Numérique et sciences informatiques

Baccalauréat général 2025 — épreuve de spécialité

Métropole, session 2025 (25-NSIJ1ME1)

Corrigé

Trois exercices

Ibrahim Alame — Adloun, édition 2026

Exercice 1 (6 points)

Sujet — Exercice 1 — 6 points

Cet exercice porte sur les bases de données relationnelles et les requêtes SQL.
Dans cet exercice, 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`, `OR`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT`, `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT`, `ORDER BY`.

Dans un schéma relationnel, on utilisera les conventions suivantes :

- la clé primaire d'une relation est définie par son attribut souligné ;
- les attributs précédés de # sont les clés étrangères.

Le guitariste Slash possède une incroyable collection de guitares. Maud est une grande fan de Slash. Elle décide de faire un inventaire de la collection de guitares sous la forme d'une base de données relationnelle.

Partie A Dans cette partie, Maud utilise la relation suivante :
`inventaire (id, marque, modele, annee, num_ser, prix)`
`num_ser` représente le numéro de série d'une guitare. Il est unique pour chaque guitare d'une même marque. Le `prix` est en euro.
Voici un extrait de la table `inventaire`.

`inventaire`

id	marque	modele	annee	num_ser	prix
1	Gibson	Les Paul Goldtop	1956	@70562	100000
2	Gibson	Les Paul Goldtop	1988	81738349	20000
3	Gibson	Les Paul Standard	1959	@90663	250000
4	Gibson	Les Paul Standard	1987	81757532	25000
5	Fender	Telecaster	1952	000230	150000
6	Fender	Telecaster	1965	81345673	10000
7	Fender	Stratocaster	1956	001359	200000
8	Fender	Stratocaster	1965	81757532	15000

1. Expliquer pourquoi l'attribut `num_ser` ne peut pas être une clé primaire de la relation `inventaire`.
2. Donner, sous forme de tableau, le résultat de la requête suivante appliquée à l'extrait de table précédent.

```
SELECT marque, modele
```

```
FROM inventaire
WHERE annee = 1956
```

3. Écrire une requête SQL permettant d'obtenir toutes les années du modèle Les Paul Standard dans la collection.
4. Écrire une requête SQL permettant d'obtenir tous les modèles de guitares de la marque Gibson par ordre croissant de l'année dans la collection.
5. Maud a fait une erreur de saisie pour la guitare d'identifiant `id`=1. L'année est en réalité 1957. Écrire une requête SQL permettant de corriger cette erreur de saisie.

Partie B Maud change de représentation pour l'inventaire de la collection. Dans cette partie, Maud utilise maintenant les trois relations suivantes :
`marque (id, nom)`
`modele (id, nom, #id_marque)`
`guitare (id, #id_modele, annee, num_ser, prix)`
Dans la relation `modele`, `#id_marque` est une clé étrangère reliée à la clé primaire `id` de la relation `marque`. Dans la relation `guitare`, `#id_modele` est une clé étrangère reliée à la clé primaire `id` de la relation `modele`.
Voici des extraits des trois tables `marque`, `modele`, `guitare`.

`marque`

id	nom
1	Gibson
2	Fender

`modele`

id	nom	id_marque
1	Les Paul Goldtop	1
2	Les Paul Standard	1
3	Telecaster	2
4	Stratocaster	2

`guitare`

id	id_modele	annee	num_ser	prix
1	1	1956	@70562	100000
2	1	1988	81738349	20000
3	2	1959	@90663	250000
4	2	1987	81757532	25000
5	3	1952	000230	150000
6	3	1965	81345673	10000
7	4	1956	001359	200000
8	4	1965	81757532	15000

6. Expliquer brièvement, en justifiant, dans quel ordre les trois tables doivent être créées.
7. Écrire une requête SQL permettant d'obtenir le numéro de série et l'année de toutes les guitares Les Paul Standard de la collection.
Maud vient d'apprendre que Slash a fait cadeau d'une de ses guitares à un ami. Elle doit donc la retirer de sa base de données.
8. Écrire une requête SQL permettant de retirer de la collection la guitare d'identifiant `id`=3.

Slash a aussi acheté une guitare d'une marque qu'il n'avait pas encore dans sa collection. Maud décide de la rajouter.

9. Écrire l'ensemble des requêtes SQL permettant d'ajouter la guitare suivante :

- marque : BC Rich
- modèle : Mockingbird
- année : 1992
- numéro de série : 92R
- prix : 5000.

On supposera que l'on peut attribuer la valeur 3 pour l'attribut `id` dans la table `marque` pour la marque BC Rich, que l'on peut attribuer la valeur 5 pour l'attribut `id` dans la table `modele` pour le modèle Mockingbird et que l'on peut attribuer la valeur 9 pour l'attribut `id` dans la table `guitare` pour cette guitare.

Maud souhaite connaître la valeur totale des modèles Stratocaster de la collection. Son ami David lui conseille de regarder la fonction `SUM`. La syntaxe pour utiliser cette fonction SQL peut être similaire à celle-ci :

```
SELECT SUM(nom_colonne)
FROM tab
```

Cette requête SQL permet de calculer la somme des valeurs contenues dans la colonne `nom_colonne` de la table `tab`.

10. Écrire une requête SQL permettant de calculer la valeur totale des modèles Stratocaster de la collection de Slash.

Corrigé

Partie A **1.** Une clé primaire doit identifier de façon *unique* chaque enregistrement de la relation : deux lignes ne peuvent jamais porter la même valeur de clé (c'est la contrainte d'intégrité d'entité). Or l'énoncé précise que `num_ser` n'est unique qu'*au sein d'une même marque* : deux guitares de marques différentes peuvent porter le même numéro de série. L'extrait le montre d'ailleurs : la Gibson Les Paul Standard d'identifiant 4 et la Fender Stratocaster d'identifiant 8 ont toutes deux le numéro de série `81757532`. La valeur `81757532` désignerait deux guitares, ce qu'une clé primaire interdit. En revanche, le couple (`marque`, `num_ser`) pourrait servir de clé, et c'est pourquoi Maud a créé l'identifiant artificiel `id`.

2. La clause `WHERE annee = 1956` ne conserve que les lignes 1 et 7, et la projection `SELECT marque, modele` n'en garde que deux colonnes :

marque	modele
Gibson	Les Paul Goldtop
Fender	Stratocaster

3. On filtre sur le modèle et on ne projette que l'année :

```
SELECT annee
FROM inventaire
WHERE modele = 'Les Paul Standard';
```

Sur l'extrait, la requête renvoie 1959 et 1987. La chaîne de caractères est entre apostrophes simples, avec exactement l'orthographe de la table (majuscules comprises).

4. On filtre sur la marque et on trie avec `ORDER BY`, croissant par défaut (on peut préciser `ASC`) :

```sql
SELECT modele
FROM inventaire
WHERE marque = 'Gibson'
ORDER BY annee ASC;
```

Sur l'extrait, le résultat est, dans l'ordre : Les Paul Goldtop (1956), Les Paul Standard (1959), Les Paul Standard (1987), Les Paul Goldtop (1988). On peut ajouter `annee` dans le `SELECT` pour rendre le tri lisible ; on n'utilise pas `DISTINCT`, qui ferait perdre la correspondance entre un modèle et chacune de ses années.
5. On modifie une ligne existante avec `UPDATE`, en ciblant la guitare par sa clé primaire :

```sql
UPDATE inventaire
SET annee = 1957
WHERE id = 1;
```

Ce que le correcteur attend : la clause `WHERE` est indispensable ; sans elle, *toutes* les guitares de la table passeraient à l'année 1957.

