

# Asie - juin 2024 - sujet 2 (corrigé)

## Exercice 1 (Listes, dictionnaires et récursivité)

1. Puisque  $10/2 = 5$ , un élément doit apparaître entre 6 et 10 fois pour être absolument majoritaire.

### Partie A : calcul des effectifs de chaque élément sans dictionnaire

2. La fonction suivante convient :

```
def effectif(val, lst):  
    cpt = 0  
    for elem in lst:  
        if elem == val:  
            cpt += 1  
    return cpt
```

3. Puisque chaque élément de la liste doit être testé avec la valeur voulue, on doit effectuer autant de comparaisons qu'il y a d'éléments dans la liste, donc, ici, 9 comparaisons.
4. La fonction suivante convient :

```
def majo_abs(lst):  
    for elem in lst:  
        if effectif(elem, lst) > len(lst)//2:  
            return elem  
    return None
```

5. La première valeur dont on calcule l'effectif est 1 (on fait donc 9 comparaisons d'après la question A)3)). Ensuite, on fait une comparaison supplémentaire pour vérifier si 1 est absolument majoritaire. Comme c'est le cas, on a fait 10 comparaisons en tout.

### Partie B : calcul des effectifs de chaque élément dans un dictionnaire

6. Le code suivant convient :

```
def eff_dico(lst):  
    dico_sortie = {}  
    for elem in lst:  
        if elem in dico_sortie:  
            dico_sortie[elem] += 1  
        else:  
            dico_sortie[elem] = 1  
    return dico_sortie
```

7. La fonction suivante convient :

```
def majo_abs2(lst):  
    eff = eff_dico(lst)  
    for cle, val in eff:  
        if val > len(lst)//2:  
            return cle  
    return None
```

### Partie C : par la méthode « diviser pour régner »

8. S'il n'y a qu'un élément dans la liste `lst`, il est nécessairement absolument majoritaire. Il s'agit donc de `lst[0]`.

9. Si `lst` admet un élément absolument majoritaire, celui-ci apparaît  $m$  fois avec  $m > n/2$ . Dans ce cas, il y a nécessairement un élément dans `lst1` ou `lst2` qui apparaîtra au moins  $m/2$  fois. Comme  $m/2 > n/4$ , cet élément sera absolument majoritaire dans `lst1` ou `lst2`. Par contraposée, si ni `lst1`, ni `lst2` n'admettent un élément absolument majoritaire, alors `lst` n'admet pas d'élément absolument majoritaire.
10. Il suffit de vérifier si son effectif dans la liste `lst` est strictement plus grand que la moitié de la longueur de `lst`.
11. Le code suivant convient :

```
def majo_abs3(lst):  
    n = len(lst)  
    if n == 1:  
        return lst[0]  
    else:  
        lst_g = lst[:n//2]  
        lst_d = lst[n//2:]  
        maj_g = majo_abs3(lst_g)  
        maj_d = majo_abs3(lst_d)  
        if maj_g is not None:  
            eff = effectif(maj_g, lst)  
            if eff > n/2:  
                return maj_g  
        if maj_d is not None:  
            eff = effectif(maj_d, lst)  
            if eff > n/2:  
                return maj_d
```

**Exercice 2 (Programmation Python, POO, piles)**

1. L'expression n'est pas bien parenthésée car la deuxième parenthèse fermante est associée au premier crochet ouvrant au lieu d'une parenthèse ouvrante.

**Partie A**

2. La fonction suivante convient :

```
def compte_ouvrante(txt):  
    cpt = 0  
    for car in txt:  
        if car in '([':  
            cpt += 1  
    return cpt
```

3. La fonction suivante convient :

```
def compte_fermante(txt):  
    cpt = 0  
    for car in txt:  
        if car in ')]':  
            cpt += 1  
    return cpt
```

4. La fonction suivante convient :

```
def bon_compte(txt):  
    return compte_ouvrante(txt) == compte_fermante(txt)
```

5. La chaîne `txt = ') ['` n'est pas bien parenthésée mais l'instruction `bon_compte(txt)` renvoie `True`.

**Partie B**

6. Le code suivant convient :

```
def empiler(self, elt):  
    self.contenu.append(elt)  
  
def depiler(self):  
    if self.est_vide():  
        return "La pile est vide."  
    return self.contenu.pop()
```

