Métropole - septembre 2025 - sujet 1

Exercice 1 (Programmation Python, réseaux, protocoles de routage - 6 points)
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`.

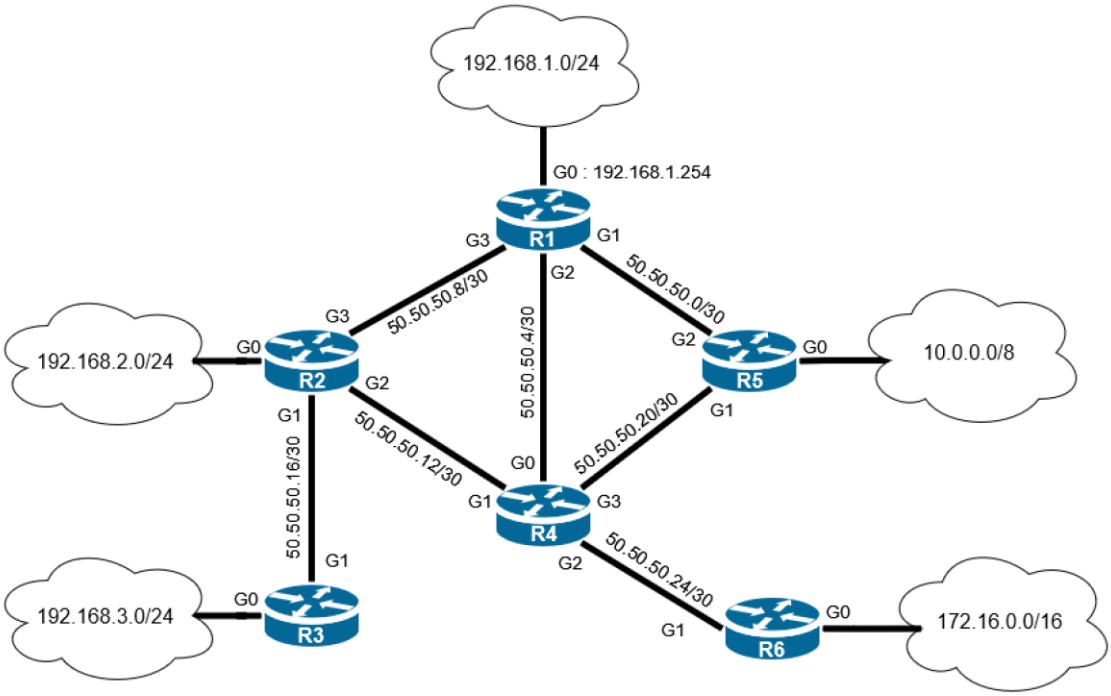

FIGURE 1 – Réseau

1. Donner le nom, ainsi que l'interface, du routeur qui constitue la passerelle pour les machines du réseau `192.168.2.0/24`.

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`.

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.

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 oeuvre, 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. Recopier et 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`		
`192.168.3.0/24`		
`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`		
`50.50.50.24/30`		

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. Recopier et compléter les lignes manquantes pour définir le tableau `t_routage` pour qu'il modélise complètement la table de routage du routeur R1 (vous écrirez les numéros de lignes complétées).

```
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              ...]
```

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,` 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. Par exemple :

```
>>> 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)
```

7. Déterminer à quel réseau appartient l'adresse IP `50.50.50.6/30` en écrivant l'opération `ET` bit à bit effectuée. On convertira `50` et `6` en binaire.
8. Recopier et compléter la fonction `is_in_network(address, 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.

