Corrigé – Bac 2026 – Numérique et sciences informatiques – Métropole, jour 2

Réseaux (adressage IP, RIP, OSPF) et programmation orientée objet, Le taquin : mise au point de programmes et graphe des déplacements, Bases de données (covoiturage), file et parcours de graphe

Corrigé détaillé rédigé par Ben, professeur particulier – benprofparticulier.fr

Exercice 1 – Réseaux (adressage IP, RIP, OSPF) et programmation orientée objet *(6 pts)*

1. Le masque /16 indique que les 16 premiers bits, c'est-à-dire les deux premiers octets (172.16), forment la partie fixe de l'adresse. L'adresse du réseau s'obtient en mettant à 0 tous les bits de la partie machine, soit les deux derniers octets : l'adresse du réseau local du site C est **172.16.0.0** (notée 172.16.0.0/16).

2. L'adresse de diffusion s'obtient en mettant à 1 tous les bits de la partie machine ; un octet dont les 8 bits valent 1 vaut 255. L'adresse de diffusion du site C est **172.16.255.255**.

3. La partie machine est codée sur $32 - 16 = 16$ bits, ce qui donne $2^{16} = 65\,536$ adresses. Deux d'entre elles sont réservées (adresse du réseau et adresse de diffusion). On peut donc connecter au plus $2^{16} - 2 =$ **65 534 machines** sur ce réseau local, machines déjà présentes comprises.

4. D'après le schéma, les liaisons entre routeurs sont R1-R3, R1-R4, R2-R3, R3-R4, R4-R5 et R2-R5. Avec RIP, on retient pour chaque destination une route de nombre de sauts minimal ; la colonne « Passe par » indique le premier routeur voisin emprunté.

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

Justification : R3 et R4 sont des voisins directs de R1 (1 saut). R2 est atteint par R1 → R3 → R2 (2 sauts), alors que R1 → R4 → R5 → R2 en compte 3. R5 est atteint par R1 → R4 → R5 (2 sauts).

5. Avec RIP, le paquet suit la route de R1 vers R2 qui a le moins de sauts, c'est-à-dire en passant par R3 (2 sauts). Le chemin est : **PC A01 → Switch A → R1 → R3 → R2 → Switch B → PC B01**.

6. Si R3 tombe en panne, les liaisons R1-R3, R2-R3 et R3-R4 deviennent inutilisables. Il reste les liaisons R1-R4, R4-R5 et R5-R2 : la nouvelle route est **R1 → R4 → R5 → R2** (3 sauts), seule route possible.

7. On applique 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**

8. Coûts des liaisons : R1-R3 (fibre) = 0,1 ; R1-R4 (Ethernet) = 10 ; R2-R3 (Ethernet) = 10 ; R5-R4 (Fast Ethernet) = 1 ; R3-R4 (Fast Ethernet) = 1 ; R5-R2 (Fast Ethernet) = 1. Chemins possibles entre R1 et R4 (sans repasser par un même routeur) :

- R1 → R4 : coût 10 ;
- R1 → R3 → R4 : coût 0,1 + 1 = 1,1 ;
- R1 → R3 → R2 → R5 → R4 : coût 0,1 + 10 + 1 + 1 = 12,1.

OSPF choisit le chemin de coût total minimal : **R1 → R3 → R4**, de coût 1,1. Il comporte plus de sauts que la liaison directe, mais il emprunte des liaisons bien plus rapides.

> ⚠ **Piège.** Ne confonds pas les deux protocoles : RIP compte les sauts (la liaison directe R1-R4 serait choisie), OSPF additionne les coûts. Ici, le chemin le plus court en sauts n'est pas le moins coûteux.

9. La fonction ajouter_route ajoute la route avec append sans jamais comparer la longueur de liste_routes à MAX_ROUTES : la constante n'est utilisée nulle part. Après cinq appels, un sixième appel ajoute encore une route, et la liste peut contenir autant de routes que l'on veut. La contrainte « au plus MAX_ROUTES routes » n'est donc pas vérifiée.

