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 :

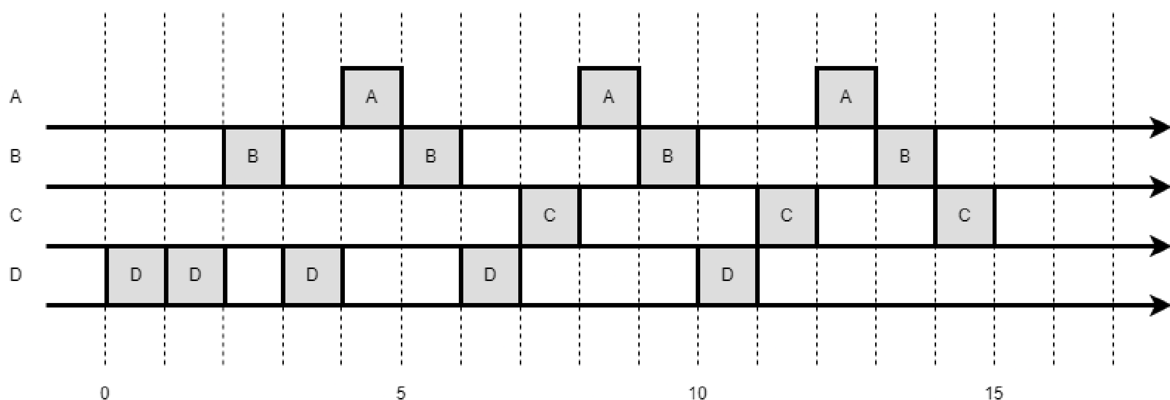

Figure 1. Chronogramme pour les processus A, B, C et D

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`.

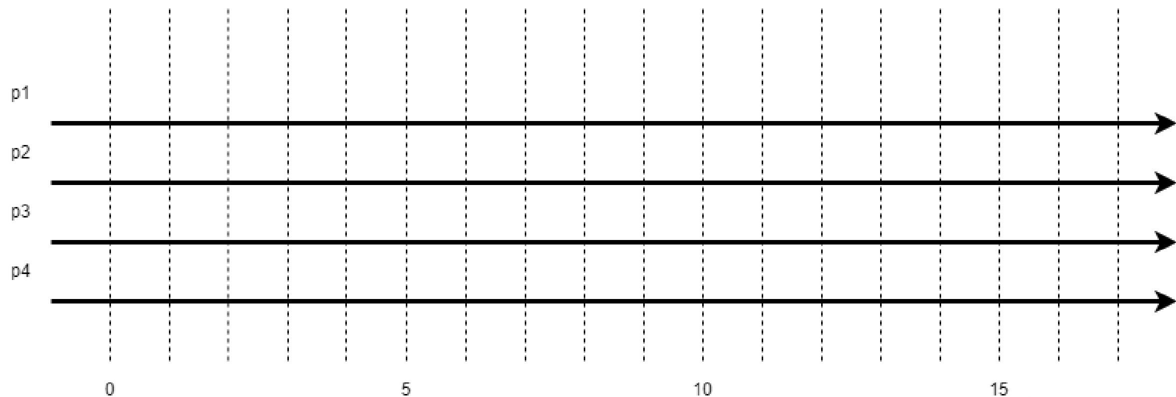

Figure 2. Chronogramme pour les processus p1, p2, p3 et p4

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 structures de données, la programmation, les graphes.

Partie A

Le *siteswap* est une notation mathématique pour codifier les figures de jonglerie. Elle est aujourd'hui utilisée par des jongleurs et jongleuses dans le monde entier. Beaucoup de figures sont alors simplement désignées par leur siteswap, comme par exemple 441, 7531 ou encore 453.

On modélise le jonglage de la manière suivante : au lieu de calculer des trajectoires complexes, on considère simplement un rythme régulier sur lequel on jongle, et une balle est lancée à chacun de ses « temps ».

Les lancers sont caractérisés par un nombre entier positif, représentant simplement le nombre de « temps » au bout duquel la balle revient dans la main du jongleur et peut être relancée.

À un instant donné, on peut représenter ce qu'on appelle un *état*, c'est-à-dire une sorte de photographie des balles « en l'air ». On notera ces états sous forme de tableaux Python, contenant des 0 et des 1. Un 0 représente un espace vide et un 1 représente une balle.

Si on considère l'état `e1 = [1, 0, 0, 1, 1, 0]` : son premier élément, `e1[0]` vaut `1`, et représente donc la balle prête à être relancée. Si `e1[0]` valait `0`, aucune balle à relancer ne serait présente. Ensuite chaque `e1[i]` représente la présence ou non d'une balle qui atterrira dans la main de la jongleuse au bout de `i` temps.