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

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
```

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

9. Écrire la fonction `choose_interface(t_routage, destination_ip)` 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`. Par exemple :

```
>>> choose_interface(t_routage, (192, 168, 1, 12))
G0
>>> choose_interface(t_routage, (192, 168, 5, 12))
None
```

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.

11. Recopier et 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`		
`192.168.3.0/24`	G3	20
`172.16.0.0/16`		
`10.0.0.0/8`		
`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`		
`50.50.50.24/30`	G3	11

Exercice 2 (Algorithmique, listes et programmation dynamique - 6 points)
Dans un jeu de stratégie, la carte est un carré de $n \times n$ cases, où n est un entier strictement positif. Certaines cases sont dites *constructibles*, d'autres ne le sont pas. Toutes les cartes contiennent au moins une case constructible.

Une telle carte est représentée en Python par une liste de listes. Les cases constructibles sont représentées par des cellules contenant la valeur `1`, celles qui sont non constructibles par des cellules contenant la valeur `0`. On fournit ci-dessous la représentation d'une carte pour laquelle n est égal à 5 :

```
carte_A = [[0, 0, 1, 1, 1],
           [1, 1, 0, 1, 1],
           [0, 1, 1, 1, 0],
           [0, 1, 1, 1, 0],
           [0, 1, 1, 1, 0]]
```

Une *base* dans la carte est un carré formé de cases constructibles. On cherche à construire la plus grande base possible, ce qui revient donc à trouver un plus grand carré, inclus dans la carte, ne contenant que des valeurs `1`. Il peut en exister plusieurs.

Dans la carte précédente, la plus grande base possible est un carré de trois cases de côté, son coin supérieur gauche est la cellule `carte_A[2][1]`, de *coordonnées* `(2,1)`. On dit que cette base est *issue* de la cellule `carte_A[2][1]` et que sa taille vaut `3`.

Partie A
On considère la `carte_B` donnée ci-dessous :

```
carte_B = [[1, 1, 1, 1],
           [0, 1, 1, 1],
           [0, 1, 1, 1],
           [1, 0, 1, 1]]
```

1. Donner la taille de la plus grande base carrée ainsi que les coordonnées de la cellule dont elle est issue.

Afin de répondre à ce problème, on envisage une recherche exhaustive de solution. Cela signifie que l'on teste tous les carrés inclus dans la carte initiale afin de vérifier s'ils ne contiennent que des cases constructibles. On considère dans un premier temps le carré de taille `n`, puis les quatre carrés de taille `n - 1` et ainsi de suite jusqu'aux carrés de taille `1`. Dès que l'on trouve un carré constructible, on renvoie sa taille et les coordonnées de son coin supérieur gauche.

2. On considère un entier `taille` strictement positif. Déterminer la somme des valeurs de toutes les cellules d'un carré constructible de `taille` cases de côté.

La fonction `est_constructible` prend en paramètres :

- la liste de listes `carte` représentant la carte ;
- les entiers `i_coin` et `j_coin` indiquant les coordonnées de la cellule dont est issu le carré considéré (`i_coin` est l'indice de la ligne et `j_coin` celui de la colonne) ;
- l'entier positif `taille` indiquant la taille de ce carré.

Cette fonction renvoie `True` si le carré décrit par les paramètres est constructible, `False` dans le cas contraire.

Cette fonction n'a pas besoin de vérifier que les coordonnées et la taille passées en paramètres définissent toujours un carré valide dont toutes les cases sont incluses dans la carte. On suppose que c'est toujours le cas.

3. Compléter le code de la fonction `est_constructible`.

```
1 def est_constructible(carte, i_coin, j_coin, taille):
2     s = 0
3     for i in range(i_coin, i_coin + taille):
4         for j in range(..., ...):
5             s = ...
6     ... # Plusieurs lignes possibles
```

La fonction `plus_grande_base_exhaustive` prend en paramètre la liste de listes `carte` représentant une carte et renvoie un triplet `(taille, i, j)` dans lequel `taille` est la taille d'un carré constructible de taille maximale et `i` et `j` sont les coordonnées de son coin supérieur gauche. Cette fonction utilise la méthode exhaustive décrite plus haut.

On rappelle qu'une carte contient toujours au moins une case constructible et que cette fonction renverra donc toujours un résultat.

4. Compléter le code de la fonction `plus_grande_base_exhaustive`.

