EXERCICE 1

Cet exercice porte sur la programmation Python, la programmation orientée objet, les structures de données (file), l'ordonnancement et l'interblocage.

On s'intéresse aux processus et à leur ordonnancement au sein d'un système d'exploitation. On considère ici qu'on utilise un monoprocesseur.

1. Citer les trois états dans lesquels un processus peut se trouver.

On veut simuler cet ordonnancement avec des objets. Pour ce faire, on dispose déjà de la classe `Processus` dont voici la documentation :

Classe Processus:

```
p = Processus(nom: str, duree: int)
    Crée un processus de nom <nom> et de durée <duree> (exprimée en
cycles d'ordonnancement)

p.execute_un_cycle()
    Exécute le processus donné pendant un cycle.

p.est_fini()
    Renvoie True si le processus est terminé, False sinon.
```

Pour simplifier, on ne s'intéresse pas aux ressources qu'un processus pourrait acquérir ou libérer.

2. Citer les deux seuls états possibles pour un processus dans ce contexte.

Pour mettre en place l'ordonnancement, on décide d'utiliser une file, instance de la classe `File` ci-dessous.

Classe File

```
1  class File:
2      def __init__(self):
3          """ Crée une file vide """
4          self.contenu = []
5
6      def enfile(self, element):
7          """ Enfile element dans la file """
8          self.contenu.append(element)
9
10     def defile(self):
11         """ Renvoie le premier élément de la file et l'enlève de
la file """
12         return self.contenu.pop(0)
13
14     def est_vide(self):
```

```
15          """ Renvoie True si la file est vide, False sinon """
16          return self.contenu == []
```

Lors de la phase de tests, on se rend compte que le code suivant produit une erreur :

```
1 f = File()
2 print(f.defile())
```

3. Rectifier sur votre copie le code de la classe `File` pour que la function `defile` renvoie `None` lorsque la file est vide.

On se propose d'ordonnancer les processus avec une méthode du type *tourniquet* telle qu'à chaque cycle :

- si un nouveau processus est créé, il est mis dans la file d'attente ;
- ensuite, on défile un processus de la file d'attente et on l'exécute pendant un cycle ;
- si le processus exécuté n'est pas terminé, on le replace dans la file.

Par exemple, avec les processus suivants

Liste des processus		
processus	cycle de création	durée en cycles
A	2	3
B	1	4
C	4	3
D	0	5

On obtient le chronogramme ci-dessous :

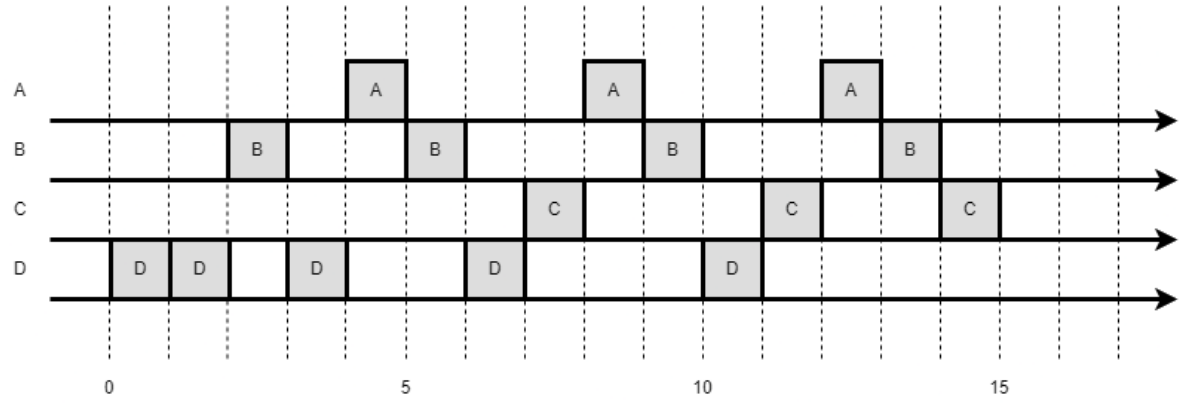

Figure 1. Chronogramme pour les processus A, B, C et D

Pour décrire les processus et le moment de leur création, on utilise le code suivant, dans lequel `depart_proc` associe à un cycle donné le processus qui sera créé à ce moment :

```
1 p1 = Processus("p1", 4)
2 p2 = Processus("p2", 3)
3 p3 = Processus("p3", 5)
4 p4 = Processus("p4", 3)
5 depart_proc = {0: p1, 1: p3, 2: p2, 3: p4}
```

Il s'agit d'une modélisation de la situation précédente où un seul processus peut être créé lors d'un cycle donné.