10. Les lignes complétées sont les lignes 3, 7, 8 et 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('Erreur : capacité maximale atteinte, route non ajoutée')
```

La condition utilise une inégalité stricte : tant que la liste contient moins de capacite routes, il reste de la place ; quand elle en contient exactement capacite, l'ajout est refusé.

11. On parcourt chaque route, on affiche chacun de ses routeurs sur une ligne, puis le séparateur --- après chaque route :

```
    def afficher(self):
        for route in self.routes:
            for routeur in route:
                print(routeur)
            print('---')
```

Avec les routes ['R1','R3','R2'] et ['R1','R4','R5','R2'], on obtient bien R1, R3, R2, ---, R1, R4, R5, R2, --- (un élément par ligne).

Exercice 2 – Le taquin : mise au point de programmes et graphe des déplacements *(6 pts)*

Partie A : Mélange aléatoire

1. D'après la figure 1, la grille mélangée est représentée par melangee = [[14, 16, 15, 12], [5, 3, 7, 1], [2, 10, 9, 4], [13, 11, 6, 8]]. melangee[2] est la troisième ligne [2, 10, 9, 4] et son élément d'indice 1 est le deuxième : **melangee[2][1] vaut 10**.

2. range(16) produit les entiers de 0 à 15. Pour obtenir les entiers de 1 à 16, on écrit :

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

(ou, de façon équivalente, valeurs = [k + 1 for k in range(16)]).

3. shuffle mélange la liste en place et renvoie None ; c'est ce None qui est affecté à valeurs. Il suffit d'appeler la fonction sans affecter son résultat :

```
shuffle(valeurs)
```

4. La liste doit contenir n × n éléments :

```
    assert len(valeurs) == n * n
```

5. Dans la liste aplatie, la case de la ligne i et de la colonne j se trouve à l'indice i × n + j (on saute i lignes complètes de n éléments, puis on avance de j). La ligne 6 corrigée est :

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

Vérification avec en_grille([1, 2, 3, 4], 2) : grille[0][1] = valeurs[1] = 2 et grille[1][0] = valeurs[2] = 3, d'où [[1, 2], [3, 4]].

Partie B : Grille résoluble

6. La liste aplatie est [1, 2, . . ., 12, 13, 16, 15, 14]. Les douze premiers nombres sont rangés et plus petits que tous ceux qui suivent : ils ne créent aucune inversion ; 13 est plus petit que 16, 15 et 14 : aucune inversion non plus. Les inversions sont donc 16 > 15, 16 > 14 et 15 > 14 : **3 inversions**.

La tuile vide (16) est sur la dernière ligne, deuxième colonne : elle est séparée de 0 ligne et de 2 colonnes du coin inférieur droit, donc la distance vaut **2**.

3 + 2 = 5 est impair : **cette grille n'est pas résoluble**.

7. La liste [2, 3, 1, 4] compte exactement deux inversions, les couples (0, 2) car 2 > 1 et (1, 2) car 3 > 1. On propose le test :

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

La fonction proposée échoue à ce test : elle renvoie 6, car elle compte tous les couples (i, j), y compris ceux où i > j.

8. L'erreur vient de la ligne 4 : j doit parcourir uniquement les indices strictement supérieurs à i, pour respecter la condition i < j de la définition. Fonction corrigée :

```
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
```

Elle renvoie 0 pour [1, 2, 3, 4] et 2 pour [2, 3, 1, 4] : les deux tests passent.

9. La tuile vide porte la valeur n × n ; sa position finale est la ligne n − 1 et la colonne n − 1. Lignes 5 et 6 :

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

Vérification sur la grille mélangée de la figure 1 : la tuile vide est en $i = 0, j = 1$ et $(4 - 1 - 0) + (4 - 1 - 1) = 3 + 2 = 5$, la valeur annoncée par l'énoncé.

10. Les inversions se comptent sur la liste aplatie et la distance se calcule sur la grille :

```
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
```

La somme est paire exactement quand son reste dans la division par 2 vaut 0.

> 💡 **Astuce.** Teste toujours ta fonction à la main sur l'exemple de l'énoncé (ici la distance 5 de la figure 1) : c'est le moyen le plus sûr de repérer une erreur d'indice.

Partie C : Mélange réaliste

11. Sur une grille 4×4, l'indice 9 correspond à la ligne $9 \,//\, 4 = 2$ et à la colonne $9 \,\%\, 4 = 1$. La case a quatre voisines : au-dessus l'indice $9 - 4 = 5$, en dessous $9 + 4 = 13$, à gauche 8 et à droite 10. Une valeur possible est **graphe_4_4**$[9] = [$**5, 8, 10, 13**$]$ (dans n'importe quel ordre).

12. On répète nb_dep fois : on choisit au hasard une case voisine de la tuile vide dans le graphe, on y prend la tuile pour la placer à la position actuelle de la tuile vide, puis on met la tuile vide (valeur num_tuile_vide) dans la case choisie.