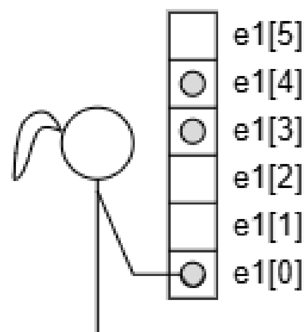

Figure 1. Représentation de l'état `e1`

L'état `e1` ci-dessus représente donc un instant d'une figure à 3 balles, l'une est dans la main de la jongleuse, et deux autres balles sont plus haut, et retomberont dans la main dans respectivement 3 et 4 temps puisque `e1[3]` et `e1[4]` sont égaux à 1 et les autres à 0.

Comme l'indice maximal est de 5 dans le tableau, on dira que la hauteur maximale est 5.

Lorsque la jongleuse attrape la balle, elle va la relancer, dans un emplacement en l'air qui est « libre », car elle ne souhaite pas recevoir à un moment donné deux balles en même temps.

Dans l'exemple `e1 = [1, 0, 0, 1, 1, 0]`, la jongleuse peut effectuer un lancer de 1, un lancer de 2 ou un lancer de 5, car les emplacements `e1[1]`, `e1[2]` et `e1[5]` sont à 0, donc « libres ». Elle ne peut pas lancer un 3 ou un 4.

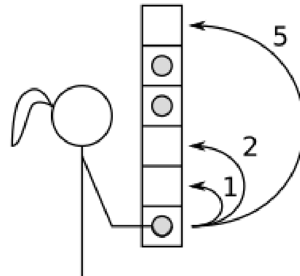

Figure 2. Transitions possibles depuis l'état `e1`

Si le premier élément de l'état est à 0, cela signifie que la jongleuse n'a aucune balle dans sa main à cet instant. Elle ne peut donc pas lancer de balle, et on appellera ça, par convention, un lancer « 0 ». Un lancer « 0 » n'est possible que dans cette situation.

1. Si on se donne l'état `e2 = [1, 1, 0, 1, 0, 0]` indiquer quels sont les lancers possibles.

2. Même question pour l'état `e3 = [0, 1, 1, 0, 1]`.

3. Recopier et compléter les lignes 4, 7 et 8 du code de la fonction `lancer_possible` ci-dessous. Elle prend en argument un tableau `etat` représentant un état et un entier `lancer`, et renvoie `True` si le lancer est possible, et `False` sinon.

```
1 def lancer_possible(etat, lancer):
2     if lancer >= len(etat) or lancer < 0:
3         return False
4     if lancer == 0 and ...
5         return False
6     if lancer > 0:
7         if etat[0] == 0 or ...
8             ...
9     return True
```

Lorsqu'on lance une balle, elle vient se placer là où on l'a prévu, puis la gravité fait son effet et toutes les balles redescendent d'un cran.

Ainsi, si depuis l'état `e1 = [1, 0, 0, 1, 1, 0]` on lance un 2, on obtient l'état `[0, 0, 1, 1, 1, 0]` puis l'état `[0, 1, 1, 1, 0, 0]` après effet de la gravité :

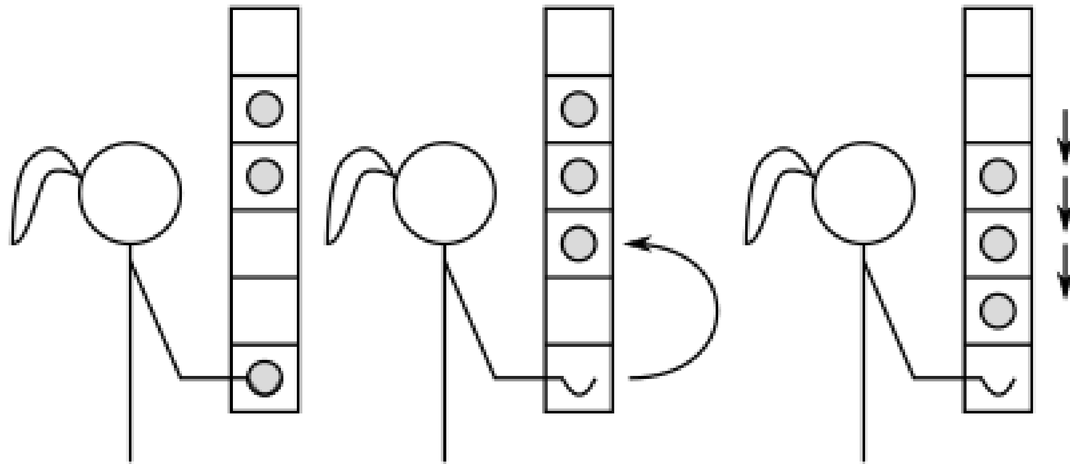

Figure 3. État `e1`, puis lancer de 2, puis effet de la gravité

4. Depuis l'état `e2 = [1, 1, 0, 1, 0, 0]`, on effectue un lancer de 5. Donner l'état qu'on obtient après le lancer et l'effet de la gravité.

