Asie - juin 2024 - sujet 2

Exercice 1 (Listes, dictionnaires et récursivité - 6 points)
Cet exercice est composé de trois parties indépendantes.

Dans cet exercice, on s'intéresse à des algorithmes pour déterminer, s'il existe, l'élément absolument majoritaire d'une liste.

On dit qu'un élément est *absolument majoritaire* s'il apparaît dans strictement plus de la moitié des emplacements de la liste.

Par exemple, la liste `[1, 4, 1, 6, 1, 7, 2, 1, 1]` admet `1` comme élément absolument majoritaire, car il apparaît cinq fois sur neuf éléments. En revanche, la liste `[1, 4, 6, 1, 7, 2, 1, 1]` n'admet pas d'élément absolument majoritaire, car celui qui est le plus fréquent est `1`, mais il n'apparaît que quatre fois sur huit, ce qui ne fait pas strictement plus que la moitié.

1. Déterminer les effectifs possibles d'un élément absolument majoritaire dans une liste de taille 10.

Partie A : calcul des effectifs de chaque élément sans dictionnaire
On peut déterminer l'éventuel élément absolument majoritaire d'une liste en calculant l'effectif de chacun de ses éléments.

2. Ecrire une fonction `effectif` qui prend en paramètres une valeur `val` et une liste `lst` et qui renvoie le nombre d'apparitions de `val` dans `lst`. Il ne faut pas utiliser la méthode `count`.
3. Déterminer le nombre de comparaisons effectuées par l'appel `effectif(1, [1, 4, 1, 6, 1, 7, 2, 1, 1])`.
4. En utilisant la fonction `effectif` précédente, écrire une fonction `majo_abs1` qui prend en paramètre une liste `lst`, et qui renvoie son élément absolument majoritaire s'il existe et renvoie `None` sinon.
5. Déterminer le nombre de comparaisons effectuées par l'appel à `majo_abs1([1, 4, 1, 6, 1, 7, 2, 1, 1])`.

Partie B : calcul des effectifs de chaque élément dans un dictionnaire
Un autre algorithme consiste à déterminer l'élément absolument majoritaire éventuel d'une liste en calculant l'effectif de tous ses éléments en stockant l'effectif partiel de chaque élément déjà rencontré dans un dictionnaire.

6. Recopier et compléter les lignes 3, 4, 5 et 7 de la fonction `eff_dico` suivante qui prend en paramètre une liste `lst` et qui renvoie un dictionnaire dont les clés sont les éléments de `lst` et les valeurs les effectifs de chacun de ces éléments dans `lst`.

```
1 def eff_dico(lst):
2     dico_sortie = {}
3     for ......... :
4         if ... in dico_sortie:
5             ...
6         else:
7             ...
8     return dico_sortie
```

7. En utilisant la fonction `eff_dico` précédente, écrire une fonction `majo_abs2` qui prend en paramètre une liste `lst`, et qui renvoie son élément absolument majoritaire s'il existe et renvoie `None` sinon.

Partie C : par la méthode « diviser pour régner »
Un dernier algorithme consiste à partager la liste en deux listes. Ensuite, il s'agit de déterminer les éventuels éléments absolument majoritaires de chacune des deux listes. Il suffit ensuite de combiner les résultats sur les deux listes afin d'obtenir, s'il existe, l'élément majoritaire de la liste initiale.

Les questions suivantes vont permettre de concevoir précisément l'algorithme.

On considère `lst` une liste de taille `n`.

8. Déterminer l'élément absolument majoritaire de `lst` si `n = 1`. *C'est le cas de base.*

On suppose que l'on a partagé `lst` en deux listes :

- `lst1 = lst[:n//2]` (`lst1` contient les `n//2` premiers éléments de `lst`);
- `lst2 = lst[n//2:]` (`lst2` contient les autres éléments de `lst`).

9. Si, ni `lst1` ni `lst2` n'admet d'élément absolument majoritaire, expliquer pourquoi `lst` n'admet pas d'élément absolument majoritaire.
10. Si `lst1` admet un élément absolument majoritaire `maj1`, donner un algorithme pour vérifier si `maj1` est l'élément absolument majoritaire de `lst`.
11. Recopier et compléter les lignes 4, 11, 13, 15 et 17 pour la fonction récursive `majo_abs3` qui implémente l'algorithme précédent. Vous pourrez utiliser la fonction `effectif` de la question 2.

```
1  def majo_abs3(lst):
2      n = len(lst)
3      if n == 1:
4          return ...
5      else:
6          lst_g = lst[:n//2]
7          lst_d = lst[n//2:]
8          maj_g = majo_abs3(lst_g)
9          maj_d = majo_abs3(lst_d)
10         if maj_g is not None:
11             eff = ......
12             if eff > n/2:
13                 return ...
14         if maj_d is not None:
15             eff = ......
16             if eff > n/2:
17                 return ...
```

Exercice 2 (Programmation Python, POO, piles - 6 points)
Cet exercice est composé de deux parties indépendantes.