```
    for k in range(nb_dep):
        prochaine = choice(graphe[actuelle])
        valeurs[actuelle] = valeurs[prochaine]
        valeurs[prochaine] = num_tuile_vide
        actuelle = prochaine
```

Chaque étape est un vrai glissement de tuile : la grille obtenue est donc résoluble par construction, il suffit de refaire les déplacements à l'envers.

> ⚠ **Piège.** Avec la ligne 4 telle qu'elle est écrite, range(n*n) produit les valeurs 0 à $n^2 - 1$, alors que la tuile vide est codée $n \times n$: c'est la même erreur qu'à la question 2. Pour une grille cohérente, on écrirait valeurs $=$ [i + 1 for i in range(n*n)] ; la réponse attendue pour les lignes 7 à 10 ne change pas.

Exercice 3 – Bases de données (covoiturage), file et parcours de graphe *(8 pts)*

Partie A

1. Une clé primaire doit identifier chaque enregistrement de façon unique. Or deux salariés peuvent avoir le même nom et le même prénom (homonymes) : le couple (nom, prenom) ne garantit pas l'unicité. De plus, un nom peut changer (mariage, correction d'une faute de saisie), alors qu'une clé primaire doit rester stable. On utilise donc un identifiant entier id_utilisateur.

2. D'après le schéma, trajet est une clé étrangère qui fait référence à id_trajet de la table trajet, et passager est une clé étrangère qui fait référence à id_utilisateur de la table utilisateur. Une clé étrangère prend ses valeurs parmi celles de l'attribut qu'elle référence, donc dans le même domaine : comme id_trajet et id_utilisateur sont des entiers, **trajet et passager sont aussi des entiers**.

3.

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

Sur les données fournies, cette requête renvoie 3 (passagers 4, 15 et 38).

4.

```
SELECT *
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 trie par ordre croissant par défaut (on peut ajouter ASC). On obtient les trajets 1292 (7 h 45) puis 1293 (8 h 00).

5. Le trajet n° 1295 n'ayant pas été modifié depuis sa création, on reprend ses valeurs actuelles :

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

6. Le trajet n° 1294 a lieu le 19 juin 2026 :

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

7. L'attribut trajet de la table inscription est une clé étrangère vers id_trajet de la table trajet. La table inscription contient l'enregistrement (1296, 18) : le passager 18 (Basil Nguema) est inscrit sur le trajet 1296. Supprimer la ligne 1296 de trajet (la ligne « parente ») laisserait dans inscription une ligne « enfant » qui ferait référence à un trajet inexistant. Le SGBD refuse cette suppression pour préserver la **contrainte d'intégrité référentielle** : toute valeur d'une clé étrangère doit exister dans la table référencée. Il faut d'abord supprimer les inscriptions de ce trajet, puis le trajet :

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

8. On relie les passagers (utilisateur) à leurs inscriptions, puis aux trajets dont la conductrice est l'utilisatrice 25 ; DISTINCT évite les doublons d'un passager transporté plusieurs fois :

```
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;
```

Sur les données fournies (trajets 1291 et 1295), on obtient Mizab Yasmine, Rey Maxime et Fabre Clément, chacun une seule fois.

> ⚠ **Piège.** Sans DISTINCT, les trois passagers apparaîtraient deux fois chacun, puisqu'ils sont inscrits à la fois sur le trajet 1291 et sur le trajet 1295.

Partie B

9. D'après la figure 2, les arêtes sont C-P1, C-P2, P1-A, A-P2, A-P3, P2-M, P3-M et M-V :

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

10. Une file fonctionne selon le principe **FIFO** (« premier entré, premier sorti ») : les éléments sont insérés à une extrémité (la queue) et extraits à l'autre (la tête), si bien que l'élément extrait est toujours le plus ancien encore présent, comme dans une file d'attente.

