Devoir ITC n°2

Autour des matrices

Le sujet suivant s'intéresse aux opérations sur les matrices ; on rappelle qu'au sens mathématique, une matrice carrée de taille n peut être vue comme un tableau de nombres : par exemple, on propose ci-dessous une matrice A de taille $n = 4$ à valeurs dans $\mathbb{Z}$ et une représentation d'une matrice M de taille n quelconque par ses éléments $M_{i,j}$, i étant le numéro de ligne et j celui de colonne de la matrice :

$$A = \begin{pmatrix} 7 & 2 & 8 & 1 \\ 1 & -1 & -3 & 0 \\ 2 & -2 & 2 & 9 \\ 4 & 7 & 5 & -7 \end{pmatrix} \quad ; \quad M = (M_{i,j}) = \begin{pmatrix} M_{0,0} & M_{0,1} & \cdots & M_{0,n-1} \\ M_{1,0} & M_{0,1} & \cdots & M_{0,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ M_{n-1,0} & M_{n-1,1} & \cdots & M_{n-1,n-1} \end{pmatrix}$$

On appelle **matrice nulle** la matrice 0_n ne contenant que des zéros et **matrice identité** la matrice I_n contenant des 1 sur la diagonale et des zéros partout ailleurs ; par exemple pour $n = 3$:

$$0_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \text{et} \quad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Dans un programme informatique en Python, les matrices sont représentées comme des listes de listes de nombres. En effet, si `m` est une liste de n listes, chacune étant une liste de taille n contenant des nombres, alors :

- `m[i]` pour $i \in [\![0; n-1]\!]$ est une liste de taille n représentant la i^{e} ligne de la matrice ;
- `m[i][j]` pour $(i, j) \in [\![0; n-1]\!]^2$ est le j^{e} élément de la liste `m[i]`, donc le j^{e} élément de la i^{e} ligne de `m` ; finalement, `m[i][j]` est bien l'élément de la matrice `m` en ligne i, colonne j.

I. Manipulations élémentaires sur une matrice

1 – Écrire une fonction `matrice_nulle(n: int) -> Mat` qui prend en paramètre un entier n et renvoie la matrice nulle de taille $n \times n$.

```
def matrice_nulle(n):
    return [[0]*n for _ in range(n)]
```

2 – Écrire une fonction `matrice_identité(n: int) -> Mat` qui prend en paramètre un entier n et renvoie la matrice identité de taille $n \times n$. On

```
def matrice_identité(n):
    mat = matrice_nulle(n)
    for i in range(n):
        mat[i][i] = 1
    return mat
```

3 – Écrire une fonction `copie_matrice(m: Mat) -> Mat` qui prend en paramètre une matrice `m` et renvoie une nouvelle matrice, indépendante, contenant les mêmes éléments que `m`.

```
# itération sur les indices
```

```python
def copie_matrice(m):
    n = len(m)
    cop = [[m[i][j] for j in range(n)] for i in range(n)]
    return cop
# itération sur les valeurs
def copie_matrice(m):
    cop = [[x for x in ligne] for ligne in m]
    return cop
```

On appelle **trace** de la matrice la somme des éléments sur sa diagonale, soit

$$\mathrm{tr}(M) = \sum_{i=0}^{n-1} M_{i,i}$$

Par exemple, la trace de la matrice identité vaut n, et celle de la matrice A donnée en exemple au début du sujet est 1.

