Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI T^le Année **2025-26**

Devoir surveillé n°1 – Correction

Exercice 1 : (12pt) *Cet exercice porte sur la programmation, les réseaux et les protocoles de routage.*

Rappels :

Une adresse IPv4 est composée de 4 octets, soit 32 bits. Elle est notée `a.b.c.d`, où `a`, `b`, `c` et `d` sont les écritures décimales des valeurs des 4 octets. Cette écriture est nommée notation décimale pointée.

La notation `a.b.c.d/n` est appelée notation CIDR (*Classless Inter Domain Routing*), l'entier `n` représentant le masque du réseau. Les `n` premiers bits à gauche dans l'adresse IP représentent la partie réseau, les bits à droite qui suivent représentent la partie machine.

- L'adresse IPv4 dont tous les bits de la partie machine sont à `0` est appelée adresse du réseau.
- L'adresse IPv4 dont tous les bits de la partie machine sont à `1` est appelée adresse de diffusion.
- Le masque du réseau est composé de 4 octets : les n premiers bits à gauche sont égaux à `1` et les bits à droite qui suivent sont égaux à `0`.

Dans un réseau informatique, lorsqu'une machine cherche à transmettre des données à une autre machine, elle les transmet sans passer par un routeur si le destinataire fait partie du même réseau, sinon, elle transmet les données à un routeur qui fait office de passerelle entre les différents réseaux.

Dans le schéma réseau de la figure 1, toutes les machines du réseau `192.168.1.0/24` ont pour adresse de passerelle celle de l'interface `G0` du routeur `R1`, soit `192.168.1.254`.

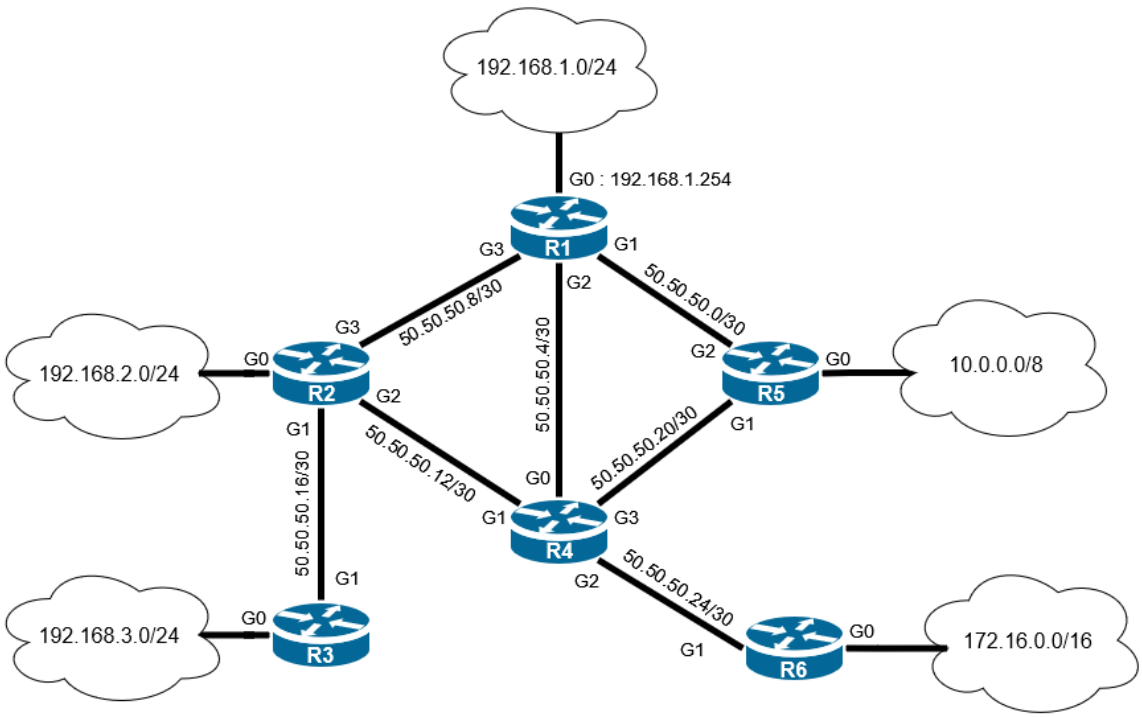

Figure 1. Réseau

1) Donner le nom, ainsi que l'interface, du routeur qui constitue la passerelle pour les machines du réseau `192.168.2.0/24`.
Solution : Il s'agit de R2, avec l'interface `G0`.

