Exercice 3 (8 points)

Cet exercice porte sur l'algorithmique des tableaux, la gestion de bugs, les listes, les piles et la programmation orientée objet.

Le but de cet exercice est d'implémenter un algorithme de pseudo-tri, appelé le tri dictatorial.

L'exercice est constitué de trois parties indépendantes.

Pour chaque question, on peut considérer acquis les résultats et les fonctions demandés dans les questions précédentes, même sans les avoir traitées.

Le pseudo-tri dictatorial d'une série d'entiers suit le principe suivant :

- s'il n'y a aucun ou un seul élément, la série est considérée comme triée et n'est donc pas modifiée ;
- sinon :
  - on conserve le premier élément de la série ;
  - pour chaque élément de la série à partir du deuxième :
    - si l'élément est plus petit que le dernier élément conservé alors on l'élimine ;
    - sinon on le conserve.

Par exemple, si on considère la série 2, 3, 1, 8 :

- on conserve le 2 qui est le premier élément ;
- le 3 n'est pas plus petit que le dernier conservé (qui est 2) donc on le conserve ;
- le 1 est plus petit que le dernier conservé (qui est 3) donc on l'élimine ;
- le 8 n'est pas plus petit que le dernier conservé (qui est toujours 3) donc on le conserve.

La série triée obtenue après cet algorithme est donc 2, 3, 8.

Partie A

Dans cette partie, on implémente le tri dictatorial en utilisant le type `list` de Python pour représenter une série d'entiers.

On souhaite coder une fonction `tri_dictatorial` qui :

- prend en paramètre une liste d'entiers `serie` de type `list` ;
- renvoie une nouvelle liste d'entiers obtenue en suivant l'algorithme présenté en introduction, c'est-à-dire une liste triée, éventuellement vide, ne contenant que les éléments de `serie` à conserver ;

- ne modifie pas `serie`.

Par exemple, si `s = [5, 2, 6, 8, 3, 7]`, l'appel `tri_dictatorial(s)` devrait renvoyer la liste `[5, 6, 8]` sans modifier `s`. On remarque que la liste obtenue est en effet triée.

1. Donner le résultat que doit renvoyer l'appel : `tri_dictatorial([31, 45, 41, 28, 37, 108, 127, 2, 124, 421])`.
2. Expliquer pourquoi le tri dictatorial n'est pas un algorithme de tri.

Edgar a écrit le programme suivant, qui prétend implémenter le tri dictatorial :

```
1 def tri_dictatorial(serie):
2     serie_triee = [serie[0]]
3     for i in range(1, len(serie)):
4         if serie[i] >= serie[i - 1]:
5             serie_triee.append(serie[i])
6     return serie_triee
```

Edgar souhaite tester si sa fonction fait bien ce qu'elle est censée faire.

3. Edgar réalise l'appel `tri_dictatorial([8, 2, 9, 6, 12])`.

   Expliquer pas à pas comment la liste `serie_triee` se construit après cet appel.

4. Edgar réalise maintenant l'appel `tri_dictatorial([])` et obtient l'erreur suivante :

   ```
   Traceback (most recent call last):
     File "tri_edgar.py", line 8, in <module>
       tri_dictatorial([])
     File "tri_edgar.py", line 2, in tri_dictatorial
       result = [serie[0]]
   IndexError: list index out of range
   ```

   Expliquer précisément l'erreur obtenue et proposer une modification du code d'Edgar afin que cet appel soit conforme à l'algorithme du tri dictatorial décrit en introduction.

Dijkstra lors de la réception de son prix Turing en 1972, a notamment déclaré :

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

ce que l'on peut traduire par :

"Tester les programmes peut être un moyen très efficace d'y trouver des bugs, mais c'est un moyen désespérément inadéquat pour prouver leur absence."

5. Expliquer pourquoi des tests ne peuvent pas prouver de façon certaine l'absence de bugs d'un programme en général.

Edgar décide de procéder à un test supplémentaire et réalise l'appel `tri_dictatorial([8, 2, 3, 5, 12])`. La fonction renvoie alors `[8, 3, 5, 12]` qui n'est pas une liste triée.

6. Expliquer la cause du problème et proposer une modification du code d'Edgar afin de la corriger.

Partie B

Dans cette partie, on implémente le tri dictatorial sur des listes chaînées. Cette fois-ci on va modifier la liste chaînée initiale au lieu de construire une nouvelle liste.

On dispose d'une classe `Maillon` :

```
1 class Maillon:
2     def __init__(self, val, suiv):
3         self.valeur = val
4         self.suivant = suiv
```

L'attribut `suivant` doit correspondre à un `Maillon` (le suivant de `self`), ou à `None` si `self` est le dernier.

