Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI T^le Année **2025-26**

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 : (16pt) *Dans cet exercice il est question de tableaux, de dictionnaires, de recherche de chemins dans un graphe, de piles, de files et de POO.*

On considère une liste de mots de quatre lettres. Par exemple `gars`, `grue`, `mars`, `mors`, `ours`, `purs`, `durs`...

Deux mots sont voisins s'ils ne diffèrent que d'une seule lettre sans se soucier de l'ordre des lettres dans les mots. Par exemple :

- `mars` et `mors` sont voisins (on change le `a` en `o`) ;
- `mors` et `ours` sont voisins (on change le `m` en `u`) ;
- `grue` et `ours` ne sont pas voisins (il faudrait changer `g` en `o` et `e` en `s`).

L'exercice consiste à partir d'un mot de départ (par exemple `mars`) pour atteindre un mot de destination (par exemple `ours`) en passant de voisins en voisins et en empruntant le moins de voisins possible.

Les mots utilisés, du mot de départ au mot de destination, forment alors le chemin emprunté. Par exemple `mars`, `mors`, `ours` est le chemin qui relie le mot `mars` au mot `ours`.

On considère qu'il est toujours possible de trouver un tel chemin.

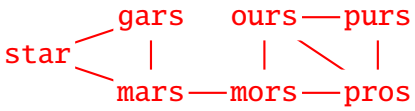
Partie A

Pour résoudre notre problème nous aurons besoin, entre autres, d'une structure de pile, d'une structure de file et d'un graphe.

1) Décrire le fonctionnement d'une Pile.
   Solution : Dans une pile, le dernier élément ajouté est mis au sommet de la pile. Lorsqu'on enlève un élément, on prend celui au sommet. C'est un mode "dernier arrivé, premier sorti".
2) Décrire le fonctionnement d'une File.
   Solution : Dans une file, le dernier élément ajouté est mis à la fin de la file. Lorsqu'on enlève un élément, on prend celui au début de la file. C'est un mode "premier arrivé, premier sorti".

On va utiliser un graphe dont les sommets sont les mots et dont les arêtes relient deux sommets si les mots sont voisins.

3) Expliquer pourquoi un graphe non orienté est adapté à la situation.
   Solution : La relation de voisinage est symétrique. Donc si on relie A à B, il faut aussi relier B à A. Un graphe orienté est inutile et il faut utiliser un graphe non orienté.
4) Dessiner le graphe si la liste de mots est : `gars`, `mars`, `mors`, `ours`, `purs`, `star` et `pros`.
   Solution :



Partie B

La distance entre deux mots est le nombre minimum de lettres qu'il faut modifier pour passer de l'un à l'autre. Par exemple, la distance entre mars et mors est 1 (on change le `a` en `o`) tandis que la distance entre grue et ours est 2 (on change le `g` en `o` et le `e` en `s`).

Deux mots sont voisins si et seulement si la distance entre eux vaut 1.

Dans toute cette partie, on dispose d'une variable globale `TAB_MOTS`, un tableau (type **`list`** en Python) dont les éléments sont des chaines de quatre caractères qui correspondent à des mots.

La fonction `chaine_vers_tab(mot)` prend en paramètre une chaine de quatre caractères et renvoie un tableau (type **`list`** en Python) dont les éléments sont les caractères de cette chaine.

Ainsi l'appel de la fonction `chaine_vers_tab('ours')` renvoie `['o', 'u', 'r', 's']`.