4. Recopier et compléter sur votre copie le chronogramme ci-dessous pour les processus `p1`, `p2`, `p3` et `p4`.

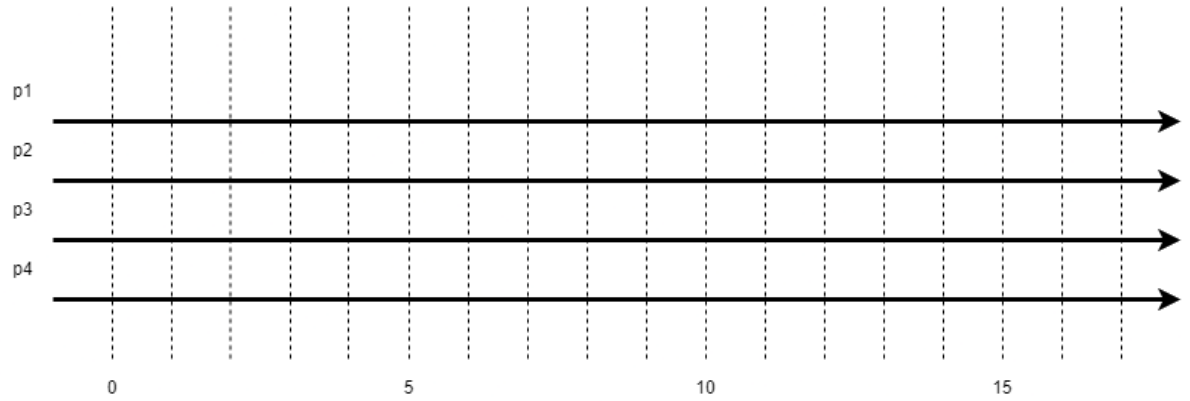

Figure 2. Chronogramme pour les processus p1, p2, p3 et p4

Pour mettre en place l'ordonnancement suivant cette méthode, on écrit la classe `Ordonnanceur` dont voici un code incomplet (l'attribut `temps` correspond au cycle en cours) :

```
1  class Ordonnanceur:
2
3      def __init__(self):
4          self.temps = 0
5          self.file = File()
6
7      def ajoute_nouveau_processus(self, proc):
8          '''Ajoute un nouveau processus dans la file de
9            l'ordonnanceur. '''
10          ...
11
12      def tourniquet(self):
13          '''Effectue une étape d'ordonnancement et renvoie le nom
14            du processus élu.'''
15          self.temps += 1
16          if not self.file.est_vide():
17              proc = ...
18              ...
19              if not proc.est_fini():
20                  ...
21              return proc.nom
22          else:
23              return None
```

5. Compléter le code ci-dessus.

À chaque appel de la méthode `tourniquet`, celle-ci renvoie soit le nom du processus qui a été élu, soit `None` si elle n'a pas trouvé de processus en cours.

6. Écrire un programme qui :

- utilise les variables `p1`, `p2`, `p3`, `p4` et `depart_proc` définies précédemment ;
- crée un ordonnanceur ;
- ajoute un nouveau processus à l'ordonnanceur lorsque c'est le moment ;
- affiche le processus choisi par l'ordonnanceur ;
- s'arrête lorsqu'il n'y a plus de processus à exécuter.

Dans la situation donnée en exemple (voir Figure 1), il s'avère qu'en fait les processus utilisent des ressources comme :

- un fichier commun aux processus ;
- le clavier de l'ordinateur ;
- le processeur graphique (GPU) ;
- le port 25000 de la connexion Internet.

Voici le détail de ce que fait chaque processus :

Liste des processus			
A	B	C	D
acquérir le GPU	acquérir le clavier	acquérir le port	acquérir le fichier
faire des calculs	acquérir le fichier	faire des calculs	faire des calculs
libérer le GPU	libérer le clavier	libérer le port	acquérir le clavier
	libérer le fichier		libérer le clavier
			libérer le fichier

7. Montrer que l'ordre d'exécution donné en exemple aboutit à une situation d'interblocage.

EXERCICE 2

Cet exercice porte sur les graphes, la programmation, la structure de pile et l'algorithmique des graphes.

On s'intéresse à la fabrication de pain. La recette est fournie sous la forme de tâches à réaliser. Cette recette est réalisée par une personne seule.

(a) Préparer 500g de farine.
(b) Préparer 1/3 de litre d'eau (33cl).
(c) Préparer 1 c. à café de sel.
(d) Préparer 20g de levure de boulanger.
(e) Faire tiédir l'eau dans une casserole.
(f) Délayer la levure dans l'eau tiède.
(g) Laisser reposer la levure 5 minutes.
(h) Préparer un grand saladier.
(i) Verser la farine dans le saladier.
(j) Verser le sel dans le saladier.
(k) Mélanger la farine et le sel puis creuser un puits.
(l) Verser l'eau mélangée à la levure dans le puits.
(m) Pétrir jusqu'à obtenir une pâte homogène.
(n) Couvrir à l'aide d'un linge humide et laisser fermenter au moins 1h30.
(o) Disposer dans le fond du four un petit récipient contenant de l'eau.
(p) Préchauffer un four à 200 degrés Celsius.
(q) Fariner un plan de travail.
(r) Verser la pâte à pain sur le plan de travail.
(s) Pétrir rapidement la pâte à pain.
(t) Disposer la pâte dans un moule à cake.
(u) Mettre au four pour 15 à 20 minutes, arrêter le four et sortir le pain.