Dans cet exercice, on appelle parenthèses les couples de caractères `()`, {} et `[]`. Pour chaque couple de parenthèses, la première parenthèse est appelée la parenthèse ouvrante du couple et la seconde est appelée la parenthèse fermante du couple.

On dit qu'une expression (chaîne de caractères) est bien parenthésée si

- chaque parenthèse ouvrante correspond à une parenthèse fermante de même type ;
- les expressions comprises entre parenthèses sont des expressions bien parenthésées.

Par exemple, l'expression `'tab[2*(i + 4)] - tab[3]'` est bien parenthésée.
En revanche, l'expression `'tab[2*(i + 4] - tab[3)'` n'est pas bien parenthésée, car la première parenthèse fermante `]` devrait correspondre à la dernière parenthèse ouvrante `(`.

1. Déterminer si l'expression `'[2*(i+1)-3) for i in range(3, 10)]'` est bien parenthésée. Justifier votre réponse.

Partie A
On peut observer que, si les parenthèses vont par couple, une expression bien parenthésée contient autant de parenthèses ouvrantes que de parenthèses fermantes.

On se propose d'écrire une fonction qui vérifie si une chaîne de caractères est bien parenthésée.

2. Ecrire une fonction `compte_ouvrante` qui prend en paramètre une chaîne de caractères `txt` et qui renvoie le nombre de parenthèses ouvrantes qu'il contient.
3. Ecrire une fonction `compte_fermante` qui prend en paramètre une chaîne de caractères `txt` et qui renvoie le nombre de parenthèses fermantes qu'il contient.
4. En utilisant les deux fonctions précédentes, écrire une fonction `bon_compte` qui prend en paramètre une chaîne de caractères `txt` et qui renvoie `True` si `txt` a autant de parenthèses ouvrantes que parenthèses fermantes et `False` sinon.
5. Donner un exemple de chaîne de caractères pour laquelle `bon_compte` renvoie `True` alors qu'elle n'est pas bien parenthésée.

Partie B
Comme l'algorithme précédent n'est pas suffisant, on se propose d'implémenter un algorithme utilisant une structure linéaire de pile.

On se propose d'écrire une classe `Pile` qui implémente la structure de pile.

```
1  class Pile:
2      def __init__(self):
3          self.contenu = []
4
5      def est_vide(self):
6          return len(self.contenu) == 0
7
8      def empiler(self, elt):
9          ...
10
11     def depiler(self):
12         if self.est_vide():
13             return "La pile est vide."
14         return ...
```

6. Compléter les lignes 9 et 14 du code précédent pour que la méthode `empiler` permette d'empiler un élément elt dans une pile et que la méthode `depiler` permette de dépiler une pile en renvoyant l'élément dépilé. *Vous n'écrirez que le code des deux méthodes.*

Un algorithme permettant de vérifier si une expression est bien parenthésée consiste à

- ⋆ créer une pile vide `p` ;
- ⋆ parcourir l'expression en testant chaque caractère :
  - si c'est une parenthèse ouvrante, on l'empile dans `p` ;
  - si c'est une parenthèse fermante, on dépile `p` ;
    - — si les deux caractères correspondent à un couple de parenthèses, on continue le parcours,

  - sinon l'expression n'est pas bien parenthésée ;
- sinon on continue le parcours.

Si l'expression a été entièrement parcourue, on teste la pile ; l'expression est bien parenthésée si la pile est vide.

7. Déterminer le nombre de comparaisons effectuées si on applique l'algorithme précédent à la chaine de caractères

```
'tab[2*(i + 4)] - tab[3]'
```

   En déduire le nombre maximum de comparaisons effectuées si on applique l'algorithme précédent à une chaine de caractères de taille `n` (attention aux comparaisons de la classe `Pile`).

8. Ecrire une fonction `est_bien_parenthesee` qui prend en paramètre une chaîne de caractères et qui implémente l'algorithme précédent.

Exercice 3 (Programmation Python, bases de données et SQL - 8 points)
L'énoncé de cet exercice utilise des mots-clés du langage SQL suivants : `SELECT`, `FROM`, `WHERE`, `JOIN ... ON`, `UPDATE ... SET`, `INSERT INTO ... VALUES ...`, `COUNT`, `ORDER BY`.

La clause `ORDER BY` suivie d'un attribut permet de trier les résultats par ordre croissant de l'attribut précisé. `SELECT COUNT(*)` renvoie le nombre de lignes d'une requête.

Amélie souhaite organiser sa collection de CD. Elle a commencé par enregistrer toutes les informations sur un fichier CSV mais elle trouve que la recherche d'informations est longue et fastidieuse. Elle repense à son cours sur les bases de données et elle se dit qu'elle doit pouvoir utiliser une base de données relationnelle pour organiser sa collection.

Partie A
Dans cette partie on utilise une seule table.

Voici un extrait de la table `Chanson`.

