BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2026

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 1

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 16 pages numérotées de 1/16 à 16/16.

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, la programmation de base en Python et l'algorithmique.

La compagnie de danse *L'air de l'art* fait des présentations de spectacle. Les danseurs et danseuses d'un spectacle se déplacent donc dans une camionnette avec le matériel nécessaire.

Pour gérer la logistique, Jessica, la secrétaire de la compagnie, a mis en place une base de données.

Partie A

La base de données permet à Jessica de relier les danseurs aux différentes chorégraphies et représentations. Pour simplifier, seules les villes seront mentionnées dans les adresses.

La base de données se schématise comme suit :

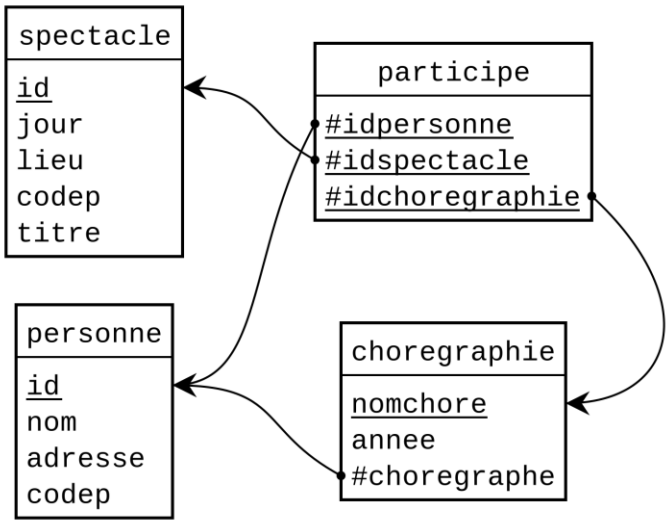


Figure 1. Schéma relationnel de la base

- La table `personne` est composée d'un identifiant `id`, d'un `nom`, d'une `adresse` et du code postal associé `codep`.
- La table `spectacle` est composée d'un identifiant `id`, d'un `jour` de représentation (au format `AAAA-MM-JJ`), d'une adresse du `lieu`, de son code postal associé `codep` et du titre de spectacle `titre`.
- La table `choregraphie` est composée d'un `nomchore` unique, d'une `annee` de création et de l'identifiant de son `choregraphe`.

- La table `participe` associe une personne qui danse `idpersonne`, un spectacle `idspectacle` et une chorégraphie `idchoregraphie`. La clef primaire est le triplet `idpersonne`, `idspectacle` et `idchoregraphie`.
- Les clefs primaires sont soulignées et les clefs étrangères sont précédées du symbole «#».

On pourra utiliser les mots clefs du langage SQL suivants : `SELECT, DISTINCT, FROM, WHERE, JOIN … ON, UPDATE … SET, DELETE, INSERT INTO … VALUES`.

La fonction d'agrégation `COUNT()` permet de compter le nombre d'enregistrements dans une table. Par exemple, la requête SQL ci-dessous permet de connaitre le nombre de lignes dans la table `ma_table` :

```
SELECT COUNT(*) FROM ma_table;
```

Un code postal est composé de cinq chiffres.

1. Écrire une requête qui ajoute la chorégraphie 'Tout autour' de Rachid Ouramdane (`id` n°9) datant de 2015 à la table `choregraphie`.
2. Écrire une requête qui affiche le nombre de chorégraphies de 1983.
3. Décrire par une phrase le résultat obtenu lors de l'exécution de la requête ci-dessous :

```
SELECT choregraphie.nomchore, personne.nom
FROM choregraphie
JOIN personne
ON choregraphie.choregraphe = personne.id;
```

4. Écrire une requête qui affiche le nom des danseurs ayant dansé dans le spectacle ayant pour titre 'Tout autour'.
5. Préciser les précautions à prendre pour supprimer un spectacle de la table `spectacle`. On ne demande pas d'écrire la requête.

Partie B

Jessica a en charge d'amener les costumes et accessoires pour les représentations. Afin de minimiser les allers-retours, elle essaie d'optimiser le chargement de son véhicule à l'aide d'un programme.

