EXERCICE 1

Cet exercice porte sur les graphes et les protocoles réseau.

Les deux parties sont indépendantes.

Partie A : Réseau dans un lycée

Dans un lycée, le réseau contient plusieurs sous-réseaux : pédagogie (noté P), administration (noté AD), vie_scolaire (noté VS).

1. L'adresse IP du réseau pédagogie est 110.217.50.0 et on utilise le masque de sous-réseau 255.255.255.0 (i.e. les trois premiers octets sont réservés au réseau).

   Déterminer le nombre de machines que l'on peut brancher au maximum sur le réseau pédagogie (remarque : l'adresse IP 110.217.50.255 est réservée : c'est l'adresse de diffusion).

2. Déterminer l'écriture binaire du nombre 217.

3. Déterminer l'écriture décimale du nombre binaire 110010.

On scinde finalement le réseau pédagogie en deux sous-réseaux pédagogie 1 (noté P1) et pédagogie 2 (noté P2) et on utilise le masque de sous-réseau 255.255.255.0 pour les deux.

Les adresses IP de ces deux sous-réseaux sont :

- 110.217.50.0 pour le réseau pédagogie 1
- 110.217.52.0 pour le réseau pédagogie 2

4. Indiquer, en justifiant, si une machine ayant l'adresse IP 110.217.53.22 fait partie du réseau pédagogie 2 ou non.

L'adresse IP du réseau administrateur est 110.217.54.0. Celle du réseau vie_scolaire est 110.217.56.0.

Les sous-réseaux sont connectés entre eux par trois routeurs R1, R2 et R3 de la façon suivante :

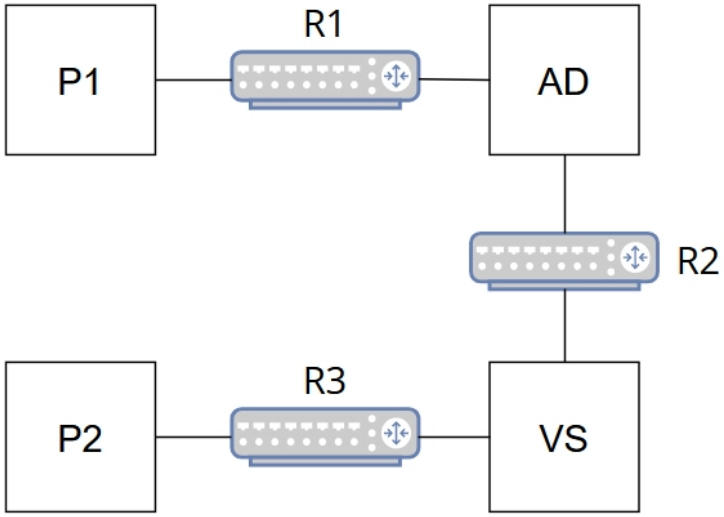

Figure 1. Plan du réseau.

Les adresses IP des routeurs dans chacun des réseaux sont données dans le tableau suivant :

Routeur	adresse dans P1	adresse dans P2	adresse dans AD	adresse dans VS
R1	110.217.50.254		110.217.54.254	
R2			110.217.54.253	110.217.56.254
R3		110.217.52.254		110.217.56.253

5. Recopier et compléter la table de routage du routeur R1 suivante en indiquant les adresses IP des passerelles et des interfaces :

Destination	Passerelle	Interface
110.217.50.0	on-link	110.217.50.254
110.217.52.0	110.217.54.253	
110.217.54.0		
110.217.56.0		

Précision : Si un réseau est directement relié à un routeur, l'adresse IP de la passerelle est remplacée par les mots "on-link".

On ajoute une liaison entre les réseaux pédagogie 1 et pédagogie 2 via un routeur R4 dont les adresses IP sont :

- 110.217.50.253 dans pédagogie 1
- 110.217.52.253 dans pédagogie 2

Le réseau devient alors :

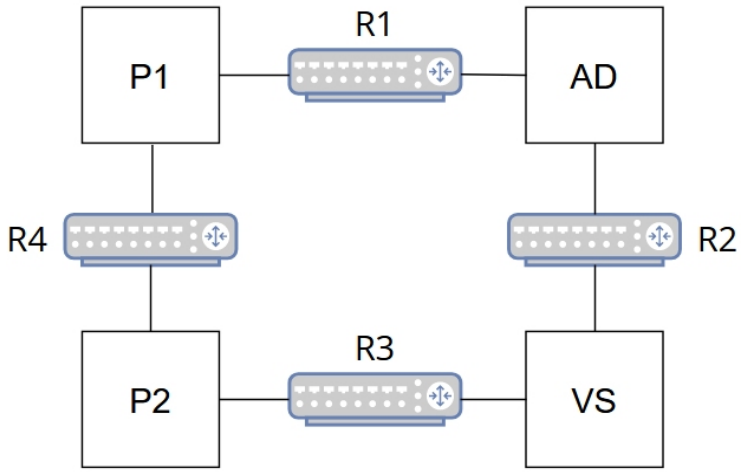

Figure 2. Plan du réseau mis à jour.