Chanson			
id	titre	album	groupe
1	Sunburn	Showbiz	Muse
2	Muscle Museum	Showbiz	Muse
3	Showbiz	Showbiz	Muse
4	New Born	Origin of Symmetry	Muse
5	Sing for Absolution	Absolution	Muse
6	Hysteria	Absolution	Muse
7	Welcome too the Jungle	Appetite for Destruction	Guns NRoses
8	Muscle Museum	Hullabaloo	Muse
9	Showbiz	Hullabaloo	Muse

1. L'attribut `titre` peut-il être une clé primaire pour la table `Chanson` ? Justifier.
2. Donner le résultat de la requête suivante :

```
SELECT titre, album FROM Chanson WHERE groupe = 'Guns NRoses';
```

3. Ecrire une requête SQL permettant d'obtenir tous les titres des chansons de l'album Showbiz dans l'ordre croissant.
4. Ecrire une requête SQL permettant d'ajouter la chanson dont le titre est Megalomania de l'album Hullabaloo du groupe Muse.

Amélie a remarqué une faute de frappe dans la chanson Welcome too the Jungle qui s'écrit normalement Welcome to the Jungle.

5. Ecrire une requête SQL permettant de corriger cette erreur.

Partie B
Dans cette partie on utilise trois tables.

Voici des extraits des trois tables `Chanson`, `Album` et `Groupe`.

Chanson		
id	titre	id_album
1	Sunburn	1
2	Muscle Museum	1
3	Showbiz	1
4	New Born	2
5	Sing for Absolution	4
6	Hysteria	4
7	Welcome to the Jungle	5
8	Muscle Museum	3
9	Showbiz	3

Album			
id	titre	année	id_groupe
1	Showbiz	1999	1
2	Origin of Symmetry	2001	1
3	Hullabaloo	2002	1
4	Absolution	2003	1
5	Appetite for Destruction	1987	2

Groupe	
id	nom
1	Muse
2	Guns NRoses

6. Expliquer l'intérêt d'utiliser trois tables `Chanson`, `Album` et `Groupe` au lieu de regrouper toutes les informations dans une seule table.
7. Expliquer le rôle de l'attribut `id_album` de la table `Chanson`.
8. Proposer alors un schéma relationnel pour cette version de la base de données. On pensera à bien spécifier les clés primaires en les soulignant et les clés étrangères en les faisant précéder par le symbole `#`.
9. Ecrire une requête SQL permettant d'obtenir tous les noms des albums contenant la chanson Showbiz.
10. Ecrire une requête SQL permettant d'obtenir tous les titres avec le nom de l'album des chansons du groupe Muse.
11. Décrire par une phrase ce qu'effectue la requête SQL suivante :

```
SELECT COUNT(*) AS tot
FROM Album AS a
JOIN Groupe AS g
ON a.id_groupe = g.i
WHERE g.nom = 'Muse';
```

Partie C

Dans cette partie, on utilise Python.

Amélie a remarqué que son professeur ne parle jamais d'ordre alphabétique mais d'ordre lexicographique lorsqu'il fait une requête avec `ORDER BY`. Elle a compris qu'il s'agissait de l'ordre du dictionnaire mais elle se demande comment elle pourrait elle-même écrire une fonction `ordre_lex(mot1, mot2)` de comparaison entre deux chaînes de caractères en utilisant l'ordre lexicographique. La fonction `ordre_lex(mot1, mot2)` prend en arguments deux chaînes de caractères et renvoie un booléen. Une rapide recherche lui permet de trouver le résultat suivant :

Lorsque l'on compare deux chaînes de caractères suivant l'ordre lexicographique, on commence par comparer les deux premiers caractères de chacune des deux chaînes, puis en cas d'égalité on s'intéresse au second, et ainsi de suite. Le classement est donc le même que celui d'un dictionnaire. Si lors de ce procédé on dépasse la longueur d'une seule des deux chaînes, elle est considérée plus petite que l'autre. Lorsqu'on dépasse la longueur des deux chaînes au même moment, elles sont nécessairement égales.

Amélie commence par écrire quelques assertions que sa fonction devra vérifier.

12. Compléter les assertions suivantes :

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

On suppose que les chaînes de caractères `mot1` et `mot2` ne sont composées que des lettres de l'alphabet, en minuscule, et la comparaison entre deux lettres peut se faire avec les opérateurs classiques == et <. Par exemple :

```
>>> "" < "a"
True
>>> "b" == "a"
False
```

Enfin, le slice `mot1[1:]` renvoie la chaîne de caractères de `mot1` privée de son premier caractère. Par exemple :

```
>>> mot1 = "abcde"
>>> mot2 = mot1[1:]
>>> mot2 4 "bcde"
```

13. Recopier et compléter la fonction récursive `ordre_lex` ci-dessous qui prend pour paramètre deux chaînes de caractères `mot1` et `mot2` et qui renvoie `True` si `mot1` précède `mot2` dans l'ordre lexicographique.

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

14. Proposer une version itérative de la fonction `ordre_lex`.