

$$\texttt{generation} : \begin{array}{l} (\texttt{int} > 0)^3 \to \texttt{list[int} > 0\texttt{]} \\ (\texttt{nb\_val, nb\_min, nb\_max}) \mapsto [\texttt{x} | \texttt{x} \in [\![\texttt{nb\_min}, \texttt{nb\_max}]\!]] \end{array}$$

Qui génère `nb_val` nombres aléatoires entiers entre `nb_min` et `nb_max`

Générer un nombre entier aléatoire

Avec `randon.random`, qui renvoie un réel aléatoire dans $[0; 1[$

```
from random import random
from math import floor

def random_gen_with_random (nb_min: int, nb_max: int) -> int:
    return floor(random() * (nb_max - nb_min) + nb_min) # floor -> arrondi inferieur (int

print(random_gen_with_random(5, 10))
```

```
5
```

Autre version avec la fonction `randint`

```
from random import randint

def random_gen (nb_min: int, nb_max: int) -> int:
    return randint(nb_min, nb_max)

# Note: cette fonction est un peu inutile, car elle est exactement similaire a
# randint.

print(random_gen(8, 18))
```

```
13
```

Générer une liste de nombres aléatoires

On utilise la fonction `random_gen` que l'on a définie plus haut.

Première version, avec une boucle :

```python
def generation(nb_val: int, nb_min: int, nb_max: int) -> list[int]:
    """Renvoie une liste de *nb_val* nombres entiers aleatoires entre nb_min,
    """
    new_list = []
    for i in range(nb_val):
        new_list.append(random_gen(nb_min, nb_max))
    return new_list

print(generation(10, 5, 10))
```

```
[8, 9, 9, 9, 6, 8, 8, 10, 6, 10]
```

Autre version, avec une *list comprehension* :

```python
def generation_comprehension(nb_val: int, nb_min: int, nb_max: int) -> list[int]:
    return [random_gen(nb_min, nb_max) for i in range(nb_val)]

print(generation_comprehension(10, 5, 10))
```

```
[9, 7, 6, 5, 6, 10, 10, 8, 8, 5]
```

2. Vitesse d'exécution

On veut mesurer (et comparer) le temps d'exécution des fonctions de génération de nombres aléatoires créées précédemment.

On veut en fait comparer les temps de génération pour des listes contenant entre 10 et 1000 éléments (avec plusieurs valeurs intermédiaires).

Pour cela, on utilise la fonction `time.time()`, du module `time`

> **Fonctionnement de la fonction `time`**
>
> La fonction `time` renvoie le nombre de secondes depuis le début de "l'époque" (Epoch en anglais), c'est-à-dire depuis la "date initiale" définie par votre système d'exploitation. Sur les systèmes UNIX et leurs dérivés, cette date est généralement fixée au 1er janvier 1970.
> Ce qui est à comprendre, c'est que c'est un nombre qui augmente de 1 chaque seconde (les chiffres après la virgule augmentent continuellement pour avoir une mesure plus précise). Donc, pour mesurer la durée d'exécution d'une fonction, il suffit de mémoriser dans une

variable le résultat de `time` avant l'exécution, puis celui après, et de faire la différence entre ces deux nombres. On obtient ainsi la durée de l'exécution en secondes.

Temps d'exécution de la première méthode de génération

Voici donc le code :

```python
from time import time

# liste des nombres d'éléments dans la liste que l'on veut tester
# on peut aussi utiliser range(10, 10000, 10) par exemple
LIST_NUMBER_OF_ELEMENTS = [10, 100, 1000, 10000]

for number_of_elements in LIST_NUMBER_OF_ELEMENTS:
    # on stocke le moment de début de la génération
    start = time()

    # on génère des nombres aléatoires
    foo = generation(number_of_elements, 42, 73)

    # on stocke le moment de fin de la génération
    end = time()

    # la durée d'exécution est la différence entre le moment de début et de fin
    # Attention : si on inverse end et start, on obtient un nombre négatif
    duration = end - start

    # on arrondi la durée, pour que le tout soit plus lisible
    duration = round(duration, 5)

    # affichage du résultat
    print(f"générer {number_of_elements} à mis {duration} secondes")
```

```
générer 10 à mis 1e-05 secondes
générer 100 à mis 4e-05 secondes
générer 1000 à mis 0.0004 secondes
générer 10000 à mis 0.00416 secondes
```

