Numérique et sciences informatiques

Baccalauréat général 2026 — épreuve de spécialité

Métropole, session 2026 (26-NSIJ2ME1)

Corrigé

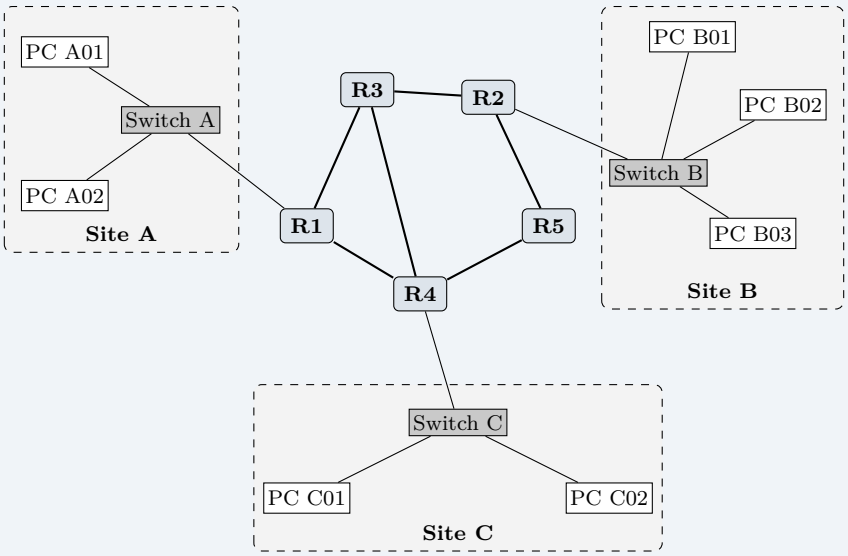
Trois exercices

Ibrahim Alame — Adloun, édition 2026

Exercice 1 (6 points)

Sujet — Exercice 1 — 6 points

Cet exercice porte sur les réseaux et la programmation orientée objet.
Une entreprise dispose d'une infrastructure réseau répartie sur plusieurs sites interconnectés à l'aide de routeurs. La figure ci-dessous représente le schéma de ce réseau, où les routeurs sont notés de R1 à R5.



Afin d'assurer la communication entre les différents postes et sous-réseaux, des protocoles de routage tels que RIP ou OSPF sont utilisés.

- Le protocole RIP minimise le nombre de sauts entre deux routeurs ; un « saut » correspond au transfert des données d'un routeur à un autre.
- Le protocole OSPF (Open Shortest Path First) minimise la somme des coûts des liaisons entre les routeurs empruntées par le paquet ; le coût entre deux routeurs voisins dépend du débit de la liaison entre ces routeurs selon la formule $\text{Coût} = \dfrac{10^8}{\text{Débit}}$ où le débit est exprimé en bit/s et le coût est sans unité.

Le bon fonctionnement de ce réseau repose sur une configuration correcte des adresses IP, des masques de sous-réseau, des routes, ainsi qu'une gestion rigoureuse des ressources (comme la mémoire et les tables de routage).
Le réseau utilise des adresses IPv4, c'est-à-dire des adresses de la forme `IP/S` où :

- IP est une suite de 4 entiers entre 0 et 255 séparés par un point ;
- S est un entier entre 1 et 32.

L'adresse IP est codée en machine sur 4 octets, soit 32 bits. L'entier S indique que les S premiers bits de IP correspondent à la partie fixe de l'adresse et les suivants à la partie propre à la machine.

Deux adresses IP sont réservées :

— l'adresse du réseau local qui est constituée de la partie fixe de l'adresse complétée par des bits égaux à 0 ;

— l'adresse de diffusion (broadcast) qui est constituée de la partie fixe de l'adresse complétée par des bits égaux à 1.

On considère le poste PC C01 du site C ayant pour adresse IPv4 : `172.16.2.1 /16`.

1. Déterminer l'adresse IP du réseau local dédié au site C.

2. Déterminer l'adresse IP de diffusion du réseau local dédié au site C.

3. Donner le nombre maximal de machines que l'on peut connecter sur le réseau local dédié au site C, y compris les machines déjà présentes.

4. Recopier et compléter la table de routage du routeur R1 obtenue avec le protocole RIP.

Table de routage R1 :

Destination	Passe par	Nombre de sauts
R2		
R3		
R4		
R5		

5. Déterminer le chemin que suit un paquet envoyé depuis PC A01 (Site A, relié à R1) vers PC B01 (Site B, relié à R2) en supposant qu'on utilise le protocole RIP.

6. Déterminer la nouvelle route qu'un paquet peut suivre de R1 à R2 si le routeur R3 tombe en panne, toujours en supposant qu'on utilise le protocole RIP.

7. Recopier et compléter le tableau suivant donnant le coût pour le protocole OSPF des liaisons réseau selon leur type de connexion.

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

Le tableau ci-dessous indique le type connexion pour chaque liaison du réseau.

Liaison	Type de connexion
R1-R3	Fibre
R1-R4	Ethernet
R2-R3	Ethernet
R5-R4	Fast Ethernet
R3-R4	Fast Ethernet
R5-R2	Fast Ethernet

8. Déterminer, en justifiant, le chemin choisi par le protocole OSPF entre R1 et R4.

On décide pour la suite de représenter en Python chaque routeur par une chaîne de caractères, par exemple `'R1'`, et de représenter les routes par des listes de routeurs, par exemple `['R1','R3','R2']`.

Un code Python permettant de définir une liste de routes (chaque route étant une liste de routeurs) a été créé. Ce code est composé :

— d'une liste vide appelée `liste_routes` qui permettra de contenir les routes ;

— d'une constante `MAX_ROUTES` indiquant le nombre maximum de routes que peut contenir la liste `liste_routes` ;

— d'une fonction `ajouter_route` qui ajoute la route donnée en paramètre à la liste `liste_routes`.

```
1  liste_routes = []
2  MAX_ROUTES = 5
3
4  def ajouter_route(route):
5      liste_routes.append(route)
```

9. Justifier que le code proposé ci-dessus ne permet pas de s'assurer que la contrainte d'un nombre maximal de routes contenues dans la liste est respectée.
On souhaite créer une classe `Routage` qui contient :

— le constructeur `__init__` qui permet d'initialiser un objet avec une capacité maximale donnée ;

— une méthode `ajouter` qui ajoute une route uniquement si la capacité maximale n'est pas atteinte ; sinon, elle affiche un message d'erreur.

