Exercice 3 (8 points)

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.
2. Décrire le fonctionnement d'une File.

   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.
4. Dessiner le graphe si la liste de mots est : *gars, mars, mors, ours* et *purs*.

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.

5. Recopier et compléter la fonction `chaine_vers_tab(mot)` ci-dessous. Cette fonction 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']`.

```
1. def chaine_vers_tab(mot):
2.     tab_lettres = ...
3.     for ... in ... :
4.         tab_lettres....
5.     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.

```
1. def distance(mot1, mot2):
2.     tab = chaine_vers_tab(mot1)
3.     for lettre in mot2:
4.         if lettre in tab:
5.             tab.remove(lettre)
6.     return len(tab)
```

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.

```
1. def renvoie_voisins(mot):
2.     tab_voisins = ...
3.     for voisin_possible in ... :
4.         if ... :
5.             tab_voisins....
6.     return ...
```

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

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’* et *‘mors’* . Ils ne sont pas encore présents dans le dictionnaire *parent* donc ils ont tous les deux pour parent *‘mars’* et on les enfile dans cet ordre dans *file_voisins*. Ainsi on obtient :
- *parent* = {*’mars’* : *None*, *‘gars’*: *’mars’*, *‘mors’*: *‘mars’*}
- file_voisins contient, de la tête à la queue, *‘gars’* puis *‘mors’*.

– Deuxième tour de boucle :

- on défile *‘gars’*
- le seul voisin de _‘gars_’ est ‘*mars*’. ‘*mars*’ est déjà dans *parent*. Ainsi on obtient :
- *parent* = {*’mars’* : *None*, *‘gars’*: *’mars’*, *‘mors’*: *‘mars’*}
- *file_voisins* contient uniquement *‘mors’*

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.

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`.

9. Recopier et compléter la fonction `renvoie_pile(parent, mot_final)`.

Cette fonction 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.

```
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
```

10. Recopier et compléter la fonction `construit_chemin` :

- son paramètre est une pile de mots obtenue grâce à la fonction `renvoie_pile` ;
- elle renvoie un tableau dont les éléments sont les mots du chemin recherché dans le bon ordre.

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