BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2025

NUMÉRIQUE ET SCIENCES INFORMATIQUES

ÉPREUVE DU MARDI 17 JUIN 2025

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 17 pages numérotées de 1/17 à 17/17 dans la version originale et **29 pages numérotées de 1/29 à 29/29 dans la version en caractères agrandis.**

Le sujet est composé de trois exercices indépendants.
Le candidat traite les trois exercices.

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.

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 page agrandie précédente.

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 :

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

produit l'affichage :

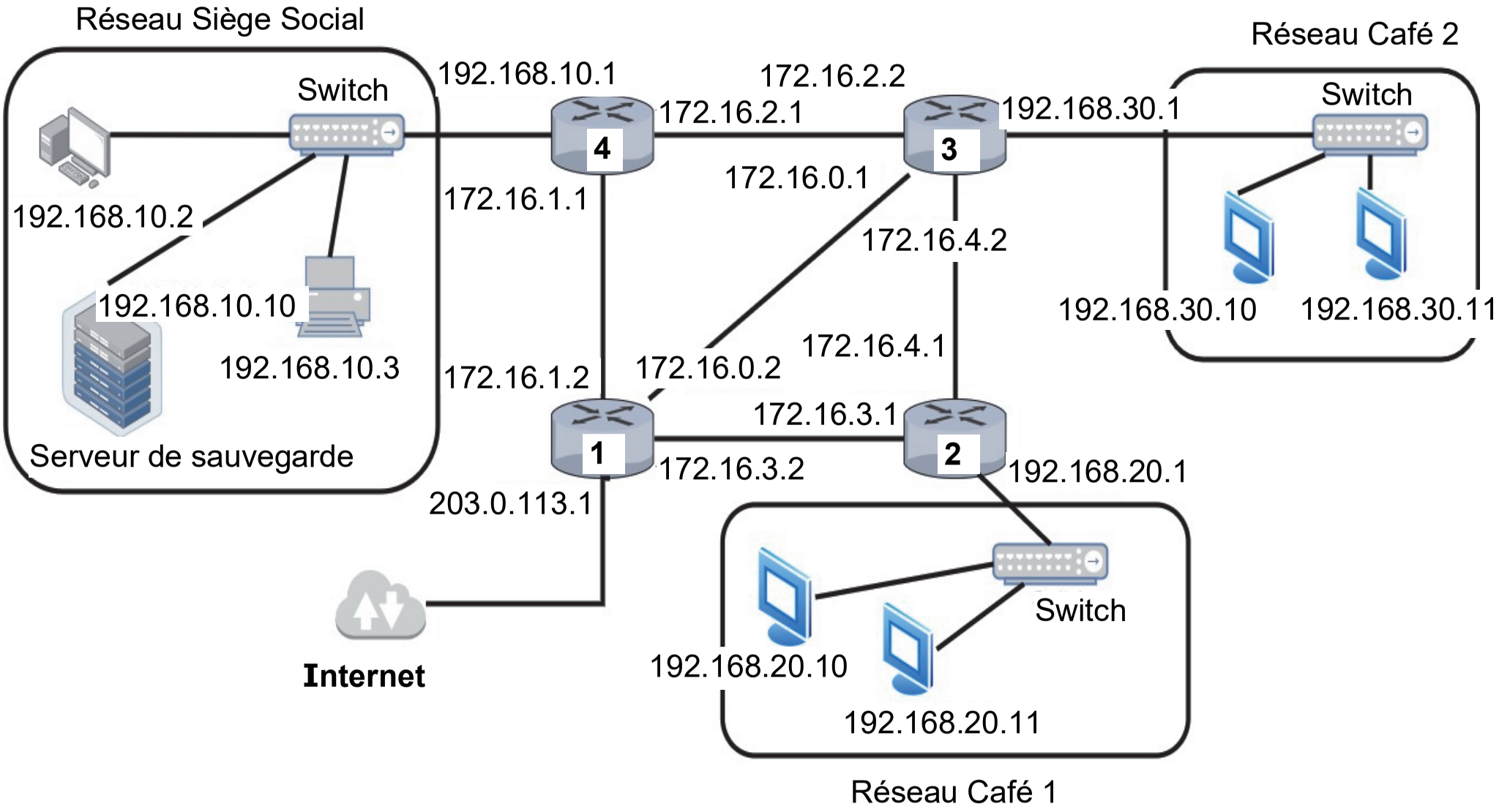
```
[<t3>, <t3>, <t3>, <t3>, <t1>, <t2>, <t1>, <t2>, <t2>]
```

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 page agrandie suivante.

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
(b) 192.168.20.157
(c) 192.168.20.261
(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 :

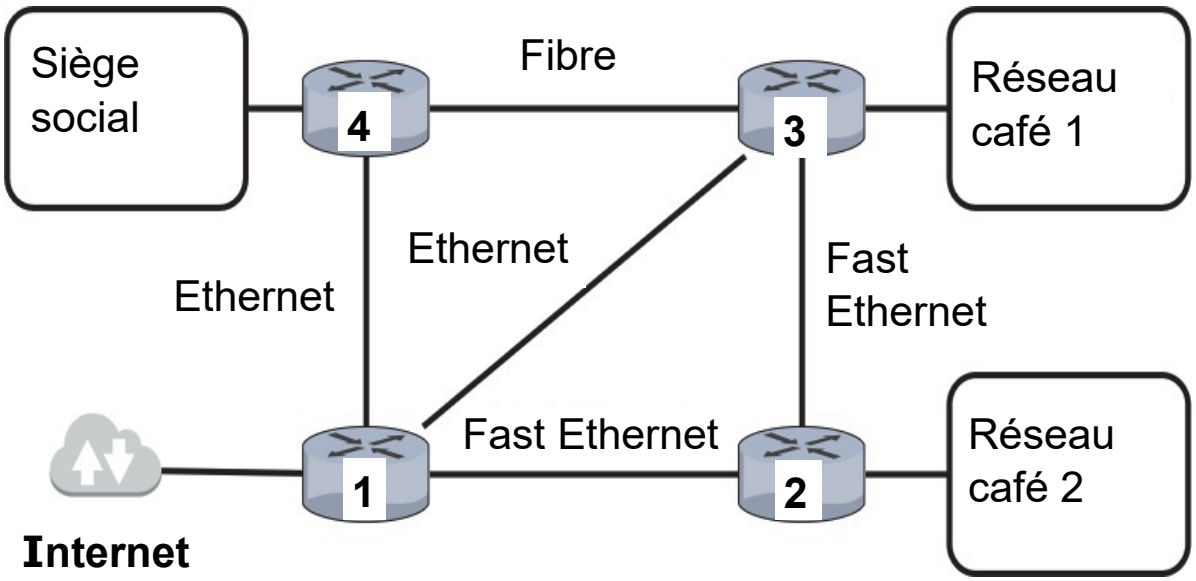
$cout = \frac{10^9}{debit}$ avec le débit en $bit.s^{-1}$.

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

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

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`.