Pour chaque représentation, elle a récupéré les informations utiles sous forme d'un dictionnaire de dictionnaires. Les sous-dictionnaires contiennent le nombre d'éléments nécessaires, le volume unitaire ainsi que le poids unitaire. Par exemple :

```
dmateriel = {
    'costume_A': {'nb': 3, 'volume':2, 'poids':1},
    'costume_B': {'nb': 1, 'volume':6, 'poids':2},
```

```
    'chaise_bois': {'nb': 40, 'volume':50, 'poids':4},
    'chapeau_A': {'nb': 2, 'volume':3, 'poids':1},
    'chapeau_B': {'nb': 4, 'volume':7, 'poids':1},
}
```

6. Recopier et compléter la ligne 3 du code ci-dessous afin que `n` soit égal au nombre total d'éléments nécessaires dans le dictionnaire `dmateriel`.

   Avec l'exemple précédent, à la fin de l'exécution du script, `n` doit valoir : $3 + 1 + 40 + 2 + 4 = 50$.

```
1 n = 0
2 for clef in dmateriel:
3     n += dmateriel[...][...]
```

Afin d'obtenir deux tableaux (de type `list`), l'un avec les clefs, l'autre avec les volumes associés, Jessica a écrit la fonction suivante :

```
1 def tab_clefs_tab_volumes(dmatos):
2     tab_clefs = []
3     tab_volumes = []
4     for clef, dico in dmatos.items():
5         tab_clefs.append(...)
6         tab_volumes.append(...)
7     return tab_clefs, tab_volumes
```

L'appel `tab_clefs_tab_volumes(dmateriel)` devra renvoyer le tuple :

```
(['costume_A', 'costume_B', 'chaise_bois', 'chapeau_A',
'chapeau_B'], [2, 6, 50, 3, 7])
```

7. Recopier et compléter les lignes 5 et 6 pour rendre effective cette fonction.

Pour optimiser le volume chargé dans son véhicule, Jessica se dit qu'une bonne stratégie serait de remplir son véhicule du matériel le plus volumineux en premier puis de finir de le remplir par du petit matériel.

8. Préciser à quel type d'algorithme vu en cours cette stratégie fait référence.

Pour le moment, on suppose que la fonction `clefs_triees_selon_volume` prend en paramètre un dictionnaire `dmatos` contenant le matériel et renvoie la liste des clefs de `dmatos` triée dans l'ordre croissant des volumes unitaires associés à chacune des clefs. Ainsi on a :

```
>>> clefs_triees_selon_volume(dmateriel)
['costume_A', 'chapeau_A', 'costume_B', 'chapeau_B',
'chaise_bois']
```

9. Citer un algorithme de tri vu en cours et donner un ordre de grandeur de son coût en temps d'exécution en fonction de la taille `n` de la liste à trier.

Jessica a commencé à coder une fonction `remplissage`, qui prend en paramètres un dictionnaire `dmatos` contenant le matériel, un entier `vmax`, qui renvoie la liste des éléments à charger dans une voiture sans dépasser le volume maximal `vmax` et qui modifie `dmatos` en mettant à jour le nombre d'éléments restants.

Par exemple :

```
>>> remplissage(dmateriel, 160)
['chaise_bois', 'chaise_bois', 'chaise_bois', 'chapeau_B',
'chapeau_A']
>>> print(dmateriel)
{
    'costume_A': {'nb': 3, 'volume':2, 'poids':1},
    'costume_B': {'nb': 1, 'volume':6, 'poids':2},
    'chaise_bois': {'nb': 37, 'volume':50, 'poids':4},
    'chapeau_A': {'nb': 1, 'volume':3, 'poids':1},
    'chapeau_B': {'nb': 3, 'volume':7, 'poids':1},
}
```

10. Recopier et compléter les lignes 6 et de 11 à 15 de la fonction `remplissage` ci-dessous :

```
 1 def remplissage(dmatos, vmax):
 2     vol = 0
 3     clefs_triees = clefs_triees_selon_volume(dmatos)
 4     i = len(clefs_triees)-1
 5     elements = []
 6     while i >= ... and vol < ...:
 7         clef = clefs_triees[i]
 8         if dmatos[clef]['nb'] == 0:
 9             i -= 1
10         elif vol + dmatos[clef]['volume'] <= vmax:
11             vol += ...
12             elements.append(...)
13             dmatos[clef]['nb'] -= 1
14         else:
15             ...
16     return elements
```

On remarquera que la fonction `remplissage` renvoie une liste vide s'il n'y a plus de matériel à charger dans `dmatos` (les valeurs associées à `'nb'` sont toutes égales à 0).