La politique d'attribution des adresses IP dans les réseaux nous impose de choisir la dernière adresse IP disponible dans le réseau pour la passerelle. Ainsi, dans le réseau `192.168.1.0/24`, la passerelle a pour adresse IP `192.168.1.254`.

2) Donner l'adresse IP à attribuer à la passerelle du réseau `172.16.0.0/16`.

Solution : La partie machine est composée des 2 dernières octets. Ce sera donc `172.16.255.254`.

Dans le réseau `50.50.50.4/30`, l'interface `G2` du routeur `R1` a pour adresse IP `50.50.50.5`.

3) Lister les quatre adresses du réseau `50.50.50.4/30` et attribuer une adresse IP à l'interface `G0` du routeur `R4`.

Solution : Les adresses sont `50.50.50.4`, `50.50.50.5`, `50.50.50.6` et `50.50.50.7`.
La seule adresse encore disponible pour l'interface `G0` de `R4` est `50.50.50.6`.

Pour choisir la bonne interface de sortie, la passerelle utilise une table de routage qui identifie une interface par où sortir pour trouver le réseau de destination des données et un nombre appelé *métrique* qui représente le coût de la liaison. Cette métrique dépend du type de routage mis en œuvre, manuel (statique) ou automatique (protocoles RIP, OSPF, ...). Dans le cas d'un routage automatique utilisant le protocole RIP, la métrique correspond au nombre minimum de routeurs à traverser pour joindre le réseau de destination.

4) Compléter les lignes manquantes de la table de routage du routeur `R1` dans le cas d'un routage automatique utilisant le protocole RIP. En cas d'égalité de métrique, on choisira l'interface de numéro le plus faible.

Table de routage du routeur `R1`		
Réseau de destination	Interface de sortie	Métrique
`192.168.1.0/24`	`G0`	0
`192.168.2.0/24`	`G3`	1
`192.168.3.0/24`	`G3`	2
`172.16.0.0/16`	`G2`	2
`10.0.0.0/8`	`G1`	1
`50.50.50.0/30`	`G1`	0
`50.50.50.4/30`	`G2`	0
`50.50.50.8/30`	`G3`	0
`50.50.50.12/30`	`G2`	1
`50.50.50.16/30`	`G3`	1
`50.50.50.20/30`	`G1` ou `G2`	1
`50.50.50.24/30`	`G2`	1

On décide de modéliser la table de routage du routeur `R1` par un tableau de triplets contenant l'adresse du réseau de destination et son masque en notation CIDR (sous forme d'un quintuplet), le nom de de l'interface de sortie vers le réseau de destination et la métrique.

5) Compléter les lignes manquantes pour définir le tableau `t_routage` pour qu'il modélise complètement la table de routage du routeur `R1`.

```
1  t_routage = [ ((192, 168, 1, 0, 24), 'G0', 0),
2                ...,
3                ...,
4                ((172, 16, 0, 0, 16), 'G2', 2),
5                ((10, 0, 0, 0, 8), 'G1', 1),
6                ((50, 50, 50, 0, 30), 'G1', 0),
7                ((50, 50, 50, 4, 30), 'G2', 0),
8                ((50, 50, 50, 8, 30), 'G3', 0),
9                ((50, 50, 50, 12, 30), 'G2', 1),
10               ((50, 50, 50, 16, 30), 'G3', 1),
11               ...,
12               ...
13             ]
```

Solution : Les lignes manquantes sont :

```
((192, 168, 2, 0, 24), 'G3', 1),  # ligne 2
((192, 168, 3, 0, 24), 'G3', 2),  # ligne 3
((50, 50, 50, 20, 30), 'G1', 1),  # ou 'G2' | ligne 11
((50, 50, 50, 24, 30), 'G2', 1),  # ligne 12
```

On trouve l'adresse du réseau auquel appartient une adresse IP en appliquant un opérateur ET bit à bit entre l'adresse IP et le masque de sous-réseau.
Par exemple, pour trouver le réseau auquel appartient l'adresse IP `192.168.1.10/24` on fait :

```
adresse IP  :  11000000.10101000.00000001.00001010
masque      :  11111111.11111111.11111111.00000000
            &  --------------------------------
               11000000.10101000.00000001.00000000
```

On peut conclure que l'adresse IP `192.168.1.10/24` appartient au réseau `192.168.1.0/24`.
On dispose d'une fonction `et_bit_a_bit` qui renvoie le résultat de l'opération ET bit à bit entre deux entiers. Ainsi `et_bit_a_bit(10, 252)` renverra `8` car $8 = 0000\ 1000_2$, $10 = 0000\ 1010_2$ et $252 = 1111\ 1100_2$.
De plus, on dispose d'une fonction `mask_for_size(size)` qui prend en paramètre un entier `size` qui représente un masque de sous-réseau en notation CIDR et le renvoie en notation décimale sous la forme d'un quadruplet `(a, b, c, d)` d'entiers compris entre 0 et 255.
Exemples :

```
>>> mask_for_size(24)
(255, 255, 255, 0)
```

```
>>> mask_for_size(26)
(255, 255, 255, 192)
```

6) Donner la sortie de l'exécution du code suivant.

```
>>> mask_for_size(30)
(255, 255, 255, 252)
```

7) Déterminer à quel réseau appartient l'adresse IP `50.50.50.23/30` en écrivant l'opération ET bit à bit effectuée. On convertira 50 et 23 en binaire.

