

Composition d'ITC 2

MP***

2026-02-13

Sommes sur un chemin dans une matrice

```
## Importations
import random

## Matrices pour les vérifications de programme
M1=[[5,1,2],[3,4,5]]
M2=[[random.randint(1,10**3)*random.random() for j in range(20)] for i in range (10)]

# Exemples de chemins dans M1
chem1=[(0,0),(0,1),(0,2),(1,2)]

# Test d'appartenance "in" à une liste
def estDans(val,liste):
    for el in liste:
        if el==val:
            return True
    return False
```

Préliminaires

Question 1

Pour une matrice de $M_{m,n}(\mathbb{R})$ on peut faire $m-1$ déplacements horizontaux et $n-1$ déplacements verticaux auquel il faut rajouter le premier élément de la liste.

```
def taille_chemins(mat:list[list[float]])->int:
    return len(mat) + len(mat[0]) - 1

taille_chemins(M1),taille_chemins(M2)
```

(4, 29)

Question 2

On doit vérifier la correction des premier et dernier élément, ainsi que la validité de la formule de récurrence. Si l'on arrive en fin de boucle c'est bien que la propriété est la bonne.

```
def chemin_coherent(chem:list[(int,int)],mat:list[list[float]])->bool:
    if chem[0] != (0, 0) or chem[-1] != (len(mat)-1,len(mat[0])-1) :
        return False
    for i in range(len(chem)-1):
        if chem[i+1] not in [(chem[i][0]+1, chem[i][1]), (chem[i][0], chem[i][1]+1)]:
            return False
    return True

chemin_coherent(chem1,M1),chemin_coherent(chem1,M2)
```

(True, False)

Question 3

Aller à droite revient à incrémenter j, d'où l'affectation du booléen à la ligne 5.

```
def chemin_itineraire(chem:list[(int,int)])->list[bool]:
    res = []
    y = 1
    for i in range(len(chem)-1):
        res.append(chem[i+1][1] == chem[i][1]+1)
    return res

chemin_itineraire(chem1)
```

[True, True, False]

Question 4

On utilise ici le transtypage natif en python : pos[ori] est interprété comme pos[0] si ori=True, ce qui revient à incrémenter le numéro de ligne, et symétriquement pour le numéro de colonne.

```
def itineraire_chemin(itin:list[bool])>list[(int,int)]:
    res = [(0, 0)]
    pos = [0, 0]
    for ori in itin:
        pos[ori] += 1
        res.append((pos[0], pos[1]))
    return res

itineraire_chemin(chemin_itineraire(chem1)),chem1,itineraire_chemin(chemin_itineraire(chem1))==chem1
```

([(0, 0), (0, 1), (0, 2), (1, 2)], [(0, 0), (0, 1), (0, 2), (1, 2)], True)

Question 5

On choisit de faire appel à la fonction chemin_coherent précédemment écrite.

```
def valide_itineraire(mat:list[list[float]],itin:list[bool])>bool:
    chem=itineraire_chemin(itin)
    return chemin_coherent(chem,mat)

valide_itineraire(M1,chemin_itineraire(chem1))
```

True

Exploration exhaustive

Question 6

Comme son nom l'indique, ce programme calcule la valeur maximale d'un chemin dans la matrice par force brute, c'est à dire en testant tous les chemins. Le programme récursifs montre le principe : la somme est initialisée à la valeur de la case en haut à gauche(110). Les appels récursifs correspondent aux situations suivantes :

- l.6 : arrivé à la dernière ligne, on ne peut progresser que selon une colonne
- l.8 : arrivé à la dernière colonne, on ne peut progresser que selon une ligne
- l.9 : on choisit d'avancer ou de descendre suivant ce qui permet de maximiser la somme de la partie aval de la matrice

Question 7

Tous les chemins sont testés, et l'on peut se poser $m - 1$ fois la question de descendre parmi $n + m - 2$ étapes. La complexité est donc en $\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1}$, soit une **complexité exponentielle** en les dimensions de la matrice.