Temps d'exécution avec des *list comprehension*

On utilise exactement le même code, mais avec la fonction `generation_comprehension` au lieu de `generation` :

```
LIST_NUMBER_OF_ELEMENTS = [10, 100, 1000, 10000]

for number_of_elements in LIST_NUMBER_OF_ELEMENTS:
    start = time()
    foo = generation_comprehension(number_of_elements, 42, 73)
    end = time()

    duration = end - start
    duration = round(duration, 5)
    print(f"générer {number_of_elements} à mis {duration} secondes")
```

```
générer 10 à mis 2e-05 secondes
générer 100 à mis 4e-05 secondes
générer 1000 à mis 0.00039 secondes
générer 10000 à mis 0.00384 secondes
```

On remarque que le code avec des *list comprehension* est effectivement plus rapide.

Pour aller plus loin

> Pour aller plus loin
>
> On peut, par exemple, définir une fonction qui mesure le temps d'exécution d'une autre fonction.
> Pour cela, il faut que cette nouvelle fonction (appelons-la `temps_execution`), prenne en argument la fonction dont on mesure le temps d'exécution.
> On obtient donc quelque chose comme ça :

```python
def temps_execution(fonction_a_tester, number_of_elements: int) -> float:
    # ici, on mesure le temps d'exécution
    start = time()
    foo = fonction_a_tester(number_of_elements, 42, 73)
    end = time()
    # l'idéal est de retourner le temps d'exécution plutôt
    # que de mettre un print à l'intérieur d'une fonction
    # (ce qui est à # éviter en général)
    return end - start

print(temps_execution(generation,               1000))
print(temps_execution(generation_comprehension, 1000))
```

```
0.0004761219024658203
0.0003988742828369140 6
```

3. Inverser une liste

On veut écrire une fonction qui inverse l'ordre des éléments d'une liste

On note que, puisqu'en python, les listes ne sont pas modifiables, on devra nécessairement créer une nouvelle liste.

Une première solution fonctionne

Avec la méthode reverse

```python
def reverse_list (liste: list) -> list:
    # retourne la liste (change le contenu de la variable)
    liste.reverse()
    return liste

print(reverse_list([1, 2, 3, 4, 5]))
```

```
[5, 4, 3, 2, 1]
```

Avec la fonction `reversed`

```
def reverse_list_reversed(liste: list) -> list:
    # on est obligé de mettre la fonction list pour que le
    # résultat soit bien une liste (voir le "pour aller plus
    # loin")
    return list(reversed(liste))
```

Pour aller plus loin - Comprendre la méthode reverse

Si on exécute ce code :

```
l = [1, 2, 3, 5, 8, 13, 21]
r = reversed(l)
print(r)
```

```
<list_reverseiterator object at 0x10e13e4a0>
```

On remarque que `r` n'est pas une liste, mais un itérateur.
Un itérateur est un objet que l'on parcourt (tous les itérateurs peuvent donc être mis dans une boucle for).
Le concept d'itérateur est très utile lorsque l'on crée soi-même un objet qui doit être parcouru, car python permet de créer assez facilement ses propres itérateurs.

Avec des slice

En python, on peut indexer des listes de façon assez riche. Cela s'appelle des *slices* (des parts en anglais, car on prend des "parts" de la liste).

```
def reverse_list_slice (liste: list) -> list:
    # :: car on prends toute la liste
    # -1 car on a un pas de -1 (donc on recule dans la liste)
    return liste[::-1]

print(reverse_list_slice([1, 2, 3, 4, 5]))
```

```
[5, 4, 3, 2, 1]
```

Avec la méthode pop

La méthode `pop` des listes permet de retirer le dernier élément d'une liste. Elle retourne l'élément qu'elle retire, ce qui permet d'utiliser cet élément dans une autre fonction

Si on répète l'opération de mettre le dernier élément de l'ancienne liste à la fin de la nouvelle, retourne bien la liste

```
def reverse_list_pop (liste: list) -> list:
    new_list = []
    for _ in range(len(liste)):
        new_list.append(liste.pop())
    return new_list

print(reverse_list_pop([1, 2, 3, 4, 5]))
```

```
[5, 4, 3, 2, 1]
```

Avec la méthode insert + Pop

La méthode `list.insert` permet d'insérer un élément dans une liste, avant l'élément à l'indice précisé.