11. Écrire une fonction `nb_voitures` qui prend en paramètres un dictionnaire `dmatos` contenant le matériel, un entier `vmax` et renvoie le nombre de voitures nécessaires pour charger l'ensemble du matériel (on supposera que toutes les voitures ont la même capacité `vmax`). On pourra faire appel à la fonction remplissage, même si le code n'a pas été complété.

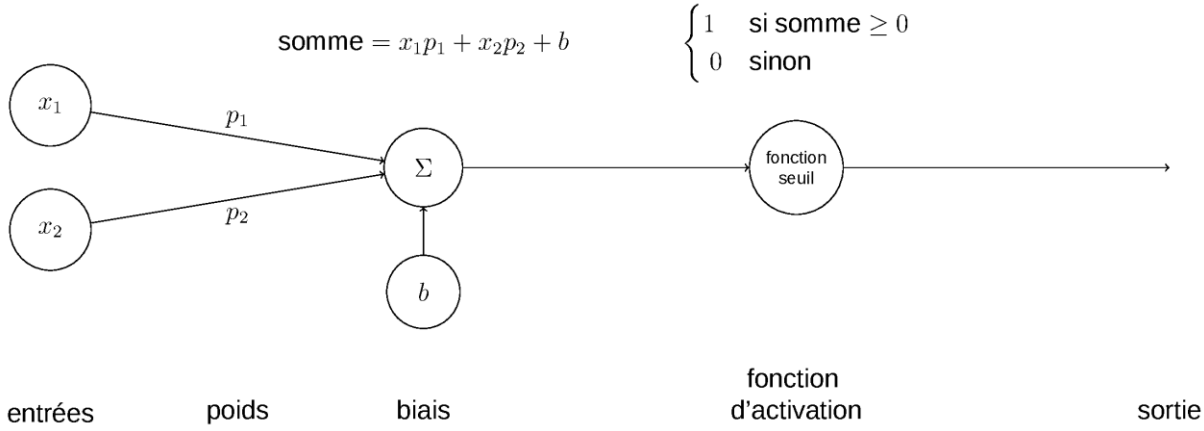
Exercice 2 (6 points)

Cet exercice porte sur les thèmes langages, programmation et algorithmique.

Dans le cadre d'un concours national intitulé "Jeunes Talents IA 2025", une équipe constituée d'élèves de terminale NSI est sélectionnée pour développer un système d'intelligence artificielle capable de détecter si un email est un spam ou non pour la messagerie d'un établissement scolaire. Après réflexion, l'équipe décide d'utiliser un perceptron pour classer les messages comme spam ou non spam.

Le perceptron est un neurone artificiel qui imite très simplement le fonctionnement d'un neurone biologique. L'équipe choisit d'entraîner le perceptron à l'aide d'un algorithme d'apprentissage supervisé : on présente au perceptron des exemples, avec les bonnes réponses, afin qu'il ajuste automatiquement ses poids dans le but d'améliorer ses prédictions.

Le fonctionnement d'un perceptron est décrit ci-après :



Le perceptron dispose d'un tuple de n poids $p = (p_1, p_2, p_3, \ldots, p_n)$ et d'un biais b. Il reçoit en entrée un tuple de n valeurs numériques d'entrée $x = (x_1, x_2, x_3, \ldots, x_n)$.

Il calcule la somme pondérée avec biais des valeurs d'entrées :
$somme = x_1 \times p_1 + x_2 \times p_2 + \ldots + x_n \times p_n + b$.

Puis il applique une fonction d'activation. Dans cet exercice la fonction d'activation est la fonction seuil qui renvoie 1 si la somme pondérée avec biais est supérieure ou égale à 0, et 0 sinon.

Le schéma ci-dessus illustre le cas $n = 2$.