La figure 1 représente les différentes tâches et les dépendances entre ces tâches sous la forme d'un graphe. Chaque sommet du graphe représente une tâche à réaliser. Les dépendances entre les tâches sont représentées par les arcs entre les sommets.

Par exemple, il y a une flèche sur l'arc qui part du sommet d'étiquette (l) et qui atteint le sommet d'étiquette (m) car il faut avoir réalisé la tâche "*Verser l'eau mélangée à la levure dans le puits.*" (l) avant de pouvoir réaliser la tâche "*Pétrir jusqu'à obtenir une pâte homogène.*" (m).

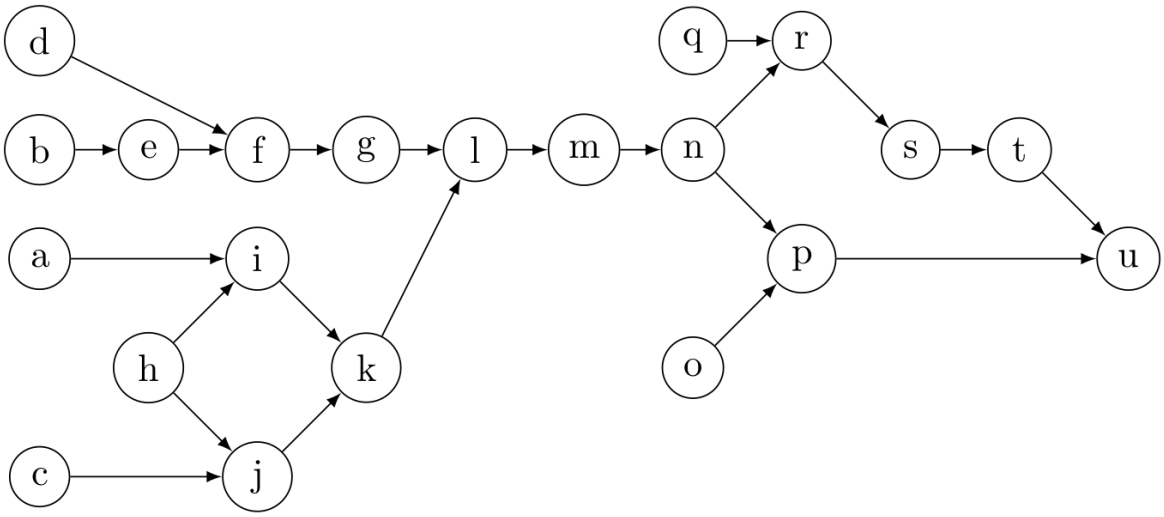

Figure 1. Recette du pain : tâches à effectuer avec leurs dépendances

1. Dire, sans justifier, s'il s'agit d'un graphe orienté ou non orienté.
2. D'après le graphe, dire s'il est possible d'effectuer les réalisations dans chacun des ordres suivants :
   - réaliser la tâche (f) puis la tâche (g)
   - réaliser la tâche (g) puis la tâche (f)
   - réaliser la tâche (i) puis la tâche (j)
   - réaliser la tâche (j) puis la tâche (i)
3. Donner toutes les tâches qu'il faut nécessairement avoir réalisées depuis le début pour pouvoir réaliser la tâche (k). Ne donner que les tâches nécessaires.
4. Indiquer, sans justifier, si le graphe de la Figure 1 contient un cycle.

Graphe de tâches

On s'intéresse désormais de manière plus générale à un graphe de tâches avec des dépendances.

Les sommets sont nommés par des indices. Comme précédemment, un arc orienté d'un sommet d'indice `i` à un sommet d'indice `j` signifie que la tâche représentée par le sommet d'indice `i` doit être réalisée avant la tâche représentée par le sommet d'indice `j`.

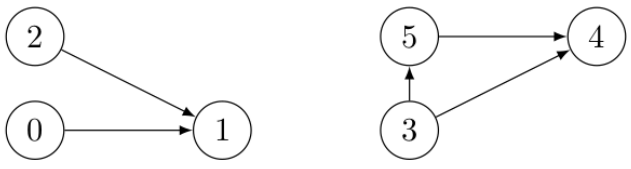

Figure 2. Exemple de graphe de dépendances entre 6 tâches

5. Déterminer un ordre permettant de réaliser toutes les tâches représentées dans le graphe de la Figure 2 en respectant les dépendances entre les tâches.

