Correction 26-NSIJ1AN1

Exercice 1

Partie A

1.
```
class Grille:
    def __init__(self):
        self.grille = [[0 for _ in range(7)] for _ in range(6)]
```

2.
```
    def joue(self, colonne, joueur):
        ligne = 5 # on part de la rangée la plus basse
        while ligne!=-1 and self.grille[ligne][colonne]!=0:
            ligne -= 1
        if ligne != -1:
            self.grille[ligne][colonne] = joueur
            return True
        else:
            return False
```

3.
```
jeu1 = Grille()
jeu1.joue(2,1)
jeu1.joue(3,2)
jeu1.joue(3,1)
```

4. On voit qu'il y a 7 alignements possibles pour la case $(5, 3)$ occupée par le joueur 2, 5 alignements pour la case $(5, 2)$ et 10 alignements pour la case $(4, 3)$ occupées par le joueur 1. On a donc un score de $7 - 5 - 10 = -8$.

5.
```
    def score(self):
        score = 0
        for ligne in range(6):
            for colonne in range(7):
                joueur = self.grille[ligne][colonne]
                if joueur != 0:
                    alignements = valeur_case(ligne, colonne)
                    if joueur == 1:
                        score -= alignements
                    else:
                        score += alignements
        return score
```

Partie B

6.
```
class Noeud:
    def __init__(self, colonne):
        self.colonne = colonne
        self.score = 0
        self.suivants = []
```

7.
```
    def colonne_score_min(self):
        colonne = self.suivants[0].colonne
        score = self.suivants[0].score
        for fils in self.suivants:
            if fils.score < score:
                score = fils.score
                colonne = fils.colonne
        return colonne, score
```

8.
```python
def calcule_score(self, niveau, joueur, grille):
    g = grille.gagnant()
    if g == 1:
        self.score = -(100 + 10 * (niveau_max - niveau))
    elif g == 2:
        self.score = 100 + 10 * (niveau_max - niveau)
    elif niveau == niveau_max:
        self.score = grille.score()
    else:
        for colonne in range(7):
            grille2 = grille.copie_grille()
            if grille2.joue(colonne, joueur):
                nouveau_noeud = Noeud(colonne)
                self.suivants.append(nouveau_noeud)
                nouveau_noeud.calcule_score(niveau + 1, 3 - joueur, grille2)
        if joueur == 1:
            self.score = self.colonne_score_min()[1]
        else:
            self.score = self.colonne_score_max()[1]
```

9. (Comme il y a $6 \times 7 = 42$ cases dans la grille, on a bien `niveau_max` ≤ 42) Comme l'algorithme explore toutes les possibilités de jeu sur une profondeur de `niveau_max`, le nombre de noeuds explorés est de l'ordre de $O(7^{\text{niveau\_max}})$, ce qui est très grand pour des valeurs de `niveau_max` supérieures à 5 ou 6, donc d'autant plus pour 42. Ici, pour simplifier, on a considéré que l'arbre de jeu était 7-aire complet, ce qui n'est pas le cas en réalité, mais cela donne une idée de l'ordre de grandeur du nombre de noeuds explorés.

Partie C

10.
```python
def choisir_coup(grille, joueur):
    racine = Noeud(-1)  # la colonne n'est pas utilisée pour la racine
    racine.calcule_score(0, joueur, grille)
    if joueur == 1:
        return racine.colonne_score_min()[0]
    else:
        return racine.colonne_score_max()[0]
```

Exercice 2

1. Comme 29 bits sont réservés à la partie "réseau", il reste $32 - 29 = 3$ bits pour la partie "machine". Avec 3 bits, on peut représenter $2^3 = 8$ adresses différentes, mais comme l'adresse de réseau et l'adresse de diffusion ne sont pas attribuables, il ne reste que $8 - 2 = 6$ adresses disponibles attribuables. **Pour son interface avec le réseau Administration**, le routeur E peut donc avoir pour adresse IP 10.42.0.81, par exemple. En effet, on a :

10.42.0.80	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
255.255.255.248	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
10.42.0.81	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1

2. Comme les trois premier octets des réseaux présentés sont tous identiques et que les masques sont tous supérieurs à $3 \times 8 = 24$, on ne s'intéresse qu'au dernier octet. En notation binaire, on a (en bleu les adresses IP, en orange les masques) :

70	0	1	0	0	0	1	1	0
96	0	1	1	0	0	0	0	0
224	1	1	1	0	0	0	0	0
64	0	1	0	0	0	0	0	0
240	1	1	1	1	0	0	0	0
32	0	0	1	0	0	0	0	0
224	1	1	1	0	0	0	0	0
88	0	1	0	1	1	0	0	0
248	1	1	1	1	1	0	0	0
128	1	0	0	0	0	0	0	0
248	1	1	1	1	1	0	0	0

La machine correspondante se situe donc dans le réseau d'adresse IP 10.42.0.64/28, sont la **Salle Gaming Online**.

