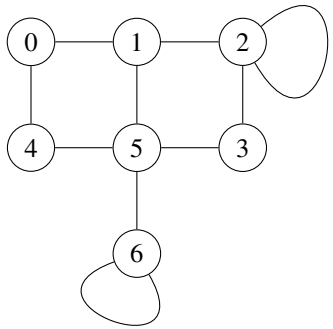
DEVOIR MAISON 3 : ITC - CORRIGÉ

Dénombrement de chemins dans un graphe

On s'intéresse dans ce problème à un graphe non orienté (pouvant contenir des boucles, c'est-à-dire des arêtes d'un sommet vers lui-même), dont on veut dénombrer les chemins en utilisant les puissances de sa matrice d'adjacence. On appliquera ce procédé pour dénombrer certains types d'entiers en les interprétant comme des chemins dans un graphe.

Partie I
Généralités sur les graphes

1. Donner la matrice d'adjacence **`M0`** du graphe non orienté G_0 défini ci-contre, ainsi que sa représentation par une liste de listes d'adjacences **`L0`** comme vous l'écririez en Python.



Corrigé :

```
M0 = [  [0, 1, 0, 0, 1, 0, 0],
        [1, 0, 1, 0, 0, 1, 0],
        [0, 1, 1, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 1, 0],
        [1, 0, 0, 0, 0, 1, 0],
        [0, 1, 0, 1, 1, 0, 1],
        [0, 0, 0, 0, 0, 1, 1]   ]

L0 = [  [1, 4],
        [0, 2, 5],
        [1, 2, 3],
        [2, 5],
        [0, 5],
        [1, 3, 4, 6],
        [5, 6]    ]
```

2. Quel est l'ordre de ce graphe ? Est-il connexe ? Justifier.

Corrigé :
L'ordre du graphe est le nombre de sommets, donc 7. Il est connexe, car chaque couple de sommets est relié par un chemin.

3. Écrire une fonction **`Degre(M,s)`** qui prend en argument un graphe représenté par sa matrice d'adjacence **`M`** et un sommet **`s`** et qui renvoie le degré du sommet **`s`**.

Corrigé :

```
def Degre(M,s):
    return sum(M[s])
```

4. Écrire une fonction **`Voisins(M,s)`** qui prend en argument un graphe représenté par sa matrice d'adjacence **`M`** et un sommet **`s`** et qui renvoie la liste des numéros des sommets voisins du sommet **`s`**.

Corrigé :

```
def Voisins(M,s):
    V = []
    for i in range(len(M)):
        if M[s][i]:
            V.append(i)
    return V
```

5. Écrire une fonction **`NbAretes(M)`** qui prend en argument un graphe représenté par sa matrice d'adjacence **`M`** et renvoie le nombre d'arêtes dans ce graphe.

Corrigé :

```
def NbAretes(M):
    n = len(M)
    NA = 0
    for i in range(n):
        for j in range(i+1):
            NA += M[i][j]
    return NA
```

Partie II
Dénombrement des chemins

6. Expliquer comment savoir s'il existe un chemin de longueur 1 du sommet i au sommet j dans un graphe défini par une matrice d'adjacence **`M`**.

Corrigé : Il y a un arc, c'est-à-dire un chemin de longueur 1 de i à j si et seulement si la matrice d'adjacence contient un 1 en position i, j.

7. On calcule la puissance 3 de la matrice d'adjacence **`M0`** du graphe G_0 et on trouve :

$$M_0^3 = \begin{pmatrix} 3 & 5 & 1 & 1 & 2 & 2 & 1 \\ 5 & 1 & 6 & 8 & 0 & 2 & 6 \\ 1 & 6 & 0 & \dots & 4 & 3 & 0 \\ 1 & 8 & 0 & 1 & 5 & 6 & 1 \\ 2 & 0 & 4 & 5 & 0 & 1 & 3 \\ 2 & 2 & 3 & 6 & 1 & 5 & 5 \\ 1 & 6 & 0 & 1 & 3 & 5 & 1 \end{pmatrix}$$

Vérifier pour les coefficients $[M_0^3]_{1,4} = 2$ et $[M_0^3]_{6,6} = 1$ de cette matrice que la valeur à la ligne i et à la colonne j de M_0^3 est égale au nombre de chemins de longueur 3 allant du sommet i au sommet j.