Solution :

```
    00110010.00110010.00110010.00010111  50.50.50.23
    11111111.11111111.11111111.11111100  255.255.255.252
 &  --------------------------------
    00110010.00110010.00110010.00010100  50.50.50.20
```

8) Compléter la fonction `is_in_network` qui prend en paramètres une adresse IP `address` et l'adresse d'un réseau `network`, chacune fournie sous la forme d'un tuple, et qui renvoie **`True`** si l'adresse IP appartient au réseau, et **`False`** sinon.
Exemples :

```
>>> is_in_network((192, 168, 1, 1), (192, 168, 1, 0, 24))
True
>>> is_in_network((192, 168, 1, 1), (192, 168, 2, 0, 24))
False
```

```
def is_in_network(address, network):
    network_mask = mask_for_size(.....................................)
    for i in range(4):
        if et_bit_a_bit(network_mask[i], address[i]) != ..............:
            return ...
    return ...
```

Solution :

```
def is_in_network(address, network):
    network_mask = mask_for_size(network[4])
    for i in range(4):
        if et_bit_a_bit(network_mask[i], address[i]) != network[i]:
            return False
    return True
```

Le routeur sélectionne l'interface de sortie vers le réseau auquel appartient l'adresse IP de destination.

9) Écrire la fonction `choose_interface` qui prend en paramètres un tableau `t_routage` qui modélise une table de routage et un quadruplet `destination_ip` qui représente une adresse IP, et qui renvoie l'interface de sortie du routeur si l'adresse IP `destination_ip` est dans un des réseaux présents dans `t_routage`. Si l'adresse IP de destination n'est pas présente, la fonction renvoie **None**.

Exemples :

```
>>> choose_interface(t_routage, (192, 168, 1, 12))
'G0'
>>> choose_interface(t_routage, (192, 168, 5, 12))
>>> # None ne s'affiche pas
```

Solution :

```
def choose_interface(t_routage, destination_ip):
    for network, interf, metrique in t_routage:
        if is_in_network(destination_ip, network):
            return interf
    return None # pas obligatoire
```

Dans le cas d'un routage automatique utilisant le protocole OSPF, la métrique tient compte du débit des liaisons dont le coût est calculé selon la formule suivante (le débit est donné en bits/seconde) :

$$\text{coût} = \frac{10^{10}}{\text{Débit}}$$

Les débits des liaisons entre les routeurs sont donnés ci-dessous :

Liaisons inter-routeurs	Type	Débit
R1 - R2	Gigabit Ethernet	1 Gb/s
R1 - R4	Fast Ethernet	100 Mb/s
R1 - R5	Gigabit Ethernet	1 Gb/s
R2 - R3	Gigabit Ethernet	1 Gb/s
R2 - R4	Fibre	10 Gb/s
R4 - R5	Gigabit Ethernet	1 Gb/s
R4 - R6	Fibre	10 Gb/s

10) Calculer les coûts des trois types de liaisons.

Solution :

- 1 Gb/s : $\frac{10^{10}}{10^9} = 10$
- 100 Mb/s : $\frac{10^{10}}{10^8} = 100$
- 10 Gb/s : $\frac{10^{10}}{10^{10}} = 1$

11) Compléter les lignes manquantes de la table de routage du routeur R1 dans le cas d'un routage automatique utilisant le protocole OSPF. Un réseau directement connecté au routeur a une métrique de 0, sinon, la métrique est le coût minimum pour joindre le réseau de destination.