4 – Écrire une fonction `trace(m: Mat) -> int` qui prend en paramètre une matrice `mat` et renvoie la trace de la matrice.

```python
def trace(mat):
    tr = 0
    n = len(mat)
    for i in range(n):
        tr += mat[i][i]
    return tr
```

La somme de deux matrices carrées de taille n est une matrice carrée de taille n dont les éléments sont les sommes des deux matrices d'entrées :

$$S_{i,j} = M_{i,j} + N_{i,j}$$

5 – Pourquoi l'instruction `s = m1 + m2` ne fait pas la somme de deux matrices au sens ci-dessus en Python ? Illustrer le problème sur le cas

$$m_1 = \begin{pmatrix} 2 & 3 \\ 1 & 1 \end{pmatrix} \quad \text{et} \quad m_2 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

En Python, l'opérateur + sur les listes produit une **concaténation** ; ainsi, sur les cas précédents,

```python
m1 = [[2, 3], [1, 1]]
m2 = [[0, 0], [0, 0]]
s = m1 + m2
```

donne pour s

```python
[[2, 3], [1, 1], [0, 0], [0, 0]]
```

et non

```python
[[2, 3], [1, 1]]
```

---

6 – Écrire une fonction `add(m1: Mat, m2:Mat) -> Mat` qui renvoie la somme de deux matrices `m1` et `m2` . Quelle est sa complexité ?

```python
def add(m1, m2):
```

```
    n = len(m1)
    mat = matrice_nulle(n)
    for i in range(n):
        for j in range(n):
            mat[i][j] = m1[i][j] + m2[i][j]
    return mat
```

L'appel à `matrice_nulle` est en $\mathcal{O}(n^2)$; la somme et l'écriture dans la boucle sont des opérations élémentaires, et la double boucle est répétée n^2 fois d'où une complexité en $\mathcal{O}(n^2)$.

II. Déterminant d'une matrice

On peut calculer[1] le déterminant d'une matrice carrée de la façon suivante :

- si $n = 1$,

$$\det(M) = M_{0,0}$$

- si $n > 1$, on peut ramener le calcul du déterminant à celui de n déterminants de taille $n-1$ en développant selon la colonne j :

$$\det(M) = \sum_{i=0}^{n-1} (-1)^{i+j} M_{i,j} \det(A_{i,j})$$

où $A_{i,j}$ est la matrice de taille $n-1$ obtenue en retirant la ligne i et la colonne j de la matrice j.

7 – Écrire une fonction `extraire_sous_matrice(m: Mat, i: int, j: int)` qui extrait la matrice A_{ij} de taille $n-1$ et la renvoie.

```
def extraire_sous_matrice(m, i, j):
    n = len(m)
    ss_mat = matrice_nulle(n-1)
    for k in range(n-1):
        for l in range(n-1):
            if k < i and l < j:
                ss_mat[k][l] = m[k][l]
            elif k >= i and l < j:
                ss_mat[k][l] = m[k+1][l]
            elif k < i and l >= j:
                ss_mat[k][l] = m[k][l+1]
            elif k >= i and l >= j:
                ss_mat[k][l] = m[k+1][l+1]
    return ss_mat
# Autre proposition
def extraire_sous_matrice(m, i, j):
    n = len(m)
    ss_mat = []
    for k in range(n):
        if k != i:
            new_ligne = m[k][:j] + m[k][j+1:]
            ss_mat.append(new_ligne)
    return ss_mat
```

---

8 – Écrire une fonction `déterminant(m: Mat) -> int` qui calcule le déterminant d'une matrice.

---

[1] La définition correcte sera dans votre cours de mathématiques

```
def déterminant(m):
    n = len(m)
    if n == 1:
        det = m[0][0]
    else:
        det = 0
        for i in range(n):
            ss_mat = extraire_sous_matrice(m, i, 0)
            det += (-1)**i * m[i][0] * déterminant(ss_mat)
    return det
```

III. Produits de matrices

Par définition, le produit P de deux matrices M et N est obtenu en sommant les termes produits de la i^{e} ligne de M et la j^{e} colonne de N :

$$P_{i,j} = \sum_{k=0}^{n-1} M_{i,k} N_{k,j}$$

Par exemple,

$$\begin{pmatrix} 2 & 1 \\ 3 & -1 \end{pmatrix} \cdot \begin{pmatrix} 5 & 4 \\ -2 & -3 \end{pmatrix} = \begin{pmatrix} 2\times 5 + 1\times(-2) & 2\times 4 + 1\times(-3) \\ 3\times 5 + (-1)\times(-2) & 3\times 4 + (-1)\times(-3) \end{pmatrix} = \begin{pmatrix} 8 & 5 \\ 17 & 15 \end{pmatrix}$$