Corrigé :

Attention, la puissance de matrice donnée ne correspond pas au graphe du sujet, mais à ce graphe :

```
L0 = [  [0, 1],
        [0, 2, 6],
        [1, 4],
        [4, 5],
        [3, 2],
        [3, 5, 6],
        [1, 5]    ]
```

De plus, on a bien $[M_0^3]_{1,4} = 0$ et non 2.

Il y a bien 0 chemins de longueur (exactement) 3 pour aller de 1 à 4 : 1-2-4 et 1-3-4 sont de longueur 2, et tout autre chemin sera de longueur au moins 4.

Il existe un seul chemin de longueur 3 pour aller de 6 à 6 : le chemin 6 - 5 - 5 - 6.

8. On admet le théorème suivant :

> *Soit G un graphe de matrice d'adjacence M et $k \in \mathbb{N}^*$. La valeur du coefficient (i, j) de la matrice M^k est égal au nombre de chemins de longueur k allant du sommet i au sommet j.*

En déduire la valeur manquante dans la matrice M_0^3.

Corrigé :

La valeur manquante correspond au nombre de chemins de longueur 3 de 2 à 3. On constate sur le graphe qu'il y en a 0.

Partie III
Puissances de matrice

On rappelle que si $A \in \mathscr{M}_{n,p}(\mathbb{R})$ et $B \in \mathscr{M}_{p,q}(\mathbb{R})$ alors les coefficients de la matrice produit $C = AB \in \mathscr{M}_{n,q}(\mathbb{R})$ se calculent selon la formule :

$$\forall (i,j) \in [\![0, n-1]\!] \times [\![0, q-1]\!], \quad c_{i,j} = \sum_{k=0}^{p-1} a_{ik} b_{kj},$$

où, comme en Python, on numérote les lignes et les colonnes à partir de 0.

9. En déduire une fonction **`Mult(A,B)`** qui renvoie la matrice produit de **`A`** par **`B`**, notée **`C`** dans ce qui précède. La fonction testera si la taille des matrices permet de calculer le produit à l'aide d'un **`assert`**.

Corrigé :

```
def Mult(A,B):
    assert len(A[0]) == len(B)
    n = len(A)
    p = len(B)
    q = len(B[0])
    C = [[0 for j in range(q)] for i in range(n)]
    for i in range(n):
        for j in range(q):
            for k in range(p):
                C[i][j] += A[i][k]*B[k][j]
    return C
```

10. Quelle est la complexité de cette fonction, en fonction de n, p et q ?

Corrigé :

L'initialisation de C est en $O(nq)$, et les trois boucles imbriquées sont en $O(npq)$, la complexité totale est donc en $O(npq)$.

11. Écrire une fonction **`Id(n)`** qui renvoie la matrice identité de taille n.

Corrigé :

```
def ID(n):
    I = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        I[i][i] = 1
    return I
```

12. Soit $k \in \mathbb{N}$. Déduire de ce qui précède une fonction **`MatrixPower(A,k)`** qui renvoie la matrice A^k pour **`A`** une matrice carrée. (Attention $A^0 = I_n$)

Corrigé :

```
def MatrixPower(A,k):
    n = len(A)
    R = ID(n)
    for i in range(k):
        R = Mult(R,A)
    return R
```

13. Déterminer la complexité de la fonction **`MatrixPower`**, en fonction de k et de n le nombre de lignes (et de colonnes) de **`A`**.

Corrigé :

L'appel `ID(n)` est en $O(n^2)$, puis on fait k appels `Mult(R,A)`, chacun en $O(n^3)$, la complexité totale est donc en $O(kn^3)$.

14. On peut améliorer le calcul de la puissance d'une matrice à l'aide de l'exponentiation rapide

qui est fondée sur l'identité :

$$\forall k \in \mathbb{N}^*, \quad M^k = \begin{cases} \left(M^{\frac{k}{2}}\right)^2 & \text{si } k \text{ est pair} \\ \left(M^{\frac{k-1}{2}}\right)^2 \times M & \text{si } k \text{ est impair} \end{cases} \quad \text{et} \quad M^0 = I_n$$

Écrire une fonction récursive **`QuickMatrixPower(A,k)`** qui calcule et renvoie A^k selon ce principe.

Corrigé :

```
def QuickMatrixPower(A,k):
    if k==0:
        return ID(len(A))
    B = QuickMatrixPower(A, k//2)
    if k%2 ==0:
        return Mult(B,B)
    else :
        return Mult(mult(B,B),A)
```

15. Déterminer la complexité de cette fonction.

    Corrigé :

    On sait qu'une exponentiation rapide réalise $O(\log k)$ multiplications. Ici, chaque multiplication est en $O(n^3)$, d'où une complexité totale en $O(n^3 \log k)$.