Table de routage du routeur R1		
Réseau de destination	Interface de sortie	Métrique
192.168.1.0/24	G0	0
192.168.2.0/24	G3	10
192.168.3.0/24	G3	20
172.16.0.0/16	G3	12
10.0.0.0/8	G1	10
50.50.50.0/30	G1	0
50.50.50.4/30	G2	0
50.50.50.8/30	G3	0
50.50.50.12/30	G3	10
50.50.50.16/30	G3	10
50.50.50.20/30	G1	10
50.50.50.24/30	G3	11

Exercice 2 : (12pt) *Cet exercice porte sur la programmation Python et la récursivité.*
Le jeu du baguenaudier est un jeu de casse-tête constitué d'une réglette comportant n cases, numérotées de 1 à n.
Chaque case peut être soit vide, soit contenir un pion.
« Jouer » une case consiste à placer un pion dans la case (remplir) si elle est vide ou enlever un pion (vider) si elle est remplie.
Initialement, toutes les cases sont vides.
Le but du jeu est de remplir toutes les cases du baguenaudier en suivant les règles suivantes :

- on ne peut jouer qu'une case à la fois ;
- chaque case ne peut contenir qu'un pion ;
- on peut toujours jouer la case 1 ;
- si le baguenaudier n'est ni vide ni rempli, on peut aussi jouer la case qui suit la première case remplie ;
- aucune autre case ne peut être jouée.

Exemple de situation avec un baguenaudier de 5 cases.

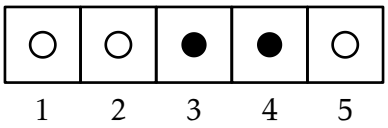

Figure 1. Situation baguenaudier 5 cases

Dans cette situation, on peut poser un pion dans la case 1 ou enlever le pion de la case 4 mais on ne peut pas jouer les cases 2, 3 et 5.

1) On considère un baguenaudier à 4 cases sur lequel les trois dernières cases sont remplies. Désigner les cases que l'on peut jouer et dessiner l'état du baguenaudier à la suite de chacun des coups possibles.

Solution :

- On peut rajouter un jeton en 1 : 
- On peut enlever le jeton en 3 : 

Pour modéliser le baguenaudier on utilise un tableau (de type `list`) de booléens. Une case vide est représentée par **`False`** et une case remplie par **`True`**.
L'exemple de situation du baguenaudier de 5 cases de la Figure 1 présentée plus haut sera ainsi représenté par le tableau suivant.

`[False, False, True, True, False]`

La fonction `initialiser` prend en paramètre un nombre entier `n` et elle renvoie une liste modélisant un baguenaudier vide de `n` cases.

Exemple :

```
>>> initialiser(3)
[False, False, False]
```

2) Écrire le code de la fonction `initialiser`.

Solution :

```
def initialiser(n):
    return [False]*n  # ou [False for _ in range(n)]
```

La fonction `victoire` renvoie un booléen indiquant si toutes les cases du baguenaudier sont remplies.

```
1 def victoire(tab):
2     for etat_case in tab:
3         if etat_case == ..........................:
4             return ...
5     ...
```

3) Compléter les lignes 3, 4 et 5 du code de la fonction `victoire`.

Solution :

```
def victoire(tab):
    for etat_case in tab:
        if etat_case == False:
            return False
    return True
```

La fonction `indice_premiere_case_occupee` prend en paramètre un tableau `tab` modélisant l'état du baguenaudier et renvoie l'indice de la première case remplie du baguenaudier, ou `None` si le baguenaudier est vide.

Exemples :

```
>>> indice_premiere_case_occupee([True, True, True])
0
>>> indice_premiere_case_occupee([False, False, False, True, True])
3
>>> indice_premiere_case_occupee([False, False, False, False])
# pas de sortie car valeur None
```

4) Écrire le code de la fonction `indice_premiere_case_occupee`.

Solution :

```
def indice_premiere_case_occupee(tab):
    for i in range(len(tab)):
        if tab[i] == True:
            return i
    return None # pas obligatoire
```

La fonction `coup_valide` prend en paramètres `tab` qui est une liste modélisant l'état d'un baguenaudier, et un entier `case` qui est l'indice de la case à jouer. La fonction renvoie `True` si le coup respecte les règles et `False` si ce n'est pas le cas.

L'indice de la première case du jeu est 0. Pour que le coup soit valide, il faut aussi s'assurer que l'indice est un entier positif inférieur à la longueur de `tab`.

5) Écrire le code la fonction `coup_valide`.

Solution :

Il faut faire attention à `indice_premiere_case_occupee(tab)` puisque s'il vaut **None**, on ne doit pas ajouter 1, sinon on provoque une erreur.

```
# En testant qu'il y a bien une première case remplie
def coup_valide(tab, case):
    if case == 0:
        return True
    k = indice_premiere_case_occupee(tab)
    return k is not None and case == k+1 and case < len(tab)

# En faisant plutôt case-1
def coup_valide(tab, case):
    if case == 0:
        return True
    k = indice_premiere_case_occupee(tab)
    return case-1 == k and case < len(tab)
```

La fonction `changer_case` prend en paramètres un tableau `tab` modélisant l'état du baguenaudier et un indice de case nommé `case`. Si le coup désigné par `case` est valide, la fonction renvoie le tableau modélisant le baguenaudier obtenu après avoir changé l'état de la case correspondante. Si le coup joué n'est pas valide, le baguenaudier passé en paramètre est renvoyé sans modification.