On souhaite écrire une fonction `lancer_balle` qui prend en paramètres un état `etat` de jonglage (comme décrit ci-dessus) et un entier positif `lancer` qui représente un lancer. Elle ne doit pas modifier l'état passé en paramètre, mais doit renvoyer un nouvel état correspondant au résultat du lancer. On suppose sans le vérifier que le lancer est forcément valide.

5. Recopier et compléter la ligne 4 du code de la fonction `lancer_balle` ci-dessous. On peut insérer plusieurs lignes si besoin.

```
1 def lancer_balle(etat, lancer):
2     # copie de l'état pour ne pas le modifier
3     nouvel_etat = [balle for balle in etat]
4     ...
5     return nouvel_etat
```

Partie B

6. Écrire une fonction `liste_lancers_possibles` qui prend en paramètre un état `etat` et qui renvoie une liste d'entiers correspondant à l'ensemble des lancers possibles à partir de cet état.

   Par exemple

```
1 >>> liste_lancers_possibles(e1)
2 [1, 2, 5]
3 >>> liste_lancers_possibles([0, 1, 1, 1, 0])
4 [0]
```

On souhaite maintenant générer toutes les suites de lancers possibles à partir d'un état donné, c'est-à-dire tous les lancers consécutifs qu'on peut faire à partir de cet état.

Par exemple, à partir de l'état `e1 = [1, 0, 0, 1, 1, 0]` on peut lancer un 1, un 2 ou un 5. Si on a lancé un 1 on obtient l'état `[1, 0, 1, 1, 0, 0]` (on rappelle que cet état est obtenu après le lancer et l'effet de gravité) et on peut lancer un 1, un 4 ou un 5. Et de même pour les états obtenus à partir de lancers 2 ou 5.

On peut alors calculer qu'à partir de `e1` on peut faire les séries de lancers de longueur 2 suivants (notés sous forme de listes Python) : `[1, 1]`, `[1, 4]`, `[1, 5]`, `[2, 0]`, ou `[5, 0]`.

On aimerait obtenir tous les lancers possibles d'une longueur donnée à partir d'un état.

Pour cela on propose la méthode suivante :

- si la longueur demandée est 0, alors la seule séquence possible est la séquence vide ;
- sinon, on calcule quels sont les lancers possibles à partir de cet état. Pour chacun de ces lancers, on va :
  - calculer le nouvel état obtenu ;
  - chercher l'ensemble des séquences possibles à partir de ce nouvel état (d'une longueur un de moins) ;
  - pour toutes ces séquences, on ajoutera le numéro du lancer au début et on la mettra dans une liste `s_possibles` à renvoyer au final.

Voici la fonction `calcule_sequences` partiellement écrite :

```
 1 def calcule_sequences(etat, n):
 2     """ etat est un état de jonglerie, n est un entier.
 3     Calcule et renvoie l'ensemble des siteswaps (listes
 4     d'entiers) de longueur n qu'on peut effectuer à
 5     partir de cet état."""
 6     if n == 0:
 7         return [[]]
 8     else:
 9         s_possibles = []
10         l_lancers = ...
11         for lancer in l_lancers:
12             etat2 = ...
13             s_etat2 = calcule_sequences(etat2, n-1)
14             for ...
15                 s_possibles.append([lancer] + ...)
16         return s_possibles
```

7. Justifier qu'il s'agit d'une fonction récursive.

8. Expliquer brièvement pourquoi elle se termine si `n` est un entier positif. On admet que les boucles `for` présentes sont bornées et donc terminent.

9. Recopier et compléter les lignes 10, 12, 14 et 15 de cette fonction.

Partie C

Plutôt que de calculer l'ensemble des séquences possibles à partir d'un état donné, on préfère calculer d'un coup, dès le début, l'ensemble des états et des lancers possibles.

On représentera ces données par un graphe orienté, dont les sommets sont les états, et on a un arc d'un état `e` à un état `f` si le lancer `n` permet de passer de l'état `e` à l'état `f`. Dans ce cas on inscrit le `n` à proximité l'arc entre `e` et `f` et on dit que c'est l'étiquette de l'arc.