Dans ce cas, on insère avant l'indice 0, donc au début de la liste. C'est pourquoi on utilise plus `pop()`, mais `pop(0)`, qui va retirer le premier élément au lieu du dernier.

```
def reverse_list_insert (liste: list) -> list:
    new_list = []
    for _ in range(len(liste)):
        new_list.insert(0, liste.pop(0))
    return new_list

print(reverse_list_insert([1, 2, 3, 4, 5]))
```

```
[5, 4, 3, 2, 1]
```

4. Un élément sur `n` dans une liste

On cherche à écrire une fonction qui, à partir d'une liste, sélectionne un élément sur `n` dans cette liste.

Par exemple, si $n = 3$, on veut transformer cette liste : $[\underline{3}, 9, 2, \underline{1}, 7, 8, \underline{4}, 3, 0, \underline{1}, 9, 7, \underline{5}, 3, 1, \underline{9}]$ en celle-ci : $[3, 1, 4, 1, 5, 9]$

Voici la liste de test que nous allons utiliser pour la suite :

Avec une boucle et une nouvelle liste

On peut utiliser une approche classique : créer la nouvelle liste au fur-et-à-mesure, en parcourant la liste de départ.

Avec une condition sur les indices

```python
def un_sur_n_indices(n: int, liste: list) -> list:
    """Sélectionne un élément sur `n` dans `liste`"""
    new_list = []
    for i in range(len(liste)):
        # si i est divisible par n (une fois sur n)
        if 0 == i % n:
            # on ajoute l'élément à l'indice actuel dans la
            # nouvelle liste
            new_list.append(liste[i])
    return new_list

print(un_sur_n_indices(3, [2, 7, 1, 8, 2, 8, 1, 8]))
```

```
[2, 8, 1]
```

En utilisant un pas sur `range`

Une technique plus simple (et plus efficace) est, plutôt que de tester pour tous les indices, d'utiliser un `range` dans lequel on met un pas de `n`.

Cela permet de n'avoir dans la boucle que les indices qui nous intéressent.

```python
def un_sur_n_range(n: int, liste: list) -> list:
    new_list = []
    # on met un 0 pour que n soit bien le 3ème argument
    for i in range(0, len(liste), n):
        new_list.append(liste[i])
    return new_list

print(un_sur_n_range(3, [2, 7, 1, 8, 2, 8, 1, 8]))
```

```
[2, 8, 1]
```

Avec une *list comprehension*

Pour être encore plus efficace, on peut simplement utiliser un *list comprehension*, en conjonction avec les techniques citées plus haut.

Le code est en fait équivalent, mais permet de créer la liste de façon plus efficace.

Avec une condition sur les indices

```python
def un_sur_n_comprehension_indices(n: int, liste: list) -> list:
    return [liste[i] for i in range(len(liste)) if 0 == i%n]

print(un_sur_n_comprehension_indices(3, [2, 7, 1, 8, 2, 8, 1, 8]))
```

```
[2, 8, 1]
```

Retours à la ligne pour plus de clarté

Pour rendre le code plus clair, on peut mettre un retour à la ligne avant le `for` et le `if` :

```python
def un_sur_n_comprehension_indices(n: int, liste: list) -> list:
    return [liste[i]
            for i in range(len(liste))
            if 0 == i%n]
```

Cela est très utile quand on construit des expressions complexes, par exemple avec des *list comprehension* à l'intérieur de *list comprehension*.

