Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI T^le Année **2024-25**

Devoir surveillé n°6 – Correction

Nom et prénom :

Exercice 1 : (14pt) *Cet exercice porte sur la programmation Python (listes, dictionnaires) et la méthode "diviser pour régner".*
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 5 fois sur 9 éléments.
Par ailleurs, 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 4 fois sur 8, ce qui ne fait pas plus que la moitié.

1) Déterminer les effectifs possibles d'un élément absolument majoritaire dans une liste de taille 10.
   Solution : Il faut que l'élément apparaîsse 6, 7, 8, 9 ou 10 fois.

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) Écrire 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`.

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

3) Déterminer le nombre de comparaisons effectuées par l'appel `effectif(1, [1, 4, 6, 1, 7, 2, 1, 1])`.
   Solution : On va comparer 1 avec tous les éléments, donc 8.

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.

```
def majo_abs1(lst):
    for v in lst:
        c = effectif(v, lst)
        if c > len(lst)/2:
            return c
    return None
```

5) Déterminer le nombre de comparaisons effectuées avec des éléments de la liste par l'appel à `majo_abs1([1, 4, 6, 1, 7, 2, 1, 1])`.
Solution : Comme il n'y a pas d'élément absolument majoritaire, on va utiliser `effectif` pour chaque élément, ce qui va faire 8 comparaisons à chaque fois. Cela fait un total de 64 comparaisons.

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) 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 v in lst:
4         if v in dico_sortie:
5             dico_sortie[v] += 1
6         else:
7             dico_sortie[v] = 1
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.
Solution :

```
def majo_abs2(lst):
    dico = eff_dico(lst)
    for v in dico:
        if dico[v] > len(lst)/2:
            return v
    return None
```

Nom et prénom :

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.
Solution : C'est `lst[0]`.

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.
Solution : S'il n'y a pas d'élément absolument majoritaire dans aucune des sous-listes, cela veut dire qu'un élément ne peut pas apparaître plus que la moitié de la taille de chacune des deux sous-listes. Au total, cela fait au plus la moitié de la liste totale. Il ne peut donc pas y avoir d'élément absolument majoritaire dans la liste.

10) Si `lst1` admet un élément absolument majoritaire `maj1`, donner un algorithme **en français** pour vérifier si `maj1` est l'élément absolument majoritaire de `lst`.
On regarde si les effectifs cumulés de `maj1` dans chacune des sous-liste. Si cela dépasse la moitié de la taille de la liste, c'est un élément absolument majoritaire. Si ce n'est pas le cas, on regarde avec un éventuel élément absolument majoritaire de la deuxième sous-liste. Sinon, il n'y a pas d'élément absolument majoritaire.

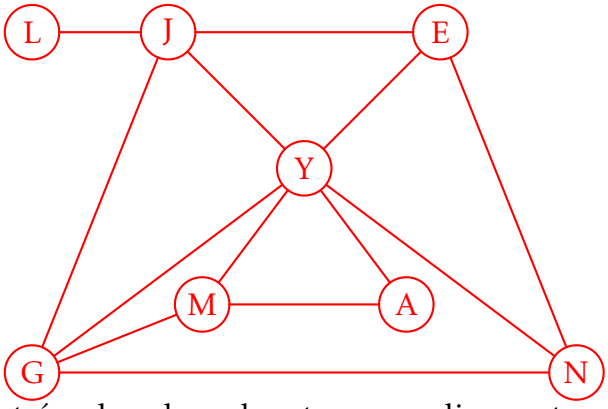
11) Compléter les lignes 4, 11, 13, 15 et 17 pour la fonction récursive `majo_abs3` qui implémente l'algorithme précédent. Vous pourrez utiliser la fonction `effectif` de la question 2.

```
1  def majo_abs3(lst):
2      n = len(lst)
3      if n == 1:
4          return lst[0]
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 = effectif(maj_g, lst_g) + effectif(maj_g, lst_d)
12             if eff > n/2:
13                 return maj_g
14         if maj_d is not None:
15             eff = effectif(maj_d, lst_g) + effectif(maj_d, lst_d)
16             if eff > n/2:
17                 return maj_d
```

Exercice 2 : (12pt) *Cet exercice porte sur les graphes.*
Dans cet exercice, on modélise un groupe de personnes à l'aide d'un graphe. Le groupe est constitué de huit personnes (Anas, Emma, Gabriel, Jade, Lou, Milo, Nina et Yanis) qui possèdent entre elles les relations suivantes :

- Gabriel est ami avec Jade, Yanis, Nina et Milo ;
- Jade est amie avec Gabriel, Yanis, Emma et Lou ;
- Yanis est ami avec Gabriel, Jade, Emma, Nina, Milo et Anas ;
- Emma est amie avec Jade, Yanis et Nina ;
- Nina est amie avec Gabriel, Yanis et Emma ;
- Milo est ami avec Gabriel, Yanis et Anas ;
- Anas est ami avec Yanis et Milo ;
- Lou est amie avec Jade.

Partie A : Matrice d'adjacence
On choisit de représenter cette situation par un graphe dont les sommets sont les personnes et les arêtes représentent les liens d'amitié.



1) Dessiner ci-contre ce graphe en représentant chaque personne par la première lettre de son prénom entourée d'un cercle et où un lien d'amitié est représenté par un trait entre deux personnes.