On dispose également d'une classe `Liste` qui implémente une liste chaînée avec pour unique attribut `tete` qui est le maillon de tête de la liste chaînée, une instance de `Maillon` :

```
class Liste:
    def __init__(self, tete):
        self.tete = tete
```

On peut représenter graphiquement une liste chaînée de la manière suivante, avec la barre à hachure symbolisant la valeur `None` :

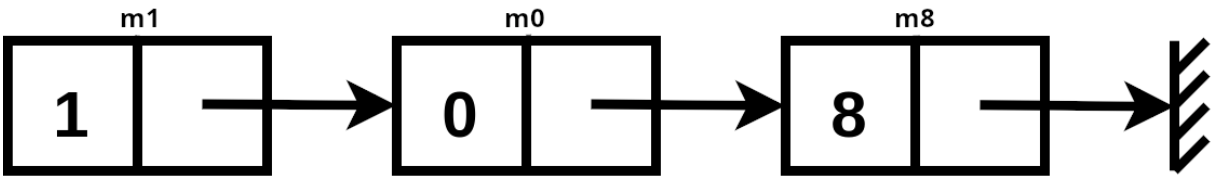


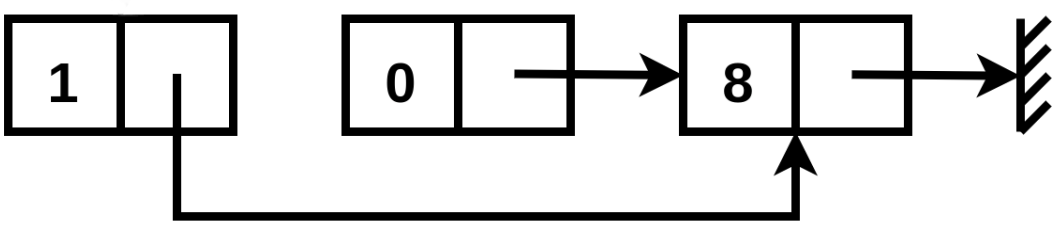
Figure 1. Liste chaînée constituée de trois maillons m1, m0 et m8.

7. Donner des instructions permettant de construire les trois maillons `m1`, `m0` et `m8` et la liste chaînée représentés ci-dessus. On nommera la liste chaînée `ma_liste`.

8. Indiquer ce que renvoie chacune des instructions ci-dessous :

```
m1.valeur == 1
m1.suivant.valeur == 8
m1.suivant.suivant == None
m1.suivant.suivant.suivant == None
```

9. Donner une instruction permettant de transformer `ma_liste` en la liste chaînée représentée ci-dessous :



On souhaite à présent une fonction `tri_dictatorial_chaine` qui prend en paramètre une instance de liste chaînée `chaine` et qui modifie cette liste chaînée démarrant en suivant l'algorithme du pseudo-tri dictatorial. La fonction ne renvoie rien.

10. Recopier et compléter la fonction `tri_dictatorial_chaine` ci-dessous.

```
def tri_dictatorial_chaine(chaine):
    maillon = chaine.tete
    while maillon.suivant ... :
        if maillon.valeur ...
            maillon = ...
        else:
            maillon.suivant = ...
```

Partie C

Une pile p, éventuellement vide, stocke des éléments entiers qu'on souhaite trier selon le pseudo-tri dictatorial. À l'issue du tri, on veut que cette pile soit modifiée et ne contienne plus que des éléments triés.

11. Rappeler le principe du fonctionnement d'une pile.

12. Remettre dans l'ordre les lignes ci-dessous afin d'obtenir l'algorithme attendu, en respectant une tabulation lorsque la ligne est à l'intérieur d'un bloc `si` ou `tant que`.

    – si p n'est pas vide :
        - tant que p n'est pas vide :
        - tant que p2 n'est pas vide :
        - on dépile p, on stocke l'élément obtenu dans la variable `dernier_conservé` et on l'empile dans p2 ;
        - on crée une pile intermédiaire p2 vide ;

- on dépile p et on stocke l'élément obtenu dans la variable `candidat` ;
- si `candidat` est supérieure ou égal à `dernier_conservé` :
- on dépile `p2` et on empile l'élément obtenu dans `p` ;
    - `dernier_conservé` prend la valeur de `candidat` et l'empile dans `p2`

On suppose maintenant que l'on dispose d'une classe `Pile` implémentant une structure de pile. L'appel `help(Pile)` entraîne l'affichage suivant :

```
Help on class Pile in module __main__:

class Pile(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |      Initialize self.  See help(type(self)) for accurate
signature.
 |
 |  __str__(self)
 |      Return str(self).
 |
 |  depiler(self)
 |
 |  empiler(self, elt)
 |
 |  est_vide(self)
```

13. Écrire en Python la fonction `tri_dictatorial_pile` qui prend en paramètre `p` une instance de `Pile` et modifie cette pile afin qu'elle ne conserve que des éléments triés selon le pseudo-tri dictatorial.