Partie B **6.** Une clé étrangère doit référencer la clé primaire d'une table qui existe déjà : le SGBD refuse de créer une contrainte référentielle vers une table absente (contrainte d'intégrité référentielle). Il faut donc créer les tables dans l'ordre des dépendances :

1. `marque`, qui ne dépend d'aucune autre table ;
2. `modele`, dont l'attribut `id_marque` référence `marque.id` ;
3. `guitare`, dont l'attribut `id_modele` référence `modele.id`.

Le même ordre s'impose pour insérer des données (question 9) : une marque avant ses modèles, un modèle avant ses guitares.
7. Le nom du modèle est dans la table `modele`, le numéro de série et l'année dans la table `guitare` : il faut une jointure sur la clé étrangère `id_modele` :

```sql
SELECT guitare.num_ser, guitare.annee
FROM guitare
JOIN modele ON guitare.id_modele = modele.id
WHERE modele.nom = 'Les Paul Standard';
```

Sur l'extrait, la requête renvoie (`@90663`, 1959) et (`81757532`, 1987) : les guitares 3 et 4, seules à porter `id_modele = 2`.
Ce que le correcteur attend : sans la jointure, on ne peut pas filtrer sur `nom`, absent de `guitare` ; la condition de jointure relie bien la clé étrangère `guitare.id_modele` à la clé primaire `modele.id`, et non les deux attributs `id`.
8. On supprime la ligne par sa clé primaire :

```sql
DELETE FROM guitare
WHERE id = 3;
```

Seule la table `guitare` est touchée : le modèle Les Paul Standard et la marque Gibson restent dans la base, car d'autres guitares (ici la guitare 4) y font encore référence. Ici encore, oublier le `WHERE` viderait toute la table.
9. La marque BC Rich n'existe pas encore, pas plus que le modèle Mockingbird : il faut trois insertions, dans l'ordre imposé par les clés étrangères (marque, puis modèle, puis guitare), avec les identifiants fournis par l'énoncé :

```sql
INSERT INTO marque (id, nom)
VALUES (3, 'BC Rich');

INSERT INTO modele (id, nom, id_marque)
VALUES (5, 'Mockingbird', 3);

INSERT INTO guitare (id, id_modele, annee, num_ser, prix)
VALUES (9, 5, 1992, '92R', 5000);
```

Chaque clé étrangère pointe vers une ligne qui vient d'être créée : `id_marque = 3` est l'identifiant de BC Rich, `id_modele = 5` celui de Mockingbird. Le numéro de série `'92R'` contient une lettre : c'est une chaîne de caractères, entre apostrophes.
Ce que le correcteur attend : insérer la guitare avant son modèle (ou le modèle avant sa marque) violerait l'intégrité référentielle et serait refusé par le SGBD.
10. Le prix est dans `guitare`, le nom du modèle dans `modele` : on joint les deux tables, on filtre sur le modèle, puis on somme les prix avec la fonction d'agrégation `SUM` :

```sql
SELECT SUM(guitare.prix)
FROM guitare
JOIN modele ON guitare.id_modele = modele.id
WHERE modele.nom = 'Stratocaster';
```

Sur l'extrait, les Stratocaster sont les guitares 7 et 8 (`id_modele = 4`) ; la requête renvoie $200\,000 + 15\,000 = 215\,000$ (euros). La fonction `SUM` s'applique après le filtrage : elle n'additionne que les lignes retenues par le `WHERE`, et renvoie une seule ligne.

Exercice 2 (6 points)

Sujet — Exercice 2 — 6 points

Cet exercice porte sur l'algorithmique, les structures de données, et la gestion de processus.
On cherche à créer une application de type *liste de tâches à faire* pour aider Alice à planifier sa journée. Pour cela Alice saisit les informations concernant chacune des tâches qu'elle doit effectuer : elle indique un nom pour la tâche, ainsi que la durée qu'elle estime nécessaire afin de la réaliser. On représente une tâche saisie par Alice à l'aide d'un objet de type `Tache`, muni de quatre attributs :

— le `numero` de la tâche, saisi par Alice ;

— le `nom` de la tâche, saisi par Alice ;

— la `duree` (un entier exprimé en minute) nécessaire à la réalisation de la tâche saisie par Alice ;

— la `duree_restante` (un entier exprimé en minute) avant la fin de la tâche. Cet attribut sera initialisé avec la durée totale nécessaire à la réalisation de la tâche.

Avancer de `n` minutes (`n` entier positif) dans une tâche consiste à diminuer de `n` la durée restante de cette tâche. Une tâche est terminée si la durée restante est négative ou nulle.
Lors de la phase de planification de ses tâches (aucune d'entre elles n'est commencée), Alice liste les tâches suivantes qui doivent être effectuées :

Numéro	Nom	Durée	Durée restante
1	Répondre aux e-mails	45	45
2	Ranger ma chambre	60	60
3	Réviser la NSI	90	90
4	S'entraîner aux échecs	30	30
5	Apprendre le vocabulaire de chinois	30	30
6	Lire Fondation	60	60
7	Écrire ma lettre au Père Noël	20	20

On dispose de la classe `Tache` ci-dessous pour représenter les tâches :

```
1  class Tache:
2      def __init__(self, numero, nom, duree):
3          self.numero = numero
4          self.nom = nom
5          self.duree_initiale = duree
6          self.duree_restante = duree
7
8      def __repr__(self):
9          return '<t'+str(self.numero)+'>'
```

1. Donner le code Python qui permet d'instancier deux variables `tache1` et `tache2` représentant les tâches :

- tâche numéro 1 : Répondre aux e-mails. Durée estimée : 45 minutes.
- tâche numéro 2 : Ranger ma chambre. Durée estimée : 60 minutes.

On supposera dans la suite que les variables `tache1`, `tache2`, ..., `tache7` représentent les tâches établies par Alice lors de la phase de planification.
La méthode `__repr__` renvoie une représentation de l'instance sous forme d'une chaîne de caractères. La fonction `print` utilise cette méthode. Ainsi on a :

```
>>> print(tache1)
<t1>
```

2. Recopier et compléter le code de la méthode `avancer` de la classe `Tache` qui permet d'avancer la tâche `self` de `n` minutes.

```
1    def avancer(self, n):
2        ...
```

3. Recopier et compléter le code de la méthode `est_terminee` de la classe `Tache` qui renvoie `True` si la tâche est terminée, ou `False` sinon.

```
1    def est_terminee(self):
2        ...
```

Afin d'aider Alice à planifier sa journée, on lui propose d'associer à chacune des tâches une priorité. La priorité d'une tâche est représentée par un entier de la manière suivante : 1 est la priorité minimale et, plus le nombre est grand, plus la tâche associée est prioritaire.
Pour stocker toutes les tâches à effectuer, on utilise une file, dans laquelle les éléments sont des tuples `(tache, priorite)`. Les éléments stockés dans la file doivent respecter les deux conditions ci-après.