On travaille donc avec un graphe orienté étiqueté.

Ce graphe est également appelé *automate des états*.

Voici par exemple l'automate des états des jonglages à deux balles, de hauteur maximale 4.

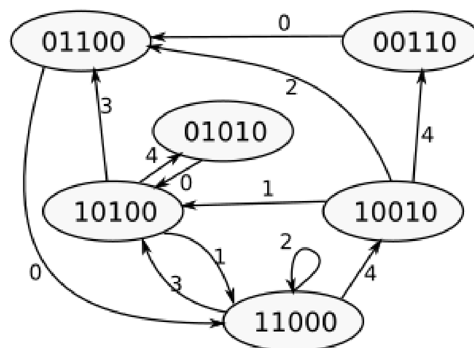

Figure 4. Ensemble des états et lancers, à deux balles et hauteur maximale 4

On a choisi de représenter les états par des chaînes de caractères : `'11000'` représente l'état `[1, 1, 0, 0, 0]` dans les parties précédentes.

On souhaite stocker ce graphe sous forme de dictionnaire de listes d'adjacences : les clés sont les états, et les valeurs sont des listes de tuple : le premier élément est un entier, le numéro du lancer possible, et le second est l'état qu'on obtient lorsqu'on applique ce lancer.

10. Recopier et compléter le code Python permettant de représenter l'automate de la figure 4 dans une variable `automate` :

```
1 automate = { '11000': [(3, '10100'), (2, '11000'), (4,
'10010')],
2              '01010': [(0, '10100')],
3              '10100': ...,
4              ...: [(0, '11000')],
5              ... : ...,
6              ... : ...}
```

11. Écrire le code de la fonction `lancer_balle_automate` qui prend en arguments un automate `automate` comme décrit plus haut, un état `etat` et un entier `lancer` représentant un lancer et qui renvoie l'état obtenu lorsqu'on lance `lancer` depuis l'état `etat`. On renvoie la chaîne vide si le lancer n'est pas possible.

    Par exemple, pour l'automate de la Figure 4,

```
1 >>> lancer_balle_automate(automate, '10010', 2)
2 '01100'
3 >>> lancer_balle_automate(automate, '11000', 1)
4 ''
```

Un *siteswap* est une suite de lancers qui correspond à un cycle dans l'automate : autrement dit cela correspond à des lancers qu'on peut répéter en boucle : c'est une « figure » de jonglage.

Par exemple dans le graphe de la Figure 4, la séquence `3, 1` est un siteswap : on part de l'état `'11000'` puis le lancer de 3 nous amène dans l'état `'10100'`, le lancer de 1 nous ramène dans l'état `'11000'` et on peut recommencer cette figure.

La séquence `1, 2, 3, 4, 0` est également un siteswap (partant de l'état `'10100'`, les lancers successifs sont possibles et on revient bien à l'état de départ).

La séquence `2` est également un siteswap (reste dans l'état `'11000'`).

On souhaite écrire une fonction `parcours_sequence_depart` qui prend en argument un automate, un état de départ, et une liste de lancers, et qui renvoie l'état dans lequel on arrive en suivant la séquence de lancers, ou bien `None` si l'un des lancers était impossible.

Par exemple :

```
1 >>> parcours_sequence_depart(automate, '11000',  [3, 1])
2 '11000'
3 >>> parcours_sequence_depart(automate, '10010',  [4, 0])
4 '01100'
5 >>> parcours_sequence_depart(automate, '10100',  [3, 4])
6 None
```

12. Écrire le code de la fonction `parcours_sequence_depart`. On peut utiliser la fonction `lancer_balle_automate`.

Grâce à la fonction précédente, il est possible de vérifier qu'un siteswap est valide, c'est-à-dire qu'il existe un état à partir duquel réaliser la figure de jonglage.

On souhaite à présent écrire une fonction `departs_siteswap` qui prend en argument un automate et une liste de lancers (un potentiel siteswap), et renvoie la liste des états de l'automate qui valide le siteswap.

Par exemple :

```
1 >>> departs_siteswap(automate, [1, 2, 3, 4, 0])
2 ['10100']
3 >>> departs_siteswap(automate, [2, 1, 0])
4 []
```

13. Écrire la fonction `departs_siteswap`. On peut utiliser la fonction `parcours_sequence_depart`, et vérifier si le siteswap est possible à partir de chaque état de l'automate.

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.

