Correction 26-NSIJ1AN1

Exercice 1

1. Comme $16 = 8 \times 2$, les 16 premiers bits, c'est à dire les deux premiers octets, de l'adresse IP de PC C01 correspondent à l'adresse du réseau. L'adresse du réseau C est donc 172.16.0.0/16.
2. Comme un octet constitué de 8 bits à 1 représente l'entier 255 en décimal, l'adresse de diffusion du réseau C est 172.16.255.255/16.
3. Il y a $32 - 16 = 16$ bits réservés pour la partie machine de l'adresse IP. Il y a donc $2^{16} - 2 = 65534$ adresses IP disponibles pour des machines du réseau C.
4.

Destination	Passe par	Nombre de sauts
R2	R3	2
R3	R3	1
R4	R4	1
R5	R4	2

5. Le chemin suivi sera : $\text{PC A01} \rightarrow \text{Switch 1} \rightarrow \text{R1} \rightarrow \text{R3} \rightarrow \text{R2} \rightarrow \text{Switch B} \rightarrow \text{PC B01}$.
6. La nouvelle route serait : $\text{R1} \rightarrow \text{R4} \rightarrow \text{R5} \rightarrow \text{R2}$.
7.

Type de connexion	Débit (bit/s)	Coût
Ethernet	10^7	10^8/10^7=10
Fast Ethernet	10^8	10^8/10^8=1
Fibre	10^9	10^8/10^9=0,1

8. On a troi chemins possibles pour aller de R1 à R4 :
   - $\text{R1} \rightarrow \text{R4}$, de coût 10
   - $\text{R1} \rightarrow \text{R3} \rightarrow \text{R4}$, de coût 0,1 + 1 = 1,1
   - $\text{R1} \rightarrow \text{R3} \rightarrow \text{R2} \rightarrow \text{R5} \rightarrow \text{R4}$, de coût 0,1 + 10 + 1 + 1 = 12,1

   Le chemin choisi par le protocole OSPF est donc $\text{R1} \rightarrow \text{R3} \rightarrow \text{R4}$.

   On pouvait aussi appliquer un algorithme de Dijkstra, mais c'est plus long et on a plus de chances de se tromper.
9. Dans la fonction `ajouter_route`, il n'y a aucun test de précondition fait pour vérifier si la taille de `liste_routes` est inférieure à `MAX_ROUTES`. Il faudrait par exemple rajouter, entre les lignes 4 et 5, le code suivant :

```
assert len(liste_routes) < MAX_ROUTES, "Nombre maximum de routes atteint"
```

10.

```
class Routage:
    def __init__(self, capacite = 5):
        self.capacite = capacite
        self.routes = []

    def ajouter(self, route):
        if len(self.routes) < self.capacite:
            self.routes.append(route)
        else:
            print("Nombre maximum de routes atteint")
```

11.
```python
def afficher(self):
    for route in self.routes:
        for r in route:
            print(r)
        print("---")
```

Exercice 2

Partie A

1. La valeur est `10`.
2.
```python
valeurs = [k for k in range(1, 17)]
# ou
# valeurs = [k + 1 for k in range(16)]
```
3. Le tri étant en place, il suffit de faire :

```python
shuffle(valeurs)
```
4.
```python
def en_grille(valeurs, n):
    assert len(valeurs) == n * n
    grille = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            grille[i][j] = valeurs[j * n + i]
    return grille
```
5.
```python
def en_grille(valeurs, n):
    assert len(valeurs) == n * n
    grille = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            grille[i][j] = valeurs[i * n + j] # inversion de i et j
    return grille
```

Partie B

6. La liste aplatie est `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 15, 14]`. On voit :
   - qu'il y a trois couples de valeurs inversées : (14, 15), (15, 16) et (14, 16), c'est à dire trois inversions de couples d'indices (13, 14), (13, 15) et (14, 15) dans la liste aplatie.
   - que la distance à calculer vaut `2` (`0`lignes et `2`colonnes).

   La somme du nombre d'inversions et de la distance vaut donc `3 + 2 = 5`, qui est un nombre impair. La grille n'est donc pas résoluble.
7.
```python
assert compte_inversion([2, 1, 4, 3]) == 2
```

8.
```python
def compte_inversion(valeurs):
    total = 0
    for i in range(len(valeurs)):
        for j in range(len(valeurs)):
            if i < j and valeurs[i] > valeurs[j]:
                total += 1
    return total
# ou
# def compte_inversion(valeurs):
#     total = 0
#     for i in range(len(valeurs)):
#         for j in range(i + 1, len(valeurs)):
#             if valeurs[i] > valeurs[j]:
#                 total += 1
#     return total
```

