Métropole - septembre 2025 - sujet 1 (corrigé)

Exercice 1 (Programmation Python, réseaux, protocoles de routage)

1. Il s'agit du routeur R2 et de son interface G0
2. Seuls les deux derniers octets sont utilisables pour définir les adresses IP disponibles. L'adresse IP à attribuer à la passerelle est donc `172.16.255.254`
3. Seuls les deux derniers bits du quatrième octets sont utilisables pour définir les adresses IP disponibles. On a donc les quatre adresses suivantes : `50.50.50.4` (adresse réseau), `50.50.50.5` (interface G2), `50.50.50.6` et `50.50.50.7` (adresse de diffusion). Ainsi, on peut attribuer à G0 l'adresse `50.50.50.6`
4. On a le tableau suivant :

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

5. Le code suivant convient :

```
t_routage = [((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', 1),
             ((50, 50, 50, 24, 30), 'G2', 1)]
```

6. Cette instruction renvoie `(255, 255, 255, 252)`
7. Puisque $50 = (0011\ 0010)_2$ et $6 = (0000\ 0110)_2$, on a l'opération suivante :

```
adresse IP  :  00110010.00110010.00110010.00000110
masque      :  11111111.11111111.11111111.11111100
            &  -----------------------
               00110010.00110010.00110010.00000100
```

Ainsi, cette adresse IP appartient au réseau `50.50.50.4/30`

8. La fonction suivante convient :

```
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
```

9. La fonction suivante convient :

```
def choose_interface(t_routage, destination_ip):
    for i in range(len(t_routage)):
        network, interface, metrique = t_routage[i]
        if is_in_network(destination_ip, network):
            return interface
    return None # ce return est facultatif
```

10. On a les coûts suivants :
    - Gigabit Ethernet : $10^{10}/10^9 = 10$
    - Fast Eternet : $10^{10}/10^8 = 100$
    - Fibre : $10^{10}/10^{10} = 1$

11. On a le tableau suivant :

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 (Algorithmique, listes et programmation dynamique)
Partie A

1. La plus grande taille est 3 et cette base est issue de la cellule `carte_B[0][1]`
2. Si ce carré est constructible, il ne contient que des valeurs `1`. La somme de ces valeurs est égale au produit de sa largeur par sa hauteur et vaut donc `taille * taille`
3. Le code suivant convient :

```
def est_constructible(carte, i_coin, j_coin, taille):
    s = 0
    for i in range(i_coin, i_coin + taille):
        for j in range(j_coin, j_coin + taille):
            s += carte[i][j]
    return s == taille * taille
```

4. Le code suivant convient :

```
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 = 0
            while j + taille <= n:
                if est_constructible(carte, i, j, taille):
                    return (taille, i, j)
                j = j + 1
            i = i + 1
```

5. Le nombre de carré à tester devient rapidement très grand. Dans le cas de grandes cartes, le temps de calcul est si important qu'une telle recherche exhaustive devient impraticable. On peut en particulier monter que le nombre de carrés à tester est égal à $\frac{n(n+1)(2n+1)}{6}$

Partie B

6. On a la liste auxiliaire suivante :

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

7.
   - Comme `carte_C[0][3] = 0`, cette cellule n'est pas constructible. On a donc `a = aux_C[0][3] = 0`
   - Comme `carte_C[1][0] = 1`, il est possible de construire une base carré de taille 1 issue de cette cellule. On lit par contre `aux_C[1][1] = 0`. Il n'existe donc pas de base carrée issue de la cellule `carte_C[1][1]` ce qui signifie que `carte_C[1][1] = 0`. Il est donc impossible de construire une base de taille supérieure ou égale à 2 issue de `carte_C[1][1]`. On a donc `b = aux_C[1][0] = 1`.
   - Des raisonnements analogues montrent que `c = aux_C[3][3] = 3` et `d = aux_C[4][0] = 2`
8. Le code suivant convient :

```
def calcule_aux(carte):
    n = len(carte)
    aux = [[0 for j in range(n)] for i in range(n)]
    # Remplissage de la dernière ligne et de la dernière colonne
    for k in range(n):
        aux[n - 1][k] = carte[n - 1][k]
        aux[k][n - 1] = carte[k][n - 1]
    # On complète les lignes de bas en haut
    for i in range(n - 2, -1, -1):
        # On complète les colonnes de droite à gauche
        for j in range(n - 2, -1, -1):
            if carte[i][j] == 1:
                aux[i][j] = 1 + min(aux[i + 1][j], aux[i][j + 1], aux[i + 1][j + 1])
    return aux
```