Un objet de type `Routage` doit contenir les attributs :

— `capacite` : un nombre entier initialisé à 5 par défaut ;

— `routes` : une liste pour stocker les routes.

```
1  class Routage:
2      def __init__(self, capacite = 5):
3          self.capacite = ...
4          self.routes = []
5
6      def ajouter(self, route):
7          if ...
8              ...
9          else:
10             ...
```

10. Recopier et compléter les lignes 3, 7, 8 et 10 du code ci-dessus pour respecter les contraintes de la classe.
11. Écrire une méthode `afficher` de la classe `Routage` qui affiche toutes les routes présentes dans la liste `routes`. Par exemple si l'attribut `routes` de l'objet contient les routes `['R1','R3','R2']` et `['R1','R4','R5','R2']`, l'affichage sera :

```
R1
R3
R2
---
R1
R4
R5
R2
---
```

Corrigé

Adressage du site C (questions 1 à 3). **1.** L'adresse `172.16.2.1 /16` a une partie fixe de $S = 16$ bits, soit exactement les deux premiers octets (le masque de sous-réseau est `255.255.0.0`). L'adresse du réseau local s'obtient en conservant cette partie fixe et en mettant à 0 les $32 - 16 = 16$ bits restants, c'est-à-dire les deux derniers octets :

$$\texttt{172.16.2.1} \longrightarrow \underbrace{\texttt{10101100.00010000}}_{\text{partie fixe conservée}}.\underbrace{\texttt{00000000.00000000}}_{\text{16 bits à 0}} = \texttt{172.16.0.0}$$

L'adresse du réseau local du site C est donc **`172.16.0.0`** (notée `172.16.0.0/16`).
2. L'adresse de diffusion se construit de la même façon, mais en mettant à 1 les 16 bits de la partie machine : chaque octet `11111111` vaut 255, d'où l'adresse de diffusion **`172.16.255.255`**. Un paquet envoyé à cette adresse est reçu par toutes les machines du réseau local.
3. La partie propre à la machine compte $32 - 16 = 16$ bits, ce qui donne $2^{16} = 65\,536$ adresses possibles. Deux d'entre elles sont réservées (l'adresse du réseau `172.16.0.0` et l'adresse de diffusion `172.16.255.255`) et ne peuvent pas être attribuées à une machine. Le nombre maximal de machines est donc

$$2^{16} - 2 = 65\,536 - 2 = \mathbf{65\,534}.$$

Ce que le correcteur attend : le « -2 » explicitement justifié par les deux adresses réservées ; répondre 65 536 est une erreur classique.
Routage RIP (questions 4 à 6). **4.** Avec RIP, chaque routeur retient, pour chaque destination, la route ayant le *moins de sauts*. Sur la figure, R1 possède deux voisins directs, R3 et R4 (liaisons R1-R3 et R1-R4) : ils sont à 1 saut, et le routeur « par lequel on passe » (le prochain saut) est alors la destination elle-même, jointe par la liaison directe ; on peut aussi écrire « liaison directe » dans la colonne. Le routeur R2 est voisin de R3 (liaison R2-R3), donc atteignable en 2 sauts par R3, alors que le chemin par R4 (R1-R4-R5-R2) demande 3 sauts. De même R5 est voisin de R4 (liaison R5-R4), donc atteignable en 2 sauts par R4, alors que le chemin R1-R3-R2-R5 demande 3 sauts.

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

5. PC A01 est relié au Switch A, lui-même relié à R1 ; PC B01 est relié au Switch B, relié à R2. D'après la table de routage précédente, R1 envoie vers R2 en passant par R3 (2 sauts). Le paquet suit donc le chemin

$$\text{PC A01} \to \text{Switch A} \to \text{R1} \to \text{R3} \to \text{R2} \to \text{Switch B} \to \text{PC B01},$$

soit, entre routeurs, la route `['R1','R3','R2']`. Les commutateurs (switchs) ne comptent pas comme des sauts : seuls les routeurs traversés sont comptés.
6. Si R3 tombe en panne, les liaisons R1-R3, R3-R2 et R3-R4 disparaissent. Le seul chemin restant de R1 vers R2 passe par R4 puis R5 : R1 $\to$ R4 $\to$ R5 $\to$ R2, soit 3 sauts. Après convergence du protocole, la ligne « R2 » de la table de R1 devient « passe par R4, 3 sauts ». C'est l'intérêt des réseaux maillés : la commutation de paquets permet de réacheminer les paquets par une autre route lorsqu'un équipement tombe.

Routage OSPF (questions 7 et 8). **7.** On applique la formule Coût $= 10^8/$Débit :

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	$10^8/10^9 = 0{,}1$

Plus la liaison est rapide, plus son coût est faible : OSPF privilégie les liens rapides.

8. OSPF choisit le chemin dont la *somme des coûts* des liaisons est minimale. Les chemins possibles de R1 à R4 (sans repasser deux fois par le même routeur) sont :

- R1-R4 en direct (Ethernet) : coût 10 ;
- R1-R3-R4 (Fibre puis Fast Ethernet) : coût $0{,}1 + 1 = 1{,}1$;
- R1-R3-R2-R5-R4 (Fibre, Ethernet, Fast Ethernet, Fast Ethernet) : coût $0{,}1+10+1+1 = 12{,}1$.

Le minimum est 1,1 : OSPF choisit le chemin R1 → R3 → R4. On remarque que RIP aurait choisi la liaison directe R1-R4 (1 saut contre 2), pourtant dix fois plus lente : c'est précisément la différence entre les deux protocoles, RIP compte les sauts, OSPF tient compte du débit.

Programmation orientée objet (questions 9 à 11). **9.** La fonction `ajouter_route` ajoute la route à `liste_routes` de façon inconditionnelle : aucune instruction ne compare la longueur de la liste, `len(liste_routes)`, à la constante `MAX_ROUTES`. Cette constante est déclarée mais *jamais utilisée*. Cinq appels, puis un sixième, puis un septième, allongent la liste sans limite : après 7 appels de `ajouter_route`, `len(liste_routes)` vaut 7, supérieur à `MAX_ROUTES`. De plus, `liste_routes` est une variable globale : n'importe quelle partie du programme peut écrire directement `liste_routes.append(...)` sans passer par la fonction, ce qu'aucun contrôle ne peut empêcher. C'est ce défaut d'*encapsulation* que la classe `Routage` va corriger, en regroupant la liste et sa capacité dans un même objet et en réservant l'ajout à une méthode qui vérifie la contrainte.

