CORRIGÉ DU DS 2 (Sans EX2, partie V)

le 18/12/2025 – Durée : 2h

- L'usage de la calculatrice n'est pas autorisé.
- La clarté et la précision des raisonnements interviendront pour une grande part dans la notation. La présentation (en particulier les indentations) est également essentielle.
- Le résultat d'une question peut être admis afin de traiter une question suivante ; une fonction peut être utilisée dans la suite d'un exercice même si elle n'a pas été écrite.
- Le barème tiendra compte de la longueur du devoir.

Exercice 1

1.
```
SELECT code_pays FROM Pays WHERE nom = 'France'
```

2.
```
SELECT Pays.nom
FROM Pays JOIN Inclusion
          ON Pays.code_pays = Inclusion.code_pays
WHERE Inclusion.continent = 'Europe'
```

3.
```
SELECT SUM(longueur)
FROM Frontieres
WHERE code_pays1 = 'F' OR code_pays2 = 'F'
```

4.
```
SELECT P.nom
FROM Frontieres AS Fr JOIN Pays AS P
                      ON Fr.code_pays2 = P.code_pays
WHERE Fr.code_pays1 = 'F'
UNION
SELECT P.nom
FROM Frontieres AS Fr JOIN Pays AS P
                      ON Fr.code_pays1 = P.code_pays
WHERE Fr.code_pays2 = 'F'
```

5. Écrire une requête SQL permettant de récupérer les noms des 10 pays les plus peuplés, par ordre décroissant.

```
SELECT nom
FROM Pays
ORDER BY population DESC LIMIT 10
```

6. Écrire une requête SQL permettant de récupérer le nom du troisième pays le plus peuplé.

```
SELECT nom
FROM Pays
ORDER BY population DESC LIMIT 1 OFFSET 3
```

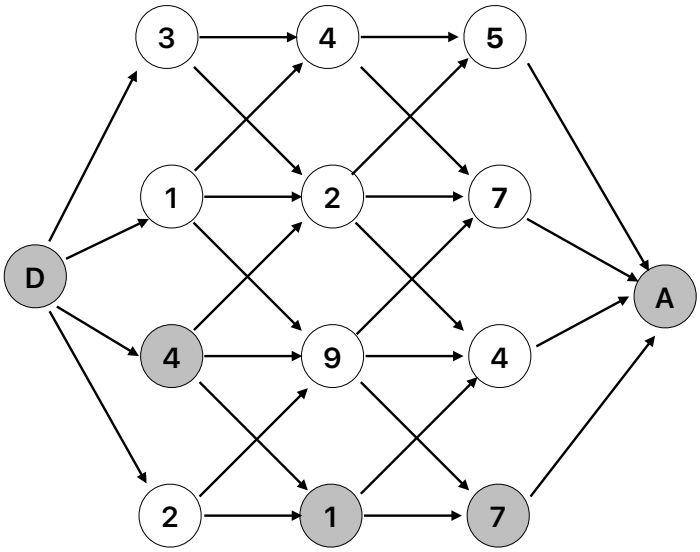
7. Écrire une requête SQL permettant de récupérer les noms des continents avec pour chacun sa population. On supposera dans cette question que chaque pays appartient à un seul continent.

```
SELECT I.continent , SUM(P.population)
FROM Pays AS P JOIN Inclusion AS I
               ON P.code_pays = I.code_pays
GROUP BY I.continent
```

Exercice 2

I. Présentation

II. Représentation des données du problème



1. Le score de ce chemin est de $4 + 1 + 7 = 12$.

2.
```
def scoreParcours(Lgrille,lparcours):
    p = len(lparcours) # on peut aussi utiliser p = len(Lgrille[0])
    s = 0
    for i in range(p):
        s += Lgrille[lparcours[i]][i]
    return s
```

3. Pour le premier disque on a n choix. Ensuite à chacune des $p-1$ étapes suivantes on a au moins 2 choix. Le nombre de parcours est donc minoré par $n \times 2^{p-1}$.
On obtient ainsi une complexité exponentielle (en la composante p).

III. Stratégie gloutonne

4.
```
def scoreMaxGlouton(L):
    n , p = len(L) , len(L[0])
    score , lparcours = 0 , []
    # On détermine la ligne contenant le poids max dans la première colonne
    iMax , poidsMax = 0 , L[0][0]
    for i in range(1,n):
        if L[i][0] > poidsMax:
            iMax , poidsMax = i , L[i][0]

    score += poidsMax
    lparcours.append(iMax)
    # On parcourt ensuite la grille de gauche à droite
    # Si à l'étape j-1 on se trouve au niveau i,
    # à l'étape j on pourra se trouver au niveau i, i-1 ou i+1
    # (si ces niveaux sont valables)
    for j in range(1,p):
        iMaxSuivant , poidsMax = iMax , L[iMax][j]
        if iMax - 1 >= 0 and L[iMax - 1][j] > poidsMax:
            iMaxSuivant , poidsMax = iMax-1 , L[iMax-1][j]
        if iMax + 1 < n and L[iMax + 1][j] > poidsMax:
            iMaxSuivant , poidsMax = iMax+1 , L[iMax+1][j]
        iMax = iMaxSuivant
        score += poidsMax
        lparcours.append(iMax)
    return score , lparcours
```

5. Par cette stratégie on obtient dans l'exemple de la figure 1 le score de 21 (avec le parcours représenté par `[0, 0, 0, 0, 1]`).
   On constate que le parcours représenté par `[0, 0, 1, 2, 2]` donne un score de 23. On a donc pas obtenu le score maximal par la stratégie gloutonne.