```
def plus_grande_base_exhaustive(carte):
    n = len(carte)
    # les tailles vont de n à 1, de -1 en -1
    for taille in range(n, 0, -1):
        i = 0
        while i + taille <= n:
            j = ...
            while ...:
                if est_constructible(...):
                    return (taille, i, j)
                j = ...
            i = i + 1
```

Un décompte du nombre de carrés à étudier dans le pire des cas en fonction de la taille de la carte initiale permet de construire la figure 1 ci-dessous.

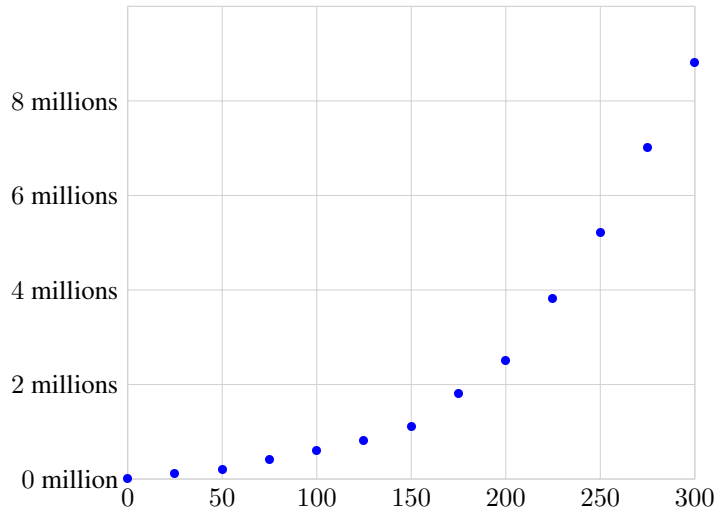

FIGURE 1 – Nombre de carrés à étudier en fonction de la largeur de la carte

5. Expliquer pourquoi cette approche exhaustive est inapplicable dans le cas de grandes cartes.

Partie B

On souhaite désormais résoudre ce problème en utilisant la *programmation dynamique*. Pour cela, on construit une liste de listes auxiliaire `aux` de mêmes dimensions que la carte et telle que `aux[i][j]` contienne la taille de la plus grande base (carrée) issue de `carte[i][j]`.

En reprenant l'exemple de `carte_A`, on obtient la liste de listes `aux_A` :

```
carte_A = [[0, 0, 1, 1, 1],
           [1, 1, 0, 1, 1],
           [0, 1, 1, 1, 0],
           [0, 1, 1, 1, 0],
           [0, 1, 1, 1, 0]]

aux_A = [[0, 0, 1, 2, 1],
         [1, 1, 0, 1, 1],
         [0, 3, 2, 1, 0],
         [0, 2, 2, 1, 0],
         [0, 1, 1, 1, 0]]
```

`aux_A[2][1]` contient la valeur `3` car la plus grande base issue de la cellule `carte_A[2][1]` a une taille de `3`.

Une fois la liste de listes auxiliaire `aux` remplie, on détermine la taille et les coordonnées de la plus grande base (carrée) de la carte en cherchant la valeur maximale dans `aux`.

6. Déterminer la liste auxiliaire `aux_B` associée à la carte représentée par `carte_B`.

```
carte_B = [[1, 1, 1, 1],
           [0, 1, 1, 1],
           [0, 1, 1, 1],
           [1, 0, 1, 1]]
```

On considère une carte `carte_C` de taille `6` ainsi que la liste auxiliaire `aux_C` associée. Seules certaines valeurs sont données ci-dessous.

```
carte_C = [[..., ..., ..., 0, ..., ...],
           [1, ..., ..., ..., ..., ...],
           [..., ..., ..., ..., ..., ...],
           [..., ..., ..., 1, ..., ...],
           [1, ..., ..., ..., ..., ...],
           [..., ..., ..., ..., ..., ...]]

