CORRECTION BACCALAURÉAT BLANC
Sujet 2

EXERCICE 1 (6 points)

i Cet exercice porte sur l'adressage IP et les protocoles de routage.

•Partie A : L'adressage IP

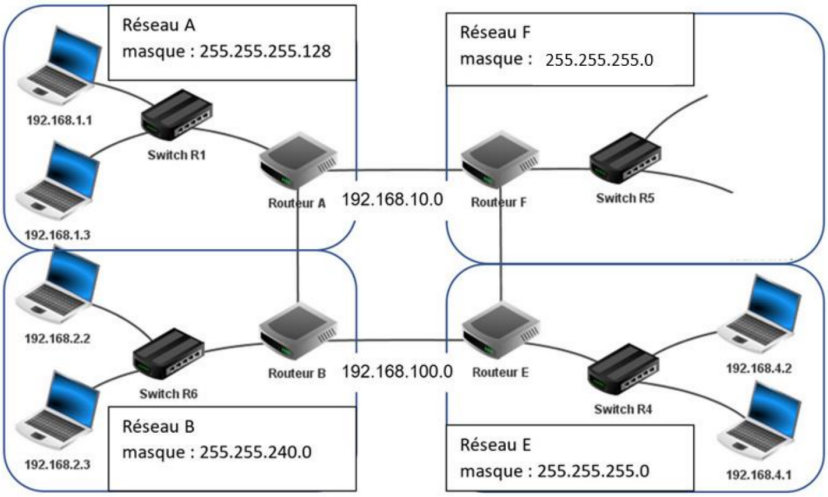

Figure 1 : Plan des réseaux étudiés

1. Nous allons considérer le réseau nommé F tel qu'illustré. Son masque de réseau étant, en décimales pointées, 255.255.255.0, les trois premiers octets d'une adresse IP sur ce réseau servent pour la partie réseau de l'adresse (appelée aussi Net ID), le dernier octet sert pour la partie hôte et est propre à chaque machine sur le réseau. Une machine connectée au switch R5 possède 192.168.5.3 comme adresse IPV4.

   (a) Proposer une adresse IP valide pour le routeur F.

   Correction

   192.168.5.254 est une adresse IP valide.

   (b) Indiquer le nombre maximum de machines que l'on pourra connecter sur ce réseau F.

   Correction

   Ce réseau a des adresses de la forme 192.168.5.X ou X est un entier entre 0 et 255.

Donc la plage IP de ce réseau peut contenir jusqu'à 255 machines. En réalité les adresses 192.168.5.0 et 192.168.5.255 sont réservées.

2. Pour déterminer la partie d'une adresse IPV4 qui correspond à l'adresse réseau, on effectue un ET logique entre chaque bit de l'adresse IP binaire de l'hôte et celle du masque de sous-réseau. Exemple pour un octet :

	1 1 1 0 1 0 1 0	extrait de l'adresse IP
ET	1 1 1 1 1 0 0 0	extrait du masque du réseau
=	1 1 1 0 1 0 0 0	extrait de l'adresse réseau

On considère le réseau B :

(a) Identifier son masque de sous-réseau sur la figure 1 ci-dessus.

Correction

Le masque de sous réseau est 255.255.240.0

(b) Déterminer l'adresse du réseau B, à partir de l'adresse IP d'une machine et du masque de ce réseau. On détaillera soigneusement chaque étape du raisonnement.

Correction

Machine : 192.168.2.2

Machine binaire : 11000000.10101000.00000010.00000010

Masque : 255.255.240.0

Masque binaire : 111111111.111111111.111110000.00000000

Le ET logique nous donne :

Réseau binaire : 11000000.10101000.00000000.00000000

Réseau : 192.168.0.0

(c) Proposer un intérêt au fait d'avoir une telle interconnexion entre les quatre routeurs A, B, E et F.

Correction

Cela permet de résister si une liaison entre 2 routeurs se cassait.