La puissance p d'une matrice M est définie comme la répétition de p multiplications de la matrice :

$$M^p = M \cdot M^{p-1} = \underbrace{M \cdot M \cdot \ldots \cdot M}_{p \text{ fois}}$$

9 – Écrire une fonction `prod(m1: Mat, m2: Mat) -> Mat` qui calcule ainsi le produit de deux matrices en appliquant la définition. La complexité doit être en $\mathcal{O}(n^3)$, et on justifiera succintement cette complexité.

```
def prod(m1, m2):
    n = len(m1)
    mat = matrice_nulle(n)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                mat[i][j] += m1[i][k]*m2[k][j]
    return mat
```

La complexité en $\mathcal{O}(n^3)$ est assurée par la triple boucle imbriquée, qui ne contient que des opérations élémentaires, et sachant que `matrice_nulle` est en $\mathcal{O}(n^2)$ donc négligeable.

---

10 – En utilisant la fonction `prod`, écrire une fonction `puiss(m: Mat, p: int) -> Mat` qui calcule la puissance p de la matrice m. La complexité attendue est $\mathcal{O}(n^3 p)$, sans justification.

```
def puiss(m, p):
    n = len(m)
    res = matrice_identité(n)
    for i in range(p):
        res = prod(res, m)
    return res
```

Pour les matrices comme pour les nombres, on peut améliorer le temps de calcul par l'algorithme d'**exponentiation rapide** : au lieu de calculer p fois le produit, on calcule $y = x^{p//2}$ puis

$$x^p = \begin{cases} y \times y & \text{si } p \text{ est pair} \\ x \times y \times y & \text{si } p \text{ est impair} \end{cases}$$

11 – À quelle famille d'algorithmes appartient l'algorithme d'exponentiation rapide ?

Il s'agit d'un algorithme **dichotomique**.

---

12 – En utilisant l'algorithme d'exponentiation rapide, proposer une fonction **récursive** `puiss2(m: Mat, p: int) -> Mat` dont on donnera la complexité sans justification.

```
def puiss2(m, p):
    if p == 1:  # cas de base
        return m
    m = puiss2(mat, p//2)  # appel récursif
    a = prod(m, m)
    if p%2 == 0:  # si p pair
        return a
    else:
        return prod(mat, a)
```

La complexité est $\mathcal{O}(n^3 \log p)$.

On s'intéresse à des améliorations possibles du calcul du produit lui-même ; dans un premier temps, on remarque que si on décompose les matrices A, B et le produit C en quatre sous-matrices de taille égale :

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad ; \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \quad ; \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

alors la formule pour calculer les sous-matrices de C est la même que pour une matrice de nombres 2×2 :

$$\begin{aligned} C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\ C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\ C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2} \end{aligned}$$

Ainsi, le calcul du produit $A \cdot B$ de matrices de taille n se ramène au calcul de 8 produits de matrices de taille $n/2$, ainsi que 4 sommes. **Dans toute la suite, on supposera que les tailles des matrices sont des puissances de 2** : $n = 2^k$. On suppose également que l'on dispose des fonctions

```
découpage(m: Mat) -> (Mat, Mat, Mat, Mat)
reconstruction(m11: Mat, m12: Mat, m21: Mat, m22: Mat) -> Mat
```

qui permettent respectivement de décomposer une matrice en ses 4 sous-matrices, et de prendre 4 sous-matrices pour recréer une grande matrice. On admet que ces fonctions peuvent avoir une complexité $\mathcal{O}(1)$.