Une matrice d'adjacence est un tableau à deux entrées dans lequel on trouve en lignes et en colonnes les sommets du graphe.
Un lien d'amitié sera représenté par la valeur 1 à l'intersection de la ligne et de la colonne qui représentent les deux amis alors que l'absence de lien d'amitié sera représentée par un `0`.

2) Compléter l'implémentation de la déclaration de la matrice d'adjacence du graphe.

```
# sommets :      G, J, Y, E, N, M, A, L
matrice_adj = [[0, 1, 1, 0, 1, 1, 0, 0], # G
               [1, 0, 1, 1, 0, 0, 0, 1], # J
               [1, 1, 0, 1, 1, 1, 1, 0], # Y
               [0, 1, 1, 0, 1, 0, 0, 0], # E
               [1, 0, 1, 1, 0, 0, 0, 0], # N
               [1, 0, 1, 0, 0, 0, 1, 0], # M
               [0, 0, 1, 0, 0, 1, 0, 0], # A
               [0, 1, 0, 0, 0, 0, 0, 0]] # L
```

On dispose de la liste suivante qui identifie les sommets du graphe :

```
sommets = ['G', 'J', 'Y', 'E', 'N', 'M', 'A', 'L']
```

On dispose d'une fonction `position(l, s)` qui prend en paramètres une liste de sommets `l` et un nom de sommet `s` et qui renvoie la position du sommet `s` dans la liste `l` s'il est présent et None sinon.

3) Indiquer quel seront les retours de l'exécution des instructions suivantes :

```
>>> position(sommets, 'G')
0
>>> position(sommets, 'Z')
None  # en fait None ne s'affiche pas
```

Nom et prénom :

4) Compléter le code de la fonction `nb_amis(L, m, s)` qui prend en paramètres une liste de noms de sommets `L`, une matrice d'adjacence `m` d'un graphe et un nom de sommet `s` et qui renvoie le nombre d'amis du sommet `s` s'il est présent dans `L` et **`None`** sinon.

```
def nb_amis(L, m, s):
    pos_s = position(L, s)
    if pos_s == None:
        return None
    amis = 0
    for i in range(len(m)):
        amis += m[pos_s][i]
    return amis
```

5) Indiquer quel est le retour de l'exécution de la commande suivante :

```
>>> nb_amis(sommets, matrice_adj, 'G')
4
```

Partie B : Dictionnaire de listes d'adjacence

6) Dans un dictionnaire Python `{c: v}`, indiquer ce que représentent `c` et `v`.
Solution : `c` est la clé et `v` la valeur associée à la clé.

On appelle `graphe` le dictionnaire de listes d'adjacence associé au graphe des amis.
On rappelle que Gabriel est ami avec Jade, Yanis, Nina et Milo.

7) Compléter le dictionnaire de listes d'adjacence `graphe` pour qu'il modélise complètement le groupe d'amis.

```
graphe = {'G': ['J', 'Y', 'N', 'M'],
          'J': ['G', 'Y', 'E', 'L'],
          'Y': ['G', 'J', 'E', 'N', 'M', 'A'],
          'E': ['J', 'Y', 'N'],
          'N': ['G', 'Y', 'E'],
          'M', ['G', 'Y', 'A'],
          'A': ['Y', 'M'],
          'L': ['J']
         }
```

8) Écrire, sur la page suivante, le code de la fonction `nb_amis2(d, s)` qui prend en paramètres un dictionnaire d'adjacence `d` et un nom de sommet `s` et qui renvoie le nombre d'amis du nom de sommet `s`. On suppose que `s` est bien dans `d`. Par exemple :

```
>>> nb_amis2(graphe, 'L')
1
```

Solution :

```
def nb_amis2(d, s):
    return len(d[s])
```

Milo s'est fâché avec Gabriel et Yanis tandis qu'Anas s'est fâché avec Yanis. Le dictionnaire d'adjacence du graphe qui modélise cette nouvelle situation est donné ci-dessous

```
graphe = {'G' : ['J', 'N'],
          'J' : ['G', 'Y', 'E', 'L'],
          'Y' : ['J', 'E', 'N'],
          'E' : ['J', 'Y', 'N'],
          'N' : ['G', 'Y', 'E'],
          'M' : ['A'],
          'A' : ['M'],
          'L' : ['J']
         }
```

Pour établir la liste du cercle d'amis d'un sommet, on utilise un parcours en profondeur du graphe à partir de ce sommet. On appelle cercle d'amis de *Nom* toute personne atteignable dans le graphe à partir de *Nom*.

9) Donner la liste du cercle d'amis de Lou.

Solution : Le cercle d'amis de Lou est composé de Jade, Gabriel, Nina, Yanis et Emma.

Un algorithme possible de parcours en profondeur de graphe est donné ci-dessous :

```
visités = liste vide des sommets déjà visités

fonction parcours_en_profondeur(d, s)
    ajouter s à la liste visités
    pour tous les sommets voisins v de s :
        si v n'est pas dans la liste visités :
            parcours_en_profondeur(d, v)
    retourner la liste visités
```

10) Compléter le code de la fonction `parcours_en_profondeur(d, s)` qui prend en paramètres un dictionnaire d'adjacence `d` et un sommet `s` et qui renvoie la liste des sommets issue du parcours en profondeur du graphe modélisé par `d` à partir du sommet `s`.

```
def parcours_en_profondeur(d, s, visites=[]):
    visites.append(s)
    for v in d[s]:
        if v not in visites:
            parcours_en_profondeur(d, v)
    return visites
```