•Partie B : Le routage

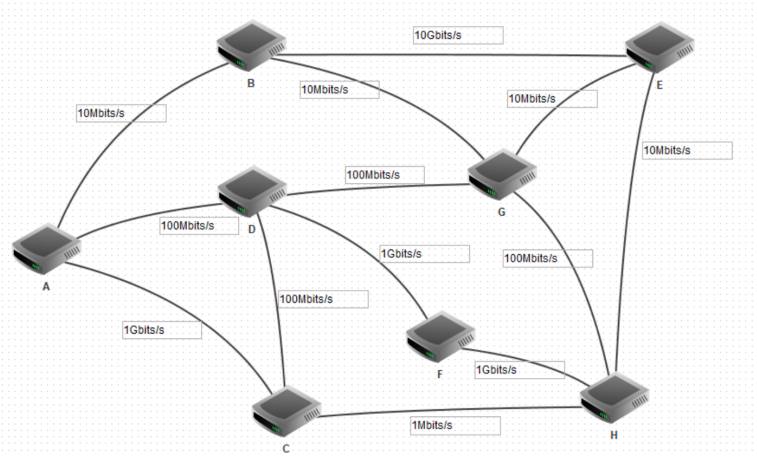

Figure 2 : Plan de routage

1. Dans le cadre du protocole RIP, le chemin emprunté par les informations est celui qui aura la distance la plus petite en nombre de sauts. En considérant le réseau présenté ci-dessus :

   (a) Donner, en respectant le protocole RIP, le(s) chemin(s) possible(s) entre les routeurs A et E, puis entre les routeurs F et B.

   Correction

   1 seule chemin entre A et E : A - B - E

   4 chemins possibles entre F et B : F - D - A - B ; F - D - G - B ; F - H - E - B ; F - H - G - B

   (b) Recopier sur votre copie les tableaux ci-dessous et les compléter pour le routeur E et le routeur G.

   Correction

Table de routage du routeur E		
Destination	Routeur suivant	Distance
A	B	2
B	B	1
C	H	2
D	G	2
F	H	2
G	G	1
H	H	1

Table de routage du routeur G		
Destination	Routeur suivant	Distance
A	D	2
B	B	1
C	D	2
D	D	1
E	E	1
F	D	2
H	H	1

2. On considère à présent le protocole OSPF qui se base sur le coût total minimal de la communication. Le coût entre deux routeurs se calcule en fonction du débit selon la formule suivante :

$$\text{coût} = \frac{10^8}{\text{débit}}$$

   (a) Recopier et compléter la table de routage du routeur F ci dessous.

Correction

Table de routage du routeur F		
Destination	Routeur suivant	Coût total
A	D	1,1
B	**H**	10,11
C	D	1,1
D	D	**0,1**
E	H	10,1
G	D	**1,1**
H	H	0,1

(b) Indiquer quel sera le chemin emprunté par les informations entre le routeur E et le routeur D.

Correction

Avec un coût total de 10,2,le chemin emprunté sera E - H - F - D

EXERCICE 2 (6 points)

i Cet exercice porte sur la programmation en Python, la manipulation des chaînes de caractères, les arbres binaires de recherche et le parcours de liste.

1. On rappelle ici quelques notions sur la manipulation des chaînes de caractères en Python. Une chaîne de caractères se comporte comme un tableau de caractères que l'on ne peut pas modifier.

   Par exemple, on a le comportement suivant :

```
>>> une_chaine = 'Bonjour'
>>> une_chaine[3]
'j'
>>> une_chaine[3] = 'z'
TypeError: 'str' object does not support item assignment
```

   On peut aussi utiliser l opérateur de concaténation +.

```
>>> une_chaine = 'a' + 'b'
>>> une_chaine
'ab'
>>> une_chaine = une_chaine + 'c'
>>> une_chaine
'abc'
```

   On définit la fonction `bonjour` par le code suivant :

```
def bonjour(nom)
    return 'Bonjour ' + nom + ' !'
```

   (a) Donner le résultat de l'exécution de `bonjour('Alan')`.

   Correction

   La fonction renverra la chaine de caractères `Bonjour Alan !`

   On exécute le programme suivant :

```
une_chaine='Bonjour'
x = (une_chaine[2] == une_chaine[3])
y = (une_chaine[4] == une_chaine[1])
```

   (b) Donner le type et les valeurs des variables `x` et `y`.

   Correction

   Ce sont des booléens (`True` / `False`)

   (c) Écrire une fonction `occurrences_lettre(une_chaine,une_lettre)` prenant en paramètres une chaîne `une_chaine` et une lettre `une_lettre` et renvoyant le nombre d'occurrences de `une_lettre` dans `une_chaine`.