Exemples :

```
>>> changer_case([False, False, False], 1) # coup non valide
[False, False, False]
>>> changer_case([False, False, False], 0) # coup valide
[True, False, False]
>>> changer_case([False, True, False], 2) # coup valide
[False, True, True]
```

6) Écrire le code de la fonction `changer_case`.

Solution :

```
def changer_case(tab, case):
    if est_valide(tab, case):
        tab[case] = not tab[case]
    return tab
```

La résolution d'un baguenaudier de 3 cases, en Figure 2 et 4 cases, en Figure 3 sont données ci-dessous.

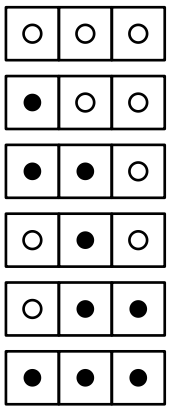

Figure 2. Résolution du baguenaudier à 3 cases

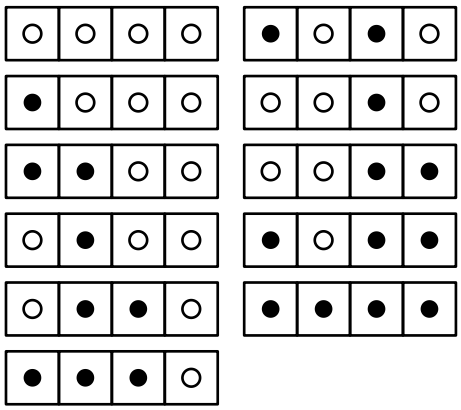

Figure 3. Résolution du baguenaudier à 4 cases

Il est possible d'obtenir la solution du jeu du baguenaudier, sous forme d'affichage, en utilisant des fonctions récursives nommées `vider` et `remplir`. Ces deux fonctions prennent en paramètre un entier `n` correspondant au nombre de cases du jeu. La fonction `vider` vide un baguenaudier de `n` cases initialement remplies. La fonction `remplir` remplit un baguenaudier de `n` cases initialement vides.

Le code de la fonction `vider`, incomplet, est donné ci-dessous. On notera que les cases sont numérotées à partir de 1 dans la suite et que l'appel `vider(0)` ne fait rien.

```
1 def vider(n):
2     if n == 1:
3         print('Vider case 1')
4     elif n > 1:
5         vider(n-2)
6         print(.....)
7         remplir(n-2)
8         vider(n-1)
```

7) Compléter la ligne 6 du code de la fonction `vider` pour qu'elle affiche le texte `'Vider case '` suivi de la valeur de l'entier `n`.

Solution : `print('Vider case', n)`

8) En supposant que l'appel `remplir(1)` affiche `'Remplir case 1'` et que l'appel `remplir(0)` ne fait rien, donner les affichages, dans la console, produits par l'exécution de `vider(3)`.

Solution :

```
vider(3) -> vider(1)                -> 'Vider case 1'
            print(...)              -> 'Vider case 3'
            remplir(1)              -> 'Remplir case 1'
            vider(2) -> vider(0)    ->
                        print(...)  -> 'Vider case 2'
                        remplir(0)  ->
                        vider(1)    -> 'Vider case 1'
```

9) En vous inspirant de la fonction `vider`, de la résolution donnée à la question précédente pour `n` = 3 et des Figures 2 et 3, écrire le code de la fonction récursive `remplir`.

Solution :

```
def remplir(n):
    if n == 1:
        print('Remplir case 1')
    elif n > 1:
        remplir(n-1)
        vider(n-2)
        print('Remplir case', n)
        remplir(n-2)
```

10) On envisage d'utiliser la fonction `vider` pour un baguenaudier de 2000 cases. Expliquer si ce code est adapté à un baguenaudier de si grande taille. Justifier votre réponse.

Solution : Le nombre d'appels est exponeniel. À un moment, il faudra vider les 198 premières cases. Pour cela, il va falloir vider les 196 cases, les re-remplir et vider les 197 premières cases. Pour cela il va falloir vider plusieurs fois les 195 puis 196 premières cases. Et aini de suite. Les 4 premières cases vont être vidées un très grand nombre de fois. Il va donc y avoir de l'ordre de 2^{2000} opérations, ce qui est gigantesque.