5) Compléter le code de la fonction chaine_vers_tab.

```
def chaine_vers_tab(mot):
    tab_lettres = ...
    for ............ in ................. :
        tab_lettres....
    return tab_lettres
```

Solution :

```
def chaine_vers_tab(mot):
    tab_lettres = []
    for lettre in mot:
        tab_lettres.append(lettre)
    return tab_lettres
```

6) Expliquer pourquoi la fonction ci-dessous renvoie effectivement la distance entre les deux mots mot1 et mot2, deux chaines de quatre caractères.

```
def distance(mot1, mot2):
    tab = list(mot1)
    for lettre in mot2:
        if lettre in tab:
            tab.remove(lettre)
    return len(tab)
```

Solution : À la fin de l'appel, tab ne contient que les lettres de mot1 qui ne sont pas dans mot2. Cela correspond donc au nombre de lettres à changer et donc à la distance.

7) Recopier et compléter la fonction renvoie_voisins(mot) qui renvoie un tableau dont les éléments sont les mots de TAB_MOTS qui sont voisins de mot, une chaine de quatre caractères, passé en paramètre.

```
def renvoie_voisins(mot):
    tab_voisins = ...
    for voisin_possible in ............ :
        if ........................... :
            tab_voisins....
    return ...
```

Solution :

```
def renvoie_voisins(mot):
    tab_voisins = []
    for voisin_possible in TAB_MOTS:
        if distance(mot, voisin_possible) == 1:
            tab_voisins.append(voisin_possible)
    return tab_voisins
```

Partie C

Il nous faut maintenant chercher le chemin le plus court entre deux mots.
On dispose pour cela d'une classe File et d'une classe Pile.
Voici les méthodes de la classe File dont nous aurons besoin :

- est_vide(self) qui renvoie True si l'instance de File est vide et False sinon ;
- defiler(self) qui, si l'instance de File n'est pas vide, lui enlève la tête et la renvoie ;
- enfiler(self, element) qui enfile element dans l'instance de File.

Voici les méthodes de la classe Pile dont nous aurons besoin :

- `est_vide(self)` qui renvoie `True` si l'instance de `Pile` est vide et `False` sinon;
- `depiler(self)` qui, si l'instance de `Pile` n'est pas vide, lui enlève le sommet et le renvoie.
- `empiler(self, element)` qui empile element dans l'instance de `Pile`.

La fonction `dic_parent(mot_depart, mot_final)` ci-dessous renvoie le chemin entre `mot_depart`, qui est le mot de départ, et `mot_final`, qui est celui qu'on cherche à atteindre, sous la forme d'un dictionnaire `{sommet parcouru: sommet précédent}` :

```
1  def dic_parent(mot_depart, mot_final):
2      file_voisins = File()
3      parent = {mot_depart: None}
4      mot = mot_depart
5      file_voisins.enfiler(mot)
6      while not file_voisins.est_vide() and not mot == mot_final:
7          mot = file_voisins.defiler()
8          for voisin in renvoie_voisins(mot):
9              if not voisin in parent:
10                 parent[voisin] = mot
11                 file_voisins.enfiler(voisin)
12     return parent
```

On donne les résultats ci-dessous qui correspondent au graphe de la partie A :

```
>>> renvoie_voisins('mars')
['gars', 'mors', 'star']
>>> renvoie_voisins('gars')
['mars', 'star']
>>> renvoie_voisins('mors')
['mars', 'ours', 'pros']
>>> renvoie_voisins('ours')
['mors', 'purs', 'pros']
```

Voici le début de l'exécution pas à pas de la fonction `dic_parent` en prenant `'mars'` pour mot de départ et `'ours'` pour mot final :

- Avant le début de la boucle :
  - `parent = {'mars': None}`
  - `file_voisins` contient uniquement `'mars'`
- Premier tour de boucle :
  - on défile `'mars'`
  - les voisins de `'mars'` sont `'gars'`, `'mors'` et `'star'`. Ils ne sont pas encore présents dans le dictionnaire `parent` donc ils ont tous les trois pour parent `'mars'` et on les enfile dans cet ordre dans `file_voisins`. Ainsi on obtient :
  - `parent = {'mars': None, 'gars': 'mars', 'mors': 'mars', 'star': 'mars'}`
  - `file_voisins` contient, de la tête à la queue, `'gars'` puis `'mors'` puis `'star'`.
- Deuxième tour de boucle :
  - on défile `'gars'`
  - les voisins de `'gars'` sont `'mars'` et `'star'` qui sont déjà dans `parent`. Ainsi on obtient :
  - `parent` n'est pas modifié
  - `file_voisins` contient `'mors'` puis `'star'`.