10. La ligne 3 mémorise la capacité reçue en paramètre dans l'attribut `self.capacite` (la valeur par défaut 5 est fournie par la signature `capacite = 5`). La ligne 7 teste que la capacité n'est pas atteinte, la ligne 8 ajoute la route, la ligne 10 affiche le message d'erreur.

```
1  class Routage:
2      def __init__(self, capacite = 5):
3          self.capacite = capacite
4          self.routes = []
5
6      def ajouter(self, route):
7          if len(self.routes) < self.capacite:
8              self.routes.append(route)
9          else:
10             print("Erreur : capacité maximale atteinte, route non ajoutée")
```

Vérification exécutée : `r = Routage()` donne `r.capacite` égal à 5 et `r.routes` égal à `[]` ; avec `r2 = Routage(2)`, trois appels successifs de `r2.ajouter(['R1','R3'])` ajoutent les deux premières routes et le troisième affiche le message d'erreur, `r2.routes` restant de longueur 2. *Ce que le correcteur attend :* la condition stricte `len(self.routes) < self.capacite` (avec `<=`, on accepterait une route de trop) et l'usage de `self.` devant chaque attribut.

11. Pour chaque route de la liste `routes`, on affiche ses routeurs un par ligne, puis une ligne de séparation formée de trois tirets :

```
    def afficher(self):
```

```
        for route in self.routes:
            for routeur in route:
                print(routeur)
            print("---")
```

Cette méthode s'écrit à l'intérieur de la classe, au même niveau d'indentation que `ajouter`. Exécutée après `r.ajouter(['R1','R3','R2'])` et `r.ajouter(['R1','R4','R5','R2'])`, l'appel `r.afficher()` produit exactement l'affichage demandé : `R1`, `R3`, `R2`, la ligne de trois tirets, puis `R1`, `R4`, `R5`, `R2` et une nouvelle ligne de trois tirets, chaque élément sur sa propre ligne. Si la liste `routes` est vide, la méthode n'affiche rien, ce qui est le comportement attendu.

Exercice 2 (6 points)

Sujet — Exercice 2 — 6 points

Cet exercice porte sur la mise au point de programme, la gestion des bugs et les graphes.
Le taquin est un jeu qui se joue avec 15 tuiles numérotées dans une grille de 4 lignes et 4 colonnes pouvant en contenir 16. La tuile manquante permet de déplacer les tuiles adjacentes en les faisant glisser. Par exemple dans la grille de gauche de la figure 1, il est possible de déplacer les tuiles 12 et 15, et dans celle de droite, il est possible de déplacer les tuiles 14, 15 ou 3.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

14		15	12
5	3	7	1
2	10	9	4
13	11	6	8

Figure 1. Une grille rangée, à gauche, et mélangée, à droite.

Le but du jeu, partant d'une grille mélangée, est de ranger la grille en procédant à des glissements successifs afin de remettre les tuiles comme sur la grille de gauche sur la figure 1. Une grille mélangée pour laquelle cela est possible est dite résoluble.
On souhaite dans cet exercice écrire des fonctions permettant de mélanger une grille de taquin tout en s'assurant que la grille mélangée est résoluble.

Partie A : Mélange aléatoire Une grille de taquin est représentée en machine par une liste Python contenant quatre sous-listes de quatre entiers. Chacune des sous-listes contient des valeurs entières de façon à ce que chaque entier entre `1` et `16` apparaisse une unique fois. La valeur `16` représente la tuile vide. La grille rangée ci-dessus est ainsi représentée par :

```
rangee = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
```

On suppose que la liste `melangee` représente la grille mélangée de droite sur la figure 1.
1. Donner la valeur de `melangee[2][1]`.
On décide dans un premier temps d'obtenir une grille mélangée en utilisant la démarche suivante :

- on crée une liste `valeurs` contenant les entiers entre `1` et `16`. Dans la suite du sujet, cette liste sera appelée liste aplatie ;
- on mélange la liste aplatie en utilisant la fonction `random.shuffle` de Python ;
- on convertit la liste aplatie ainsi mélangée en une liste de listes afin d'obtenir la grille.

Afin de créer la liste aplatie `valeurs` on saisit tout d'abord l'instruction `valeurs = [k for k in range(16)]`. On constate que les valeurs ne sont pas celles attendues.

2. Réécrire cette instruction en la corrigeant pour que la liste `valeurs` contienne bien les entiers de `1` à `16`.
La liste aplatie `valeurs` étant désormais correctement créée, on la mélange à l'aide de la fonction `random.shuffle`. La documentation de Python indique que `random.shuffle(x)`, mélange la séquence x sans créer de nouvelle instance (« en place »).
On saisit donc :

```
>>> from random import shuffle
>>> valeurs = shuffle(valeurs)
>>> print(valeurs)
None
```

On constate que la variable `valeurs` est désormais affectée à `None`.
3. Proposer une modification de la ligne `valeurs = shuffle(valeurs)` afin de corriger cette erreur.
La fonction `en_grille` donnée ci-dessous prend en paramètres une liste d'entiers `valeurs` et un entier `n`. Cette fonction permet de transformer la liste `valeurs` (contenant $n \times n$ éléments) en une liste de `n` listes contenant chacune `n` entiers.

```
1  def en_grille(valeurs, n):
2      assert ...
3      grille = [[0 for j in range(n)] for i in range(n)]
4      for i in range(n):
5          for j in range(n):
6              grille[i][j] = valeurs[j * n + i]
7      return grille
```

4. Recopier et compléter la ligne 2 de cette fonction afin qu'elle vérifie que la liste passée en paramètre contient le bon nombre d'éléments.
L'appel `en_grille([1, 2, 3, 4], 2)` renvoie `[[1, 3], [2, 4]]` au lieu de `[[1, 2], [3, 4]]`.
5. Recopier et modifier la ligne 6 de la fonction `en_grille` afin de corriger cette erreur.

Partie B : Grille résoluble Certaines grilles obtenues par la méthode précédente ne sont pas résolubles. Pour savoir si une grille est résoluble, on doit calculer :

- le **nombre d'inversions** présentes dans la liste aplatie ;
- la distance, mesurée en nombre de déplacements, séparant la tuile vide de sa position finale en bas à droite de la grille.

Une inversion dans une liste aplatie `valeurs` est un couple d'indices `(i, j)` vérifiant `i < j` et `valeurs[i] > valeurs[j]`. Par exemple, la liste `valeurs = [6, 9, 7, 8]` compte 2 inversions : les couples `(1, 2)` (car `9 > 7`) et `(1, 3)` (car `9 > 8`).
La distance séparant la tuile vide de sa position finale est égale à la somme du nombre de lignes et du nombre de colonnes séparant la tuile vide et le coin inférieur droit de la grille. Dans la grille de droite de la figure 1 cette distance vaut `5`. En effet, la tuile vide est séparée de `3` lignes et de `2` colonnes de sa position finale.
On admet que la grille est résoluble si et seulement si la somme du nombre **d'inversions et de la distance** est un nombre pair.
6. Indiquer si la grille représentée en machine par la liste suivante est résoluble ? Justifier.

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 16, 15, 14]]
```

On propose ci-dessous la fonction `compte_inversions` qui permet de compter les inversions dans une liste aplatie passée en paramètre.