Voici une matrice d'adjacence d'un graphe écrite en langage Python et telle que si `M[i][j] = 1` alors il existe un arc qui va du sommet d'indice `i` au sommet d'indice `j`. Par exemple, `M[0][1] = 1` alors il existe un arc qui va du sommet d'indice `0` au sommet d'indice `1`.

```
M = [ [0, 1, 0, 0, 0],
      [0, 0, 1, 0, 0],
      [0, 0, 0, 1, 0],
      [0, 1, 0, 0, 1],
      [0, 0, 0, 0, 0] ]
```

6. Représenter le graphe associé à cette matrice d'adjacence. Les noms des sommets seront leurs indices.

7. Déterminer s'il est possible de trouver un ordre permettant de réaliser les tâches représentées par le graphe de la question 6 en respectant leurs dépendances. Si oui, donner l'ordre. Si non, expliquer pourquoi.

Voici le code Python d'une fonction `mystere`.

```
1  def mystere(graphe, s, n, ouverts, fermes, resultat):
2    """ Paramètres :
3        graphe    un graphe représenté par une matrice d'adjacence
4        s         l'indice d'un sommet du graphe
5        n         le nombre de sommets du graphe
6        ouverts   une liste de booléens permettant de savoir
7                  si le traitement d'un sommet a été commencé
8        fermes    une liste de booléens permettant de savoir
9                  si le traitement d'un sommet a été terminé
10       Retour :  False s'il y a eu un "problème", True sinon.
11       Le paramètre resultat sera modifié ultérieurement.
12   """
13   if ouverts[s]:
14       return False
15   if not fermes[s]:
16       ouverts[s] = True
17       for i in range(n):
18           if graphe[s][i] == 1:
19             val = mystere(graphe, i, n, ouverts, fermes,
                             resultat)
20             if not val:
21                 return False
22       ouverts[s] = False
23       fermes[s] = True
24       # ...
25   return True
```

8. En utilisant la matrice `M` donnée précédemment, déterminer si la variable `ok` vaut `True` ou `False` à l'issue des instructions suivantes :

```
1 n = len(M)
2 ouverts = [ False for i in range(n) ]
3 fermes = [ False for i in range(n) ]
4 ok = mystere(M, 1, n, ouverts, fermes, None)
```

Décrire précisément les appels effectués à la fonction `mystere` et les valeurs des tableaux `ouverts` et `fermes` lors de chaque appel. On pourra recopier et compléter le tableau ci-dessous.

Appel `mystere`	variable `ouverts`	variable `fermes`
Avant l'appel `mystere`	`[F,F,F,F,F]`	`[F,F,F,F,F]`
`mystere(M,1,5,[F,F,F,F,F], [F,F,F,F,F],None)`	`[F,T,F,F,F]`	`[F,F,F,F,F]`
`mystere(M,2,5,[F,T,F,F,F], [F,F,F,F,F],None)`	`[F,T,T,F,F]`	`[F,F,F,F,F]`
`...`		

9. De manière générale, expliquer dans quel cas cette fonction `mystere` renvoie `False`.

L'objectif est d'utiliser la fonction `mystere` pour écrire une fonction `ordre_realisation` qui, lorsque c'est possible, détermine l'ordre de réalisation des tâches d'un graphe donné par sa matrice d'adjacence en respectant les dépendances entre les tâches.

Une structure de données de pile est représentée par une classe `Pile` qui possède les méthodes suivantes :

- la méthode `estVide` qui renvoie `True` si la pile représentée par l'objet est vide, `False` sinon ;
- la méthode `empiler` qui prend en paramètre un élément et l'ajoute au sommet de la pile ;
- la méthode `depiler` qui renvoie la valeur du sommet de la pile et enlève cet élément.

10. Déterminer la valeur associée à la variable `elt` après l'exécution des instructions suivantes :

```
>>> essai = Pile()
>>> essai.empiler(3)
>>> essai.empiler(2)
>>> essai.empiler(10)
>>> elt = essai.depiler()
>>> elt = essai.depiler()
```

Lorsqu'il en existe un, un ordre de réalisation des tâches sera représenté par un objet de classe `Pile` contenant tous les sommets du graphe de manière à ce que les tâches qu'il faut réaliser en premier se retrouvent au sommet de la pile.

La fonction `ordre_realisation` est écrite de la manière suivante :

```
1  def ordre_realisation(graphe):
2      n = len(graphe)
3      ouverts = [ False for i in range(n) ]
4      fermes = [ False for i in range(n) ]
5      ordre = Pile()
6      ok = True
7      s = 0
8      while (ok and s < n):
9          ok = mystere(graphe, s, n, ouverts, fermes, ordre)
10          s = s + 1
11     if ok :
12         return ordre
13     return None
```

11. Sachant que dans la fonction `mystere`, la ligne 24 peut être remplacée par une ou plusieurs instructions, donner ce qu'il faut écrire pour que, lorsque c'est

possible, `ordre_realisation` renvoie effectivement un ordre de réalisation des tâches du graphe.