Pour commencer, l'équipe considère un perceptron à deux entrées ayant pour caractéristiques les poids $p_1 = 0.7$ et $p_2 = -0.2$, et un biais $b = 0.1$.

1. Calculer la valeur $somme$ de la somme pondérée avec biais du perceptron lorsque les valeurs d'entrée sont $x_1 = 1$ et $x_2 = 3$ et en déduire la valeur de sortie $sortie$ du perceptron si la fonction d'activation est la fonction seuil (sortie à 1 si somme supérieure ou égale à 0, sinon 0).

Voici le début d'une classe `DetecteurSpam` basée sur un perceptron :

```
 1 class DetecteurSpam:
 2     def __init__(self, nb_entrees):
 3         self.biais = 0.0
 4         self.poids = [0.0 for i in range(nb_entrees)]
 5
 6     def get_biais(self):
 7         return self.biais
 8
 9     def set_biais(self, nouveau_biais):
10         self.biais = nouveau_biais
11
12     def get_poids(self):
13         return self.poids
14
15     def set_poids(self, nouveaux_poids):
16         assert len(nouveaux_poids) == len(self.poids)
17         self.poids = nouveaux_poids
```

2. Donner le nom d'une méthode et le nom d'un attribut de cette classe.

3. Donner une suite d'instructions Python permettant de créer un objet `detecteur` (instance de la classe `DetecteurSpam`) correspondant à un détecteur à deux entrées, puis de remplacer ses poids et son biais par les valeurs `p1 = 0.7`, `p2 = -0.2` et `b = 0.1` en utilisant les méthodes de la classe.

4. Expliquer le rôle de la ligne 16 du code (`assert len(nouveau_poids) == len(self.poids)`).

5. Écrire la méthode `fonction_activation` qui prend en paramètres `valeur` un nombre et qui renvoie 1 si `valeur` est supérieur ou égal à 0, et 0 sinon.

La méthode `prediction` ci-après :

- prend en paramètre une liste `entrees` de nombres ;
- calcule la somme pondérée avec biais des valeurs de la liste `entrees` ;
- applique la fonction d'activation ;
- renvoie la valeur de sortie de la fonction d'activation.

```
1     def prediction(self, entrees):
2         somme = 0
3         for i in range(len(entrees)):
4             somme += ...
5         somme += ...
```

```
6           sortie = self.fonction_activation(...)
7           return ...
```

On considère que le nombre d'éléments de la liste `entrees` est identique au nombre d'entrées (ou paramètres) du détecteur. Exemple d'utilisation :

```
>>> # exemple avec un détecteur à trois entrées
>>> detecteur.get_biais()
0.2
>>> detecteur.get_poids()
[0.5, -0.1, 0.2]
>>> detecteur.prediction([0.1, 0.3, 0.2])
1
```

6. Recopier et compléter la méthode `prediction`.

7. Donner le coût d'exécution temporel de la méthode `prediction` en fonction de la taille n des entrées.

La méthode `entrainement` ci-après permet d'ajuster les poids et le biais du perceptron selon un algorithme d'apprentissage supervisé. Cette méthode prend en paramètres une liste `echantillon` d'entrainement constituée de listes de nombres, une liste `cibles` de bonnes réponses attendues, un taux d'apprentissage `taux` et renvoie le nombre d'erreurs obtenues à la fin de l'entrainement.

```
 1      def entrainement(self, echantillon, cibles, taux=0.1):
 2          nb_erreurs = 0
 3          for i in range(len(echantillon)):
 4              entrees = echantillon[i]
 5              cible = ...
 6              # calcul de la prédiction
 7              predict = ...
 8              if ...:
 9                  # prédiction incorrecte
10                  nb_erreurs += 1
11                  erreur = cible - predict
12                  for j in range(len(self.poids)):
13                      self.poids[j] += ... * ... * entrees[j]
14                  self.biais += taux * erreur
15          return nb_erreurs
```

Pour l'ajustement des poids et du biais, on applique la méthode `prediction` pour chaque élément de la liste `echantillon` :

- si la prédiction est correcte, on ne change rien ;
- si la prédiction est incorrecte :
  - on calcule l'erreur `erreur = cible - prediction` ;

– on ajuste les poids selon la règle : `nouveau_poids = ancien_poids + taux * erreur * entree` ;
– puis on ajuste le biais selon la règle : `nouveau_biais = ancien_biais + taux * erreur`.