```
1  def compte_inversions(valeurs):
2      total = 0
3      for i in range(len(valeurs)):
4          for j in range(len(valeurs)):
5              if valeurs[i] > valeurs[j]:
6                  total = total + 1
7      return total
```

On remarque que la grille non mélangée de 2 lignes et 2 colonnes compte 0 inversion, ainsi on peut tester cette fonction, en exécutant le test suivant :

```
assert compte_inversions([1, 2, 3, 4]) == 0
```

7. Proposer un test faisant intervenir une liste aplatie de quatre éléments et comptant deux inversions permettant de vérifier que la fonction `compte_inversions` renvoie le résultat attendu.

8. La fonction proposée ne passe pas les tests. Proposer une correction de la fonction `compte_inversions`.

On propose désormais la fonction `distance_tuile_vide` qui prend en paramètre une liste de listes et un entier `n`, où la liste représente une grille de taquin de `n` lignes et `n` colonnes, et qui renvoie la distance séparant la tuile vide de sa position finale.

```
1  def distance_tuile_vide(grille, n):
2      num_tuile_vide = n * n
3      for i in range(n):
4          for j in range(n):
5              if grille[...][...] == ...:
6                  return ...
```

9. Recopier et compléter les lignes 5 et 6 de cette fonction afin qu'elle renvoie la distance attendue.

La fonction `est_resoluble` prend en paramètre la liste aplatie `valeurs` et un entier `n`, où la liste `valeurs` a $n \times n$ éléments, et renvoie le booléen indiquant si la grille de taquin qu'elle représente est résoluble ou non.

```
1  def est_resoluble(valeurs, n):
2      grille = en_grille(valeurs, n)
3      inv = ...(...)
4      dis = ...(...)
5      return (... + ...) ... == ...
```

10. Recopier et compléter cette fonction afin qu'elle renvoie la valeur attendue. On rappelle que `7 % 2` est égal à 1 alors que `8 % 2` est égal à 0.

Partie C : Mélange réaliste On souhaite désormais utiliser une méthode de mélange garantissant que la grille obtenue est résoluble. Pour ce faire, on va effectuer des déplacements réalistes en échangeant, à plusieurs reprises, la tuile vide avec une de ses tuiles voisines. On effectue ces échanges directement sur la liste de valeurs aplatie.

Pour représenter les déplacements réalisables, on définit un graphe dans lequel les sommets sont les indices des cases dans la liste aplatie. Deux sommets `i` et `j` sont reliés par une arête

s'il est possible, lorsque la tuile vide est sur l'indice `i`, de l'échanger avec la tuile d'indice `j`. On représente ce graphe en machine par une liste d'adjacence `graphe` sous la forme d'une liste Python dans laquelle la valeur à l'indice `i` contient la liste des indices des cases avec lesquelles on peut échanger la tuile d'indice `i`.
On fournit ci-dessous, **à titre d'exemple**, le graphe des déplacements réalisables pour une grille de taquin de 3 lignes et 3 colonnes (soit 9 tuiles au total).

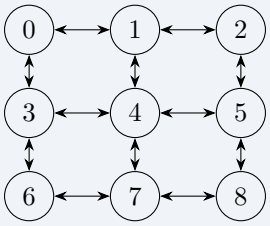


Figure 2. Le graphe des déplacements pour un taquin à 3 lignes et 3 colonnes

On suppose que `graphe_3_3` représente ce graphe par sa liste d'adjacence. Ainsi `graphe_3_3[8]` vaut `[5, 7]` ou `[7, 5]`.
11. Donner une valeur possible de `graphe_4_4[9]` en supposant que `graphe_4_4` est la liste d'adjacence représentant le graphe associé à une grille à 4 lignes et 4 colonnes.
La fonction `melange_graphe` prend en paramètre un nombre entier `nb_dep`, un entier `n` et une liste d'adjacence `graphe` représentant un graphe de déplacements sur un taquin à `n` lignes et `n` colonnes et effectue `nb_dep` déplacements successifs de la tuile vide. Ces déplacements sont choisis aléatoirement parmi ceux enregistrés dans le graphe grâce à la fonction `random.choice` qui renvoie un élément pioché aléatoirement dans la liste passée en paramètre.

```
1  from random import choice
2
3  def melange_graphe(nb_dep, n, graphe):
4      valeurs = [i for i in range (n*n)]#liste aplatie rangée
5      num_tuile_vide = n * n
6      actuelle = num_tuile_vide - 1#position de la tuile vide
7      for k in range(...):
8          prochaine = choice(...)
9          valeurs[actuelle] = valeurs[...]
10         valeurs[prochaine] = ...
11         actuelle = prochaine
12     return en_grille(valeurs, n)
```

12. Recopier et compléter les lignes 7 à 10 de la fonction `melange_graphe`.

Corrigé

Partie A : Mélange aléatoire **1.** La grille de droite de la figure 1 se lit ligne par ligne :

```
melangee = [[14, 16, 15, 12], [5, 3, 7, 1], [2, 10, 9, 4], [13, 11, 6, 8]]
```

(la tuile vide est codée `16`). Les indices commencent à 0 : `melangee[2]` est la troisième sous-liste, `[2, 10, 9, 4]`, et son élément d'indice 1 est le deuxième. Donc `melangee[2][1]` vaut **10**.
2. `range(16)` produit les entiers de 0 à 15 (la borne supérieure est exclue) : la liste obtenue est `[0, 1, ..., 15]`, elle contient un 0 et pas de 16. Il faut décaler d'un cran, en donnant à `range` sa borne de départ :

```
valeurs = [k for k in range(1, 17)]
```

On peut aussi écrire `valeurs = [k + 1 for k in range(16)]` ; les deux instructions donnent exactement `[1, 2, ..., 16]` (vérifié).
3. La fonction `shuffle` mélange la liste *en place* : elle modifie directement l'objet qu'on lui passe et ne renvoie rien, c'est-à-dire `None`. En écrivant `valeurs = shuffle(valeurs)`, on remplace donc la liste (bien mélangée) par la valeur de retour `None`. La correction consiste à ne pas affecter le résultat :

```
shuffle(valeurs)
```

Après cette ligne, `valeurs` est toujours la même liste, mais ses éléments ont été mélangés (exécuté : `shuffle(v)` renvoie `None` et `v` contient ensuite les 16 entiers dans un ordre aléatoire).
4. La précondition à vérifier est que la liste contient $n \times n$ éléments ; on l'exprime par une assertion (avec un message facultatif) :