EXERCICE 3

Cet exercice porte sur la programmation Python, la modularité, les bases de données relationnelles et les requêtes SQL.

Une *flashcard*, autrement appelée *carte de mémorisation*, est une carte papier sur laquelle se trouve au recto une question et au verso la réponse à cette question. On les utilise en lisant la question du recto puis en vérifiant notre réponse à celle du verso. Une étudiante souhaite réaliser des *flashcards* numériquement.

Partie A

L'étudiante souhaite stocker les questions/réponses de ses *flashcards* dans un fichier au format `csv`. Ce format permet de stocker textuellement des données tabulaires. La première ligne du fichier contient les descripteurs : les noms des champs renseignés par la suite. Pour être en mesure de les identifier, chaque champ est séparé par un caractère appelé séparateur. C'est la virgule qui est le plus couramment utilisée, mais cela peut être d'autres caractères de ponctuation.

Le langage Python dispose d'un module natif nommé `csv` qui permet de traiter de tels fichiers. La méthode `DictReader` de ce module prend en argument un fichier `csv` et le séparateur utilisé. Elle permet d'extraire les données contenues dans le fichier. Voici un exemple de fonctionnement.

fichier `exemple.csv`

```
champ1,champ2
a,7
b,8
c,9
```

code Python

```
import csv
with open('exemple.csv','r') as fichier:
    donnees = list(csv.DictReader(fichier,delimiter=','))
print(donnees)
```

affichage généré à l'exécution

```
[{'champ1': 'a', 'champ2': '7'},
 {'champ1': 'b', 'champ2': '8'},
 {'champ1': 'c', 'champ2': '9'}]
```

Voici un extrait du fichier `flashcards.csv` réalisé par l'étudiante :

```
discipline;chapitre;question;réponse
histoire;crise de 1929;jeudi noir - date;24 octobre 1929
histoire;crise de 1929;jeudi noir - quoi;krach boursier
histoire;2GM;l'Axe;Allemagne, Italie, Japon
histoire;2GM;les Alliés;Chine, États-Unis, France, Royaume-Uni, URSS
```

```
histoire;2GM;Pearl Harbor - date;7 décembre 1941
philosophie;travail;Marx;aliénanation de l'ouvrier
philosophie;travail;Beauvoir;donne de la valeur à l'homme
philosophie;travail;Locke;permet de fonder le droit de propriété
philosophie;travail;Crawford;satisfaction et estime de soi
```

1. Donner le séparateur choisi par l'étudiante pour son fichier `flashcards.csv`.
2. Justifier pourquoi l'étudiante a choisi ce séparateur.

Voici le code écrit par l'étudiante pour utiliser ses flashcards.

```
 1  import csv
 2  import time
 3
 4  def charger(nom_fichier):
 5      with ...
 6          donnees = ...
 7      return ...
 8
 9  def choix_discipline(donnees):
10      disciplines = []
11      for i in range(len(donnees)):
12          disc = donnees[i]['discipline']
13          if not disc in disciplines:
14              disciplines.append(disc)
15       for i in range(len(disciplines)):
16           print(i + 1, disciplines[i])
17       num_disc = int(input('numéro de la discipline ? '))
18       return disciplines[num_disc - 1]
19
20  def choix_chapitre(donnees, disc):
21      chapitres = []
22      for i in range(len(donnees)):
23          if flashcard[i]['discipline'] == disc:
24              ch = flashcard[i]['chapitre']
25              if not ch in chapitres:
26                  chapitres.append(ch)
27      for i in range(len(chapitres)):
28          print(i +1, chapitres[i])
29      num_ch = int(input('numéro du chapitre ? '))
30      return chapitres[num_ch - 1]
31
32  def entrainement(donnees, disc, ch):
33      for i in range(len(donnees)):
34          if donnees[i]['discipline'] == disc \
35          and donnees[i]['chapitre'] == ch:
36              print('QUESTION : ', donnees[i]['question'])
37              time.sleep(5)
38              print(donnees[i]['réponse'])
```

```
39              time.sleep(1)
40
41  flashcard = ...
42  d = ...
43  c = ...
44  entrainement(...)
```

3. Recopier et compléter le code de la fonction `charger(nom_fichier)` qui lit le fichier dont le nom est fourni en argument et qui renvoie les données lues sous la forme d'un dictionnaire comme dans l'exemple fourni précédemment.

4. Le module `time` est importé à la ligne 2 de ce programme. Quelle est la méthode du module `time` utilisée dans ce code ?