Lorsque tous les éléments de la liste `echantillon` ont été traités, on dit que l'on a réalisé une **époque** d'entrainement.

Un des membres de l'équipe propose les listes `echantillon` et `cibles` suivantes pour tester la méthode `entrainement`.

```
1 echantillon = [[0.5, 0.3], [0.2, 0.1], [0.4, 0.8], [0.7,
0.9]]
2 cibles = [0, 1, 1, 1]
```

8. Donner les valeurs de `echantillon[1]` et `echantillon[2][1]`.

9. Recopier et compléter les lignes 5, 7, 8 et 13 du code de la méthode `entrainement`.

L'équipe veut maintenant créer un détecteur de spam à 2 caractéristiques d'entrée et l'entraîner sur les données du tableau ci-dessous pour apprendre à détecter les emails avec "URGENT" ET "GRATUIT".

Contient "URGENT" ?	Contient "GRATUIT" ?	Est un SPAM ?
0 (non)	0 (non)	0 (non)
0 (non)	1 (oui)	0 (non)
1 (oui)	0 (non)	0 (non)
1 (oui)	1 (oui)	1 (oui)

10. Écrire un programme permettant de créer et d'entrainer le détecteur sur les listes `emails = [[0, 0], [0, 1], [1, 0], [1, 1]]` et `spams = [0, 0, 0, 1]`. pendant au maximum 1000 époques, puis d'afficher dans la console le nombre d'époques nécessaires pour que le détecteur ne fasse plus d'erreurs (si c'est possible), et -1 sinon.

Exercice 3 (8 points)

Cet exercice porte sur les protocoles de routage, la programmation orientée objet et la notion de file.

Le but de cet exercice est de simuler la façon dont des routeurs construisent leur table de routage selon le protocole RIP.

On considère à titre d'exemple le réseau ci-dessous :

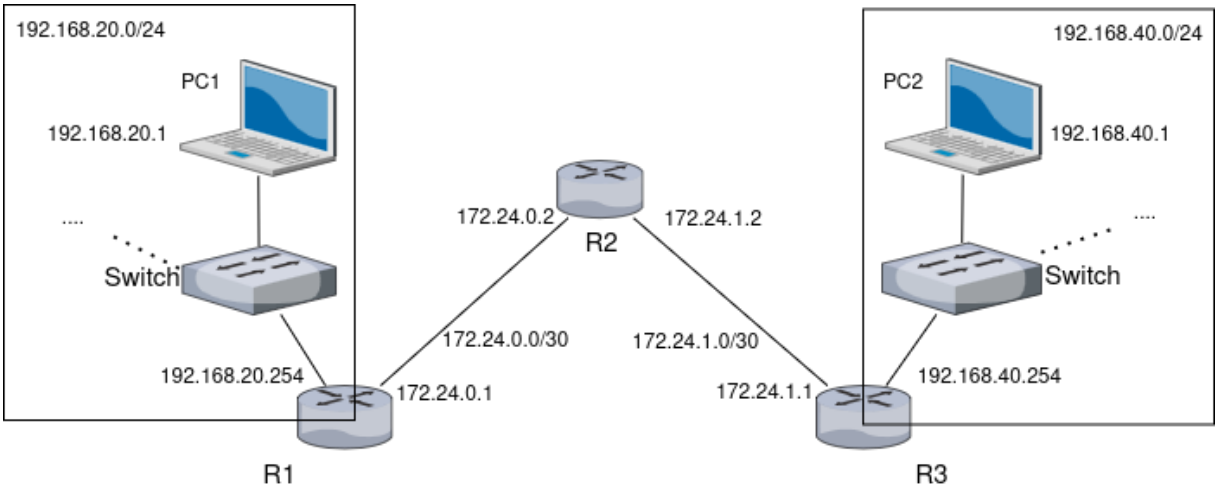


Figure 1. Exemple de réseau

Les adresses des quatre réseaux sont notées sous la forme `IP/S` où `IP` est une adresse IP donnée par quatre octets écrits en base 10 et `S` est un entier inférieur à 32 appelé *suffixe*. Une adresse IP peut être écrite sur 32 bits en binaire. Le suffixe indique que les `S` premiers bits de l'adresse correspondent à la partie fixe de l'adresse et les suivants à la partie propre à la machine.