En utilisant un pas sur `range`

On peut à nouveau utiliser un pas sur le `range` pour ne pas avoir à tester toutes les itérations.

```
def un_sur_n_comprehension_range(n: int, liste: list) -> list:
    return [liste[i] for i in range(0, len(liste), n)]

print(un_sur_n_comprehension_range(3, [2, 7, 1, 8, 2, 8, 1, 8]))
```

```
[2, 8, 1]
```

5. Maximum de nombes dans deux tableaux

On veut écrire une fonction `maxDes2(tab1, tab2)` qui prend en argument deux tableaux de nombres `tab1` et `tab2` de même longueur et retourne un tableau formé des valeurs maximales observées pour chaque indice entre les tableaux tab1 et tab2. Par exemple, `maxDes2([1, 4, 5], [2, 2, 3])` retourne le tableau `[2, 4, 5]`.

En parcourant les indices des deux tableaux

L'approche classique est de parcourir les indices `i` des deux tableaux (que l'on suppose de même taille), et de calculer le maximum pour chaque indice, que l'on mettra dans une nouvelle liste.

```
def max_des_2_indices(tab1: list[int], tab2: list[int]) -> list[int]:
    new_list: list[int] = []
    # on parcoure les indices des deux tableaux en même
    # temps avec i
    for i in range(len(tab1)):
        # la fonction max calcul le maximum de ses arguments
        # ici, les arguments sont les valeurs des deux
        # tableaux pour un même indice i
        new_list.append(max(tab1[i], tab2[i]))
    return new_list

print(max_des_2_indices([1, 4, 5], [2, 2, 3]))
```

```
[2, 4, 5]
```

En utilisant la fonction `zip`

La fonction `zip` va permettre de regrouper les éléments exactement comme on le souhaite. En effet, si on essaie de l'appliquer :

```
z = zip([3, 1, 4, 1, 5, 9, 2, 6], [2, 7, 1, 8, 2, 8, 1, 8])
print(list(z))  # on utilise list pour que le contenu soit bien affiché
```

```
[(3, 2), (1, 7), (4, 1), (1, 8), (5, 2), (9, 8), (2, 1), (6, 8)]
```

On observe que le résultat contient les paires d'éléments dont on veut faire le maximum : les deux premiers de chaque tableau, plus les deux deuxièmes, les deux troisièmes etc…

On peut alors proposer la solution suivante :

```
def max_des_2_zip(tab1: list[int], tab2: list[int]) -> list[int]:
    new_list: list[int] = []
    for couple in zip(tab1, tab2):
        # on note que la fonction `max` peut s'appliquer sur
        # une liste d'élément (ici `couple`)
        new_list.append(max(couple))
    return new_list

print(max_des_2_zip([1, 4, 5], [2, 2, 3]))
```

```
[2, 4, 5]
```

Avec `zip` et `map`

On remarque dans cet exercice une structure que l'on a déjà vue dans les exercices suivants : on veut appliquer une fonction particulière sur chaque élément d'une liste, puis récupérer le résultat.

L'approche classique consiste à parcourir la liste, et à créer au fur-et-à-mesure une nouvelle liste.

Cependant, une des fonctions de base de python, la fonction `map`, permet directement d'appliquer une fonction sur tous les éléments d'une liste, et de récupérer le résultat.

On peut donc tout simplement écrire :

```python
def max_des_2_map(tab1: list[int], tab2: list[int]) -> list[int]:
    # on utilise list pour bien récupérer une liste
    # on applique la fonction max sur le résultat du zip
    return list(
        map(max, zip(tab1, tab2))
    )

print(max_des_2_map([1, 4, 5], [2, 2, 3]))
```

```
[2, 4, 5]
```

Cette approche est une approche *fonctionnelle* du problème, puisque la solution est créée en composant des fonctions existantes (`map`, `max`, `zip`…) et sans structures de contrôles comme des boucles ou des conditions.