16. Une matrice A de taille $n \times n$ est nilpotente s'il existe $k \in \mathbb{N}$ tel que $A^k = 0$, et dans ce cas son indice de nilpotence est le plus petit tel k, qui vérifie alors $k \leq n$.

    Écrire une fonction **`IndiceNilpotence(A)`** prenant en argument une matrice carrée A, et renvoyant son indice de nilpotence, ou **`None`** si A n'est pas nilpotente. On cherchera à avoir la meilleure complexité possible (en fonction de n), que l'on précisera.

    Corrigé :

    Version basique en $O(n^4)$:

```
def IndiceNilpotence(A):
    n = len(A)
    Ak = A
    Z = [[0 for _ in range(n)] for _ in range(n)]
    if Ak == Z:
        return 1
    for k in range(2,n+1):
        Ak = Mult(Ak,A)
        if Ak == Z:
            return k
    return None
```

    Version avancée en $O(n^3 \log^2 n)$:

```
def IndiceNilpotence(A):
    n = len(A)
    Z = [[0 for _ in range(n)] for _ in range(n)]
    g = 0
    d = n
    while g < d - 1:
        m = (d+g)//2
        Am = QuickMatrixPower(A,m)
        if Am == Z:
            d = m
        else:
            g = m
    return d
```

    Version balèze en $O(n^3 \log n)$:

```
def IndiceNilpotence(A):
    n = len(A)
    D = {1 : A}
    k = 1
    Ak = A
    Z = [[0 for _ in range(n)] for _ in range(n)]
    if Ak == Z:
        return 1
    while Ak != Z and k <= n:
        k = 2*k
        Ak = Mult(Ak, Ak)
        D[k] = Ak
    if Ak != Z:
        return None
    g = k//2
    d = k
    k = k//2
    Ag = D[g]
    while g < d - 1:
        k = k//2
        m = g + k
        Am = Mult(Ag, D[k])
        if Am == Z:
            d = m
        else:
            g = m
            Ag = Am
    return d
```

Partie IV
Nombres élégants

On dit qu'un nombre écrit en base p (donc avec des chiffres de 0 à $p-1$) sur k chiffres est élégant si l'écart entre chaque chiffres consécutifs est d'au plus 1. Formellement, pour

$$n = \sum_{i=0}^{k-1} c_i\, p^i$$

n écrit en base p sur k chiffres est élégant si $\forall i \in [\![0, k-2]\!]$, $|c_{i+1} - c_i| \leq 1$.
On s'intéresse dans la suite au nombre de nombres élégants en base p sur k chiffres.

17. les écritures en base 5 123323434 et 0012243 sont-elles élégantes ?

    Corrigé :

    123323434 est élégant.

    0012243 n'est pas élégant, car un 2 est suivi d'un 4.

18. Écrire une fonction **`Ecriture(n,p,k)`** prenant en argument un entier naturel **`n`**, un entier **`p`** supérieur ou égal à 2 et un entier naturel **`k`**, et renvoyant l'écriture de **`n`** en base **`p`** sur **`k`**, sous forme de liste des chiffres, chiffre de poids fort en premier (on suppose qu'une telle écriture existe sur k chiffres).

    Par exemple, **`Ecriture(28,5,4)`** doit renvoyer **`[0, 1, 0, 3]`**

    On utilisera la méthode du cours pour l'écriture binaire, en utilisant des divisions euclidiennes par **`p`** plutôt que par 2.

    Corrigé :

```
def Ecriture(n, p, k):
    if k==0:
        return []
    L = Ecriture(n//p, p, k-1)
    L.append(n%p)
    return L
```

19. Écrire une fonction **`EstElegante(L)`** prenant en argument une liste **`L`** et testant si c'est une écriture élégante.

    Corrigé :

```
def est_elegante(L):
    n = len(L)
    for i in range(n-1):
        if abs(L[i] - L[i+1]) > 1:
            return False
    return True
```

20. En déduire une fonction **`NbElegant(p,k)`** dénombrant le nombre d'écritures élégantes en base **`p`** sur **`k`** chiffres.