13 – Écrire une fonction **récursive** `prod2(m1: Mat, m2: Mat) -> Mat` qui utilise cette décomposition pour calculer le produit de deux matrices de taille n.

```python
def prod2(m1, m2):
    if len(m1) == 1:  # condition d'arrêt
        a = m1[0][0]
        b = m2[0][0]
        return [[a*b]]
    # construction récursive
    a11, a12, a21, a22 = découpage(m1)  # O(1)
    b11, b12, b21, b22 = découpage(m2)  # O(1)
    c11 = somme(prod2(a11, b11), prod2(a12, b21))  # 2C(n/2) + an/2
    c12 = somme(prod2(a11, b12), prod2(a12, b22))  # 2C(n/2) + an/2
    c21 = somme(prod2(a21, b11), prod2(a22, b21))  # 2C(n/2) + an/2
    c22 = somme(prod2(a21, b12), prod2(a22, b22))  # 2C(n/2) + an/2
    return reconstruction(c11, c12, c21, c22)            # O(1)
```

---

14 – On note c_n le temps de calcul de `prod2` pour une matrice de taille n, et an le temps de calcul de la somme de deux matrices de taille n avec a une constante, montrer que c_n est de la forme

$$c_n = \alpha c_{n/2} + \beta an + b$$

avec b une constante, et α et β à préciser.

La fonction fait 8 appels récursifs pour calculer des produits de matrices de taille $n/2$, donc de coût $c_{n/2}$; par ailleurs elle fait 4 sommes de matrices de taille $n/2$, donc un coût $4an/2 = 2an$. En ajoutant les opérations de temps constant b, on obtient finalement :

$$c_n = 8c_{n/2} + 2an + b \quad \text{soit} \quad \alpha = 8 \quad \text{et} \quad \beta = 2$$

---

15 – En introduisant $w_k = c_{2^k}$, et en négligeant le temps de calcul βan et b devant $\alpha c_{n/2}$, montrer que $c_n = \mathcal{O}(n^3)$. A-t-on gagné une meilleure complexité par rapport à la fonction `prod` ?

En partant du résultat précédent,

$$c_n \simeq 8c_{n/2} \quad \Rightarrow \quad w_k = 8w_{k-1} = 8^k w_0 = (2^k)^3 w_0 \quad \Rightarrow \quad c_n = n^3 w_0 = \mathcal{O}(n^3)$$

Il s'agit de la même complexité que `prod` , il n'y a donc pas de gain à ce stade.

Pour améliorer cette complexité, on utilise l'algorithme de Strassen : on décompose les matrices comme précédemment, mais on effectue le calcul des sous-matrices de C de la façon suivante :

$$\begin{cases} M_1 = (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\ M_2 = (A_{2,1} + A_{2,2}) \cdot B_{1,1} \\ M_3 = A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\ M_4 = A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\ M_5 = (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\ M_6 = (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \\ M_7 = (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) \end{cases} \quad \text{et} \quad \begin{cases} C_{1,1} = M_1 + M_4 - M_5 + M_7 \\ C_{1,2} = M_3 + M_5 \\ C_{2,1} = M_2 + M_4 \\ C_{2,2} = M_1 - M_2 - M_3 + M_6 \end{cases}$$

16 – En adaptant le calcul de complexité précédent, calculer la complexité de l'algorithme de Strassen.

On fait à nouveau des appels récursifs sur des sous-matrices de taille $n/2$, mais on n'effectue cette fois que 7 multiplications de sous-matrices au lieu de 8. Le calcul précédent s'adapte donc sous la forme

$$c_n \simeq 7c_{n/2} \quad \Rightarrow \quad w_k = 7w_{k-1} = 7^k w_0 = (2^k)^{\log_2 7} w_0 \quad \Rightarrow \quad c_n = n^{\log_2 7} w_0 = \mathcal{O}(n^{\log_2 7})$$

---

17 – En combinant les résultats précédents, quelle complexité peut-on espérer au mieux pour le calcul de la puissance p d'une matrice de taille n ?

En combinant l'algorithme de Strassen pour les multiplications et l'exponentiation rapide pour diminuer le nombre de multiplications à effectuer, on obtient une complexité $\mathcal{O}(n^{\log_2 7} \log p)$.