aux_C = [[..., ..., ..., a, 2, ...],
         [b, 0, ..., 1, 1, ...],
         [3, 2, ..., ..., ..., ...],
         [..., ..., ..., c, 2, ...],
         [d, 2, ..., 2, 2, ...],
         [1, 1, ..., ..., ..., ...]]
```

7. Déterminer les valeurs des coefficients `a`, `b`, `c` et `d`.

Pour une liste de listes `carte` donnée, lorsqu'on construit la liste auxiliaire `aux` associée, on commence par remplir les cellules de la dernière ligne et de la dernière colonne de `aux` en recopiant celles de `carte`.

On admet de façon générale que, pour toutes les autres cellules, si `carte[i][j]` vaut `0` alors `aux[i][j]` prend aussi la valeur `0`, et, si `carte[i][j] = 1`, alors `aux[i][j]` se calcule à l'aide de l'expression suivante :

```
1 + min(aux[i + 1][j], aux[i][j + 1], aux[i + 1][j + 1])
```

8. Recopier et compléter les lignes 8, 9, 14 et 15 de la fonction `calcule_aux` qui prend en paramètre une liste de listes `carte` et renvoie la liste auxiliaire associée.

```
1  def calcule_aux(carte):
2      n = len(carte)
3      aux = [[0 for j in range(n)] for i in range(n)]
4
5      # Remplissage de la dernière ligne
6      # et de la dernière colonne
7      for k in range(n):
8          aux[n - 1][k] = ...
9          aux[...][...] = ...
10
11     # On complète les lignes de bas en haut
12     for i in range(n - 2, -1, -1):
13         # On complète les colonnes de droite à gauche
14         for j in range(...):
15             if ...:
16                 aux[i][j] = 1 + min(aux[i + 1][j], aux[i][j + 1], aux[i + 1][j + 1])
17     return aux
```

La fonction `plus_grande_base` prend en paramètre une liste de listes `carte` et renvoie la taille et les coordonnées du coin supérieur gauche d'une base de taille maximale. Cette fonction utilise la liste auxiliaire `aux` renvoyée par l'appel `calcule_aux(carte)`.

9. Compléter la fonction `plus_grande_base` à partir de la ligne 7. Il est possible d'écrire plusieurs lignes.

```
1 def plus_grande_base(carte):
2     n = len(carte)
3     aux = calcule_aux(carte)
4     taille_max = aux[0][0]
5     i_max = 0
6     j_max = 0
7     ...
```

`carte_1000` est la liste représentant une carte de 1000×1000 cases et `carte_3000` celle représentant une carte de 3000×3000 cases. L'appel `plus_grande_base(carte_1000)` s'exécute en $0{,}4$ seconde.

10. Parmi les durées suivantes, indiquer celle qui permet d'estimer le temps d'exécution de l'appel `plus_grande_base(carte_3000)` :

    - $0{,}4$ seconde ;
    - $1{,}2$ seconde ;
    - $3{,}6$ secondes.

    Justifier.

Exercice 3 (Langage SQL, programmation Python et recherche textuelle - 8 points)
Le sujet d'une étude porte sur les papillons, la corrélation entre leur présence et celle de certaines plantes ainsi que sur la classification de nouvelles espèces.

Partie A : corrélation avec la présence des plantes
Dans cet exercice, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT`.

Dans le cadre de cette étude, une base de données `faune_flore.db` a été créée pour étudier la corrélation entre la présence d'espèces de papillons et celle de certaines plantes. Cette base de données regroupe les tables `papillon`, `plante` et `zone_geographique`.

La table `papillon` comporte les informations suivantes :

- l'identification du papillon (`num`) ;
- le nom commun du papillon (`nomCo`) ;
- le nom scientifique du papillon (`nomSc`) ;
- la taille moyenne du papillon en millimètres (`taille`) ;
- le principal habitat du papillon (`habitat`) ;
- la zone géographique où le papillon est le plus présent (`zone`).
  Cet attribut fait référence à l'attribut `num` de la table `zone_geographique`.

Un extrait de cette table est donné ci-après.

papillon					
num	nomCo	nomSc	taille	habitat	zone
458	Monarque	Danaus plexippu	100	Prairies	3
459	Citron de Provence	Gonepteryx cleopatra	30	Prairies	1
460	Paon-du-jour	Aglais io	6	Jardins	6
461	Machaon	Papilio machaon	85	Forêts	2
462	Petite Tortue	Aglais urticae	30	Prairies	5
463	Robert-le-Diable	Polygonia c-album	25	Forêts	4

La table `plante` comporte les informations suivantes :

- l'identification de la plante (`num`) ;
- le nom commun de la plante (`nomCo`) ;
- le nom scientifique de la plante (`nomSc`) ;
- le principal habitat de la plante (`habitat`) ;
- la zone géographique où elle est la plus présente (`zone`).
  Cet attribut fait référence à l'attribut `num` de la table `zone_geographique`.

Un extrait de la table `plante` est donné ci-dessous.

plante				
num	nomCo	nomSc	habitat	zone
128	Orchidée Phalaenopsis	Phalaenopsis	Forêts	5
129	Bambou	Bambusoideae	Forêts	3
130	Rose	Rosa	Haies	2
131	Lilas	Syringa	Haies	6
132	Coquelicot	Papaver rhoeas	Jardins	4
133	Lavande	Lavandula	Collines	1

La table `zone_geographique` contient les informations suivantes :

- l'identification de la zone géographique (`num`) ;
- le nom de la zone (`zone`).

Un extrait de la table `zone_geographique` est donné ci-après.

zone_geographique	
num	zone
1	Afrique du Nord
2	Amérique du Nord
3	Amérique du Sud
4	Asie
5	Asie du Sud
6	Europe

1. Donner la définition d'une clé primaire.
2. Expliquer pourquoi l'attribut `habitat` de la table `papillon` ne peut pas être une clé primaire.
3. Donner le résultat obtenu suite à l'exécution de la requête suivante si on l'applique sur l'extrait de table donné :