```
    assert len(valeurs) == n * n, "la liste doit contenir n*n éléments"
```

Si la condition est fausse, une exception `AssertionError` est levée et le programme s'arrête, ce qui vaut mieux qu'un résultat faux : `en_grille([1, 2, 3], 2)` déclenche bien l'assertion (vérifié).
5. L'élément situé à la ligne `i` et à la colonne `j` de la grille est précédé, dans la liste aplatie, de `i` lignes complètes de `n` éléments puis de `j` éléments : son indice est `i * n + j`. La ligne 6 du sujet utilise `j * n + i`, qui échange les rôles de la ligne et de la colonne (la grille obtenue est la transposée de la grille attendue, d'où `[[1, 3], [2, 4]]`). Correction :

```
            grille[i][j] = valeurs[i * n + j]
```

Trace pour `en_grille([1, 2, 3, 4], 2)` : $(i, j) = (0, 0)$ lit l'indice 0, soit 1 ; $(0, 1)$ lit l'indice 1, soit 2 ; $(1, 0)$ lit l'indice $1 \times 2 + 0 = 2$, soit 3 ; $(1, 1)$ lit l'indice 3, soit 4. La fonction renvoie bien `[[1, 2], [3, 4]]`, et `en_grille([1, ..., 16], 4)` renvoie la grille `rangee` (vérifié).

Partie B : Grille résoluble **6.** La liste aplatie correspondante est `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 15, 14]`. Les treize premiers éléments sont dans l'ordre croissant et plus petits que tous les suivants : ils ne créent aucune inversion. Il reste les couples d'indices parmi 13, 14 et 15 :

- `(13, 14)` : $16 > 15$, inversion ;
- `(13, 15)` : $16 > 14$, inversion ;
- `(14, 15)` : $15 > 14$, inversion.

Le nombre d'inversions vaut 3. La tuile vide (`16`) est en ligne 3, colonne 1 ; sa position finale est la ligne 3, colonne 3 : elle est séparée de 0 ligne et de 2 colonnes, la distance vaut 2. La somme $3 + 2 = 5$ est **impaire** : d'après le critère admis, la grille **n'est pas résoluble**.
Ce que le correcteur attend : compter les inversions sur la liste aplatie *avec* la valeur 16 (la tuile vide compte), et donner la parité de la somme.
7. Le sujet fournit lui-même une liste de quatre éléments à deux inversions, `[6, 9, 7, 8]` ; le test s'écrit :

```
assert compte_inversions([6, 9, 7, 8]) == 2
```

Tout autre exemple convient, par exemple `assert compte_inversions([2, 1, 4, 3]) == 2` (inversions `(0, 1)` et `(2, 3)`).
8. Dans la fonction proposée, la boucle intérieure parcourt *tous* les indices `j`, y compris ceux qui sont inférieurs à `i` : la condition `i < j` de la définition n'est pas respectée. Pour deux

valeurs distinctes, l'une des deux comparaisons `valeurs[i] > valeurs[j]` ou `valeurs[j] > valeurs[i]` est toujours vraie : la fonction compte donc *toutes* les paires, soit $\binom{4}{2} = 6$ pour n'importe quelle liste de quatre éléments distincts. Exécutée, elle renvoie 6 pour `[1, 2, 3, 4]` (au lieu de 0), 6 pour `[6, 9, 7, 8]` (au lieu de 2) : aucun des deux tests ne passe. Il suffit de faire démarrer `j` à `i + 1` :

```
def compte_inversions(valeurs):
    total = 0
    for i in range(len(valeurs)):
        for j in range(i + 1, len(valeurs)):
            if valeurs[i] > valeurs[j]:
                total = total + 1
    return total
```

Les tests `compte_inversions([1, 2, 3, 4]) == 0`, `compte_inversions([6, 9, 7, 8]) == 2`, `compte_inversions([4, 3, 2, 1]) == 6` et `compte_inversions([]) == 0` passent tous (exécutés). La fonction reste de complexité quadratique : pour une liste de longueur p, on effectue $p(p-1)/2$ comparaisons, soit $O(p^2)$.

9. On cherche la case contenant la valeur `num_tuile_vide` (soit $n \times n$, c'est-à-dire 16 pour le taquin classique). Si elle est en ligne `i` et colonne `j`, la position finale étant la ligne `n - 1` et la colonne `n - 1`, la distance vaut `(n - 1 - i) + (n - 1 - j)` :

```
            if grille[i][j] == num_tuile_vide:
                return (n - 1 - i) + (n - 1 - j)
```

Vérifications exécutées : pour `melangee` (tuile vide en $(0, 1)$) la fonction renvoie $3 + 2 = 5$, la valeur annoncée par le sujet ; pour la grille rangée elle renvoie 0 ; pour la grille de la question 6 elle renvoie 2.

10. On calcule le nombre d'inversions sur la liste aplatie, la distance sur la grille, puis on teste la parité de leur somme avec l'opérateur `%` :

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

Exécutée, `est_resoluble` renvoie `False` pour la liste de la question 6 (somme 5), `True` pour la grille rangée (somme 0) et `True` pour la grille mélangée de la figure 1 (73 inversions, distance 5, somme 78 paire).

Ce que le correcteur attend : `compte_inversions` reçoit la liste *aplatie* `valeurs`, `distance_tuile_vide` reçoit la *grille* et `n` ; ne pas inverser les deux arguments.

Partie C : Mélange réaliste **11.** Dans une grille 4×4, l'indice `k` de la liste aplatie correspond à la ligne `k // 4` et à la colonne `k % 4` : l'indice 9 est en ligne 2, colonne 1, c'est-à-dire une case intérieure de la grille qui possède quatre voisines : au-dessus (ligne 1, colonne 1, indice $9 - 4 = 5$), à gauche (indice $9 - 1 = 8$), à droite (indice $9 + 1 = 10$) et au-dessous (ligne 3, colonne 1, indice $9 + 4 = 13$). Une valeur possible est donc

```
graphe_4_4[9] = [5, 8, 10, 13]
```

dans n'importe quel ordre. (Vérifié par construction du graphe complet : `graphe_3_3[8]` redonne bien `[5, 7]` par la même règle.)

12. À chaque tour de boucle, on tire au hasard une case `prochaine` voisine de la case `actuelle` de la tuile vide (dans `graphe[actuelle]`), on fait glisser la tuile qui s'y trouve vers la case `actuelle`, puis on place la tuile vide en `prochaine` :