— Condition 1 : les éléments sont rangés par ordre décroissant de priorité. L'élément de priorité maximale se trouve au début de la file, l'élément le moins prioritaire se trouve à la fin de la file.

— Condition 2 : parmi les éléments de même priorité, les éléments sont rangés dans l'ordre dans lequel ils ont été insérés dans la file. Ainsi, le premier élément de priorité p inséré se trouve devant les éléments de même priorité p insérés plus tard.

Par exemple, si la file de tâches `f` est la file :

```
[début] (<t3>, 4) (<t1>, 3) (<t2>, 3) (<t4>, 1) (<t5>, 1) [fin]
```

Cela signifie que :

— la tâche de priorité maximale est la tâche numéro 3 ;

— les deux tâches à exécuter en priorité après la tâche numéro 3 sont les tâches numéro 1 et numéro 2. La tâche numéro 1 a été ajoutée à la file des tâches à traiter avant la tâche numéro 2 ;

— il n'y a pas de tâche de priorité 2 ;

— les tâches les moins prioritaires de la file sont les tâches numéro 4 et numéro 5. La tâche numéro 4 a été ajoutée avant la tâche numéro 5.

4. Représenter l'état de la file `f` lorsqu'on lui ajoute successivement la tâche numéro 6 avec la priorité 2, puis la tâche numéro 7 avec la priorité 4 en respectant les conditions 1 et 2 décrites ci-dessus.
On suppose déjà définies les méthodes suivantes pour la classe `File` :

- `File()` : crée et renvoie un objet de type `File`, vide.
- `enfiler(self, e)` : ajoute l'élément `e` à la fin de la file `f`.
- `defiler(self)` : renvoie, en le supprimant de la file, le premier élément de la file si cela est possible.
- `examiner(self)` : renvoie, sans le supprimer de la file, le premier élément de la file si cela est possible.
- `est_vide(self)` : renvoie `True` si la file est vide, ou `False` sinon.

5. En repartant de la file `f` suivante :

```
[début](<t3>, 4)(<t1>, 3)(<t2>, 3)(<t4>, 1)(<t5>, 1)[fin]
```

donner la valeur de `f.defiler()[0]`, et représenter le contenu de la file `f` après l'exécution de cette instruction.
6. En repartant de la file `f` suivante :

```
[début](<t3>, 4)(<t1>, 3)(<t2>, 3)(<t4>, 1)(<t5>, 1)[fin]
```

donner la valeur de `f.examiner()[1]`, et représenter le contenu de la file `f` après l'exécution de cette instruction.
On souhaite écrire une fonction `ajouter_file_prio` qui prend en paramètres :

- une file `f` dont les éléments sont des tuples `(tache, priorite)` respectant les deux conditions de l'énoncé ;
- une tâche `t` ;
- la priorité `p` de la tâche `t` ;

et qui ajoute le tuple `(t, p)` à la bonne position dans la file `f`.
On utilise une file auxiliaire `f_aux` que l'on remplit en défilant les éléments en début de file `f` tant que la priorité du premier élément de la file est supérieure ou égale à `p`. Puis on enfile l'élément `(t, p)` dans la file auxiliaire. On défile ensuite tous les éléments restants de `f` dans `f_aux` et enfin on enfile dans `f` tous les éléments de `f_aux`.
7. Recopier et compléter le code de la fonction `ajouter_file_prio`.

```
1  def ajouter_file_prio(f, t, p):
2      f_aux = File()
3      while ...:
4          ...
5      ...enfiler(...)
6      while not ...:
7          ...
8      while not ...:
9          ...
```

8. Donner le coût d'exécution temporel dans le pire des cas de la fonction `ajouter_file_prio`, en fonction du nombre `m` d'éléments de la file `f`.
Une fois qu'Alice a entré les tâches qu'elle doit effectuer, leur durée estimée, ainsi que la priorité à laquelle elle doit les effectuer, l'application lui propose un planning en utilisant la technique dite Pomodoro :

— la tâche à effectuer est la tâche qui se trouve en tête de file ;

— on défile cette tâche de la file des tâches à effectuer ;

— on avance cette tâche de 25 minutes ;

— si cette tâche n'est pas terminée, on rajoute cette tâche dans la file des tâches à effectuer, avec la même priorité qu'initialement (en utilisant la fonction `ajouter_file_prio`) ;

— si cette tâche se termine au cours des 25 minutes, alors Alice attend la fin des 25 minutes en se reposant ;

— on continue ces étapes tant que la file des tâches à effectuer n'est pas vide.

On rappelle les tâches à effectuer ci-dessous, classées par ordre de priorité. On considérera que les tâches sont ajoutées à la file de priorité dans l'ordre du tableau ci-dessous :

Numéro	Nom	Durée	Priorité
3	Réviser la NSI	90	4
7	Écrire ma lettre au Père Noël	20	4
1	Répondre aux e-mails	45	3
2	Ranger ma chambre	60	3
6	Lire Fondation	60	2
4	S'entraîner aux échecs	30	1
5	Apprendre le vocabulaire de chinois	30	1

9. Indiquer pour chaque bloc de 25 minutes la tâche qui avance, en suivant le modèle proposé, jusqu'à la fin de toutes les tâches.
On fera particulièrement attention au cas où la tâche n'est pas terminée : celle-ci est rajoutée à la file des tâches à effectuer (dont elle avait été supprimée) avec la même priorité qu'initialement, en respectant les conditions 1 et 2 de l'énoncé.
10. Écrire le code d'une fonction `planning` qui prend en paramètre une file de priorité `f` dont les éléments sont des tuples `(tache, prio)`, et qui renvoie une liste de tâches, dans l'ordre dans lequel elles vont être effectuées par tranche de 25 minutes avec la méthode Pomodoro.
Par exemple, si `tache1`, `tache2` et `tache3` sont les tâches numéro 1, numéro 2 et numéro 3, alors le programme suivant :

```
file = File()
for t, p in [(tache1, 3), (tache2, 3), (tache3, 4)]:
    ajouter_file_prio(file, t, p)
print(planning(file))
```

produit l'affichage :

```
[<t3>, <t3>, <t3>, <t3>, <t1>, <t2>, <t1>, <t2>, <t2>]
```

Corrigé

1. On appelle le constructeur de la classe avec, dans l'ordre des paramètres de `__init__`, le numéro, le nom et la durée :

```
tache1 = Tache(1, 'Répondre aux e-mails', 45)
tache2 = Tache(2, 'Ranger ma chambre', 60)
```

Le constructeur initialise lui-même `duree_restante` à 45 (puis 60) : on ne passe que trois

arguments, `self` étant fourni automatiquement.
2. Avancer de `n` minutes, c'est diminuer de `n` l'attribut `duree_restante` :

```
    def avancer(self, n):
        self.duree_restante = self.duree_restante - n
```

(ou `self.duree_restante -= n`). La méthode ne renvoie rien : elle modifie l'état de l'objet. Par exemple, pour une tâche de 45 minutes, `avancer(25)` laisse 20, puis un second `avancer(25)` laisse -5 : la durée restante peut devenir négative, l'énoncé le prévoit.
3. Une tâche est terminée si sa durée restante est négative ou nulle :

```
    def est_terminee(self):
        return self.duree_restante <= 0
```

