Exercice 1 (6 points)

Cet exercice porte sur les protocoles réseaux, l'algorithmique et la POO.

Partie A

Un cœur de réseau est constitué d'un maillage dans lequel les routeurs R_1 à R_6 sont reliés par des liaisons physiques dont le débit en Mbps (Méga-bits par seconde) est indiqué sur la figure suivante.

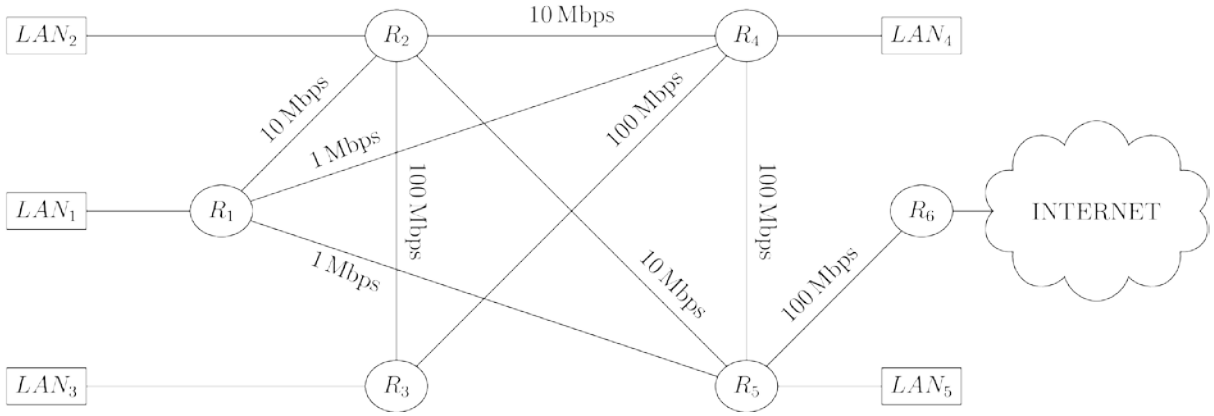

Figure 1. Réseau

Derrière chaque routeur R_1 à R_5, un switch distribue un réseau local LAN_1 à LAN_5.

Les protocoles utilisés sont le protocole RIP et le protocole OSPF, qui minimisent respectivement le nombre de routeurs traversés et la somme des coûts des liaisons physiques empruntées.

1. Recopier et compléter la table de routage de R_1 sachant que le protocole de routage RIP est utilisé.

   La destination est le routeur à atteindre, le prochain saut est le premier routeur traversé et la distance est le nombre de routeurs traversés.

Table de routage R_1		
destination	prochain saut	distance
R_2	R_2	0
R_3	R_2	1
R_4		
R_5		
R_6		

2. Un utilisateur du réseau local LAN_1 interroge, via son navigateur web, un serveur d'Internet. Détailler, suivant le protocole RIP, la route suivie dans le maillage par la requête à destination de ce serveur.

Le coût d'une liaison est donné par la relation coût = $\frac{10^2}{d}$, où d est le débit de la liaison en Mbps.

Exemple : le coût de la liaison entre R_1 et R_2 est $\frac{10^2}{10} = 10$.

Le tableau suivant donne le coût de chacune des liaisons physiques entre R_1 et les autres routeurs du graphe qui lui sont connectés.

Coût des liaisons depuis R_1			
routeur	R_2	R_4	R_5
coût	10	100	100

3. Recopier et compléter la table de routage de R_1 sachant que le protocole de routage OSPF est utilisé.

   La distance est la somme totale des coûts des liaisons physiques empruntées pour atteindre la destination.

Table de routage R_1		
destination	prochain saut	distance
R_2	R_2	10
R_3		
R_4		
R_5		
R_6		

Un utilisateur du réseau local LAN_1 interroge, via son navigateur web, un serveur d'Internet.

4. Donner, selon le protocole OSPF, la route suivie dans le maillage par la requête à destination de ce serveur.

Le protocole de routage OSPF est toujours utilisé et le routeur R_2 tombe en panne.

5. Donner la nouvelle route suivie dans le maillage par une requête partant du réseau LAN_1 à destination d'Internet et en donner la nouvelle valeur de la distance.

Partie B

Le réseau LAN_1 est supposé disposer d'un serveur DHCP (Dynamic Host Control Protocol) gérant l'attribution d'adresses IPv4. Il est rappelé, ici, qu'une adresse IPv4 est une séquence de 4 octets, généralement représentée par les 4 entiers correspondants en écriture décimale, séparés par des points (notation décimale pointée). Un réseau est caractérisé par des machines dont les adresses ont en commun les n mêmes premiers bits. La notation CIDR du réseau a la forme "adresse réseau / n". Appliqué à une adresse du réseau, le masque permet de calculer le préfixe réseau commun. Il est constitué de 4 octets consécutifs, dont (de gauche à droite) les n premiers bits sont égaux à `1` et les suivants à `0`. On obtient l'adresse du réseau en effectuant un `ET` logique, bit à bit, entre chaque octet composant l'adresse d'une machine et l'octet qui lui correspond dans le masque.

Exemple de `ET` :

```
  l'entier 192 s'écrit : 1 1 0 0 0 0 0 0
  l'entier 255 s'écrit : 1 1 1 1 1 1 1 1
-------------------------------------------------------
ET logique, bit à bit  : 1 1 0 0 0 0 0 0 ( soit l'entier 192 )
```

6. Recopier et compléter le tableau suivant correspondant à une machine d'un réseau d'adresse `192.168.1.100` et de masque `255.192.0.0`. Ce tableau détermine l'adresse du réseau (en binaire puis en décimale pointée).