5. Donner le type de la variable `donnees[i]` (par exemple ligne 12).

6. Recopier et compléter les lignes 41 à 44.

Partie B

Pour améliorer sa mémorisation sur le long terme, l'étudiante décide de mettre en œuvre le concept des boites de Leitner. Dans cette méthode, il s'agit d'espacer dans le temps la révision des *flashcards* si l'étudiante répond correctement. Elle imagine donc une base de données qui lui permettra de conserver pour chaque question la date à laquelle elle doit de nouveau être posée. Elle décide que les questions seront réparties en 5 boites. Initialement, tous les questions seront placées dans la boite 1. Les questions de la boite 1 sont posées tous les jours, celles de la boite 2 tous les deux jours, celles de la boite 3 tous les quatre jours, celles de la boite 4 tous les huit jours et celles de la boite 5 tous les quinze jours. Si l'étudiante donne la bonne réponse à une question et que la question n'appartient pas à la boite 5, son numéro de boite est incrémenté (augmenté de 1). Si l'étudiante ne donne pas la bonne réponse, la question revient dans la boite 1.

Elle met en œuvre une base de données relationnelle contenant 4 tables `discipline`, `chapitre`, `boite` et `question`.

La table `discipline` contient la liste des disciplines étudiées. Elle a deux attributs :

- `id`, de type `INT`, l'identifiant de la discipline qui est une clé primaire pour cette table ;

- `lib`, de type `TEXT`, le libellé de la discipline.

La table `chapitre` contient la liste des chapitres des disciplines étudiées. Elle a trois attributs :

- `id`, de type `INT`, l'identifiant du chapitre qui est une clé primaire pour cette table ;

- `lib`, de type `TEXT`, le libellé du chapitre ;
- `id_disc`, de type `INT`, l'identifiant de la discipline à laquelle appartient ce chapitre.

La table `boite` contient l'ensemble des cinq boites existantes. Elle a trois attributs :

- `id`, de type `INT`, l'identifiant numéro de la boite qui est une clé primaire pour cette table ;
- `lib`, de type `TEXT`, le libellé de la boite ;
- `frequence`, de type `INT`, indiquant le nombre de jours séparant deux interrogations d'une question appartenant à cette boite.

La table `flashcard` contient les questions-réponses. Elle a six attributs :

- `id`, de type `INT`, l'identifiant de la *flashcard* qui est une clé primaire pour cette table ;
- `id_ch`, de type `INT`, l'identifiant du chapitre auquel appartient la *flashcard* ;
- `id_boite`, de type `INT`, l'identifiant numéro de la boite de la *flashcard* ;
- `question`, de type `TEXT`, le texte au recto de la *flashcard* ;
- `reponse`, de type `TEXT`, le texte au verso de la *flashcard* ;
- `date_interro`, de type `DATE`, la date de la prochaine interrogation pour cette question.

Initialement `date_interro` sera la date d'insertion de la question dans la base de données.

Table boite		
Id	lib	frequence
1	tous les jours	1
2	tous les deux jours	2
3	tous les quatre jours	4
4	tous les huit jours	8

7. Écrire une requête SQL qui complète la table `boite` et insère la boite 5 de libellé 'tous les quinze jours' et de fréquence 15.

Une requête sur la table `flashcard` affiche l'enregistrement suivant :

```
5, 2, 1, Pearl Harbor – date, 6 décembre 1941
```

8. Écrire une requête SQL pour mettre à jour la date de Pearl Harbor renvoyée. La bonne date est le 7 décembre 1941.
9. Écrire une requête SQL qui permet d'obtenir la liste des libellés des disciplines.
10. Écrire une requête SQL qui permet d'obtenir la liste des libellés des chapitres de la discipline 'histoire'.
11. Écrire une requête SQL qui permet d'obtenir la liste des identifiants des flashcards de la discipline 'histoire'.
12. Écrire une requête SQL pour supprimer toutes les flashcards de la boite d'identifiant 3.

EXERCICE 4

Cet exercice porte sur les réseaux, les protocoles de routage et les graphes.

Partie A

Le réseau informatique d'une société est constitué d'un ensemble de routeurs interconnectés à l'aide de fibres optiques.

La figure ci-dessous représente le schéma de ce réseau. Il est composé de deux réseaux locaux L1 et L2. Le réseau local L1 est relié au routeur R1 et le réseau local L2 est relié au routeur R9.