```
1  def melange_graphe(nb_dep, n, graphe):
2      valeurs = [i for i in range (n*n)]#liste aplatie rangée
3      num_tuile_vide = n * n
4      actuelle = num_tuile_vide - 1#position de la tuile vide
5      for k in range(nb_dep):
6          prochaine = choice(graphe[actuelle])
7          valeurs[actuelle] = valeurs[prochaine]
8          valeurs[prochaine] = num_tuile_vide
9          actuelle = prochaine
10     return en_grille(valeurs, n)
```

L'ordre des lignes 9 et 10 est essentiel : on copie d'abord la tuile voisine dans la case de la tuile vide, puis seulement on écrase la case voisine avec `num_tuile_vide` ; dans l'ordre inverse, la tuile voisine serait perdue.
Ce que le correcteur attend : `range(nb_dep)` et non `range(n)`, et `choice` appliqué à la liste des voisins de la case courante, `graphe[actuelle]`, pas à `graphe` tout entier.
Remarque sur la ligne 4. Telle qu'elle est écrite dans le sujet, `[i for i in range (n*n)]` contient les entiers de 0 à $n^2 - 1$, alors que la convention de l'exercice (et la valeur `num_tuile_vide = n * n`) suppose la liste rangée `[1, ..., n*n]` : c'est la même erreur de borne qu'à la question 2. Avec `range(1, n*n + 1)`, la fonction complétée a été exécutée sur 2 000 mélanges (grilles 2×2, 3×3 et 4×4, de 0 à 200 déplacements) : elle renvoie chaque fois une permutation de 1 à n^2 que `est_resoluble` déclare résoluble. C'est attendu : chaque échange est un glissement légal, donc la suite des glissements inverses ramène la grille rangée.

Exercice 3 (8 points)

Sujet — Exercice 3 — 8 points

Cet exercice porte sur les bases de données ainsi que les structures de file et de graphe.

Partie A Dans cette partie, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY`.

Dans une entreprise qui possède plusieurs sites de production, on met en place une application de covoiturage pour aider les salariés à limiter leurs déplacements en voiture individuelle, et ainsi, diminuer l'empreinte carbone de l'entreprise. Avec cette application, chaque membre du personnel peut proposer un trajet ou s'inscrire sur un trajet proposé.
L'application repose sur une base de données avec trois tables, que l'on décrit par le schéma suivant où les attributs qui servent de clé primaire sont soulignés et les attributs qui servent de clé étrangère sont précédés du caractère `#`.

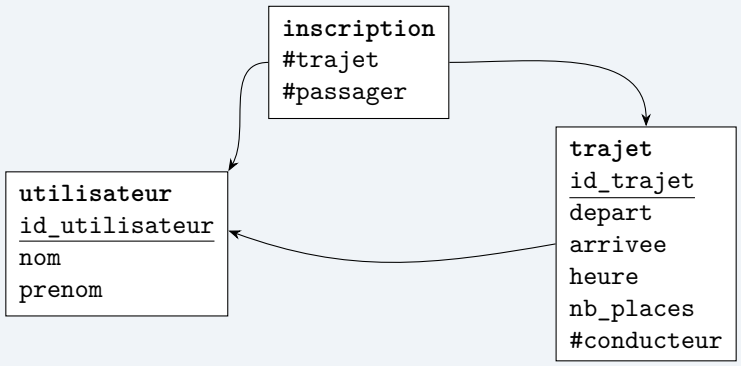


Figure 1. Schéma de la base de données

1. Expliquer pourquoi les attributs `nom` et `prenom` n'ont pas été retenus comme clé primaire de la table `utilisateur`.
Les attributs `id_utilisateur` et `id_trajet` sont des nombres entiers.
2. Justifier que les attributs de la relation `inscription` sont aussi des nombres entiers.
La requête :

```
SELECT *
FROM trajet
WHERE heure > '2026-06-19 00:00:00'
      AND heure < '2026-06-20 00:00:00';
```

renvoie le tableau suivant :

Table `trajet` :

id_trajet	depart	arrivee	heure	nb_places	conducteur
1291	Liverdun	Toul	2026-06-19 07:30:00	3	25
1292	Allain	Nancy	2026-06-19 07:45:00	2	30
1293	Messein	Nancy	2026-06-19 08:00:00	2	10
1294	Nancy	Messein	2026-06-19 18:20:00	2	10
1295	Nancy	Allain	2026-06-19 17:45:00	3	25
1296	Toul	Liverdun	2026-06-19 18:00:00	2	30

Avec des requêtes adaptées, on peut obtenir les informations suivantes concernant les inscriptions et les utilisateurs concernés par ces trajets :

Table `inscription` :

trajet	passager
1291	4
1291	15
1291	38
1292	18
1295	4
1295	15
1295	38
1296	18

Table `utilisateur` :

id_utilisateur	nom	prenom
4	Mizab	Yasmine
10	Di Maria	Alexis
15	Rey	Maxime
18	Nguema	Basil
25	Daniel	Valérie
30	Sanches	Nathalie
38	Fabre	Clément

3. Recopier et compléter la requête suivante afin de connaître le nombre de passagers inscrits sur le trajet n° 1291.

```
SELECT COUNT(*)
FROM ...
WHERE ...;
```

4. Écrire une requête permettant d'obtenir la liste des trajets qui arrivent à Nancy le 19 juin 2026 dans l'ordre croissant de l'heure de départ.
On suppose que le trajet n° 1295 a été ajouté dans la base de données lors de sa création par une simple requête et qu'il n'a pas été modifié depuis.

5. Écrire une requête qui aurait permis d'ajouter le trajet n° 1295 dans la base de données lors de sa création.
6. Écrire une requête permettant de modifier l'heure de départ à 18h40 pour le trajet n° 1294.
On pourra utiliser les valeurs présentes dans les tableaux extraits de la base de données.
Pour supprimer le trajet de 18h00 entre Toul et Liverdun, on essaye la requête suivante :

```
DELETE FROM trajet
WHERE id_trajet=1296;
```

Le gestionnaire de base de données donne alors l'erreur suivante :

```
ERROR 1451 (23000) at line 95 in file: 'covoit.sql': Cannot
delete or update a parent row: a foreign key constraint fails
```

7. Expliquer en détail les raisons de cette erreur.
8. Écrire une requête permettant d'obtenir la liste des noms et prénoms des passagers transportés au moins une fois par l'utilisatrice dont l'identifiant est 25.

Partie B Alice, Maxime, Valérie et Clément doivent choisir un point de rendez-vous pour leur trajet de covoiturage. Ils ont identifié trois points de rendez-vous possibles. Dans le graphe suivant :

— les sommets A, M, C, et V représentent les domiciles des quatre covoitureurs ;
— les sommets P1, P2, et P3 représentent les points de rendez-vous possibles ;
— les arêtes représentent des trajets entre ces lieux qui possèdent tous approximativement les mêmes conditions de parcours (distance et durée).

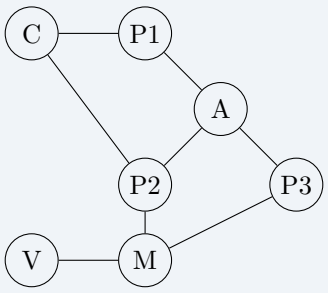


Figure 2. Graphe des domiciles et points de rendez-vous

9. Écrire un dictionnaire Python représentant le graphe de la Figure 2. Ce dictionnaire doit associer à chaque sommet la liste de ses sommets voisins.
On utilise la fonction `dico_distance` donnée ci-après pour déterminer toutes les distances en nombre d'arêtes entre un point de rendez-vous et les autres sommets du graphe.