Ce que le correcteur attend : on renvoie directement le booléen de la comparaison (pas de `if ... return True else return False`), et la comparaison est bien large (≤ 0) : une tâche dont il reste exactement 0 minute est terminée.
4. On applique les deux conditions. La tâche 6 a la priorité 2 : elle se place après tous les éléments de priorité supérieure ou égale à 2 (`<t3>`, `<t1>`, `<t2>`) et avant ceux de priorité 1 :

```
[début] (<t3>, 4) (<t1>, 3) (<t2>, 3) (<t6>, 2) (<t4>, 1) (<t5>, 1) [fin]
```

Puis la tâche 7 a la priorité 4, comme `<t3>` déjà présente : par la condition 2, elle se place *derrière* `<t3>` (insérée plus tôt) mais devant les éléments de priorité 3 :

```
[début] (<t3>, 4) (<t7>, 4) (<t1>, 3) (<t2>, 3) (<t6>, 2) (<t4>, 1) (<t5>, 1) [fin]
```

5. `f.defiler()` retire et renvoie le premier élément de la file, le tuple `(<t3>, 4)` ; l'indice `[0]` en extrait la première composante : la valeur de `f.defiler()[0]` est la tâche `<t3>` (l'objet `tache3`). Comme `defiler` supprime l'élément, la file devient :

```
[début] (<t1>, 3) (<t2>, 3) (<t4>, 1) (<t5>, 1) [fin]
```

6. `f.examiner()` renvoie le premier élément `(<t3>, 4)` *sans* le retirer ; `[1]` en extrait la seconde composante : `f.examiner()[1]` vaut 4, la priorité de la tâche 3. La file n'est pas modifiée :

```
[début] (<t3>, 4) (<t1>, 3) (<t2>, 3) (<t4>, 1) (<t5>, 1) [fin]
```

Ce que le correcteur attend : la différence entre `defiler` (qui consomme la tête) et `examiner` (qui la consulte seulement) est le point de la question.
7. On traduit ligne à ligne la description : on transvase dans `f_aux` la tête de `f` tant que sa priorité (deuxième composante du tuple, d'indice 1) est supérieure ou égale à `p`, on enfile le nouvel élément, on vide le reste de `f` dans `f_aux`, puis on remet tout dans `f` :

```
1  def ajouter_file_prio(f, t, p):
2      f_aux = File()
3      while not f.est_vide() and f.examiner()[1] >= p:
4          f_aux.enfiler(f.defiler())
5      f_aux.enfiler((t, p))
6      while not f.est_vide():
7          f_aux.enfiler(f.defiler())
8      while not f_aux.est_vide():
9          f.enfiler(f_aux.defiler())
```

Le test `>= p` (et non `> p`) garantit la condition 2 : un élément de même priorité déjà présent reste devant le nouveau. La condition `not f.est_vide()` est placée *avant* `f.examiner()` dans le `and` : si `f` est vide, ou si toute la file a été transvasée (nouvelle priorité plus petite que toutes les autres), Python n'évalue pas `examiner()` sur une file vide (évaluation paresseuse). À la ligne 5, c'est un *tuple* que l'on enfile, d'où les doubles parenthèses `enfiler((t, p))` : les parenthèses extérieures sont celles de l'appel, les intérieures celles du couple. Aux lignes 4, 7 et 9, en revanche, on enfile directement l'élément renvoyé par `defiler()`, qui est déjà un tuple. Vérification sur l'exemple de la question 4 : avec `p = 2`, les trois premiers éléments (priorités 4, 3, 3) passent dans `f_aux`, `(<t6>, 2)` est enfilé, puis `(<t4>, 1)` et `(<t5>, 1)` : on retrouve bien la file attendue.

8. Chaque opération du type abstrait file (`enfiler`, `defiler`, `examiner`, `est_vide`) est de coût constant, en $O(1)$: c'est l'hypothèse usuelle, vérifiée par l'implémentation du cours à l'aide d'une file à deux bouts (un `deque`). Dans le pire des cas, la nouvelle priorité `p` est plus petite que toutes celles de la file : la première boucle transvase les m éléments (un tour par élément), la deuxième n'en fait aucun, et la troisième remet les $m+1$ éléments dans `f`. On effectue donc de l'ordre de $2m+2$ opérations élémentaires : le coût est *linéaire*, en $O(m)$. (Si `p` est plus grande que toutes les priorités, c'est la deuxième boucle qui fait m tours : même total. Dans tous les cas, chaque élément de la file est défilé deux fois et enfilé deux fois : une fois de `f` vers `f_aux`, une fois de `f_aux` vers `f`.)

9. On construit d'abord la file en insérant les tâches dans l'ordre du tableau avec `ajouter_file_prio` :

```
[début] (<t3>, 4) (<t7>, 4) (<t1>, 3) (<t2>, 3) (<t6>, 2) (<t4>, 1) (<t5>, 1) [fin]
```

Puis, à chaque bloc de 25 minutes, on défile la tête, on l'avance de 25, et on la réinsère avec sa priorité si elle n'est pas terminée : par la condition 2 elle repasse *derrière* les tâches de même priorité, ce qui fait alterner `<t1>` et `<t2>`, puis `<t4>` et `<t5>`.

Bloc	Minutes	Tâche	Restant	État de la file après le bloc
1	0–25	`<t3>`	65	`(t7,4)(t3,4)(t1,3)(t2,3)(t6,2)(t4,1)(t5,1)`
2	25–50	`<t7>`	-5, finie	`(t3,4)(t1,3)(t2,3)(t6,2)(t4,1)(t5,1)`
3	50–75	`<t3>`	40	`(t3,4)(t1,3)(t2,3)(t6,2)(t4,1)(t5,1)`
4	75–100	`<t3>`	15	`(t3,4)(t1,3)(t2,3)(t6,2)(t4,1)(t5,1)`
5	100–125	`<t3>`	-10, finie	`(t1,3)(t2,3)(t6,2)(t4,1)(t5,1)`
6	125–150	`<t1>`	20	`(t2,3)(t1,3)(t6,2)(t4,1)(t5,1)`
7	150–175	`<t2>`	35	`(t1,3)(t2,3)(t6,2)(t4,1)(t5,1)`
8	175–200	`<t1>`	-5, finie	`(t2,3)(t6,2)(t4,1)(t5,1)`
9	200–225	`<t2>`	10	`(t2,3)(t6,2)(t4,1)(t5,1)`
10	225–250	`<t2>`	-15, finie	`(t6,2)(t4,1)(t5,1)`
11	250–275	`<t6>`	35	`(t6,2)(t4,1)(t5,1)`
12	275–300	`<t6>`	10	`(t6,2)(t4,1)(t5,1)`
13	300–325	`<t6>`	-15, finie	`(t4,1)(t5,1)`
14	325–350	`<t4>`	5	`(t5,1)(t4,1)`
15	350–375	`<t5>`	5	`(t4,1)(t5,1)`
16	375–400	`<t4>`	-20, finie	`(t5,1)`
17	400–425	`<t5>`	-20, finie	file vide

La suite des tâches qui avancent, bloc par bloc, est donc :

```
<t3> <t7> <t3> <t3> <t3> <t1> <t2> <t1> <t2> <t2> <t6> <t6> <t6> <t4> <t5> <t4> <t5>
```

soit 17 blocs, 425 minutes. Contrôle : chaque tâche occupe $\lceil \text{durée}/25 \rceil$ blocs, $4+1+2+3+3+2+2=17$.