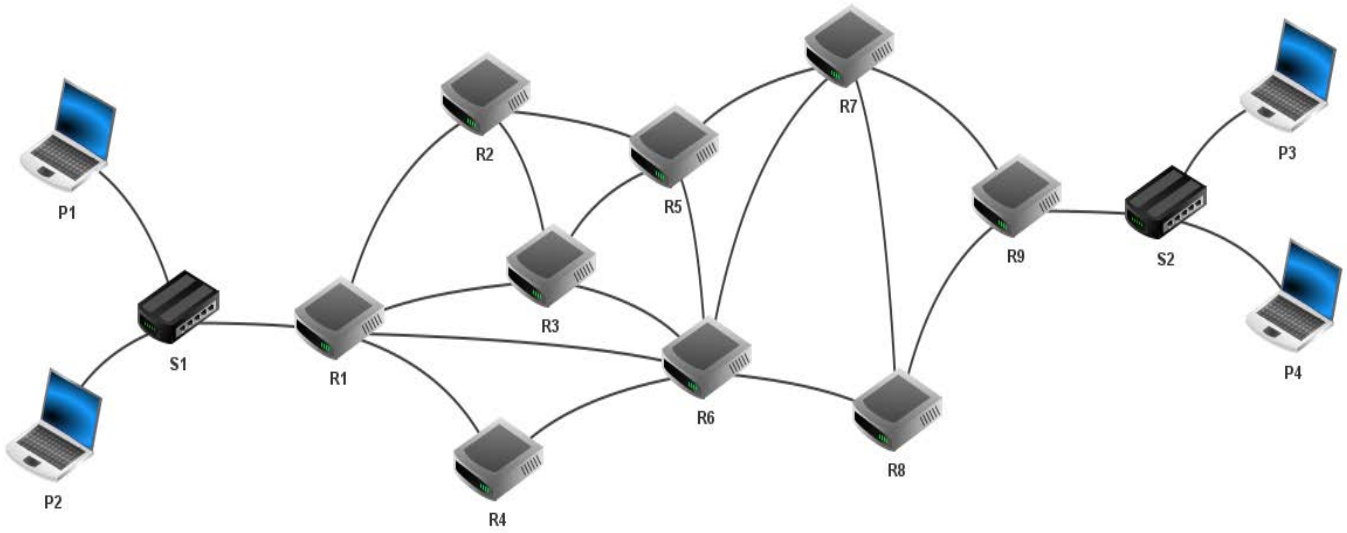

Figure 1. Réseau

Dans cette partie, les adresses IP sont composées de 4 octets, soit 32 bits. Elles sont notées X1.X2.X3.X4, où X1, X2, X3 et X4 sont les valeurs des 4 octets, converties en notation décimale. La notation X1.X2.X3.X4/n signifie que les n premiers bits de poids forts de l'adresse IP représentent la partie « réseau », les bits suivants représentent la partie « hôte ».

Toutes les adresses des machines connectées à un réseau local ont la même partie réseau.

Le tableau suivant indique les adresses IPv4 des machines constituant le réseau de la société.

NOM	TYPE	ADRESSE IPV4
R1	Routeur	Interface 1 :192.168.1.1/24 Interface 2 :192.168.2.1/24 Interface 3 :192.168.3.1/24 Interface 4 :192.168.4.1/24 Interface 5 :192.168.5.1/24

NOM	TYPE	ADRESSE IPV4
R2	Routeur	Interface 1 :192.168.2.2/24 Interface 2 :192.168.7.1/24 Interface 3 :192.168.8.1/24
R3	Routeur	Interface 1 :192.168.3.2/24 Interface 2 :192.168.7.2/24 Interface 3 :192.168.9.1/24 Interface 4 :192.168.10.1/24
R4	Routeur	Interface 1 :192.168.5.2/24 Interface 2 :192.168.6.1/24
R5	Routeur	Interface 1 :192.168.8.2/24 Interface 2 :192.168.9.2/24 Interface 3 :192.168.11.1/24 Interface 4 :192.168.12.1/24
R6	Routeur	Interface 1 :192.168.4.2/24 Interface 2 :192.168.6.2/24 Interface 3 :192.168.10.2/24 Interface 4 :192.168.11.2/24 Interface 5 :192.168.13.1/24 Interface 6 :192.168.14.1/24
R7	Routeur	Interface 1 :192.168.12.2/24 Interface 2 :192.168.13.2/24 Interface 3 :192.168.15.1/24 Interface 4 :192.168.16.1/24
R8	Routeur	Interface 1 :192.168.14.2/24 Interface 2 :192.168.15.2/24 Interface 3 :192.168.17.1/24
R9	Routeur	Interface 1 :192.168.16.2/24 Interface 2 :192.168.17.2/24 Interface 3 :192.168.18.1/24
P1	Portable	192.168.1.10
P2	Portable	Non fourni
P3	Portable	Non fourni
P4	Portable	Non fourni

1. En utilisant les adresses IP des différentes interfaces et des ordinateurs portables, en déduire une adresse possible pour le portable P2.

2. Donner l'adresse du réseau local L2 ainsi que le nombre d'adresses possibles pour les ordinateurs portables P3 et P4.

Partie B

Le graphe G, représenté ci-dessous, schématise l'architecture du réseau de la société. Les sommets représentent les routeurs et les arêtes représentent les liaisons.