```
SELECT taille FROM papillon WHERE nomCo='Machaon'
```

Après avoir mesuré l'envergure de plusieurs papillons Petite Tortue, un des scientifiques de l'étude a calculé la nouvelle moyenne des tailles pour ce papillon, qui est maintenant de 50 mm.

4. Écrire une requête qui met à jour la table `papillon`, suite au calcul de cette nouvelle moyenne.
5. Écrire une requête qui affiche le nom commun de tous les papillons présents dans les prairies et dont la taille est strictement inférieure à 55 mm.
6. Écrire le résultat obtenu suite à l'exécution de la requête suivante si on l'applique sur les extraits des tables donnés.

```
SELECT nomSc FROM plante
JOIN zone_geographique
ON plante.zone = zone_geographique.num
WHERE zone_geographique.zone = 'Asie'
```

7. Écrire une requête qui affiche le nom commun des papillons et celui des plantes qui se trouvent dans le même habitat et dont la taille des papillons est strictement inférieure à 55 mm.
8. Écrire le résultat obtenu suite à l'exécution de la requête suivante si on l'applique sur les extraits des tables donnés.

```
SELECT papillon.nomCo, plante.nomCo
FROM papillon
JOIN zone_geographique
ON papillon.zone = zone_geographique.num
JOIN plante
ON plante.zone = zone_geographique.num
WHERE zone_geographique.zone = 'Europe'
```

9. Écrire une requête qui affiche le nom commun des papillons qui se trouvent dans la même zone géographique que les coquelicots.

Partie B : classification d'une nouvelle espèce

Les espèces de papillons sont regroupées dans une liste de dictionnaires. Pour simplifier, seuls les attributs `num` (l'identifiant du papillon), `nomCo` (son nom commun), `nomSc` (son nom scientifique) et `taille` (sa taille) seront considérés dans cette partie. Une partie de la liste papillon est donnée ci-dessous :

```
papillon = [{'num': 458, 'nomCo': 'Monarque', 'nomSc': 'Danaus plexippu', 'taille': 100},
            {'num': 459, 'nomCo': 'Citron de Provence', 'nomSc': 'Gonepteryx cleopatra', 'taille': 30},
            {'num': 460, 'nomCo': 'Paon-du-jour', 'nomSc': 'Aglais io', 'taille': 6},
            {'num': 461, 'nomCo': 'Machaon', 'nomSc': 'Papilio machaon', 'taille': 85},
            {'num': 462, 'nomCo': 'Petite Tortue', 'nomSc': 'Aglais urticae', 'taille': 50},
            {'num': 463, 'nomCo': 'Robert-le-Diable', 'nomSc': 'Polygonia c-album', 'taille': 25}]
```

Le but de cette partie est de trier la liste des papillons par ordre croissant de taille et de classifier une nouvelle espèce photographiée.

La fonction `tri_collec` renvoie la liste de dictionnaires des papillons triée par ordre croissant de taille.