8) Poursuivre le déroulement de la fonction pas à pas sur le modèle ci-dessus en détaillant chaque tour de boucle jusqu'à l'issue de la fonction. On pourra n'indiquer que la valeur défilée, les voisins et les valeurs de `parent` et de `file_voisins`, en précisant bien le numéro du tour de boucle.

Solution :

- Troisième tour de boucle :
  - on défile `'mors'`
  - les voisins de `'mors'` sont `'mars'`, `'ours'` et `'pros'`. Les deux derniers ne sont pas encore dans `parent`. Ainsi on obtient :
  - `parent = {'mars': None, 'gars': 'mars', 'mors': 'mars', 'star': 'mars', 'ours': 'mors', 'pros': 'mors'}`
  - `file_voisins` contient `'star'` puis `'ours'` puis `'pros'`.
- Quatrième tour de boucle :
  - on défile `'star'`
  - les voisins de `'star'` sont `'mars'` et `'gars'` qui sont déjà dans `parent`. Ainsi on obtient :
  - `parent` n'est pas modifié
  - `file_voisins` contient `'ours'` puis `'pros'`.
- Cinquième tour de boucle :
  - on défile `'ours'`
  - on a trouvé le mot final. On renvoie :
  - `parent = {'mars': None, 'gars': 'mars', 'mors': 'mars', 'star': 'mars', 'ours': 'mors', 'pros': 'mors'}`

Dans cette question on souhaite construire une instance de `Pile` dans laquelle on va empiler chaque mot du chemin en remontant les mots, depuis le mot final jusqu'au mot de départ, grâce au dictionnaire renvoyé par la fonction `dic_parent`.
La fonction `renvoie_pile(parent, mot_final)` prend en paramètres :

- `parent`, un dictionnaire obtenu grâce à la fonction `dic_parent` ;
- `mot_final`, le mot final.

Elle renvoie une pile dont le premier élément empilé est le mot final et dont le sommet est le mot de départ.

9) Compléter la fonction renvoie_pile.

```
1 def renvoie_pile(parent, mot_final):
2     ma_pile = Pile()
3     mot = mot_final
4     while mot != ................. :
5         ma_pile....
6         mot = ...
7     return ma_pile
```

Solution :

```
def renvoie_pile(parent, mot_final):
    ma_pile = Pile()
    mot = mot_final
    while mot != None:
        ma_pile.empiler(mot)
        mot = parent[mot]
    return ma_pile
```

La fonction construit_chemin a pour paramètre une pile de mots obtenue grâce à la fonction renvoie_pile et renvoie un tableau dont les éléments sont les mots du chemin recherché dans le bon ordre.

10) Compléter la fonction construit_chemin :

```
1 def construit_chemin(ma_pile):
2     tab = ...
3     while ........................... :
4         mot = ...
5         tab....
6     return tab
```

Solution :

```
def construit_chemin(ma_pile):
    tab = []
    while not ma_pile.est_vide():
        mot = ma_pile.depiler()
        tab.append(mot)
    return tab
```

11) Coder, en utilisant les fonctions précédentes, la fonction chercher_chemin de paramètres mot_depart, le mot de départ, et mot_final, le mot à atteindre, et qui renvoie un tableau dont les éléments sont les mots qui constituent le chemin du mot de départ jusqu'au mot final.

Solution :

```
def chercher_chemin(mot_depart, mot_final):
    parent = dico_parent(mot_depart, mot_final)
    pile = renvoie_pile(paren, mot_final)
    return construit_chemin(pile)
```