    Corrigé :

```
def NbElegant(p, k):
    total = 0
    for n in range(p**k):
        if est_elegante(Ecriture(n, p, k)):
            total = total + 1
    return total
```

21. Déterminer la complexité de la fonction précédente, en fonction de **`p`** et **`k`**.

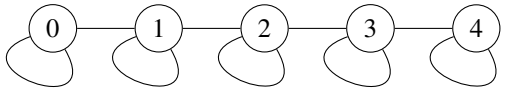
    Corrigé :

    Chaque passage dans le `for` réalise un appel à `Ecriture` en $O(k)$ et un appel à `est_elegante` également en $O(k)$. Il y a p^k passages, d'où une complexité totale en $O(kp^k)$.

    On cherche une autre façon de procéder à ce dénombrement, en traduisant le problème sous forme de graphe. Pour $p = 5$ et $k = 8$, pour construire un nombre élégant, on part d'un premier chiffre et on choisit les 7 suivants en respectant les règles suivantes :

    - 0 est suivi de 0 ou 1 ;
    - 1 est suivi de 0, 1 ou 2 ;
    - 2 est suivi de 1, 2 ou 3 ;
    - 3 est suivi de 2, 3 ou 4 ;
    - 4 est suivi de 3 ou 4.

    On construit donc le graphe dont les sommets sont les chiffres de 0 à 4, et tel qu'il y a une arête entre deux sommets si la différence entre les sommets vaut au plus 1. On obtient le graphe suivant :



    Un nombre élégant à 8 chiffres en base 5 correspond alors à une suite de 8 sommets reliés par des arêtes qui forment un chemin, c'est-à-dire à un chemin de longueur 7 dans ce graphe.

22. Écrire M_1 la matrice d'adjacence correspondant au graphe du problème des nombres élégants en base 5.

    Corrigé :

```
M1 = [
[1, 1, 0, 0, 0],
[1, 1, 1, 0, 0],
[0, 1, 1, 1, 0],
[0, 0, 1, 1, 1],
[0, 0, 0, 1, 1]]
```

23. Que dénombre le coefficient tout en haut à gauche de la matrice M_1^7 ?

Corrigé : Le nombre de chemins de longueur 7 allant de 0 à 0, c'est-à-dire le nombre d'écritures élégantes en base 5 sur 8 chiffres commençant et terminant par un 0.

24. Écrire une fonction **`MatrixElegant(p)`** qui renvoie la matrice correspondant au problème pour des nombres élégants en base p.

Corrigé :

```
def MatrixElegant(p):
    M = ID(p)
    for i in range(p-1):
        M[i][i+1]=1
        M[i+1][i]=1
    return M
```

25. En déduire une fonction **`NbElegant2(p,k)`** qui renvoie le nombre de nombres élégants écrits en base p sur k chiffres en exploitant la représentation du problème par un graphe.

Corrigé :

```
def SumMatrix(M):
    n = len(M)
    S = 0
    for i in range(n):
        for j in range(n):
            S += M[i][j]
    return S

def NbElegant2(p, k):
    M = MatrixElegant(p)
    M = QuickMatrixPower(M,k-1)
    return SumMatrix(M)
```

26. Déterminer la complexité de cette fonction.

Corrigé :

L'appel `MatrixElegant(p)` est en $O(p^2)$ (à cause de l'appel à `ID(p)`).

L'appel `QuickMatrixPower(M,k-1)` est en $O(p^3 \log k)$.

L'appel `SumMatrix(M)` est en $O(p^2)$.

La complexité totale est alors en $O(p^3 \log k)$ (bien meilleur que le $O(kp^k)$ de la méthode précédente).

27. Démontrer le théorème de la question 8. On pourra procéder par récurrence sur k.

Corrigé :

Initialisation : Pour $k = 1$, $M^k = M$ et il est vrai que la case d'indice i, j de M contient le nombre de chemins de longueur 1 (donc d'arcs) allant de i à j (ce nombre ne pouvant être que 0 ou 1).

Hérédite : Soit $k \in \mathbb{N}^*$ tel que la propriété est vraie au rang k. Soient $i, j \in [\![0, n-1]\!]$. On a $M^{k+1} = M^k M$, donc

$$[M^{k+1}]_{i,j} = \sum_{s=0}^{n-1} [M^k]_{i,s}[M]_{s,j} = \sum_{s \text{ voisin de } j} [M^k]_{i,s}$$

Or par hypothèse de récurrence, $[M^k]_{i,s}$ est le nombre de chemins de longueur k allant de i à s. La somme

$$\sum_{s \text{ voisin de } j} [M^k]_{i,s}$$

dénombre donc bien les chemins de longueur $k+1$ de i à j, en distinguant les cas selon le dernier sommet s avant j dans un tel chemin.