C'est beaucoup trop et en pratique l'utilisation d'une récursion risque fort de déclencher un dépassement de pile pour des matrices que l'on pourrait rencontrer comme arguments.

La complexité en espace est également élevée (toujours en rapport avec la pile d'appels), donc à ce stade pour éviter que Python cause un problème on peut simuler la pile soi-même en ajoutant quelques lignes, ce qui permet de se passer de la récursion.

Algorithme glouton v.1

Question 8

Pour suivre les indications de l'énoncé, on doit conserver l'appel récursif mais sans chercher à trouver lequel des deux chemins est maximal. Cela se traduit par la modification des lignes 7 (on suit le max des deux valeurs) et 9 (pas d'hésitation dans l'appel récursif).

```
def somme_maximale_glouton1(mat:list[list[ float ]])>->float :
def somme (i:int , j:int)>->float :
    if i == len(mat)-1 and j == len(mat[0])-1:
        return mat[i][j]
    if i == len(mat)-1:
        return mat[i][j] + somme (i, j+1)
    if j == len(mat[0])-1:
        return mat[i][j] + somme(i+1, j)
    # lignes modifiées
    if mat[i+1][j] >= mat[i][j+1]:
        return mat[i][j] + somme(i+1, j)
    else:
        return mat[i][j] + somme(i, j+1)
return somme (0, 0)

somme_maximale_glouton1(M2)
```

7982.774615308495

Question 9

Cette fois-ci un seul chemin est considéré, donc la complexité passe en $\mathcal{O}(m + n)$, toujours avec les mêmes notations.

Question 10

Soit la matrice suivante :

$$M = \begin{pmatrix} 1 & 2 \\ 1 & 0 \\ 1 & 0 \\ \dots & \dots \\ 1 & 0 \end{pmatrix}$$

L'algorithme glouton fera d'abord aller à droite, puis il faudra forcément descendre jusqu'à la dernière, pour une somme de 3, alors que descendre constamment donne une somme de n .

Algorithme glouton v.2

Question 11

On choisit de procéder en deux temps. On commence par écrire un programme `max_mat` qui détermine la position de la valeur maximale d'une matrice. Il faut veiller à exclure le cas des coins supérieurs et inférieurs

```
def max_mat(mat:list[list[ float ]])->(int,int):
    imax,jmax = 0, 1
    m,n=len(mat),len(mat[0])
    for i in range(m):
        for j in range(n):
            if (i,j) != (0,0) and (i,j) != (m-1,n-1) and mat[i][j] > mat[imax][jmax]:
                imax,jmax=i,j
    return imax,jmax

max_mat(M1),max_mat(M2)
```

((1, 1), (8, 6))

On définit ensuite deux sous-matrices qui ont pour coin inférieur droit et supérieur gauche le coefficient précédemment déterminé.

On applique à chacune de ces sous-matrices le programme précédent.

```
def somme_maximale_glouton2(mat)->int:
    imax, jmax = max_mat(mat)
    m,n=len(mat),len(mat[0])
    mat1 = [[mat[i][j] for j in range(jmax+1)] for i in range(imax+1)]
    mat2 = [[mat[i][j] for j in range(jmax, n)] for i in range(imax, m)]
    somme1 = somme_maximale_glouton1(mat1)
    somme2 = somme_maximale_glouton1(mat2)
    return somme1 + somme2 - mat[imax][jmax]

somme_maximale_glouton2(M2)
```

7982.774615308496

Question 12

Le calcul du maximum de la matrice est en $\mathcal{O}(mn)$. Par la suite, on construit avec la même complexité les deux sous-matrices.

La consommation en mémoire n'était en pratique pas nécessaire, mais adapter la fonction `somme_maximale_glouton1`

aurait nécessité plus de temps et d'énergie.

Les deux appels `somme_maximale_glouton1` sont de complexité $\mathcal{O}(m + n)$ d'après ce qui précède, donc on a une complexité totale en $\boxed{\mathcal{O}(mn)}$

Question 13

Le contre-exemple précédent reste valable, car il n'y a qu'un chemin possible une fois le maximum calculé et considéré comme un point de passage nécessaire.