9. Le code suivant convient :

```
def plus_grande_base(carte):
    n = len(carte)
    aux = calcule_aux(carte)
    taille_max = aux[0][0]
    i_max = 0
    j_max = 0
    for i in range(1, n):
        for j in range(1, n):
            if taille_max < aux[i][j]:
                taille_max = aux[i][j]
                i_max = i
                j_max = j
    return taille_max, i_max, j_max
```

10. La fonction `plus_grande_base` complète la liste auxiliaire puis cherche la valeur maximale de celle-ci. Si la carte initiale est de dimension n, il faut donc calculer n^2 valeurs pour compléter la liste auxiliaire et chercher le maximum parmi ces n^2 valeurs. Donc, lorsque la largeur de la carte triple, le nombre de valeurs à calculer est multiplié par $3^2 = 9$. On peut donc estimer le temps d'exécution à $0{,}4 \times 9 = 3{,}6$ secondes

Exercice 3 (Langage SQL, programmation Python et recherche textuelle)

Partie A

1. Une clé primaire est un attribut dont la valeur est unique dans la table
2. L'attribut `habitat` ne peut pas en être une car il existe plusieurs espèces de papillons qui ont le même habitat
3. On obtient `85`
4. La requête suivante convient :

```
UPDATE papillon SET taille = 50 WHERE numCo = 'Petite Tortue'
```

5. La requête suivante convient :

```
SELECT nomCo FROM papillon
WHERE habitat = 'Prairies' AND taille < 55
```

6. On obtient `Papaver rhoeas`
7. La requête suivante convient :

```
SELECT papillon.nomCo, plante.nomCo FROM papillon
JOIN plante ON plante.habitat = papillon.habitat
WHERE taille < 55
```

8. On obtient `Paon-du-jour lilas` (le papillon et la plante qui sont tous deux en Europe)
9. La requête suivante convient :

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

Partie B

10. Le code suivant convient :

```
def tri_collec(collec):
    for i in range(1, len(collec)):
        pap = collec[i]
        j = i
        while j > 0 and collec[j - 1]['taille'] > pap['taille']:
            collec[j] = collec[j - 1]
            j = j - 1
        collec[j] = pap
    return collec
```

11. La fonction `tri_collec` réalise un tri par insertion
12. Le coût est quadratique, c'est-à-dire de l'ordre de $O(n^2)$, car il y a la présence de deux boucles imbriquées, où le corps est en temps constant : pour chaque valeur de `i`, la valeur de `j` est testée.
13. L'algorithme des k plus proches voisins (kNN pour *k Nearest Neighbors* en anglais) est un algorithme d'apprentissage automatique (machine learning). Un nouvel élément peut être catégorisé selon la classe majoritaire de ses k plus proches voisins déterminés grâce à un grand jeu de données
14. Le code suivant convient :

```
def recherche_seq(seq, chaine):
    for i in range(len(chaine)-len(seq) + 1):
        j = 0
        while j < len(seq) and chaine[i + j] == seq[j]:
            j += 1
        if j == len(seq):
            return i
    return -1
```

15. Dans cet algorithme, la séquence `seq` est comparée à la chaîne `chaine` de droite à gauche à partir de sa dernière lettre ; puis un décalage est réalisé selon la comparaison effectuée. Cet algorithme permet de diminuer le nombre de comparaisons à effectuer