6. Reprogrammer la fonction zip

On veut écrire une fonction `myzip(tab1, tab2)` qui retourne une liste dont chaque élément d'indice `i` est lui-même une liste possédant deux valeurs issues des listes `tab1` et `tab2` à l'indice `i`. Par exemple, `myzip({1, 4, 5}, {2, 2, 3})` retourne la liste `{{1, 2}, {4, 2}, {5, 3}}` ; comparer votre solution à la fonction `zip()` de Python.

En parcourant les indices des deux listes

```python
def myzip_indices(tab1: list[int], tab2: list[int]) -> list[int]:
    zipped_list: list[int] = []
    for idx in range(len(tab1)):
        # on ajoute le couple (tab1[idx], tab2[idx]) à la
        # liste de résultat. On a bien un couple d'éléments
        # aux mêmes indices
        zipped_list.append((tab1[idx], tab2[idx]))
    return zipped_list

print(myzip_indices([1, 4, 5], [2, 2, 3]))
```

```
[(1, 2), (4, 2), (5, 3)]
```

Si les deux listes ne font pas la même taille

Si les deux listes ne font pas la même taille, il faut s'arrêter quand la première liste est arrivée au bout. On peut donc simplement parcourir les indices de `1` à `min(len(tab1), len(tab2))`.

```
def myzip_indices(tab1: list[int], tab2: list[int]) -> list[int]:
    zipped_list: list[int] = []
    for idx in range(min(len(tab1), len(tab2))):
        # on ajoute le couple (tab1[idx], tab2[idx]) à la
        # liste de résultat. On a bien un couple d'éléments
        # aux mêmes indices
        zipped_list.append((tab1[idx], tab2[idx]))
    return zipped_list

print(myzip_indices([1, 4, 5], [2, 2, 3, 99, 0]))
print(myzip_indices([1, 4, 5, 7, 13, 4], [2, 2, 3, 99]))
```

```
[(1, 2), (4, 2), (5, 3)]
[(1, 2), (4, 2), (5, 3), (7, 99)]
```

7. Générer une matrice aléatoire

On veut écrire une fonction `genMat(row, col, mini, maxi)` qui construit une liste de liste contenant `row` lignes et `col` colonnes et dont les valeurs sont comprises entre `mini` et `maxi`

Avec des *list comprehension*

```
def genMat(row: int, col: int, mini: int, maxi: int) -> list[list[int]]:
    """Initialiser une matrice aléatoire de taille (row, col), avec des valeurs dans [mini
    """
    return [[randint(mini, maxi) for i in range(col)] for j in range(row)]

print(genMat(3, 3, 0, 10))
```

```
[[6, 6, 9], [6, 10, 5], [3, 4, 7]]
```

En créant la liste au fur-et-à-mesure

Les *list comprehension* sont plus rapides, plus courtes et beaucoup plus simples à utiliser. Cet exemple est simplement là pour montrer d'autres techniques de programmation.

```
def genMat(row: int, col: int, mini: int, maxi: int) -> list[list[int]]:
    """Initialiser une matrice aléatoire de taille (row, col), avec des valeurs dans [mini
    """
    mat = []
    for i in range(row):
        line = []
        for j in range(col):
            line.append(randint(mini, maxi))
        mat.append(line)
    return mat


print(genMat(3, 3, 0, 10))
```

```
[[0, 9, 10], [3, 8, 9], [8, 7, 0]]
```

8. Diagonale d'une matrice

On veut écrire une fonction diagonale(mat) qui prend en argument une matrice de réels mat et retourne une liste contenant les éléments de sa diagonale.

```
M = [[1, 6, 1],
     [8, 0, 3],
     [9, 8, 8]]
```

Avec des *list comprehension*

```
def diagonale(mat: list[list[float]]) -> list[float]:
    """Diagonale d'une matrice.
    Args:
        mat (list[list[float]]): Une matrice qui doit être carrée
                                 (sinon la diagonale n'existe pas).
    Returns:
```

```
        list[float]: La liste des coefficients diagonaux de mat.
    """
    return [mat[i][i] for i in range(len(mat))]

print(diagonale(M))
```

```
[1, 0, 8]
```