Pour que deux machines soient sur un même réseau local avec un suffixe `S`, il faut que les `S` premiers bits de leurs adresses IP soient identiques.

Partie A

1. Identifier la partie fixe des adresses IP des machines branchées dans le réseau `192.168.20.0/24`.

2. Dans chaque réseau, il y a deux adresses réservées. Donner le nombre maximum de machines qui peuvent être branchées dans un réseau dont le suffixe est 30.

Le protocole RIP permet de minimiser la distance, c'est-à-dire le nombre de routeurs traversés par un paquet de données. Voici la table de routage du routeur R1 dans le réseau représenté sur la figure 1, selon le protocole RIP :

Table de routage de R1			
Destination	Interface	Passerelle	Distance
192.168.20.0/24	192.168.20.254	–	0
172.24.0.0/30	172.24.0.1	–	0
172.24.1.0/30	172.24.0.1	172.24.0.2	1
192.168.40.0/24	172.24.0.1	172.24.0.2	2

3. Donner la table de routage du routeur R2 pour le réseau représenté sur la figure 1, selon le protocole RIP.

Dans les parties B et C, les adresses sont représentées en Python par des tuples de 4 entiers. Par exemple, l'adresse `192.168.40.1` est représentée par `(192, 168, 40, 1)`.

Partie B

Afin de comparer les adresses IP pour savoir si elles appartiennent à un même réseau, on calcule le *masque de réseau* à partir du suffixe. Les premiers bits, correspondant à la partie fixe, sont mis à 1 et les autres sont mis à 0.

4. Donner le tuple d'entiers correspondant à un masque pour un réseau dont le suffixe est 30.

Pour isoler la partie fixe d'une adresse IP, on utilise le ET bit à bit. Cela consiste à appliquer l'opération booléenne ET à tous les bits se trouvant à la même position dans les deux nombres écrits en binaire.

```
   01110110  (118)
ET 11010010  (210)
-----------
   01010010   (82)
```

En Python, il est possible de faire cette opération en utilisant l'opérateur `&` directement entre deux nombres écrits en base 10 :

```
>>> 118 & 210
82
```

5. Expliquer pourquoi `255 & A` est égal à `A`, si `A` est un entier compris entre 0 et 255.

6. Déterminer, en justifiant la réponse, si `(172, 20, 189, 8)` et `(172, 20, 191, 203)` sont dans un même réseau dont le masque est `(255, 255, 252, 0)`.

La fonction `meme_reseau` prend en paramètres trois tuples de quatre entiers `adresse1`, `adresse2` et `masque`, et renvoie un booléen indiquant si `adresse1` et `adresse2` sont dans le même réseau en utilisant le masque de réseau `masque`.

```
def meme_reseau(adresse1, adresse2, masque):
    for i in range(4):
        octet1 = adresse1[...] & masque[...]
        octet2 = ...
        if ...:
            return ...
    return ...
```

7. Recopier et compléter le code de la fonction `meme_reseau`.

Partie C

Nous allons maintenant définir les classes permettant de simuler un réseau de réseaux locaux.

On dispose d'une classe `File` qui permet d'instancier une file. Cette classe dispose, en plus du constructeur qui renvoie une file vide, des méthodes `enfiler`, `defiler` et `est_vide` : pour une file `file1`,

- `file1.enfiler(e)` ajoute l'élément `e` à la fin de la file `file1` et ne renvoie rien ;
- `file1.defiler()` enlève le premier élément de la file et le renvoie ;
- `file1.est_vide()` renvoie `True` si la file `file1` est vide et `False` sinon.