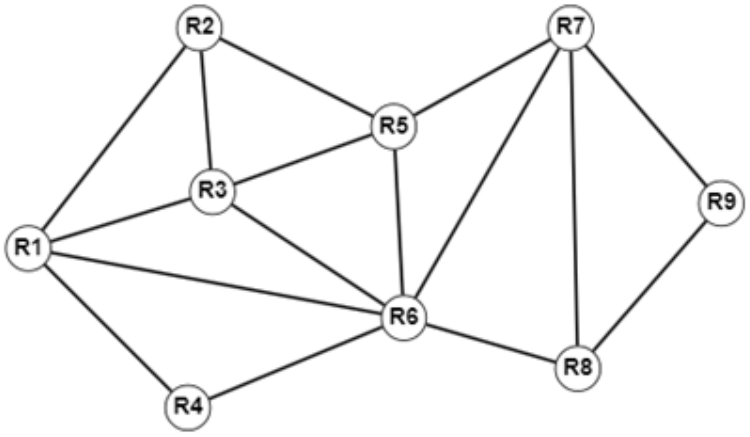

Figure 2. Graphe non pondéré

3. Donner l'implémentation Python des listes d'adjacence de ce graphe à l'aide d'un dictionnaire dont les clés sont les sommets et les valeurs la liste des sommets adjacents du sommet clé. On nomme G ce dictionnaire.

   Afin de faciliter la notation, on s'autorise à écrire chaque couple clé/valeur sur une nouvelle ligne.

On suppose que le protocole de routage RIP est utilisé.

4. Recopier et compléter, en rajoutant autant de lignes que nécessaire, la table de routage simplifiée suivante du routeur R1.

Destination	Suivant	Nombre de sauts
R2	R2	1
R3		

L'ordinateur P1 envoie un paquet de données à l'ordinateur P3.

5. Donner l'un des chemins empruntés par le paquet ainsi que le nombre de sauts.

La société doit vérifier l'état physique de la fibre optique installée sur le réseau. Un robot inspecte toute la longueur de la fibre optique afin de s'assurer qu'elle ne présente pas de détérioration apparente.

On appelle M la matrice d'adjacence du graphe de la figure 2. Les sommets sont rangés par ordre croissant des numéros des routeurs (R1, R2, …, R9).

6. Donner l'écriture en Python de cette matrice d'adjacence sous la forme d'une liste de listes.

Le degré d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité.

7. Recopier et compléter les lignes 3, 5 et 6 de le fonction `degre` qui prend en paramètre la matrice d'adjacence d'un graphe donné sous forme d'une liste de listes et qui renvoie la liste des degrés de tous les sommets du graphe rangés dans le même ordre que les sommets de la matrice d'adjacence.

```
1 def degre(MATRICE):
2     d = []
3     for ... in ...:
4         cpt = 0
5         for ... in ...:
6             cpt = cpt +...
7         d.append(cpt)
8     return d
```

8. Donner la liste renvoyée par `degre(M)`.

On appelle chaîne eulérienne d'un graphe non orienté un chemin qui passe une et une seule fois par toutes les arêtes du graphe. Un graphe connexe admet une chaîne eulérienne si et seulement si le graphe possède, au plus, deux sommets de degré impair.

9. En utilisant le résultat de la question précédente et en admettant que le graphe est connexe, indiquer si le robot peut parcourir l'ensemble du réseau en suivant les fibres optiques et en empruntant chaque fibre optique une et une seule fois.

Partie C

Le poids sur chaque arête représente la bande passante en megabits par seconde (Mb/s) de chaque liaison.

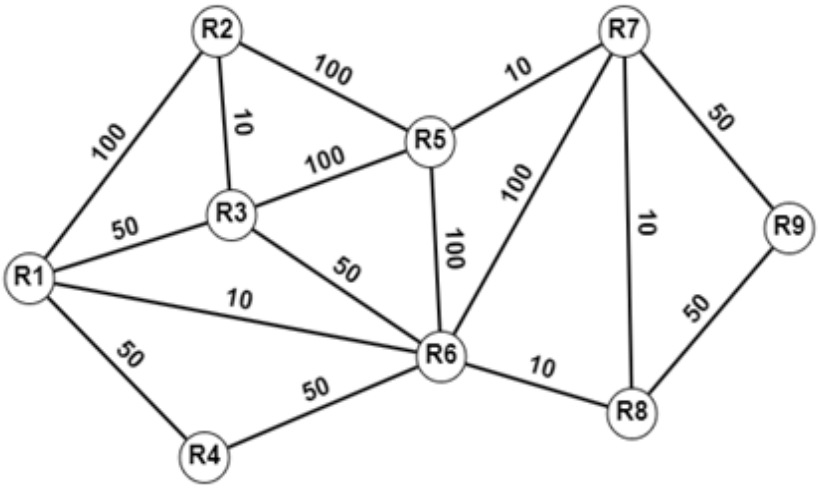

Figure 3. Graphe pondéré

Dans cette partie, on utilise le protocole de routage OSPF. Pour calculer le coût d'une liaison, on utilise la formule :

$$C = \frac{10^8}{BP}$$

où BP est la bande passante en bits par seconde.

10. Déterminer la route qui sera empruntée par le paquet pour aller de l'ordinateur P1 à l'ordinateur P3. Préciser le coût de ce trajet.