```
1  def tri_collec(collec):
2      """Renvoie la collection des papillons triées
3      par ordre croissant de leur taille.
4      Paramètre:
5          collec : liste de dictionnaires des papillons
6      Renvoie:
7          liste triée par ordre croissant des tailles
8          des papillons.
9      """
10     for i in range(1, len(collec)):
11         pap = collec[i]
12         j = ...
13         while j > 0 and collec[j - 1]['taille'] > ...:
14             collec[j] = collec[j - 1]
15             j = ...
16         collec[j] = pap
17     return ...
```

10. Recopier et compléter les lignes 12, 13, 15 et 17 de la fonction `tri_collec`.
11. Nommer le tri utilisé.
12. Indiquer, en justifiant, parmi les propositions suivantes quel est le coût en temps de ce tri, dans le pire cas, pour un tableau de taille n : *linéaire*, *quadratique*, *logarithmique* ou *exponentiel*.

L'algorithme des k plus proches voisins est utilisé pour classifier la nouvelle espèce photographiée.

13. Expliquer brièvement le principe de cet algorithme.

Cette nouvelle espèce montre beaucoup de ressemblance avec l'espèce `'Aglais io'` mais diffère par la taille et la couleur des motifs des ailes. Pour vérifier l'hypothèse que la nouvelle espèce est l'espèce `'Aglais io'` comportant une mutation génétique, une recherche naïve d'une séquence caractéristique des papillons `'Aglais io'` est réalisée sur la chaîne d'ADN extraite de la nouvelle espèce. Une chaîne d'ADN est représentée en Python par une chaine de caractères. Cette recherche utilise la fonction `recherche_seq(seq, chaine)` qui renvoie l'indice du premier caractère de `seq` si la séquence `seq` est présente dans la chaîne d'ADN `chaine` et `-1` sinon.

14. Recopier et compléter les lignes 15 et 17 de la fonction `recherche_seq`.

```
1  def recherche_seq(seq, chaine):
2      """Renvoie l'indice du premier caractère de
3         chaine où commence `seq` si la séquence `seq`
4         se trouve dans la chaine de caractères chaine,
5         -1 sinon
6      Paramètres:
7         seq : séquence à rechercher
8         chaine : chaine d'ADN
9      Renvoie:
10        indice du premier caractère de seq dans
11        la chaine, -1 sinon.
12     """
13     for i in range(len(chaine)-len(seq) + 1):
14         j = 0
15         while j < len(seq) and ...:
16             j += 1
17         if ...:
18             return i
19     return -1
```

La fonction `recherche_BMH(seq, chaine)`, donnée ci-dessous, implémente l'algorithme de Boyer-Moore Horspool.

```
1  def dico_lettres(seq):
2      d = {}
3      for i in range(len(seq)-1):
4          d[seq[i]] = i
5      return d
6
7  def recherche_BMH(seq, chaine):
8      decalage = dico_lettres(seq)
9      i = 0
10     n = len(seq)
11     while i <= len(chaine) - n:
12         j = n-1
13         while j >= 0 and chaine[i + j] == seq[j]:
14             j -= 1
15         if j == -1:
16             return i
17         else:
18             if chaine[i + n - 1] in decalage:
19                 i += n - decalage[chaine[i + n-1]] - 1
20             else:
21                 i += n
22     return -1
```

15. Expliquer le principe de cet algorithme et son avantage par rapport à la fonction naïve `recherche_seq`.