Ce que le correcteur attend : au bloc 1, `<t3>` non terminée est réinsérée avec la priorité 4 *derrière* `<t7>` (condition 2) ; c'est donc `<t7>` qui avance au bloc 2, et non `<t3>` une seconde fois. De même `<t1>` et `<t2>` alternent, puis `<t4>` et `<t5>`. Ce mécanisme est celui de l'ordonnancement par tourniquet (*round robin*) d'un système d'exploitation, avec un quantum de 25 minutes et des niveaux de priorité : d'où le thème « gestion de processus » annoncé.

10. La fonction reproduit la boucle de la question 9 et accumule dans une liste la tâche traitée à chaque bloc :

```
def planning(f):
    resultat = []
    while not f.est_vide():
        tache, prio = f.defiler()
        tache.avancer(25)
        resultat.append(tache)
        if not tache.est_terminee():
            ajouter_file_prio(f, tache, prio)
    return resultat
```

On dépaquette le tuple de tête en `tache, prio` pour réinsérer la tâche avec sa priorité initiale. Sur l'exemple du sujet, la file construite est `(<t3>, 4)(<t1>, 3)(<t2>, 3)` ; la tâche 3 (90 min) occupe quatre blocs de suite (elle est seule à la priorité 4), puis `<t1>` (45 min) et `<t2>` (60 min) alternent : `<t1>` (reste 20), `<t2>` (35), `<t1>` (finie), `<t2>` (10), `<t2>` (finie). On obtient bien `[<t3>, <t3>, <t3>, <t3>, <t1>, <t2>, <t1>, <t2>, <t2>]`, l'affichage attendu (les `<tk>` viennent de `__repr__`). Terminaison : à chaque tour de boucle, la tâche défilée perd 25 minutes, donc le nombre total de blocs encore nécessaires, somme des $\lceil$`duree_restante`$/25\rceil$ des tâches de la file, diminue strictement de 1 (une tâche terminée n'est plus réinsérée) : cet entier positif est un variant, la file finit par être vide.

Exercice 3 (8 points)

Sujet — Exercice 3 — 8 points

Cet exercice porte sur l'architecture matérielle (réseau), les arbres binaires de recherche et la programmation Python.

L'entreprise CaféNet possède plusieurs cafés répartis dans différentes villes. Le réseau de la chaîne de cafés est représenté en Figure 1.

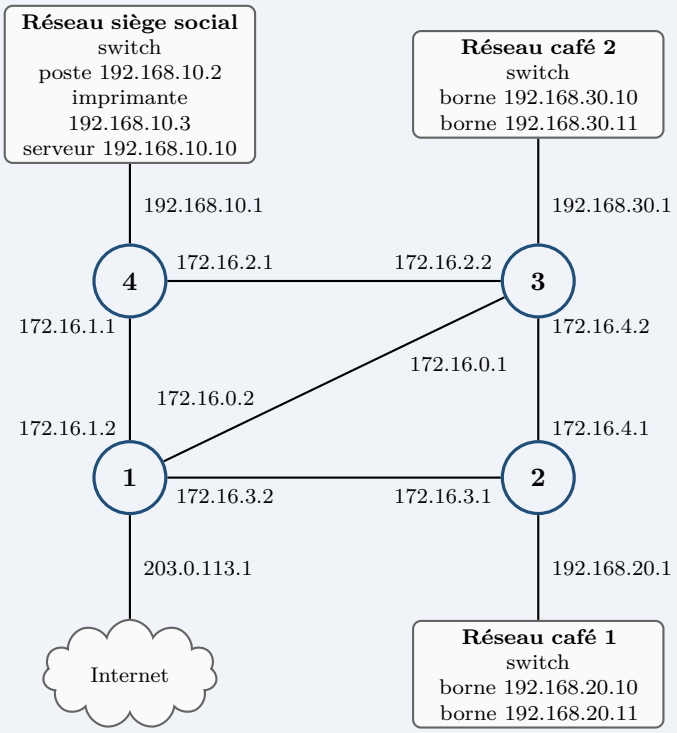

Figure 1. Schéma d'une partie du réseau

Sur le schéma sont représentés 4 routeurs, le réseau du siège social, le réseau du café 1, le réseau du café 2. Dans les réseaux du café 1 et du café 2, des bornes de commandes sont connectées à des switchs (ce sont des boitiers de connexion qui n'ont pas eux-mêmes d'adresse IP). Les 4 routeurs représentés sont composés d'au moins 3 interfaces réseau capable de relier des réseaux ensemble. Chaque interface possède donc une adresse IPV4 sur le réseau auquel elle est reliée.

Les masques des sous-réseaux sont tous 255.255.255.0. Avec ce masque, les trois premiers octets des adresses IP codent l'adresse réseau. Le dernier octet, c'est-à-dire les 8 derniers bits, code l'adresse des machines à l'intérieur de chaque sous-réseau.

Partie A Le gérant veut faire installer une troisième borne de commande dans le café 1.

1. Indiquer les deux seules adresses IP valides pour cette nouvelle borne, parmi les quatre adresses IP proposées.

(a) `192.168.20.2` (c) `192.168.20.261`
(b) `192.168.20.157` (d) `192.168.24.10`

L'adresse de diffusion, appelée aussi adresse de broadcast, est la dernière adresse disponible à l'intérieur d'un réseau local.
2. Déterminer l'adresse de diffusion du réseau du café 1.
3. Déterminer combien de machines informatiques il est encore possible de connecter au réseau du café 1 après l'installation de la troisième borne de commande.
Le réseau local du café 1 n'a pas besoin de plus de 8 adresses IP différentes. Ce décompte d'adresses IP inclut les adresses IP réservées (à savoir l'adresse de diffusion et l'adresse du réseau). Il est rappelé que la longueur du masque de sous-réseau est actuellement de 24 bits (c'est-à-dire 3 octets).
4. Expliquer quelle est la longueur maximale du masque de sous-réseau que l'on pourrait choisir pour le réseau local du café 1.

Partie B RIP (*Routing Information Protocol*) est un protocole de routage utilisé dans les réseaux IP. Il est conçu pour réduire le nombre de sauts entre deux réseaux. Un « saut » correspond au transfert des données d'un routeur à un autre. Le protocole RIP utilise le nombre de sauts comme critère principal pour évaluer le coût d'un chemin. Autrement dit, il considère que le chemin le plus optimal est celui qui traverse le moins de routeurs.
La table de routage du routeur 2 de la Figure 1 est représentée ci-dessous :

Routeur 2

Réseau destination	Interface de sortie	Prochain routeur	Nombre de sauts
192.168.20.0	192.168.20.1	aucun	0
172.16.3.0	172.16.3.1	aucun	0
172.16.4.0	172.16.4.1	aucun	0
192.168.10.0	172.16.3.1	172.16.3.2	2
172.16.0.0	172.16.4.1	172.16.4.2	1
172.16.2.0	172.16.4.1	172.16.4.2	1
192.168.30.0	...	...	...
172.16.1.0	...	...	...