9.
```python
def distance_tuile_vide(grille, n):
    num_tuile_vide = n * n
    for i in range(n):
        for j in range(n):
            if grille[i][j] == num_tuile_vide:
                return n - 1 - i + n - 1 - j
```

10.
```python
def est_resoluble(valeurs, n):
    grille = en_grille(valeurs, n)
    inv = compte_inversion(valeurs)
    dis = distance_tuile_vide(grille, n)
    return (inv + dis) % 2 == 0
```

Partie C

11. Une valeur possible est `[5, 8, 10, 13]`. En effet, on a le graphe suivant:

12.
```python
def melange_graphe(nb_dep, n, graphe):
    valeurs = [i + 1 for i in range(n * n)]  # liste aplatie rangée # erreur dans le sujet
    num_tuile_vide = n * n
    actuelle = num_tuile_vide - 1  # position de la tuile vide
    for k in range(nb_dep):
        prochaine = choice(graphe[actuelle])
        valeurs[actuelle] = valeurs[prochaine]
        valeurs[prochaine] = num_tuile_vide
        actuelle = prochaine
    return en_grille(valeurs, n)
```

Exercice 3

Partie A

1. Ils ne sont pas retenus car on peut avoir des homonymes, comme deux personnes qui s'appellent "Jean Dupont" au sein de d'entreprise.
2. La relation `inscription` possède deux attributs, `trajet` et `passager` qui sont des clés étrangères, associées respectivement aux clés primaires `id_trajet` et `id_utilisateur` des relations `trajet` et `utilisateur`. Les clés étrangères ayant le même domaine que les clés primaires auxquelles elles font référence, leurs valeurs sont donc également des nombres entiers.

3. ```sql
   SELECT COUNT(*)
   FROM inscription
   WHERE trajet = 1291;
   ```

4. ```sql
   SELECT *
   FROM trajet
   WHERE heure > '2026-06-19 00:00:00'
         AND heure < '2026-06-20 00:00:00'
         AND arrivee = 'Nancy'
   ORDER BY heure;
   ```

5. ```sql
   INSERT INTO trajet
   VALUES (1295, 'Nancy', 'Allain', '2026-06-19 17:45:00', 3, 25);
   ```

6. ```sql
   UPDATE trajet
   SET heure = '2026-06-19 18:40:00'
   WHERE id_trajet = 1294;
   ```

7. L'erreur vient sans doute du fait que le trajet 1296 possède déjà des inscripts : il est donc référencé par la relation `inscription`, et on ne peut pas supprimer un trajet qui est référencé par une clé étrangère. Il faudrait d'abord supprimer les inscriptions associées à ce trajet avant de pouvoir le supprimer.

8. ```sql
   SELECT nom, prenom
   FROM utilisateur
   JOIN inscription ON utilisateur.id_utilisateur = passager
   JOIN trajet ON id_trajet = trajet
   WHERE conducteur = 25;
   ```

Partie B

9. ```python
   graphe = {
       'V': ['M'],
       'M': ['V', 'P2', 'P3'],
       'P2': ['M', 'A', 'C'],
       'P3': ['M', 'A'],
       'A': ['P2', 'P3', 'P1'],
       'P1': ['A', 'C'],
       'C': ['P1', 'P2']
   }
   ```

10. Le principe de fonctionnement d'une file est de stocker les éléments dans l'ordre dans lequel ils sont ajoutés, et de les retirer dans le même ordre, en suivant le principe FIFO (First In, First Out). Ainsi, le premier élément ajouté à la file sera le premier à être retiré.

11. Un ordre possible est le suivant : $\leftarrow \mathrm{P1} - \mathrm{A} - \mathrm{C} - \mathrm{P2} - \mathrm{P3} - \mathrm{M} - \mathrm{V} \leftarrow$.

12. C'est un parcours en largeur.

13.
```
def excentricite(graphe, sommet):
    """ graphe : dictionnaire
        {sommet : liste de sommets adjacents}
        renvoie l'excentricité du sommet
        dans le graphe (int) """
    dico = dico_distance(graphe, sommet)
    m = 0
    for distance in dico.values():
        if distance > m:
            m = distance
    return m
    # ou
    # m = 0
    # for s in dico:
    #     if dico[s] > m:
    #         m = dico[s]
    # return m
```

14. Le meilleur point de rendez-vous est le sommet `P2`, car il a l'excentricité la plus faible, qui est `2`, parmis les points de rendez-vous. En effet, on a :

- Excentricité de `P1` : `4`,
- Excentricité de `P2` : `2`,
- Excentricité de `P3` : `3`.

A noter que l'excentricité aurait été à calculer uniquement entre les points de rendez-vous et les sommets représentant les domiciles des 4 covoitureurs.