3. Dans le réseau d'adresse IP 10.42.0.16/30, le routeur C a pour adresse IP 10.42.0.17 et le routeur F 10.42.0.18.

   Dans le réseau d'adresse IP 10.42.0.12/30, le routeur C a pour adresse IP 10.42.0.13 et le routeur D 10.42.0.14.

4.

Routeur C		
Réseau	Passerelle	Nombre de sauts
DMZ	10.42.0.1	1
Gaming Online	connecté	0
Internet	10.42.0.1	2
Gaming VR	10.42.0.18	1
Administration	10.42.0.1 ou 10.42.0.14	2
Application	10.42.0.6	1
SGBD	10.42.0.14	2

5. On a les coût suivants :
   - 100 Mb/s : $\frac{10^{10}}{100\times10^{6}} = 100$,
   - 1 Gb/s : $\frac{10^{10}}{1\times10^{9}} = 10$,
   - 10 Gb/s : $\frac{10^{10}}{10\times10^{9}} = 1$.
6. L'information peut passer par A - B - E - D, ou encore par A - B - C - D, pour un coût de $1 + 10 + 1 = 12$ dans les deux cas.
7. Les données d'un paquet TCP sont contenues dans un paquet IP.
8. Ici, le but du routeur est de traiter les paquets selon leur ordre d'arrivée, ce qui correspond à la politique de routage "First In, First Out" (FIFO), qu'on retrouve bien dans la structure de données "file", contrairement à la politique "Last In, First Out" (LIFO) qui correspond à la structure de données "pile".
9. ```
   class Routeur_DROP_TAIL:
       def __init__(self, t_max):
           self.f = cree_file()
           self.t_max = t_max
           self.t = 0
   ```
   ```

10.
```
def recoit(self, p):
    if self.t < self.t_max:
        enfile(self.f, p)
        self.t = self.t + 1
        return True
    return False
```

11.
```
def recoit(self, p):
    if self.t < self.t_min:
        enfile(self.f, p)
        self.t = self.t + 1
        return True
    elif self.t_min <= self.t < self.t_max:
        if self.tirage_au_sort():
            enfile(self.f, p)
            self.t = self.t + 1
            return True
    return False
```

Exercice 3

Partie A

1. On n'a pas choisit ce numéro parce qu'il y peut y avoir 2 imeubles différents avec le même numéro de rue, mais dans des rues différentes.

2.
```
SELECT id_immeuble FROM immeuble
WHERE rue_immeuble = "la mer"
ORDER BY id_immeuble;
```

3.
```
SELECT id_appart FROM appartement JOIN immeuble
ON appartement.id_immeuble = immeuble.id_immeuble
WHERE id_immeuble = 16
AND etage_appart > 4;
```

4. Il faut d'abord supprimer les appartements de l'immeuble 16, puis supprimer l'immeuble lui-même, sinon on ne pourra pas supprimer l'immeuble à cause de la contrainte d'intégrité référentielle (les valeurs des clés étrangères doivent correspondre à des clés primaires existantes).

5.
```
INSERT INTO appartement VALUES (140, 6, 13, 'Turing');
```

6.
```
UPDATE appartement
SET prix_appart = prix_appart * 2
WHERE id_appart = 603;
```

7.
```
SELECT MAX(prix_appart) FROM appartement JOIN immeuble
ON appartement.id_immeuble = immeuble.id_immeuble
WHERE rue_immeuble = "la mer";
```

Partie B

8. On a : `[3, 8]`, `[3, 5]`, `[1, 8]`, `[1, 2]`, `[1, 5]`, `[2, 5]`.

9. La plus longue sous-séquence strictement croissante de `L2` est `[1, 2, 5]`.

10.
```
def strict_croissante(seq):
    for i in range(len(seq) - 1):
        if seq[i] >= seq[i + 1]:
            return False
    return True
```

11.
```python
def llsc_fin(tab, i):
    if i == 0:
        return 1
    max_len = 1
    for j in range(i):
        if tab[j] < tab[i]:
            max_len = max(max_len, llsc_fin(tab, j) + 1)
    return max_len
```

12.
```python
def llsc_dyn(tab):
    n = len(tab)
    dyn = [1] * n
    for i in range(1, n):
        for j in range(i):
            if tab[j] < tab[i]:
                dyn[i] = max(dyn[i], dyn[j] + 1)
    return max(dyn)
```

13. La version par programmation dynamique est plus efficace que la version récursive, car elle évite les calculs redondants en stockant les résultats intermédiaires dans un tableau. La complexité de la version récursive est exponentielle, tandis que la complexité de la version par programmation dynamique est quadratique !