On pourrait par ailleurs trouver des cas où un des algorithmes gloutons donne une réponse strictement meilleure que l'autre et vice-versa.

Programmation dynamique

Question 14

Le principe de l'algorithme est de déterminer pour chaque case de la matrice quelle est la plus grande valeur d'un chemin qui terminerait dans cette case. Cette valeur s'obtient en additionnant l'entier stocké dans la case et la plus grande valeur d'un chemin qui terminerait dans la case du haut (si elle existe) ou de gauche (si elle existe), en prenant le maximum quand il y a deux possibilités.

Trois éléments sont donc à initialiser : la première case, ainsi que les premières ligne et colonne.

$$\begin{cases} s_{0,0} &= m_{0,0} \\ s_{i,0} &= m_{i,0} + s_{i-1,0} \\ s_{0,j} &= m_{0,j} + s_{0,j-1} \end{cases}$$

Pour le cas général, on a vu que :

$$s_{i,j} = m_{i,j} + \max(s_{i-1,j}, s_{i,j-1})$$

Question 15

```
def maxsomme(mat:list[list[float]])->float:
    nl,nc = len(mat),len(mat[0])
    maxsomm = [[None for _ in range(nc)] for _ in range(nl)]
    maxsomm[0][0] = mat[0][0]
    for i in range(1, nl):
        maxsomm[i][0] = maxsomm[i-1][0] + mat[i][0]
    for i in range(1, nc):
        maxsomm[0][i] = maxsomm[0][i-1] + mat[0][i]
    for i in range(1, nl):
        for j in range(1, nc):
            maxsomm[i][j] = mat[i][j] + max(maxsomm[i-1][j], maxsomm[i][j-1])
    return maxsomm[-1][-1]
```

`maxsomme(M2)`

11811.878983288692

Question 16

Chaque case n'est considérée qu'une fois avec un traitement constant, la complexité est donc linéaire en le nombre total d'éléments de la matrice, soit $\mathcal{O}(m \times n)$.

La complexité en espace est également de l'ordre de la taille de la matrice.

Problème 2 : Un peu de rugby

Mini-base de vérification

```
joueurs={"France":["Lucu","Barré","Dupont","Ramos"],"Angleterre":["George","Ford"]}
cr1=[["Dupont","essai","2"],["Ramos","transformation","3"]]
cr2=[["Ramos","transformation","3"],["Dupont","essai","2"]]
```

Question 17

Commençons par remarquer que les transformations ne pouvant exister sans essai, les 3 possibilités d'incrément du score sont +3,+5 et +7 (sympatiquement tous premiers).

On peut donc proposer une approche *force brute* : on écrit $n = 7m + 5p + 3q$ et l'on teste tous les triplets m, p, q possibles.

```
def degenerescence_score(n:int)->int:
    res=0
    m,p,q=n//7+1,n//5+1,n//3+1
    for i in range(m):
        for j in range(p):
            for k in range(q):
                if 7*i+5*j+3*k==n:
                    res+=1
    return res

degenerescence_score(50)
```

16

Un peu plus élégamment, on peut réaliser deux boucles imbriquées où l'on retire au score les multiples de 7 et 5 successifs et l'on teste si le résultat est un multiple de 3.

```
def degenerescence_score(n:int)->int:
    res = 0
    for n1 in range(n, -1, -7):
        for n2 in range(n1, -1, -5):
            if n2 % 3 == 0:
                res+=1
    return res

degenerescence_score(50)
```

16

Question 18

```
def trouve_equipe(joueur:str)->str :
    for equipe in joueurs:
        if estDans(joueur,joueurs[equipe]):
            return equipe
    return False

print(trouve_equipe("Dupont"),trouve_equipe("Lomu"))
```

France False

La **complexité est linéaire** en le nombre total de joueurs, car il s'agit de parcourir des listes. Il aurait pu être intéressant d'utiliser une structure d'ensemble / de dictionnaire pour les valeurs associées à chaque équipe, pour accélérer virtuellement la recherche.