```
1  def dico_distance(graphe, depart):
2      """ graphe : dictionnaire
3          {sommet : liste de sommets adjacents }
4          depart : un sommet du graphe
5          renvoie un dictionnaire
6          {sommet atteignable depuis depart :
7          distance entre depart et ce sommet }"""
8
9      # initialisation d'une file
10     file = File()
```

```
    file.inserer(depart)

    # initialisation du dictionnaire de sortie
    dico = {depart : 0}

    while not file.est_vide():
        # extraire le prochain sommet à explorer
        sommet = file.extraire()
        # pour chaque voisin encore non atteint
        for voisin in graphe[sommet]:
            if voisin not in dico:
                # attribuer la distance à ce voisin
                dico[voisin] = dico[sommet] + 1
                # placer ce voisin dans la file
                # pour une exploration future
                file.inserer(voisin)
    return dico
```

La fonction `dico_distance` utilise une file pour gérer l'ordre d'exploration des sommets du graphe. Cette file est implémentée avec une classe `File`. En tapant la commande `help(File)` dans la console de Python on obtient :

```
class File(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |      initialise une file vide
 |
 |  consulter(self)
 |      renvoie le premier élément disponible
 |          sans le retirer de la file
 |      lève une exception IndexError si la file est vide
 |
 |  est_vide(self)
 |      renvoie True si la file est vide, False sinon
 |
 |  extraire(self)
 |      renvoie le premier élément disponible et
 |          le retire de la file
 |      lève une exception IndexError si la file est vide
 |
 |  inserer(self, element)
 |      ajoute l'élément donné dans la file
```

10. Rappeler le principe de fonctionnement d'une file.
11. Donner un ordre possible d'insertion des sommets dans la file lorsqu'on applique la fonction `dico_distance` sur le graphe de la Figure 2 avec le sommet P1 pour sommet de départ.
12. Donner le nom du type de parcours de graphe, utilisé dans la fonction `dico_distance`.
On définit l'excentricité d'un sommet dans un graphe comme la plus grande distance parmi les distances entre ce sommet et un autre sommet quelconque du graphe.
13. Recopier et compléter la fonction `excentricite` suivante qui calcule l'excentricité d'un sommet dans un graphe. L'utilisation de la fonction `max`, disponible sans importation, n'est pas autorisée.

```
def excentricite(graphe, sommet):
```

```
    """ graphe : dictionnaire
        {sommet : liste de sommets adjacents }
        renvoie l'excentricité de sommet
        dans le graphe (int) """
    dico = dico_distance(graphe, sommet)
    ...
    ...
    ...
    ...
    ...
```

14. Donner, en le justifiant avec un indicateur, le meilleur point de rendez-vous pour ces quatre personnes (Alice, Maxime, Valérie et Clément).

Corrigé

Partie A : la base de données du covoiturage Le schéma relationnel, avec clés primaires soulignées et clés étrangères en `#` :

- `utilisateur`(`id_utilisateur`, `nom`, `prenom`) ;
- `trajet`(`id_trajet`, `depart`, `arrivee`, `heure`, `nb_places`, `#conducteur`) où `conducteur` référence `utilisateur.id_utilisateur` ;
- `inscription`(`#trajet`, `#passager`) où `trajet` référence `trajet.id_trajet` et `passager` référence `utilisateur.id_utilisateur` ; c'est la table d'association qui relie un trajet à ses passagers, le couple (`trajet`, `passager`) identifiant une inscription.

1. Une clé primaire doit identifier de façon *unique* chaque n-uplet (contrainte d'intégrité d'entité) : deux lignes ne peuvent jamais porter la même valeur de clé. Or rien n'empêche deux membres du personnel d'une entreprise à plusieurs sites d'être homonymes, avec le même nom *et* le même prénom : le couple (`nom`, `prenom`) ne garantit pas l'unicité, il ne peut donc pas servir de clé primaire. S'y ajoutent deux raisons pratiques : un nom peut changer (mariage, correction d'une faute de saisie), alors qu'une clé primaire doit rester stable, et la comparaison de deux chaînes de caractères est plus coûteuse que celle de deux entiers. On préfère donc un identifiant artificiel entier, `id_utilisateur`, unique et jamais modifié.

2. Les deux attributs de `inscription` sont des clés étrangères (le `#` du schéma) : `trajet` référence la clé primaire `id_trajet` de la table `trajet`, et `passager` référence la clé primaire `id_utilisateur` de la table `utilisateur`. La contrainte d'intégrité référentielle impose qu'une clé étrangère prenne ses valeurs parmi les valeurs existantes de la clé primaire qu'elle référence : elle a donc le même domaine qu'elle. Comme `id_trajet` et `id_utilisateur` sont des entiers, `trajet` et `passager` sont aussi des entiers. On le vérifie sur l'extrait : la ligne (1291, 4) désigne le trajet Liverdun-Toul et Yasmine Mizab.

3. Les passagers inscrits sur un trajet sont les lignes de `inscription` dont l'attribut `trajet` vaut 1291 :

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

Sur l'extrait, cette requête renvoie **3** (les passagers 4, 15 et 38).

4. On filtre sur la ville d'arrivée et sur la journée du 19 juin 2026 (toutes les heures de `'2026-06-19 00:00:00'` inclus à `'2026-06-20 00:00:00'` exclu), puis on trie par heure croissante avec `ORDER BY` :

```
SELECT *
```

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

(`ORDER BY heure ASC` est équivalent, l'ordre croissant étant celui par défaut.) Résultat sur l'extrait, exécuté sur une base de test : le trajet 1292 (Allain, `07:45:00`) puis le trajet 1293 (Messein, `08:00:00`).
Ce que le correcteur attend : la borne du jour écrite sous la même forme que dans le sujet (`'AAAA-MM-JJ HH:MM:SS'`), et le tri explicitement demandé par `ORDER BY`.
5. On reprend la ligne 1295 du tableau, en respectant l'ordre des attributs et les apostrophes pour les chaînes et la date :