On rappelle que le protocole de routage RIP minimise le nombre de routeurs traversés.

6. Indiquer les modifications à apporter à la table de routage de R1, en respectant le protocole RIP sur l'ensemble du réseau du lycée.

7. Indiquer si l'ajout du routeur R4 entraîne des modifications dans la table de routage du routeur R2, toujours en respectant le protocole RIP. Justifier la réponse.

Partie B : Réseaux et graphes

Dans un réseau dont la structure est connue, on veut pouvoir déterminer si deux routeurs peuvent communiquer entre eux, c'est-à-dire s'il existe entre eux une route constituée de routeurs reliés par des sous-réseaux. On représente ce réseau de routeurs par un graphe G dont les routeurs sont les sommets et les sous-réseaux reliant les routeurs sont les arêtes.

On propose la fonction de recherche d'un chemin entre deux sommets R1 et R2 suivante :

```
1 def recherche(R1,R2):
2     if R1 == R2 :
3         return True
4     for S in adjacents(R1,G) :
5          if recherche(S,R2) :
6              return True
7     return False
```

où `adjacents(R1,G)` renvoie la liste des sommets adjacents à R1 dans le graphe G, classés dans l'ordre alphabétique de leur nom.

8. Indiquer ce qui se passera si on utilise cette fonction de recherche entre deux sommets non reliés par un chemin dans le graphe.

9. Proposer une solution pour résoudre ce problème.

EXERCICE 2

Cet exercice porte sur les graphes.

Dans cet exercice, on modélise un groupe de personnes à l'aide d'un graphe.

Le groupe est constitué de huit personnes (Anas, Emma, Gabriel, Jade, Lou, Milo, Nina et Yanis) qui possèdent entre elles les relations suivantes :

- Gabriel est ami avec Jade, Yanis, Nina et Milo ;
- Jade est amie avec Gabriel, Yanis, Emma et Lou ;
- Yanis est ami avec Gabriel, Jade, Emma, Nina, Milo et Anas ;
- Emma est amie avec Jade, Yanis et Nina ;
- Nina est amie avec Gabriel, Yanis et Emma ;
- Milo est ami avec Gabriel, Yanis et Anas ;
- Anas est ami avec Yanis et Milo ;
- Lou est amie avec Jade.

Partie A : Matrice d'adjacence

On choisit de représenter cette situation par un graphe dont les sommets sont les personnes et les arêtes représentent les liens d'amitié.

1. Dessiner sur votre copie ce graphe en représentant chaque personne par la première lettre de son prénom entourée d'un cercle et où un lien d'amitié est représenté par un trait entre deux personnes.

Une matrice d'adjacence est un tableau à deux entrées dans lequel on trouve en lignes et en colonnes les sommets du graphe.

Un lien d'amitié sera représenté par la valeur `1` à l'intersection de la ligne et de la colonne qui représentent les deux amis alors que l'absence de lien d'amitié sera représentée par un `0`.

2. Recopier et compléter l'implémentation de la déclaration de la matrice d'adjacence du graphe.

```
# sommets :      G, J, Y, E, N, M, A, L
matrice_adj = [[0, 1, 1, 0, 1, 1, 0, 0], # G
               [.......................], # J
               [.......................], # Y
               [.......................], # E
               [.......................], # N
               [.......................], # M
               [.......................], # A
               [.......................]] # L
```

On dispose de la liste suivante qui identifie les sommets du graphe :

```
sommets = ['G', 'J', 'Y', 'E', 'N', 'M', 'A', 'L']
```

On dispose d'une fonction `position(l, s)` qui prend en paramètres une liste de sommets `l` et un nom de sommet `s` et qui renvoie la position du sommet `s` dans la liste `l` s'il est présent et `None` sinon.

3. Indiquer quel seront les retours de l'exécution des instructions suivantes :

```
>>> position(sommets, 'G')
>>> position(sommets, 'Z')
```

4. Recopier et compléter le code de la fonction `nb_amis(L, m, s)` qui prend en paramètres une liste de noms de sommets `L`, une matrice d'adjacence `m` d'un graphe et un nom de sommet `s` et qui renvoie le nombre d'amis du sommet `s` s'il est présent dans `L` et `None` sinon.

```
1 def nb_amis(L, m, s):
2     pos_s = ...
3     if pos_s == None:
4         return ...
5     amis = 0
6     for i in range(len(m)):
7         amis += ...
8      return ...
```

5. Indiquer quel est le retour de l'exécution de la commande suivante :

```
>>> nb_amis(sommets, matrice_adj, 'G')
```

Partie B : Dictionnaire de listes d'adjacence

6. Dans un dictionnaire Python `{c : v}`, indiquer ce que représentent `c` et `v`.

On appelle `graphe` le dictionnaire de listes d'adjacence associé au graphe des amis. On rappelle que Gabriel est ami avec Jade, Yanis, Nina et Milo.

```
graphe = {'G' : ['J', 'Y', 'N', 'M'],
          'J' : ...
          ...
         }
```

7. Recopier et compléter le dictionnaire de listes d'adjacence `graphe` sur votre copie pour qu'il modélise complètement le groupe d'amis.
8. Écrire le code de la fonction `nb_amis(d, s)` qui prend en paramètres un dictionnaire d'adjacence `d` et un nom de sommet `s` et qui renvoie le nombre d'amis du nom de sommet `s`. On suppose que `s` est bien dans `d`.