5. Recopier et compléter les deux dernières lignes de la table de routage du routeur 2.
La table de routage du routeur 2 contient un réseau de destination pour lequel deux routes différentes sont possibles. La ligne correspondante dans la table de routage aurait donc pu être remplie différemment tout en respectant le protocole RIP.
6. Identifier, dans la table de routage du routeur 2, le réseau de destination que l'on peut atteindre d'une autre façon et indiquer comment cette ligne de la table de routage pourrait être modifiée.
Une adresse IP qui n'est pas référencée dans la table de routage doit être routée par défaut vers Internet.
7. Recopier et compléter la ligne à ajouter à la table de routage du routeur 2.

Réseau destination	Interface de sortie	Prochain routeur
autre	...	...

Partie C OSPF est également un protocole d'échanges de données entre les routeurs qui prend en compte le coût des routes. Le coût est lié au débit des liaisons entre les routeurs par la formule suivante :

$$\text{coût} = \frac{10^9}{\text{débit}} \quad \text{avec le débit en bit} \cdot \text{s}^{-1}.$$

8. Recopier et compléter la dernière colonne du tableau ci-dessous :

Tableau des coûts

Type de connexion	Débit en bit $\cdot$ s^{-1}	coût
Ethernet	10 Mbit $\cdot$ s^{-1} $= 10^7$ bit $\cdot$ s^{-1}	100
Fast Ethernet	100 Mbit $\cdot$ s^{-1} $= 10^8$ bit $\cdot$ s^{-1}	...
Fibre optique	1 Gbit $\cdot$ s^{-1} $= 10^9$ bit $\cdot$ s^{-1}	...

Le schéma ci-dessous met en évidence les types de connexion qui relient les routeurs.

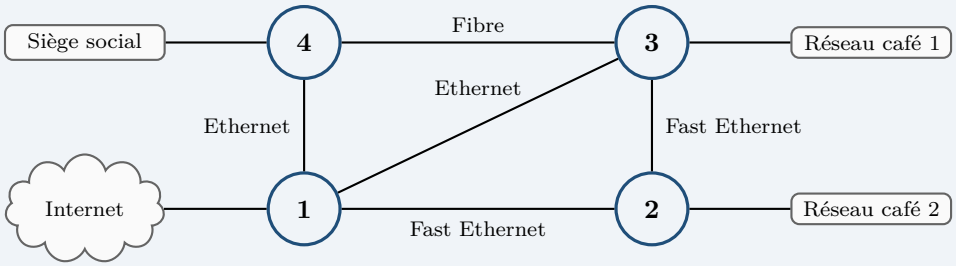

Figure 2. Schéma des types de connexion

9. Déterminer la route dont le coût est minimal pour aller du routeur 1 jusqu'au routeur 4 et calculer son coût au sens du protocole OSPF.

Partie D Le but de cette partie est de classer les adresses IP des différents réseaux afin de faciliter leur recherche.
La fonction `ip_bin` prend en argument une chaîne de caractères décrivant une adresse IP en notation décimale, et renvoie une chaîne de caractères, de longueur 35 (32 bits et les 3 points), décrivant l'adresse IP en notation binaire. Exemple :

```
>>> ip_bin('192.168.10.1')
'11000000.10101000.00001010.00000001'
```

10. Donner la chaîne de caractères renvoyée par `ip_bin('192.168.20.12')`.
La fonction `precede` prend en paramètres deux adresses IP en notation binaire, sous forme de chaînes de caractères identiques à celles renvoyées par la fonction `ip_bin`. La fonction `precede` renvoie un booléen qui vaut `True` si la première adresse IP en paramètre précède la seconde adresse IP. Exemple :

```
>>> a = '11000000.10101000.00001010.00000001'
>>> b = '11000000.10101000.00001111.00000001'
>>> precede(a, b)
True
```

L'algorithme compare bit à bit les deux chaînes binaires, en lisant les chaînes de caractères dans le sens usuel (de gauche à droite). Dans l'exemple ci-dessus, tous les caractères sont identiques

jusqu'au sixième caractère du troisième octet. Comme le bit de l'adresse `a` est inférieur à celui de l'adresse `b`, on en déduit que l'adresse IP `a` précède l'adresse IP `b`.
Si la première adresse IP ne précède pas la seconde, la fonction doit renvoyer `False`. L'algorithme de comparaison est traduit dans le langage Python sous la forme suivante :

```
1  def precede(ip_1, ip_2):
2      for i in range(35):
3          if ip_1[i] < ip_2[i]:
4              return ...
5          elif ip_1[i] > ip_2[i]:
6              return ...
7      return ...
```

11. Expliquer dans quel cas la fonction `precede` exécutera la dernière instruction `return` de la ligne 7.
12. Recopier et compléter les lignes 4, 6 et 7 du code de la fonction `precede`.
Les tables de routage de chaque routeur sont implémentées sous la forme d'arbre binaire de recherche avec la classe `Abr`.

```
1  class Abr:
2      def __init__(self, adresse_ip,
3                   interface, passerelle,
4                   cout):
5          self.adresse_ip = adresse_ip
6          self.interface  = interface
7          self.passerelle = passerelle
8          self.cout       = cout
9          if adresse_ip != '':
10             self.gauche = Abr('','','',0)
11             self.droite = Abr('','','',0)
12
13     def est_vide(self):
14         return ...
```

Dans cette représentation :

- `adresse_ip` désigne l'adresse IP de la destination ;
- `interface` désigne l'interface réseau ;
- `passerelle` désigne l'adresse IP du prochain routeur ;
- `cout` désigne le nombre de sauts pour atteindre la destination ;
- par convention, l'arbre binaire vide est une instance de `Abr` pour laquelle `adresse_ip` est une chaîne de caractères vide ;
- un arbre binaire de recherche non vide possède nécessairement un sous-arbre gauche et un sous-arbre droit, éventuellement vides, qui sont tous les deux des arbres binaires de recherche. Ces sous-arbres sont désignés par `gauche` et `droite` dans la classe `Abr` ;
- si elle n'est pas vide, l'adresse IP du sous-arbre gauche précède l'adresse IP de l'instance parent ;
- si le sous-arbre droit n'est pas vide, alors l'adresse IP de l'instance parent précède l'adresse IP du sous-arbre droit.

13. Citer un attribut et citer une méthode de la classe `Abr`.
14. Recopier et compléter la ligne 14 du code de la classe `Abr`.