Pour aller plus loin

Lever une erreur si la matrice n'est pas carrée

Pour bien faire, il faudrait lever une erreur si la matrice n'est pas carrée. Pour cela, on utilise le mot clef `raise`, ainsi qu'une erreur classique de python. Ici, on utilisera `ValueError` (on pourrait également créer une classe d'erreurs nous-même, puisque les erreurs sont simplement des objets particuliers).

```
def diagonale(mat: list[list[float]]) -> list[float]:
    """Diagonale d'une matrice.
    Args:
        mat (list[list[float]]): Une matrice qui doit être carrée
                                 (sinon la diagonale n'existe pas).
    Returns:
        list[float]: La liste des coefficients diagonaux de mat.
    Raises:
        ValueError: Si la matrice donnêé en entrée n'est pas carrée.
    """
    # si la matrice n'est pas carrée
    if not all(len(mat) == len(ligne) for ligne in mat):
        # on lève une exception.
        raise ValueError("La matrice n'est pas carrée.")
    return [mat[i][i] for i in range(len(mat))]

print(diagonale(M))
```

```
[1, 0, 8]
```

9. Trace d'une matrice

On veut écrire une fonction `trace(mat)` qui prend en argument une matrice de réels `mat` et retourne la somme de ses éléments diagonaux.

```
M = [[0, 1, 1],
     [2, 3, 5],
     [8, 1, 3]]
```

Avec une *list comprehension* et la fonction `sum`

```
def trace(mat: list[list[float]]) -> float:
    return sum([mat[i][i] for i in range(len(mat))])

print(trace(M))
```

```
6
```

En réutilisant la fonction `diagonale`

Comme on a déjà programmé la fonction `diagonale`, on peut l'utiliser, car la trace d'une matrice est la somme de ses coefficients diagonaux.

```
def trace(mat: list[list[float]]) -> float:
    return sum(diagonale(mat))

print(trace(M))
```

```
6
```

Cela rend le code moins redondant et plus clair. C'est l'intérêt d'utiliser des fonctions.

10. Somme de deux matrices

On veut écrire une fonction `somme(mat1, mat2)` qui prend en argument deux matrices de réels `mat1` et `mat2`, et retourne la matrice somme de ces deux matrices.

```
A = [[0, 1, 0, 1],
     [1, 0, 1, 0],
     [0, 1, 0, 1]]

B = [[0, 1, 2, 3],
     [0, 1, 2, 3],
     [2, 4, 6, 8]]
```

Avec une *list comprehension*

```
def somme(mat1: list[list[float]], mat2: list[list[float]]) -> list[list[float]]:
    """Somme de deux matrices que l'on suppose de même taille.
    """
    # nombre de lignes et de colonnes de mat1 (on la prends comme référence)
    rows = len(mat1)
    cols = len(mat1[0])
    return [[mat1[i][j] + mat2[i][j] for j in range(cols)] for i in range(rows)]

print(somme(A, B))
```

```
[[0, 2, 2, 4], [1, 1, 3, 3], [2, 5, 6, 9]]
```

Avec des `zip` et des `map`

Cette solution est plus complexe, mais elle peut avoir des avantages.

Par exemple, si on retire les fonctions `list` du code, la fonction va retourner un objet `map`, qui est une structure paresseuse ("*lazy*"). Cela veut dire qu'un élément donné ne sera calculé que lorsque l'on en aura besoin (lorsque l'on parcourra la matrice, par exemple).

Ce mécanisme est utile si, quand une fonction est longue à calculer, vous ne voulez pas être obligé d'attendre que toutes les valeurs soient passées par cette fonction avant de pouvoir passer à l'étape suivante : la fonction ne sera exécutée que sur les valeurs nécessaires, au fur-et-à-mesure.

```
def somme(mat1: list[list[float]], mat2: list[list[float]]) -> list[list[float]]:
    """Somme de deux matrices que l'on suppose de même taille.
    """
    return list(map(lambda x: list(map(sum, zip(*x))), zip(mat1, mat2)))
```

```
print(somme(A, B))
```

```
[[0, 2, 2, 4], [1, 1, 3, 3], [2, 5, 6, 9]]
```

11. Produit matriciel

On veut écrire une fonction `produit(mat1, mat2)` qui prend en argument deux matrices de réels `mat1` de dimension $n \times k$ et `mat2` de dimension $k \times m$, et retourne la matrice produit de ces deux matrices de dimension $n \times m$.