Par exemple :

```
>>> nb_amis(graphe, 'L')
1
```

Milo s'est fâché avec Gabriel et Yanis tandis qu'Anas s'est fâché avec Yanis. Le dictionnaire d'adjacence du graphe qui modélise cette nouvelle situation est donné ci-dessous :

```
graphe = {'G' : ['J', 'N'],
          'J' : ['G', 'Y', 'E', 'L'],
          'Y' : ['J', 'E', 'N'],
          'E' : ['J', 'Y', 'N'],
          'N' : ['G', 'Y', 'E'],
          'M' : ['A'],
          'A' : ['M'],
          'L' : ['J']
        }
```

Pour établir la liste du cercle d'amis d'un sommet, on utilise un parcours en profondeur du graphe à partir de ce sommet. On appelle cercle d'amis de *Nom* toute personne atteignable dans le graphe à partir de *Nom*.

9. Donner la liste du cercle d'amis de Lou.

Un algorithme possible de parcours en profondeur de graphe est donné ci-dessous :

```
visités = liste vide des sommets déjà visités

fonction parcours_en_profondeur(d, s)
    ajouter s à la liste visités
    pour tous les sommets voisins v de s :
        si v n'est pas dans la liste visités :
            parcours_en_profondeur(d, v)
    retourner la liste visités
```

10. Recopier et compléter le code de la fonction `parcours_en_profondeur(d, s)` qui prend en paramètres un dictionnaire d'adjacence `d` et un sommet `s` et qui renvoie la liste des sommets issue du parcours en profondeur du graphe modélisé par `d` à partir du sommet `s`.

```
1 def parcours_en_profondeur(d, s, visites = []):
2     ...
3     for v in d[s]:
4         ...
5             parcours_en_profondeur(d, v)
6     ...
```

EXERCICE 3

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).

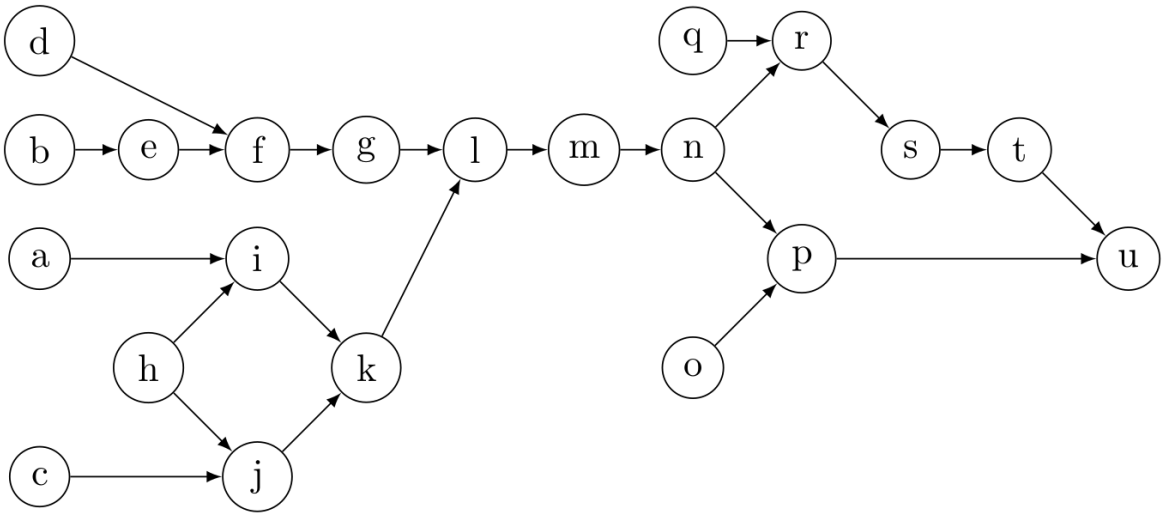

Figure 1. Recette du pain : tâches à effectuer avec leurs dépendances

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`.

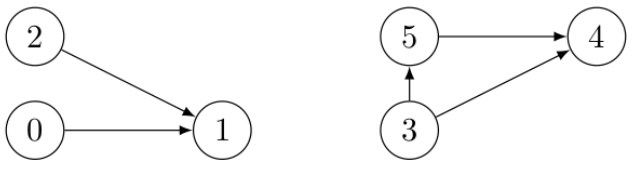

Figure 2. Exemple de graphe de dépendances entre 6 tâches

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.