7.
  - Il y a 17 caractères hors parenthèses, 3 parenthèses ouvrantes et 3 parenthèses fermantes. De plus, la chaîne est bien parenthésée. Pour chaque parenthèses fermantes, on teste si la pile est vide, puis si la parenthèse fermante coïncide avec la parenthèse ouvrante. A la fin, on reteste si la pile est vide, ce qui rajoute encore une comparaison. Ainsi, il y a  $17 + 3 + 3 \times 3 + 1 = 30$  comparaisons.
  - Le nombre maximum de comparaisons pour une chaîne de taille  $n$  est obtenue lorsque la chaîne est bien parenthésée et ne contient que des paires de parenthèses (donc  $n // 2$  à chaque fois ; en particulier,  $n$  est nécessairement pair). En reprenant le calcul précédent, on en déduit qu'il y a  $n // 2 + 3 \times n // 2 + 1 = 2n + 1$  comparaisons.

8. La fonction suivante convient :

```
def est_bien_parenthesee(txt):  
    p = Pile()  
    for car in txt:  
        if car in '([':  
            p.empiler(car):  
        elif car in ')]':  
            val = p.depiler()  
            if car != val:  
                return False  
    return p.est_vide()
```

**Exercice 3 (Programmation Python, bases de données et SQL)****Partie A**

1. Pour être une clé primaire, les valeurs de l'attribut doivent être unique. Ici, la valeur `Muscle Museum` apparaît deux fois dans l'attribut `titre`, donc cet attribut ne peut pas être une clé primaire pour la table `Chanson`.
2. On obtient le tableau suivant :

|                        |                          |
|------------------------|--------------------------|
| Welcome too the Jungle | Appetite for Destruction |
|------------------------|--------------------------|

3. La requête suivante convient :

```
SELECT titre FROM Chanson WHERE album = 'Showbiz' ORDER BY titre
```

4. La requête suivante convient :

```
INSERT INTO Chanson VALUES (10, 'Megalomania', 'Hullabaloo', 'Muse')
```

5. La requête suivante convient :

```
UPDATE Chanson SET titre = 'Welcome to the Jungle' WHERE id = 7
```

**Partie B**

6. Cela permet d'éviter les redondances d'informations et de faciliter les ajouts, suppressions et modifications de données.
7. L'attribut `id_album` de la table `Chanson` est une clé étrangère pointant sur la clé primaire `id` de la table `Album`. Il permet ainsi de relier (jointure) les deux tables.
8. On a le schéma relationnel suivant :

`Chanson(id: INT, titre: TEXT, #id_album: INT)`

`Album(id: INT, titre: TEXT, année: INT, #id_groupe: INT)`

`Groupe(id: INT, nom: TEXT)`

9. La requête suivante convient :

```
SELECT Album.titre FROM Album
JOIN Chanson ON Album.id = Chanson.id_album
WHERE Chanson.titre = 'Showbiz'
```

10. La requête suivante convient :

```
SELECT Chanson.titre, Album.titre FROM Chanson
JOIN Album ON Chanson.id_album = Album.id
JOIN Groupe ON Album.id_groupe = Groupe.id
WHERE Groupe.nom = 'Muse'
```

11. Cette requête renvoie le nombre d'album du groupe `Muse`.

**Partie C**

12. On a le code suivant :

```
assert ordre_lex("", "a") == True
assert ordre_lex("b", "a") == False
assert ordre_lex("aaa", "aaba") == True
```

13. Le code suivant convient :

```
def ordre_lex(mot1, mot2):  
    if mot1 == "":  
        return True  
    elif mot2 == "":  
        return False  
    else:  
        c1 = mot1[0]  
        c2 = mot2[0]  
        if c1 < c2:  
            return True  
        elif c1 > c2:  
            return False  
        else:  
            return ordre_lex(mot1[1:], mot2[1:])
```

14. La fonction suivante convient :

```
def ordre_lex_iter(mot1, mot2):  
    i = 0  
    while i < len(mot1) and i < len(mot2):  
        if mot1[i] < mot2[i]:  
            return True  
        elif mot1[i] > mot2[i]:  
            return False  
        i = i + 1  
    return len(mot1) <= len(mot2)
```