> Rappel de la formule
>
> Soient $mat_1 \in \mathcal{M}_{n,k}(\mathbb{R})$ et $mat_2 \in \mathcal{M}_{k,m}(\mathbb{R})$ deux matrices.
> On sait que le produit $mat_1 \times mat_2$ est une matrice de taille $n \times m$.
> Alors :
> $$\forall i \in [\![1, n]\!], \quad \forall j \in [\![1, m]\!], \quad (mat_1 \times mat_2)_{i,j} = \sum_{l=1}^{k} \Big(mat_1(i, l) \times mat_2(l, j) \Big)$$

```
A = [[1, 1, 0],
     [0, 2, 1],
     [0, 1, 3],
     [0, 0, 0]]

B = [[1, 5, 2, 5],
     [1, 1, 3, 1],
     [7, 6, 2, 6]]
```

Avec des *list comprehension*

En utilisant presque directement la formule de définition du produit de matrices, on obtient cette fonction :

```
def produit(mat1: list[list[float]], mat2: list[list[float]]) -> list[list[float]]:
    """Produit matriciel mat1 * mat2.
    On suppose que les matrices sont de la bonne taille, c'est-à-dire que la largeur de ma
    """
    # largeur et hauteur de la matrice résultat
    width = len(mat1)
```

```
    height = len(mat2[0])
    common_length = len(mat2)
    # on applique la formule :
    return [[sum(mat1[j][l] * mat2[l][i] for l in range(common_length)) for i in range(hei

print(produit(A, B))
```

```
[[2, 6, 5, 6], [9, 8, 8, 8], [22, 19, 9, 19], [0, 0, 0, 0]]
```

Avec le module `numpy`

Le module `numpy` (qui n'est pas un module standard, il faudra donc l'installer avec `pip3 --install numpy`) possède des fonctions pour l'algèbre linéaire et pour les tableaux en général.

Un objet `matrix` est implémenté, et il permet de faire des multiplications de matrices… Avec l'opérateur `*` ! (Attention : si on utilise l'objet `array` de numpy plutôt que l'objet `matrix`, la multiplication sera une multiplication élément-par-élément plutôt qu'une vraie multiplication matricielle).

```
import numpy as np

mA = np.matrix(A)
mB = np.matrix(B)

print(mA * mB)
```

```
[[ 2  6  5  6]
 [ 9  8  8  8]
 [22 19  9 19]
 [ 0  0  0  0]]
```

Note : Avec cette méthode, le résultat n'est pas une liste de listes, mais une matrice.

12. Divisibilité par récursion

On veut écrire une fonction **récursive** `estDivisible(n, m)` qui retourne `True` si et seulement si `m` divise `n`, et `false` sinon.

La fonction ne doit pas utiliser les opérateurs de division, `/`, ou le modulo, `%`.

Récursion simple

On va simplement utiliser cette propriété : `n` divise `m` si et seulement si `n` divise `m - n` : $\forall (m,n) \in \mathbb{Z}^2, \quad n \mid n \iff n \mid m-n$

On utilise aussi le fait que `n` divise `m` si et seulement si la valeur absolue de `n` divise la valeur absolue de `m`.

```
def estDivisible(n: int, m: int) -> bool:
    if m == 0:
        return True
    if m < 0:
        return False
    return estDivisible(abs(n), abs(m) - abs(n))

print(estDivisible(3, 6))    # True
print(estDivisible(-3, 12))  # True
print(estDivisible(-3, 11))  # False
```

```
True
True
False
```