Calcul d'une adresse de réseau				
machine (binaire)	11000000	10101000		
masque (binaire)	11111111	11000000	00000000	00000000
réseau (binaire)	11000000			
réseau (déc. pointée)	`192`			

On obtient le complémentaire d'un octet en échangeant respectivement chacun des 0 et des 1 qui le composent, par un 1 ou un 0.

Par exemple, le complémentaire de `1 1 0 0 1 0 1 0` est `0 0 1 1 0 1 0 1`.

Par un procédé analogue à celui de la question précédente, l'adresse de broadcast d'un réseau est obtenue en effectuant un `OU` logique bit à bit entre chaque octet composant l'adresse réseau et le complémentaire de l'octet correspondant dans l'écriture binaire du masque.

Exemple de OU :

```
          12 s'écrit : 0 0 0 0 1 1 0 0
           9 s'écrit : 0 0 0 0 1 0 0 1
-------------------------------------------------------------
OU logique, bit à bit : 0 0 0 0 1 1 0 1 ( soit l'entier 13 )
```

7. Recopier et compléter le tableau ci-après pour déterminer l'adresse de broadcast du réseau suivant (en binaire puis en décimale pointée).

Calcul d'une adresse de broadcast				
réseau (binaire)	11000000	10000000	00000000	00000000
masque (binaire)	11111111	11000000	00000000	00000000
complément du masque (binaire)				
broadcast (binaire)				
broadcast (déc. pointée)				

Une machine du réseau LAN_1 a reçu du serveur DHCP l'adresse `172.16.1.100`, avec le masque `255.255.0.0`.

Sa passerelle par défaut a pour adresse `172.16.255.254`.

8. En déduire les informations suivantes :
   - l'adresse du réseau LAN_1 (en décimale pointée) ;
   - l'adresse de broadcast (en décimale pointée) ;
   - le nombre total d'adresses pouvant être distribuées par le serveur, en ne tenant pas compte des éventuelles restrictions possiblement posées par l'administrateur du réseau.

On souhaite, désormais, simuler en `Python` le fonctionnement du réseau LAN_1.

On donne, ci-après, les documentations des méthodes `split` et `join`.

- la documentation de la méthode `split` de la classe `str` est la suivante :

```
split(self, séparateur)
  Renvoie la liste des sous-chaines de caractères
délimitées par le séparateur dans l'instance courante de
chaîne de caractères.
  Exemple :
>>> 'ab-pq-rs'.split('-')
['ab', 'pq', 'rs']
```

- la documentation de la méthode `join` de l'objet `str` :

```
join(self, iterable)
Fusionne les chaînes de caractères contenues dans iterable
en le liant par la sous-chaîne depuis laquelle cette
méthode est appelée.
Exemple:
>>>'.'.join(['ab', 'pq', 'rs'])
'ab.pq.rs'
```

On commence par créer la classe `IPv4` suivante.

```
 1 class IPv4(object):
 2     def __init__(self, adresse:str):
 3         """
 4         Constructeur de la classe de calcul sur une
 5         adresse IPv4 dont la notation décimale pointée
 6         est passée en paramètre.
 7         """
 8         self.adresse = adresse
 9 
10     def octets(self)->list[int]:
11         """
12         Découpe l'adresse décimale pointée en la liste
13         des 4 entiers correspondants.
14         """
15         return [int(i) for i in self.adresse.split(".")]
```

Pour mémoire, l'opérateur `&` entre deux entiers effectue le `ET` logique bit à bit de la représentation binaire de ces entiers et renvoie l'entier correspondant à la représentation binaire ainsi obtenue.

Exemple :

```
>>> 192 & 255
192
```

9. Recopier sur la copie et compléter les lignes 15 et 16 du code de la méthode `masquer` de la classe `IPv4`, donné ci-dessous. La méthode doit correspondre au docstring.

```
 1      def masquer(self, masque: str)->str:
 2          """
 3          Détermine le préfixe masqué de l'adresse,
 4          le masque (décimal pointé) étant passé en
 5          paramètre.
 6          >>> add  = IPv4('192.168.1.100')
 7          >>> add.masquer('255.192.0.0')
 8          '192.128.0.0'
 9          """
10          tmp = []
11          ip = self.octets()
12          crible = IPv4(masque).octets()
13          for i in range(4):
14              # Opération booléenne :
15              tmp.append(... & ...)
16          return ".".join(...)
```

10. Recopier sur la copie et compléter les lignes 19 et 20 du code de la méthode `adresse_suivante` de la classe `IPv4`, donné ci-dessous. La méthode doit correspondre au docstring.

```
 1      def adresse_suivante(self, adresse_max:str)->str:
 2          """
 3          Détermine l'adresse décimale pointée suivant
 4          immédiatement l'adresse courante, sous réserve
 5          d'existence d'une adresse disponible
 6          >>> add = IPv4('192.168.1.100')
 7          >>> add.adresse_suivante('192.168.1.254')
 8          '192.168.1.101'
 9          >>> add = IPv4('192.168.1.255')
10          >>> add.adresse_suivante('192.168.255.254')
11          '192.168.2.0'
12          """
13          assert self.adresse < adresse_max
14          liste_courante = self.octets()
15          liste_suivante = list()
16          retenue = 1
17          for index in range(4):
18              somme = liste_courante[3 - index] + retenue
19              valeur, retenue = ..., ...
20              liste_suivante = ...
21          return '.'.join(liste_suivante)
```