```sql
INSERT INTO trajet (id_trajet, depart, arrivee, heure, nb_places, conducteur)
VALUES (1295, 'Nancy', 'Allain', '2026-06-19 17:45:00', 3, 25);
```

Le conducteur 25 (Valérie Daniel) doit exister dans `utilisateur` au moment de l'insertion, sans quoi la contrainte référentielle rejetterait la ligne.
6. On modifie le seul attribut `heure` de la ligne identifiée par sa clé primaire :

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

La clause `WHERE` est indispensable : sans elle, tous les trajets de la table seraient mis à 18h40.
7. Le trajet 1296 est *référencé* par la table `inscription` : la ligne (1296, 18) indique que l'utilisateur 18 (Basil Nguema) y est inscrit, et l'attribut `trajet` de cette ligne est une clé étrangère vers `trajet.id_trajet`. Supprimer la ligne 1296 de `trajet` laisserait dans `inscription` une référence « pendante » vers un trajet qui n'existe plus : ce serait une violation de la contrainte d'intégrité référentielle. Le SGBD la fait respecter et refuse donc la suppression, ce que dit le message : `trajet` est la table « parent » (celle dont la clé primaire est référencée), `inscription` la table « enfant », et « a foreign key constraint fails » signale la contrainte de clé étrangère violée. Pour supprimer ce trajet, il faut d'abord supprimer les inscriptions qui le référencent, puis le trajet lui-même :

```sql
DELETE FROM inscription WHERE trajet = 1296;
DELETE FROM trajet WHERE id_trajet = 1296;
```

(Une autre solution consiste à déclarer la clé étrangère avec `ON DELETE CASCADE` lors de la création de la table, ce qui supprime automatiquement les inscriptions liées.) L'erreur a été reproduite sur une base de test : la suppression directe est rejetée, la suppression en deux temps est acceptée.
8. L'utilisatrice 25 est la conductrice des trajets 1291 et 1295. On relie les trois tables : `utilisateur` donne le nom du passager, `inscription` le lie à un trajet, `trajet` donne le conducteur. `DISTINCT` évite de répéter une personne transportée plusieurs fois :

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

Résultat sur l'extrait (exécuté) : Mizab Yasmine, Rey Maxime, Fabre Clément. Sans `DISTINCT`, chacun apparaîtrait deux fois, puisqu'ils sont inscrits à la fois sur le trajet 1291 et sur le trajet 1295.
Ce que le correcteur attend : deux jointures (une seule ne relie pas le passager au conducteur), la condition sur `trajet.conducteur` et non sur `inscription.passager`, et le `DISTINCT` pour « au moins une fois ».

Partie B : le choix du point de rendez-vous **9.** Le graphe non orienté de la figure 2 a huit arêtes : C-P1, C-P2, P1-A, A-P2, A-P3, P2-M, P3-M et V-M. Chaque arête apparaît dans les listes de ses deux extrémités :

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

(L'ordre des voisins dans chaque liste est libre. Vérification exécutée : la symétrie est respectée et la somme des degrés vaut $16 = 2 \times 8$.)
10. Une file est une structure linéaire de type **FIFO** (*First In, First Out*, « premier entré, premier sorti ») : les éléments sont ajoutés à une extrémité, la *queue* (méthode `inserer`, « enfiler »), et retirés à l'autre extrémité, la *tête* (méthode `extraire`, « défiler ») ; l'élément retiré est toujours le plus ancien encore présent. On peut consulter l'élément de tête sans le retirer (`consulter`) et tester si la file est vide (`est_vide`). C'est le fonctionnement d'une file d'attente, à l'opposé de la pile (LIFO), où l'on retire le dernier élément entré.
11. On déroule `dico_distance(graphe, 'P1')` avec le dictionnaire de la question 9. Le sommet de départ est inséré, puis, à chaque extraction, les voisins non encore présents dans `dico` sont insérés dans l'ordre de leur liste :

Sommet extrait	Voisins	Insérés (distance)	File après l'étape
—	—	P1 (0)	[P1]
P1	C, A	C (1), A (1)	[C, A]
C	P1, P2	P2 (2)	[A, P2]
A	P1, P2, P3	P3 (2)	[P2, P3]
P2	C, A, M	M (3)	[P3, M]
P3	A, M	aucun	[M]
M	P2, P3, V	V (4)	[V]
V	M	aucun	[]

Un ordre possible d'insertion est donc **P1, C, A, P2, P3, M, V** (exécuté), et la fonction renvoie `{'P1': 0, 'C': 1, 'A': 1, 'P2': 2, 'P3': 2, 'M': 3, 'V': 4}`. Si la liste des voisins de P1 était `['A', 'C']`, on obtiendrait P1, A, C, P2, P3, M, V : l'ordre dépend de l'ordre des listes d'adjacence, mais les distances sont toujours les mêmes.
12. C'est un **parcours en largeur** (*Breadth-First Search*, BFS) : la file fait explorer les sommets « par couches », d'abord tous les sommets à distance 1 du départ, puis ceux à distance 2, etc. C'est ce qui garantit que `dico[voisin] = dico[sommet] + 1` est bien la plus courte distance en nombre d'arêtes. Un parcours en profondeur utiliserait une pile.
13. On parcourt les valeurs du dictionnaire des distances en conservant la plus grande rencontrée, sans la fonction `max` :

```python
def excentricite(graphe, sommet):
    """ graphe : dictionnaire
        {sommet : liste de sommets adjacents }
        renvoie l'excentricité de sommet
        dans le graphe (int) """
    dico = dico_distance(graphe, sommet)
    maxi = 0
    for s in dico:
        if dico[s] > maxi:
            maxi = dico[s]
    return maxi
```

Initialiser `maxi` à 0 est correct puisque toutes les distances sont positives ou nulles (celle du sommet de départ vaut 0). Exécuté : `excentricite(graphe, 'P1')` renvoie 4 (le sommet V), `excentricite(graphe, 'P2')` renvoie 2 et `excentricite(graphe, 'P3')` renvoie 3.

14. L'indicateur naturel est l'**excentricité** du point de rendez-vous, c'est-à-dire la distance (en nombre d'arêtes) que doit parcourir le covoitureur le plus éloigné. Les distances calculées par `dico_distance` depuis chaque point sont :

Point	vers A	vers M	vers V	vers C	Excentricité	Somme des distances
P1	1	3	4	1	4	9
P2	1	1	2	1	2	5
P3	1	1	2	3	3	7

(Ici l'excentricité prise sur tout le graphe coïncide avec la plus grande distance aux quatre domiciles.) Le point **P2** a l'excentricité minimale, 2 : personne n'a plus de deux trajets élémentaires à faire, et Alice, Maxime et Clément n'en ont qu'un. Un second indicateur, la somme des distances aux quatre domiciles (5 pour P2 contre 7 pour P3 et 9 pour P1), confirme ce choix. Le meilleur point de rendez-vous est donc **P2**.

Ce que le correcteur attend : un indicateur *nommé* (excentricité ou somme des distances) et *calculé* pour les trois points, pas seulement une lecture de la figure.