11. Avec le dictionnaire de la question 9 et P1 comme départ :

- on insère P1 ;
- on extrait P1 : ses voisins C et A ne sont pas encore atteints, on insère C puis A (distance 1) ;
- on extrait C : P1 est déjà atteint, on insère P2 (distance 2) ;
- on extrait A : P1 et P2 sont déjà atteints, on insère P3 (distance 2) ;
- on extrait P2 : C et A sont atteints, on insère M (distance 3) ;
- on extrait P3 : A et M sont atteints, rien à insérer ;
- on extrait M : P2 et P3 sont atteints, on insère V (distance 4) ;
- on extrait V : la file est vide, la fonction s'arrête.

Un ordre possible d'insertion est : **P1, C, A, P2, P3, M, V**. (Selon l'ordre des voisins dans les listes, on peut aussi obtenir P1, A, C, P2, P3, M, V par exemple.)

12. L'utilisation d'une file, qui explore tous les sommets à distance 1, puis tous ceux à distance 2, etc., caractérise un **parcours en largeur** (BFS, *breadth-first search*).

13. On cherche la plus grande valeur du dictionnaire des distances, sans utiliser max :

```
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)
    exc = 0
    for s in dico:
        if dico[s] > exc:
            exc = dico[s]
    return exc
```

L'initialisation à 0 est correcte, car toutes les distances sont positives ou nulles (le sommet lui-même est à distance 0).

14. Distances obtenues par parcours en largeur depuis chaque point de rendez-vous :

Départ	**A**	**M**	**C**	**V**	**P1**	**P2**	**P3**	**Excentricité**
P1	1	3	1	4	0	2	2	4
P2	1	1	1	2	2	0	2	2

Départ	A	M	C	V	P1	P2	P3	Excentricité
P3	1	1	3	2	2	2	0	3

On prend comme indicateur l'excentricité : elle donne la distance du domicile le plus éloigné du point de rendez-vous, c'est-à-dire le trajet du covoitureur le plus pénalisé. L'excentricité de P2 vaut 2, celles de P1 et P3 valent 4 et 3. Le meilleur point de rendez-vous est donc **P2** : personne n'a plus de 2 trajets à faire pour le rejoindre (Alice, Maxime et Clément sont à 1 arête, Valérie à 2). P2 est aussi le meilleur si l'on regarde la somme des distances aux quatre domiciles : 5, contre 9 pour P1 et 7 pour P3.

> 💡 **Astuce.** Quand on te demande de « justifier avec un indicateur », réutilise la fonction que tu viens d'écrire (ici l'excentricité) : calcule-la pour chaque candidat, présente les résultats dans un petit tableau et conclus par le minimum.

Conseils pour le jour J

Ce sujet est long mais très progressif : trois exercices indépendants, dont le troisième vaut 8 points. Garde au moins 1 h 20 pour l'exercice 3, dont la partie SQL rapporte vite si tu connais la syntaxe exacte de SELECT, INSERT INTO … VALUES, UPDATE … SET … WHERE et des jointures. Commence par l'exercice où tu es le plus à l'aise, mais traite-les tous : beaucoup de questions se font en une ou deux lignes.

En réseau, recopie d'abord la liste des liaisons lue sur le schéma, puis calcule les coûts OSPF un par un : la plupart des erreurs viennent d'une liaison oubliée ou d'une confusion entre nombre de sauts (RIP) et somme des coûts (OSPF). Pour l'adressage, raisonne en bits : /16 signifie deux octets fixes, et il faut retirer 2 adresses réservées au nombre de machines.

Dans les questions « recopier et compléter », recopie les lignes entières avec leur indentation et vérifie ton code à la main sur l'exemple de l'énoncé : indices $i \times n + j$, borne range$(i + 1, \ldots)$ pour les inversions, modulo 2 pour la parité. C'est ce test mental, écrit en une phrase, qui convainc le correcteur que ton code est juste.

Ce document est la propriété intellectuelle de Ben, professeur particulier (benprofparticulier.fr), protégée par le Code de la propriété intellectuelle (art. L. 111-1 et L. 122-4). Il est mis à disposition pour un usage personnel de révision. Toute reproduction, diffusion, mise en ligne ou revente, totale ou partielle, sans autorisation écrite est interdite et constitue une contrefaçon (art. L. 335-2).