13. Palindrome par récursion

```
def palindrome(tab: list) -> bool:
    # les listes de longueur 0 ou 1 sont toutes des palindrômes
    if len(tab) <= 1:
        return True
    # si les deux extrémités sont différentes, ce n'est pas un palindrome
    if tab[0] != tab[-1]:
        return False
    # récursion en enlevant les deux extrémités, que l'on a déjà vérifiées
    return palindrome(tab[1:-1])

print(estDivisible(3, 6))    # True
print(estDivisible(-3, 12))  # True
print(estDivisible(-3, 11))  # False
```

```
True
True
False
```

14. Nombre de chiffres par récursion

Récursion classique

```python
def longueur(n: int) -> int:
    n = abs(n)
    if n < 10:
        return 1
    return 1 + longueur(n / 10)

print(longueur(314159265358))
print(longueur(73))
print(longueur(1732))
```

```
12
2
4
```

Récursion terminale

Détails sur la récursion terminale

La récursion terminale est un récursion dans laquelle la dernière opération est l'appel récursif. Cela veut dire que le `return` qui contient l'appel récursif ne contient pas d'autre opération.
Par exemple, la définition précédente de `longueur` n'est pas terminale, car on doit ajouter 1 après l'appel récursif (la ligne de l'appel récursif est `return 1 + longueur(n / 10)`).
La récursion terminale à plusieurs avantages :

- dans certains langages, elle est optimisée (l'optimisation de pile d'appel) et rend l'exécution plus rapide et moins coûteuse en mémoire
- Elle peut être très facilement convertie en une boucle (la variable de boucle est l'accumulateur de la récursion terminale)
  - Dans certains livres, comme SICP, on voit que la récursion terminale est appelée "itérative"

Ici, on a aussi optimisé le programme en travaillant uniquement sur des entiers, ce qui permet d'éviter des calculs de division de flottants, qui sont inutiles.

```
def longueur(n: int, acc: int =0) -> int:
    n = int(abs(n))
    if n < 10:
        return acc + 1
    return longueur(n // 10, acc + 1)

print(longueur(314159265358))
print(longueur(73))
print(longueur(1732))
```

```
12
2
4
```

15. Chiffres < 4 par récursion

On veut écrire une fonction **récursive** `combienInf4(n)`, qui prend en argument un entier naturel `n`, et retourne le nombre de chiffres qui le compose et qui sont strictement inférieurs à 4.

Méthode : On va simplement utiliser les opérateurs `//` et `%`, qui donnent respectivement le quotient et le reste d'une division euclidienne. En prenant en boucle le reste de la division par 10, on obtient chaque chiffre du nombre de départ.

Récursion classique

```python
def combienInf4(n: int) -> int:
    # si le dernier chiffre de n est un 4
    if n % 10 < 4:
        # si n est un chiffre
        if n < 10:
            # le résultat est 1
            return 1
        # on ajoute 1 à la récursion car n finit par 4
        return 1 + combienInf4(n // 10)
    # si n ne finit pas par 4 :
    # si n est un chiffre
    if n < 10:
        # aucun 4 dans n
        return 0
    # on ajoute rien à la récursion car n ne finit pas par 4
    return combienInf4(n // 10)

print(combienInf4(123456))
print(combienInf4(314159265358))
print(combienInf4(789456))
```

```
3
5
0
```

Avec des conversions de types

Pour rendre le code plus simple (et plus lisible pour un programmeur averti), on utilise le fait que la fonction `int` puisse convertir des booléens en entiers.

```python
def combienInf4(n: int) -> int:
    if n < 10:
        # (n < 4) est un booléen
```

```
        # int(True) vaut 1, et int(False) vaut 0
        return int(n < 4)
    # on ajoute 1 à la récursion si le dernier chiffre de n est 4
    return int(n % 10 < 4) + combienInf4(n // 10)

print(combienInf4(123456))
print(combienInf4(314159265358))
print(combienInf4(789456))
```

```
3
5
0
```

Récursion terminale

```
def combienInf4(n: int, acc: int =0) -> int:
    if n < 10:
        return int(n < 4) + acc
    return combienInf4(n // 10, acc + int(n % 10 < 4))


print(combienInf4(123456))
print(combienInf4(314159265358))
print(combienInf4(789456))
```

```
3
5
0
```