Correction

```
def occurrences_lettre(une_chaine,une_lettre):
    compteur = 0
    for l in une_chaine:
        if l == une_lettre:
            compteur += 1
    return compteur
```

2. On rappelle qu'un arbre binaire de recherche est un arbre binaire pour lequel chaque noeud possède une étiquette dont la valeur est supérieure ou égale à toutes les étiquettes des nœuds de son fils gauche et strictement inférieure à celles des nœuds de son fils droit.
   Sa taille est son nombre de nœuds ; sa hauteur est le nombre de niveaux qu'il contient.
   On rappelle aussi que l'on peut comparer des chaînes de caractères en utilisant l'ordre alphabétique. On a par exemple :

```
>>> 'ab' < 'aa'
False
>>> 'abc' < 'acb'
True
```

   On considère la liste de mots `animaux = ['python', 'chameau', 'pingouin', 'renard', 'gnou']`
   (a) Dessiner un arbre binaire de recherche contenant tous les mots de la liste animaux et de hauteur minimale.
   (b) Dessiner un arbre binaire de recherche contenant tous ces mots et de hauteur maximale.
3. On considère l'implémentation objet suivante d'un arbre binaire de recherche :
   On dispose d'une classe **`Abr`** contenant notamment les méthodes et attributs suivants :
   - Si **`un_abr`** est une instance d'Abr alors **`un_abr.est_vide()`** renvoie **`True`** si l'arbre est vide et **`False`** sinon.
   - Si **`un_abr`** est une instance d'Abr et si **`un_abr.est_vide()`** renvoie **`False`**, alors un_abr.valeur contient une chaîne de caractères représentant la valeur de la racine de l'arbre.
   - Si **`un_abr`** est une instance d'Abr et si **`un_abr.est_vide()`** renvoie **`False`**, alors **`un_abr.sag`** et **`un_abr.sad`** contiennent chacun une instance d'Abr.

     On considère que la variable **`liste_mots_francais`** est une liste de 336531 mots en français et que la variable **`abr_mots_francais`** est une instance d'Abr contenant les mots de la liste.

   On considère la fonction suivante :

```
def mystere(un_abr):
    if un_abr.est_vide():
        return 0
    else:
        return 1 + mystere(un_abr.sag) + mystere(un_abr.sad)
```

(a) Donner le résultat de `mystere(abr_mots_francais)`, en justifiant le résultat.

Correction

La fonction `mystere` renvoie la taille de l'arbre passé en paramètre.
Ainsi `mystere(abr_mots_francais)` renverra 336531.

On veut calculer la hauteur de `abr_mots_francais`.

(b) Donner le code d'une fonction `hauteur(un_abr)` permettant de faire ce calcul, en vous inspirant du code précédent.

Correction

```
1  def hauteur(un_abr):
2      if un_abr.est_vide():
3          return 0
4      else:
5          return 1 + max(hauteur(un_abr.sag), hauteur(un_abr.sad))
```

4. Dans cette question, nous nous servirons uniquement de `liste_mots_francais` et plus de `abr_mots_francais`.
Pour aider à la résolution de mots croisés, on a décidé d'écrire une fonction `chercher_mots(liste_mots, longueur, lettre, position)` où `liste_mots` est une liste de mots français, `longueur` est la taille du mot recherché,`lettre` est une lettre du mot se trouvant à l'indice `position`.
Par exemple `chercher_mots(liste_mots_francais,3,'x',2)` renverra `['aux', 'box', 'dix', 'eux', 'fax', 'fox', 'lux', 'max', six']`.

(a) Recopier et compléter la ligne 4 de la fonction ci-dessous :

Correction

```
1  def chercher_mots(liste_mots,longueur,lettre,position):
2      res = []
3      for i in range(len(liste_mots)):
4          if len(liste_mots[i]) == longueur and liste_mot[i][position] ==
       ↪  lettre :
5              res.append(liste_mots[i])
6      return res
```

(b) Expliquer ce que donne la commande suivante.

```
1  chercher_mots(chercher_mots(liste_mots_francais,3,'x',2),3,'a',1)
```

Correction

l'instruction `chercher_mots(chercher_mots(liste_mots_francais,3,'x',2),3,'a',1)` permet de trouver les mots de 3 lettres se terminant par `ax`

On cherche un mot de 5 lettres dont on connaît la fin 'ter'.

(c) Écrire la commande permettant de chercher les mots candidats dans liste_mots_francais.

Correction

```
chercher_mots(chercher_mots(chercher_mots(liste_mots_francais,5,'r',4),5,'e',3),5,'t',2)
```

EXERCICE 3 (8 points)

i Cet exercice porte sur la programmation Python (dictionnaire), la programmation orientée objet, les bases de données relationnelles et les requêtes SQL. Cet exercice est composé de 3 parties indépendantes.