15. Justifier, en mobilisant des connaissances de cours, l'intérêt qu'il peut y avoir à représenter la table de routage par un arbre binaire de recherche.
La section de code qui définit `modifie` est incluse dans la classe `Abr`.

```
16      def modifie(self, adresse_ip,
17                  interface, passerelle,
18                  cout):
19          if self.est_vide():
20              self.adresse_ip = adresse_ip
21              self.interface  = interface
22              self.passerelle = passerelle
23              self.cout       = cout
24              self.gauche = Abr('','','',0)
25              self.droite = Abr('','','',0)
26          else:
27              self.adresse_ip = adresse_ip
28              self.interface  = interface
29              self.passerelle = passerelle
30              self.cout       = cout
```

Les lignes 20 à 23 sont exactement les mêmes que les lignes 27 à 30.
16. Réécrire le code de la fonction `modifie` en évitant cette répétition.
La classe `Abr` est complétée afin de permettre l'ajout de nouvelles lignes à la table de routage, tout en conservant les propriétés que doit posséder un arbre binaire de recherche.

```
32      def rechercher(self, adresse_ip):
33          if self.est_vide() or adresse_ip==self.adresse_ip:
34              return self
35          elif precede(...):
36              return self.gauche.rechercher(adresse_ip)
37          else:
38              return self.droite.rechercher(adresse_ip)
39
40      def inserer(self, adresse_ip,
41                  interface, passerelle,
42                  cout):
43          destination = self.rechercher(adresse_ip)
44          destination.modifie(adresse_ip,
45                              interface, passerelle,
46                              cout)
```

On rappelle que la fonction `precede` prend en arguments des adresses IP écrites sous forme binaire.
17. Recopier et compléter la ligne 35 du code de la fonction `rechercher`.

Corrigé

Partie A **1.** Le réseau du café 1 est le réseau `192.168.20.0` de masque `255.255.255.0` : les trois premiers octets `192.168.20` sont imposés à toute machine du réseau, et seul le dernier octet, codé sur 8 bits, distingue les machines.

- (a) `192.168.20.2` : **valide**. Elle appartient bien au réseau du café 1, son dernier octet est compris entre 1 et 254, et elle est encore libre (le routeur 2 occupe `.1`, les deux bornes existantes `.10` et `.11`).

– (b) `192.168.20.157` : **valide**, pour les mêmes raisons.
– (c) `192.168.20.261` : impossible. Un octet est codé sur 8 bits, donc compris entre 0 et $2^8 - 1 = 255$: 261 ne s'écrit pas sur un octet.
– (d) `192.168.24.10` : le troisième octet vaut 24 et non 20. Cette adresse appartient à un *autre* réseau ; la borne ne pourrait pas dialoguer avec les machines du café 1 sans passer par un routeur.

Les deux adresses valides sont donc **(a)** et **(b)**.
2. L'adresse de diffusion est la dernière adresse du réseau local, c'est-à-dire celle dont tous les bits de la partie machine valent 1 : le dernier octet vaut $11111111_2 = 255$. L'adresse de diffusion du café 1 est `192.168.20.255`.
3. Le dernier octet offre $2^8 = 256$ combinaisons, mais deux sont réservées : `192.168.20.0` (adresse du réseau) et `192.168.20.255` (adresse de diffusion). Il reste $256 - 2 = 254$ adresses attribuables. Sont déjà attribuées : `192.168.20.1` (interface du routeur 2), `192.168.20.10` et `192.168.20.11` (les deux bornes), plus la nouvelle borne, soit 4 adresses. Les switchs, eux, n'ont pas d'adresse IP et ne comptent pas. On peut donc encore connecter $254 - 4 = 250$ machines.
4. Il faut que le réseau contienne au moins 8 adresses, adresse du réseau et adresse de diffusion comprises. Or $8 = 2^3$: il faut *au moins* 3 bits pour la partie machine, donc *au plus* $32 - 3 = 29$ bits pour le masque. La longueur maximale du masque est **29 bits**, soit `255.255.255.248` en notation décimale. Un masque de 30 bits ne laisserait que $2^2 = 4$ adresses, ce qui est insuffisant. En passant de 24 à 29 bits, on libère les $256 - 8 = 248$ autres adresses pour d'autres sous-réseaux.

Partie B **5.** On raisonne à partir de la Figure 1, en comptant les transferts de routeur à routeur.

Réseau destination	Interface de sortie	Prochain routeur	Nombre de sauts
192.168.30.0	172.16.4.1	172.16.4.2	1
172.16.1.0	172.16.3.1	172.16.3.2	1

`192.168.30.0` est le réseau du café 2, relié au routeur 3 : depuis le routeur 2, on sort par l'interface `172.16.4.1` vers le routeur 3 (`172.16.4.2`), soit 1 saut. L'autre chemin, par le routeur 1 puis le routeur 3, coûterait 2 sauts : RIP retient le plus court.
`172.16.1.0` est le réseau qui relie le routeur 4 au routeur 1 : le routeur 1 y possède l'interface `172.16.1.2`, il suffit donc de l'atteindre. On sort par `172.16.3.1` vers le routeur 1 (`172.16.3.2`), soit 1 saut ; en passant par le routeur 3 puis le routeur 4, il en faudrait 2.
6. La réponse attendue est le réseau `192.168.10.0` (le siège social, relié au routeur 4) : il est atteignable de deux façons de même coût : par le routeur 1 puis le routeur 4 (2 sauts, comme l'indique la table), ou par le routeur 3 puis le routeur 4 (2 sauts également). La ligne pourrait donc s'écrire :

Réseau destination	Interface de sortie	Prochain routeur	Nombre de sauts
192.168.10.0	172.16.4.1	172.16.4.2	2

Ce que le correcteur attend : le nombre de sauts, lui, ne change pas (les deux routes ont le même coût au sens de RIP) ; seuls l'interface de sortie et le prochain routeur changent. Honnêteté du corrigé : le réseau `172.16.0.0`, qui relie les routeurs 1 et 3, est en réalité dans la même situation : il est à 1 saut par le routeur 3 (`172.16.4.1` / `172.16.4.2`, comme dans la table) comme par le routeur 1 (`172.16.3.1` / `172.16.3.2`) ; l'énoncé n'en attend qu'un, et `192.168.10.0` est celui dont les deux routes empruntent des chemins vraiment distincts. Une

copie qui cite `172.16.0.0` en le justifiant ne dit rien de faux. Ce sont les deux cycles du graphe des routeurs qui créent cette ambiguïté ; RIP ne la tranche pas, le routeur conserve la première route apprise.

7. Internet est relié au routeur 1, dont l'interface porte l'adresse `203.0.113.1`. Depuis le routeur 2, tout paquet dont la destination n'est pas dans la table doit donc partir vers le routeur 1 :

Réseau destination	Interface de sortie	Prochain routeur
autre	172.16.3.1	172.16.3.2

L'interface de sortie est celle du routeur 2 sur cette liaison, et le prochain routeur est l'adresse de l'interface du routeur 1 sur la même liaison. Cette ligne, la route par défaut, est examinée en dernier : elle ne sert que si aucune autre ne correspond.

Partie C **8.** On applique coût $= 10^9/$débit :

Type de connexion	Débit	coût
Ethernet	10^7	$10^9/10^7 = 100$
Fast Ethernet	10^8	$10^9/10^8 = 10$
Fibre optique	10^9	$10^9/10^9 = 1$

Le coût de Fast Ethernet vaut 10, celui de la fibre optique vaut 1. Plus le débit est grand, plus le coût est petit : OSPF préfère les liaisons rapides.