Chaque réseau local est modélisé à l'aide de la classe `Reseau` dont voici le constructeur :

```
class Reseau:
    def __init__(self, adresse, masque):
        self.adresse = adresse
        self.masque = masque
        self.machines = {}
        self.file = File()
```

Dans cette modélisation, c'est le réseau qui s'occupe de l'acheminement des paquets. Lorsqu'une machine envoie un paquet, il est ajouté à la file du réseau. Pour simplifier, tout paquet envoyé sur un réseau local sera distribué à toutes les machines de ce réseau, à l'exception de celle qui a émis le paquet. Un paquet est un couple de la forme `(orig, donnees)` où `orig` est l'adresse IP de la machine envoyant le paquet et `donnees` correspond aux données envoyées.

Cette classe possède également un méthode `connexion`, non présentée ici, qui prend en paramètre une file et qui permet d'attribuer une nouvelle adresse à une machine, en associant cette adresse à la file dans le dictionnaire `machines` du

réseau concerné puis de renvoyer l'adresse. La méthode `diffuser`, de la classe `Reseau` permet de traiter les paquets contenus dans sa file et de les déposer dans les files des machines destinataires.

```
    def diffuser(self):
        while not ...:
            paquet = ...
            orig, donnees = paquet
            for m in self.machines:
                if ...:
                    self.machines[m].enfiler(...)
```

8. Recopier et compléter le code de la méthode `diffuser`.

La classe `Routeur` permet de définir les routeurs. Ils possèdent une interface pour chaque réseau auquel ils sont connectés. On utilise pour cela la classe `Interface`. Chaque interface d'un routeur est identifiée par son indice dans le tableau `interfaces`. On donne ci-dessous une partie du code de ces deux classes :

```
class Interface:
    def __init__(self, reseau):
        self.sortie = reseau.file
        self.file = File()
        self.adresse = reseau.connexion(self.file)
        self.masque = reseau.masque

class Routeur:
    def __init__(self, nom):
        self.nom = nom
        self.interfaces = []
        self.table = {}

    def branchement(self, reseau):
        k = len(self.interfaces)
        self.table[reseau.adresse] = (k, None, 0)
        self.interfaces.append(Interface(reseau))
```

Les routeurs communiquent entre eux en envoyant des *vecteurs de distance* qui sont des couples (réseau, distance). Par exemple, le routeur R1 du réseau représenté sur la figure 1 envoie à ses voisins les vecteurs `((192, 168, 20, 0), 0)`, `((172, 24, 0, 0), 0)`, `((172, 24, 1, 0), 1)` et `((192, 168, 40, 0), 2)`.

La construction des tables de routage, selon RIP, repose sur l'algorithme de Bellman-Ford :

- Initialement, la table de chaque routeur ne contient que les réseaux locaux auxquels il est connecté, qui sont à la distance 0.
- Lorsque les vecteurs d'un voisin sont reçus :

- Si la table du voisin contient une route vers un nouveau réseau local, ce réseau est ajouté à la table.
- Si elle contient une route vers un réseau déjà connu et que cette nouvelle route est plus courte, la table est mise à jour.
- Si elle contient une route vers un réseau qui n'améliore pas le chemin actuel, la table n'est pas modifiée.

Les tables de routage seront représentées sous la forme de dictionnaires associant à chaque adresse réseau un triplet `(i_interf, passerelle, distance)`, où `i_interf` est l'indice de l'interface dans le tableau `interfaces` du routeur. Lorsque le routeur est directement connecté au réseau local, on met `None` à la place de la passerelle.

La méthode `m_a_j` de la classe `Routeur` prend trois paramètres, en plus de `self` :

- `passerelle` qui est l'adresse de l'interface du routeur qui envoie ses vecteurs ;
- `i_interf` qui est l'indice dans `self.interfaces` de l'interface concernée du routeur dont on fait la mise à jour ;
- `vecteurs` qui est le tableau des vecteurs de distance reçus.

Cette méthode met à jour la table de routage selon l'algorithme de Bellman-Ford. Si la table a été modifiée suite à cette mise à jour, le routeur utilise la méthode `envoyer_vecteurs` qui prend en paramètre, en plus de `self`, un entier `i_interf` qui correspond à l'interface par laquelle il a reçu les informations et qui envoie les vecteurs sur toutes les autres interfaces. Cette méthode sera étudiée par la suite.