La société CarteMap veut créer une application permettant de stocker et de traiter des informations sur des livres de science-fiction. On désire stocker les informations suivantes :

- l'identifiant de la ville (`id`) : chaque ville possède un id unique ;
- le nom de la ville (`nom`) ;
- le numéro du département où se situe la ville (`num_dep`) ;
- le nombre d'habitants (`nombre_hab`) ;
- la superficie de la ville en km^2 (`superficie`).

Voici un extrait des informations que l'on cherche à stocker :

`ville`

id	nom	num_dep	nombre_hab	superficie
1	Annecy	74	125694	67
2	Tours	37	136252	34.4
3	Lyon	69	513275	47.9
4	Chamonix	74	8906	246
5	Rennes	35	215366	50.4
6	Nice	06	342522	72
7	Bordeaux	33	249712	49.4

•Partie A

Dans cette première partie, on utilise un dictionnaire Python. On considère le programme suivant :

```
dico_villes = {
    'id' : [1, 2, 3, 4, 5, 6 ,7],
    'noms' : ['Annecy', 'Tours', 'Lyon', 'Chamonix', 'Rennes',
            'Nice', 'Bordeaux'],
    'num_dep' : [74, 37, 69, 74, 35, 06, 33],
    'nombre_hab' : [125694, 136252, 513275, 8906, 215366, 342522, 249712],
    'superficie' : [67, 34.4, 47.9, 246, 50.4, 72, 49.4]
 }
a = dico_villes['num_dep']
b = dico_villes['noms'][4]
```

1. Déterminer les valeurs des variables `a` et `b` après l'exécution de ce programme.

Correction

La valeur de `a` sera `[74, 37, 69, 74, 35, 06, 33]` La valeur de `b` sera `Rennes`

La fonction `nom_ville` prend en paramètre un dictionnaire (de même structure que `dico_villes`) et un identifiant, et renvoie le nom de la ville qui correspond à cet identifiant.

Dans le cas où l'identifiant passé en paramètre n'est pas présent dans le dictionnaire, la fonction renvoie `None`.

Correction

```
    def titre_livre(dico, id_ville):
        for i in range(len(dico['id'])):
            if dico['id'][i] == id_ville :
                return dico['noms'][i]
        return None
```

2. Écrire une fonction **moyenne_hab** qui prend en paramètre un dictionnaire **dico** (de même structure que **dico_villes**) et qui renvoie la moyenne du nombre d'habitants par ville.

Correction

```
def moyenne_hab(dico):
    somme = 0
    for hab in dico["nombre_hab"]:
        somme += hab
    return somme / len(dico["nombre_hab"])
```

3. Écrire une fonction **villes_dep** qui prend en paramètre un dictionnaire **dico** (de même structure que **dico_villes**) et un numéro de département **num**, et qui renvoie la liste des villes du département dont le numéro correspond à **num** (on rappelle que **t.append(a)** permet de rajouter l'élément a à la fin de la liste **t**).

Correction

```
def villes_dep(dico,num):
    villes = []
    for i in range(len(dico[note])):
        if dico["num_dep"][i] == num:
            villes.append(dico["nom"][i])
    return villes
```

4. Écrire une fonction **ville_peuplee** qui prend en paramètre un dictionnaire **dico** (de même structure que **dico_ville**) et qui renvoie la liste des villes ayant une population supérieur à la moyenne sous la forme d'une liste Python.

Correction

```
def ville_peuplee(dico):
    moyenne = moyenne_hab(dico)
    ville_peuplee = []
    for i in range(len(dico["nom"])):
        if dico["nombre_hab"][i] > moyenne:
            ville_peuplee.append(dico["nom"][i])
    return ville_peuplee
```

•Partie B

Dans cette partie, on utilise le paradigme orientée objet (POO). On propose la classe : `Ville` et `Bibliotheque`.

```
class Ville:
    def __init__(self, id_ville, nom, num_dep, nombre_hab, superficie):
        self.id = id_ville
        self.nom = nom
        self.num_dep = num_dep
        self.nombre_hab = nombre_hab
        self.superficie = superficie

    def get_id(self):
        return self.id

    def get_nom(self):
        return self.nom
```

5. Citer un attribut et une méthode de la classe `Ville`.

Correction

`id` est un attribut de la classe `Livre` et `get_nom` est une méthode de la classe `Livre`

6. Écrire la méthode `get_superficie` de la classe `Ville`. Cette méthode devra renvoyer la superficie de la ville.