9. La Figure 2 donne le type de chaque liaison, donc son coût : routeur 4 – routeur 3 en fibre (coût 1) ; routeur 4 – routeur 1 en Ethernet (100) ; routeur 1 – routeur 3 en Ethernet (100) ; routeur 1 – routeur 2 en Fast Ethernet (10) ; routeur 3 – routeur 2 en Fast Ethernet (10). On énumère les chemins du routeur 1 au routeur 4 :

Chemin	Coût
$1 \to 4$	100
$1 \to 3 \to 4$	$100 + 1 = 101$
$1 \to 2 \to 3 \to 4$	$10 + 10 + 1 = 21$

La route de coût minimal est **routeur 1** $\to$ **routeur 2** $\to$ **routeur 3** $\to$ **routeur 4**, de coût **21**.

Ce que le correcteur attend : la route la plus courte en nombre de sauts (la liaison directe, 1 saut) n'est pas la moins coûteuse au sens d'OSPF. Traverser trois liaisons rapides coûte 21, quand une seule liaison Ethernet lente coûte 100. C'est toute la différence entre RIP, qui compte les routeurs traversés, et OSPF, qui tient compte du débit. La Figure 2 échange par ailleurs les étiquettes des deux cafés par rapport à la Figure 1 ; cela ne change rien au calcul, qui ne porte que sur les liaisons entre routeurs.

Partie D **10.** On convertit chaque octet en binaire sur 8 bits : $192 = 128 + 64 = 11000000_2$, $168 = 128 + 32 + 8 = 10101000_2$, $20 = 16 + 4 = 00010100_2$, $12 = 8 + 4 = 00001100_2$.

```
>>> ip_bin('192.168.20.12')
'11000000.10101000.00010100.00001100'
```

La chaîne compte bien 35 caractères : 32 chiffres binaires et 3 points.

11. La boucle `for` examine les 35 caractères. Le premier caractère où les deux chaînes diffèrent déclenche immédiatement un `return` (ligne 4 si `ip_1[i]` est le plus petit, ligne 6 sinon) et

la fonction s'arrête. La ligne 7 n'est donc atteinte que si la boucle est allée jusqu'au bout sans jamais trouver de différence : les deux chaînes sont identiques, c'est-à-dire que les deux paramètres désignent *la même adresse IP*. Les points occupent les mêmes positions dans les deux chaînes et ne peuvent jamais créer d'écart.

12. Une adresse ne se précède pas elle-même : le cas de la ligne 7 doit donc renvoyer `False`, comme le cas où la première adresse est la plus grande.

```
1  def precede(ip_1, ip_2):
2      for i in range(35):
3          if ip_1[i] < ip_2[i]:
4              return True
5          elif ip_1[i] > ip_2[i]:
6              return False
7      return False
```

Vérification sur l'exemple du sujet : avec `a` et `b`, les 23 premiers caractères coïncident ; au vingt-quatrième (le sixième bit du troisième octet) on a `a[23] = '0'` et `b[23] = '1'`, donc `'0' < '1'` et la fonction renvoie `True`. Elle renvoie `False` pour `precede(b, a)` et pour `precede(a, a)`. La comparaison `<` porte ici sur des *caractères* : elle compare leurs codes, et `'0' < '1'`, ce qui donne bien l'ordre des bits.

13. Un attribut : `adresse_ip` (on accepte aussi `interface`, `passerelle`, `cout`, `gauche`, `droite`). Une méthode : `est_vide` (on accepte aussi `__init__`, `modifie`, `rechercher`, `inserer`). Un attribut est une donnée portée par chaque instance, accessible par `self` ; une méthode est une fonction définie dans la classe, dont le premier paramètre est `self`.

14. Par convention, l'arbre est vide lorsque son adresse IP est la chaîne vide :

```
13     def est_vide(self):
14         return self.adresse_ip == ''
```

On renvoie directement le booléen de la comparaison ; attention à l'opérateur de comparaison `==` et non à l'affectation `=`.

15. Une table de routage est consultée à chaque paquet transmis : la vitesse de recherche est décisive. Rangée dans une liste, la table impose une recherche séquentielle, donc n comparaisons dans le pire des cas pour n lignes : un coût *linéaire*. Dans un arbre binaire de recherche, chaque comparaison avec la racine élimine d'un coup tout un sous-arbre : on descend d'un niveau à chaque étape, et le nombre de comparaisons est majoré par la hauteur de l'arbre. Si l'arbre est équilibré, cette hauteur est de l'ordre de $\log_2 n$: le coût est *logarithmique*. Pour une table de 1 000 lignes, cela fait une dizaine de comparaisons au lieu d'un millier. De plus, le parcours infixe de l'arbre rend les adresses déjà triées, et l'insertion d'une nouvelle ligne ne demande aucun décalage. *Réserve à signaler :* l'avantage disparaît si l'arbre dégénère en peigne (adresses insérées déjà triées) : la recherche redevient linéaire.

16. Les quatre affectations sont communes aux deux cas : on les sort du test. Seule la création des deux sous-arbres vides est propre au cas où le nœud était vide. Il faut donc mémoriser la réponse de `est_vide()` *avant* de modifier `adresse_ip` :

```
16     def modifie(self, adresse_ip,
17                 interface, passerelle,
18                 cout):
19         etait_vide = self.est_vide()
20         self.adresse_ip = adresse_ip
21         self.interface  = interface
22         self.passerelle = passerelle
23         self.cout       = cout
```

```
        if etait_vide:
            self.gauche = Abr('','','',0)
            self.droite = Abr('','','',0)
```

Ce que le correcteur attend : si l'on garde `if self.est_vide():` *après* les affectations, le test est toujours faux, puisque `self.adresse_ip` vient de recevoir une adresse non vide : les sous-arbres ne sont jamais créés et l'insertion suivante échoue sur `AttributeError: 'Abr' object has no attribute 'gauche'`. C'est le piège de la question.

17. L'arbre est ordonné par la relation `precede`, qui attend des adresses *binaires* : il faut donc convertir les deux adresses avec `ip_bin` avant de les comparer. Si l'adresse cherchée précède celle du nœud, elle ne peut se trouver que dans le sous-arbre gauche :

```
    def rechercher(self, adresse_ip):
        if self.est_vide() or adresse_ip==self.adresse_ip:
            return self
        elif precede(ip_bin(adresse_ip), ip_bin(self.adresse_ip)):
            return self.gauche.rechercher(adresse_ip)
        else:
            return self.droite.rechercher(adresse_ip)
```

Le cas d'arrêt de cette fonction récursive est double : ou bien l'adresse est trouvée, ou bien on aboutit à un arbre vide, qui est précisément l'emplacement où `inserer` viendra écrire la nouvelle ligne par un appel à `modifie`.

Vérification : en insérant dans un arbre vide les huit lignes de la table du routeur 2 (avec les deux lignes complétées à la question 5), le parcours infixe rend les adresses dans l'ordre croissant, `172.16.0.0`, `172.16.1.0`, `172.16.2.0`, `172.16.3.0`, `172.16.4.0`, `192.168.10.0`, `192.168.20.0`, `192.168.30.0` : la propriété de l'arbre binaire de recherche est bien respectée. Un `inserer` portant sur une adresse déjà présente met la ligne à jour sans créer de doublon, puisque `rechercher` retourne le nœud existant.