Prenons l'exemple d'une instance de la classe `Routeur`, qui serait contenue dans une variable `routeur_m` :

```
>>> routeur_m.table
{(172, 64, 0, 0): (1, None, 0),
 (172, 60, 0, 0): (0, (172, 58, 0, 2), 1),
 (192, 168, 47, 0): (0, (172, 58, 0, 2), 2),
 (172, 62, 0, 0): (0, (172, 58, 0, 2), 2)}
```

On considère également le tableau de vecteurs suivant :

```
vecteurs = [((192, 168, 72, 0), 0), ((172, 62, 0, 0), 0),
            ((172, 64, 0, 0), 0), ((172, 60, 0, 0), 1),
            ((192, 168, 47, 0), 1)]
```

9. Déterminer la valeur de `routeur_m.table` après l'appel `routeur_m.m_a_j((172, 64, 0, 2), 1, vecteurs)`.

10. Recopier et compléter les lignes 4 et 6 du code ci-après de la méthode `m_a_j`.

```
1     def m_a_j(self, passerelle, i_interf, vecteurs):
2         a_ete_modif = False
3         for adresse, d in vecteurs:
4             if ... or ...:
5                 self.table[adresse] = (i_interf,
passerelle, d+1)
6                 a_ete_modif = ...
7         if a_ete_modif:
8             self.envoyer_vecteurs(i_interf)
```

La méthode `vecteurs` de la classe `Routeur` permet de générer, pour le routeur considéré, le tableau des vecteurs de distance à envoyer via une interface dont l'indice `i` est donné. Pour éviter de propager des erreurs ou de générer des cycles, selon le protocole RIP, les vecteurs ne contiennent pas les routes qui passent par l'interface sur laquelle ils sont envoyés.

```
1     def vecteurs(self, i):
2         res = []
3         for adresse in self.table:
4             i_interf, passerelle, dist = self.table[adresse]
5             if ...:
6                 res.append(...)
7         return res
```

11. Recopier et compléter les lignes 5 et 6 du code de la méthode `vecteurs`.

On rappelle que lorsque les vecteurs reçus sur une interface `i_interf` entraînent une modification de la table de routage d'un routeur, ce routeur envoie ses vecteurs sur toutes les autres interfaces. La méthode `envoyer_vecteurs` ci-après prend en paramètre, en plus de `self`, un entier `i_interf` et envoie les vecteurs sur toutes les autres interfaces.

```
1     def envoyer_vecteurs(self, i_interf):
2         for i in range(len(self.interfaces)):
3             if ...:
4                 adresse = self.interfaces[i].adresse
5                 v = self.vecteurs(...)
6                 paquet = (adresse, v)
7                 ... .enfiler(paquet)
```

12. Recopier et compléter les lignes 3 à 7 du code de la méthode `envoyer_vecteurs`.

La méthode `traitement` de la classe `Routeur` parcourt toutes les files des interfaces et traite les paquets qu'elles contiennent.

```
1     def traitement(self):
2         for i in range(len(self.interfaces)):
3             file = self.interfaces[i].file
4             while not file.est_vide():
```

```
5                   orig, donnees = file.defiler()
6                   self.m_a_j(orig, i, donnees)
7
```

Afin de simuler la construction des tables de routage, nous allons procéder de la manière suivante, une fois les routeurs connectés aux différents réseaux :

- Chaque routeur va envoyer ses vecteurs initiaux sur toutes ses interfaces (on utilise la valeur `-1` pour le paramètre `i_interf` pour ne sauter aucune interface). Les routeurs sont tous contenus dans le tableau `routeurs`.
- On diffuse les paquets se trouvant dans toutes les files des réseaux (qui sont tous contenus dans le tableau `reseaux`).
- On utilise la méthode `traitement` de tous les routeurs afin de mettre à jour leurs tables et éventuellement de diffuser à nouveaux des vecteurs.
- S'il n'y a plus de paquets en attente dans les réseaux, alors on a fini.

13. Recopier et compléter la suite du script ci-dessous à partir de la ligne 4 afin de construire les tables de routage des routeurs contenus dans le tableau `routeurs` suivant l'algorithme ci-dessus.

```
1 for routeur in routeurs:
2     routeur.envoyer_vecteurs(-1)
3
4 continuer = True
5 while continuer:
6     ...
```