Correction

```
def get_superficie(self):
    return self.superficie
```

•Partie C

La société CarteMap décide d'ajouter à son logiciel de cartographie des données sur les différentes villes, notamment des données classiques : nom, département, nombre d'habitants, superficie, ..., mais également d'autres renseignements pratiques, comme par exemple, des informations sur les infrastructures sportives proposées par les différentes municipalités.
Dans un premier temps, la société a pour projet de stocker toutes ces données dans un fichier texte. Mais, après réflexion, les développeurs optent pour l'utilisation d'une base de données relationnelle.

7. Expliquer en quoi le choix d'utiliser un système de gestion de base de données (SGBD) est plus pertinent que l'utilisation d'un simple fichier texte.

Correction

Le SGBD permet d'éviter la redondance des données en séparant les données concernant le sport et la ville.

On donne les deux tables suivantes :

Table **ville**

id	nom	num_dep	nombre_hab	superficie
1	Annecy	74	125694	67
2	Tours	37	136252	34.4
3	Lyon	69	513275	47.9
4	Chamonix	74	8906	246
5	Rennes	35	215366	50.4
6	Nice	06	342522	72
7	Bordeaux	33	249712	49.4

Table **sport**

id	nom	type	note	id_ville
1	Richard Bozon	piscine	9	4
2	Bignon	terrain multisport	7	5
3	Ballons perdus	terrain multisport	6	1
4	Mortier	piscine	8	2
5	Block'Out	mur d'escalade	8	2
6	Trabets	mur d'escalade	7	4
7	Centre aquatique du lac	piscine	9	2

Dans la table ville, on peut trouver les informations suivantes :

- l'identifiant de la ville (**id**) : chaque ville possède un id unique ;
- le nom de la ville (**nom**) ;
- le numéro du département où se situe la ville (**num_dep**) ;
- le nombre d'habitants (**nombre_hab**) ;
- la superficie de la ville en km^2 (**superficie**).

Dans la table **sport**, on peut trouver les informations suivantes :

- l'identifiant de l'infrastructure (**id**) : chaque infrastructure a un id unique ;
- le nom de l'infrastructure (**nom**) ;
- le type d'infrastructure (**type**) ;
- la note sur 10 attribuée à l'infrastructure (**note**) ;
- l'identifiant de la ville où se situe l'infrastructure (**id_ville**).

En lisant ces deux tables, on peut, par exemple, constater qu'il existe une piscine Richard Bozon à Chamonix.

8. Donner le schéma relationnel de la table **sport**.

Correction

sport
id : INT nom : VARCHAR type : VARCHAR note : INT #id_ville : INT

9. Expliquer le rôle de l'attribut **id_ville** dans la table sport.

Correction

`id_ville` est une clé étrangère qui fait référence à la clé primaire `id` de la table `ville`.
Elle permet de faire une jointure entre les deux tables.

10. Donner le résultat de la requête SQL suivante :

```
SELECT nom
FROM ville
WHERE num_dep = 74 AND superficie > 70;
```

Correction

Le résultat sera `(Chamonix)`

11. Écrire une requête SQL permettant de lister les noms de l'ensemble des piscines présentes dans la table `sport`.

Correction

```
SELECT nom
FROM sport
WHERE type = "piscine";
```

Suite à de bons retours d'utilisateurs, la note du terrain multisport "Ballons perdus" est augmentée d'un point (elle passe de 6 à 7).

12. Écrire une requête SQL permettant de modifier la note du terrain multisport "Ballon perdus" de 6 à 7.

Correction

```
UPDATE sport
SET note = 7
WHERE nom = "Ballons perdus";
```

13. Écrire une requête SQL permettant d'ajouter la ville de Toulouse dans la table ville. Cette ville est située dans le département de la Haute-Garonne (31). Elle a une superficie de 118 km2. En 2023, Toulouse comptait 471941 habitants. Cette ville aura l'identifiant 8.

Correction

```
INSERT INTO  ville(id,nom,num_dep,nombre_hab,superficie )
VALUES (8, "Toulouse", 31, 471941, 118);
```

14. Écrire une requête SQL permettant de lister les noms des murs d'escalade disponibles à Annecy.

Correction

```sql
SELECT sport.nom FROM sport
JOIN ville
ON ville.id = sport.id_ville
WHERE sport.type = "piscine" AND ville.nom = "Annecy";
```