6. La première partie de la fonction consiste à trouver un maximum parmi n nombre. Cela nécessite de l'ordre de n opérations.
   Ensuite on effectue une boucle de longueur $p-1$ avec à chaque itération un nombre d'opérations borné par une constante (car on recherche un maximum parmi 3 valeurs).
   La complexité de cette fonction est donc en $\prime(n+p)$.

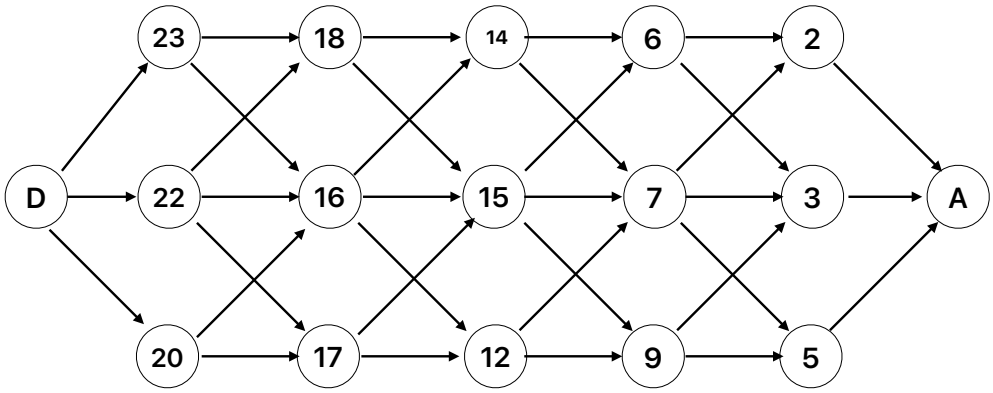
IV. Algorithme récursif naïf

7.
```
def scoreMaxRecPart(L,i,j):
    n , p = len(L) , len(L[0])
    if j == p-1:
        return  L[i][j]
    else:
        # on recherche le score max parmi les trois (ou 2) successeurs de (i,j)
        # en effectuant un appel récursif à cette fonction pour chaque successeur
        scoreMax = scoreMaxRecPart(L,i,j+1)
        if i-1 >= 0:
            score = scoreMaxRecPart(L,i-1,j+1)
            if score > scoreMax:
                scoreMax = score
        if i+1 < n:
            score = scoreMaxRecPart(L,i+1,j+1)
            if score > scoreMax:
                scoreMax = score
        return scoreMax + L[i][j]
```

```
def scoreMaxRec(L):
    n , p = len(L) , len(L[0])
    sMax = scoreMaxRecPart(L,0,0)
    for i in range(1,n):
        score = scoreMaxRecPart(L,i,0)
        if score > sMax:
            sMax = score
    return sMax
```

8. La boucle externe de `scoreMaxRec(L)` nécessite n appels à la fonction `scoreMaxRecPart(L,i,0)`.
Notons $C(j)$ le nombre d'opérations nécessaires (pour i quelconque) au calcul de `scoreMaxRecPart(L,i,j)`.
L'appel à `scoreMaxRecPart(L,i,j)` nécessite au moins 2 appels au niveau $j+1$. On en déduit que $C(j) \geq 2C(j+1)$. Ainsi $C(0)$ va être de l'ordre de 2^{p-1}. On retombe donc sur une complexité minorée par un fonction de l'ordre de 2^{p-1}.
Réponse plus sommaire : la résolution se fera en $p-1$ étapes avec à chaque étape au moins 2 appels récursifs. D'où une complexité minorée par une fonction de l'ordre de 2^p.

V. Résolution par mémoïzation



10.

11.
```
def grilleZeros(Lgrille):
    n , p = len(Lgrille) , len(Lgrille[0])
    return [[0 for j in range(p)] for i in range(n)]
```

12.
```
def remplitLscore(L,Lscore,i,j):
    n , p = len(L) , len(L[0])
    if Lscore[i][j] == 0:
        if j == p-1:
            Lscore[i][j] = L[i][j]
        else :
            remplitLscore(L,Lscore,i,j+1)
            scoreMax = Lscore[i][j+1]
            if i-1 >= 0:
                remplitLscore(L,Lscore,i-1,j+1)
                if Lscore[i-1][j+1] > scoreMax:
                    scoreMax = Lscore[i-1][j+1]
            if i+1 < n :
                remplitLscore(L,Lscore,i+1,j+1)
                if Lscore[i+1][j+1] > scoreMax:
                    scoreMax = Lscore[i+1][j+1]
            Lscore[i][j] = scoreMax + L[i][j]
```

13.
```
def scoreMaxMemo(L):
    n , p = len(L) , len(L[0])
    Lscore = grilleZeros(L)
    remplitLscore(Lgrille,Lscore,0,0)
    scoreMax = Lscore[0][0]
    for i in range(1,n):
        remplitLscore(Lgrille,Lscore,i,0)
        if Lscore[i][0] > scoreMax:
            scoreMax = Lscore[i][0]
    return scoreMax
```