Question 19

Il n'y a ici qu'à parcourir les listes et stocker les éléments au fil de leur lecture.

```
def construit_dict_equipe()->dict:
    dic_joueurs = dict()
    for equipe in joueurs:
        for joueur in joueurs[equipe]:
            dic_joueurs[joueur] = equipe
    return dic_joueurs

dic_joueurs=construit_dict_equipe()
dic_joueurs
```

```
{'Lucu': 'France',
 'Barré': 'France',
 'Dupont': 'France',
 'Ramos': 'France',
 'George': 'Angleterre',
 'Ford': 'Angleterre'}
```

Question 20

Sans indication particulière de l'énoncé, on choisit ici le tri insertion (en se disant que les listes doivent normalement être quasi-triées).

```
def ordonne_cr(cr:list[list[str,float]]):
    for indice,even in enumerate(cr):
        valeur=even[2]
        j=indice
        while j>0 and valeur<cr[j-1][2]:
            cr[j]=cr[j-1]
            j=j-1
        cr[j]=even

# print(cr2)
# ordonne_cr(cr2)
# print(cr2)
```

Question 21

Les causes d'erreurs sont liées aux transformations :

- la première entrée est une transformation
- une transformation est tapée par un joueur de l'équipe adverse
- une transformation n'est pas précédée d'un essai

```
def coherence(cr:list[list[str,str,float]])->bool :
    dic_joueurs=construit_dict_equipe()
    for i in range(len(cr)):
        if cr[i][1] == "transformation":
            if i == 0 or dic_joueurs[cr[i][0]] != dic_joueurs[cr[i-1][0]] or cr[i-1][1] != "essai":
                return False
    return True

print(coherence(cr2))
ordonne_cr(cr2)
print(cr2,coherence(cr2))
```

False

```
[['Dupont', 'essai', '2'], ['Ramos', 'transformation', '3']] True
```

Question 22

Il suffit ici de parcourir tout le compte-rendu est d'incrémenter les scores des deux équipes au fur et à mesure de la lecture. On choisit de créer un dictionnaire des points pour alléger l'écriture.

```
def resultat(cr, e1, e2):
    dic_joueurs=construit_dict_equipe()
    points_par_action = { "essai" : 5, "transformation" : 2, "drop" : 3, "penalite" : 3 }
    resultat = { e1 : 0, e2 : 0 }
    for j, t, _ in cr:
        resultat[dic_joueurs[j]] += points_par_action[t]
    return resultat[e1], resultat[e2]

resultat(cr2,"France","Afrique du Sud")
```

(7, 0)

Question 23

Il suffit ici de parcourir la table Equipe.

```
SELECT eq_nom
FROM Equipe
```

Question 24

On utilise ici des fonctions d'agrégations sans nécessité de faire de regroupement.

```
SELECT MIN(de_tempsJeu), MAX(de_tempsJeu)
FROM Deroulement
```

Question 25

On réalise ici une jointure selon l'identifiant de l'équipe pour déterminer tous les joueurs.

```
SELECT jo_nom
FROM Equipe
JOIN Joueur
ON Equipe.eq_id = Joueur.eq_id
WHERE eq_nom = "France"
```


Question 26

On repart de la jointure précédente, en ajoutant la table `Deroulement` qui permet d'associer aux joueurs les points qu'ils ont inscrit.

```
SELECT DISTINCT jo_nom
  FROM Equipe
JOIN Joueur
  ON Equipe.eq_id = Joueur.eq_id
JOIN Deroulement
  ON Joueur.jo_id = Deroulement.jo_id
WHERE eq_nom = "France"
  AND de_evenement = "essai"
```

Question 27

On réalise ici une jointure pour déterminer tous les pays ayant joué contre la France. On utilise ensuite une fonction d'agrégation pour déterminer les noms des pays intéressants.

```
SELECT Eq2.eq_nom, AVG(pa_score1) AS moy
  FROM Partie
JOIN Equipe AS Eq1
  ON Eq1.eq_id = pa_equipe1
JOIN Equipe AS Eq2
  ON Eq2.eq_id = pa_equipe2
WHERE Eq1.eq_nom = "France"
GROUP BY pa_equipe2
HAVING moy > 20
```