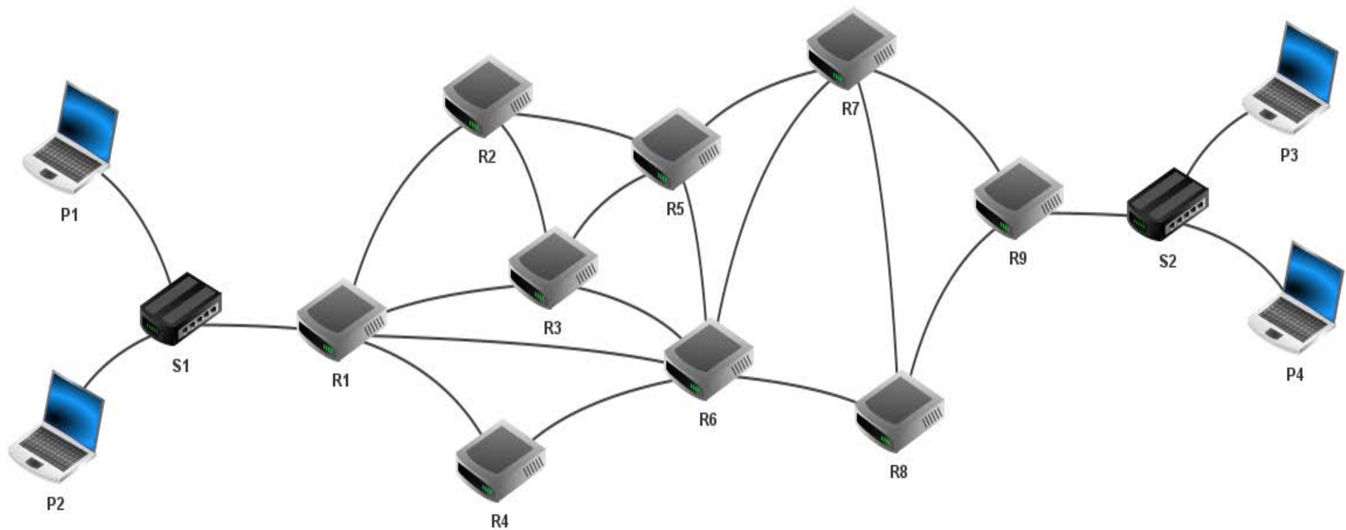

Figure 1. Réseau

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.

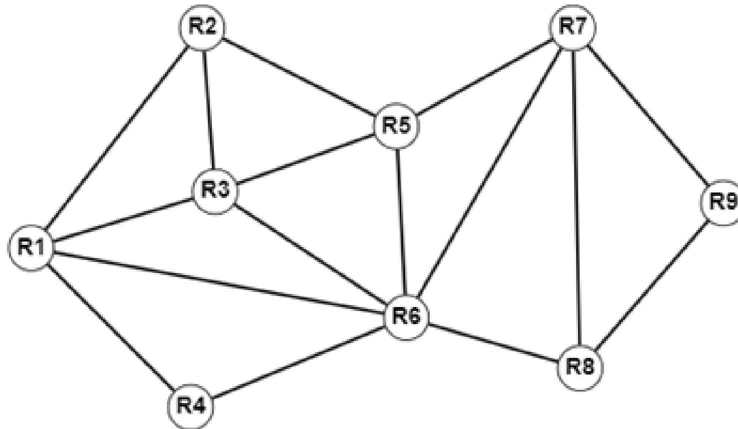

Figure 2. Graphe non pondéré

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.

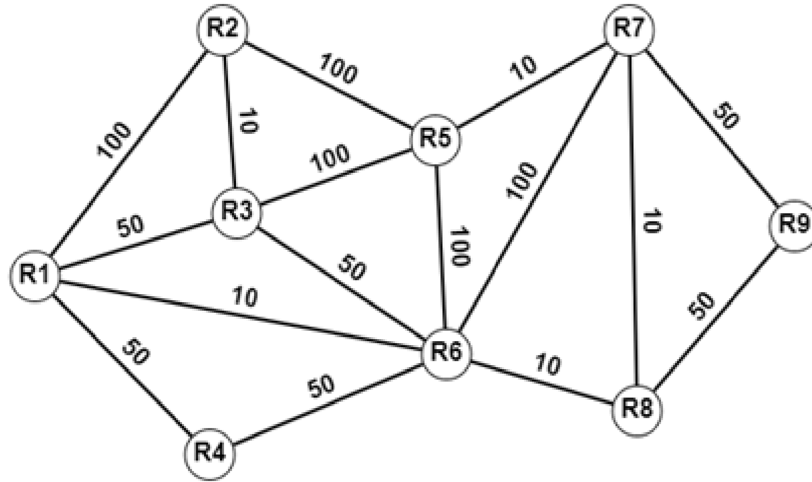

Figure 3. Graphe pondéré

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.