X - TP 2

Algorithmes de tri

1 Rappels

En première, deux tris sont étudiés, le tri par insertion, et le tri par sélection. Nous reprenons dans ce tp ces deux tris, en en donnant une implémentation itérative opérant sur les tableaux, et une implémentation récursive opérant sur les listes chaînées. Nous verrons également d'autres tris, qui réalisent en moyenne moins d'opération de comparaison entre éléments.

On pourra également utiliser *en complément* les ressources suivantes :

- https://visualgo.net/bn/sorting Visualisation des algorithmes principaux de tri
- https://fr.wikipedia.org/wiki/Tri_par_insertion
- https://fr.wikipedia.org/wiki/Tri_par_sélection

Dans tout ce tp, le mot "élément" désignera des éléments de $\mathbb{Z}$. Lorsque demandé par l'énoncé, on utilisera l'implémentation à l'aide des listes chaînées que l'on trouvera dans le fichier `data_structures.py`. On rappelle que la fonction `ajoute` prend deux arguments, une liste `l` et un élément `e` et renvoie la liste constituée de l'élément `e` suivi des éléments de `l`.

Code

```
from ds import *
```

1.1 Vérifier qu'un tableau est trié

Écrire une fonction itérative `tab_est_trié` qui étant donné un tableau `tab` renvoie `True` si et seulement si ses éléments sont triés par ordre croissant, c'est à dire si pour tout $i \in \mathbb{N}$ avec $0 \leq i < n-1$ (n est la taille du tableau), `tab[i] <= tab[i + 1]`.

Code

```
def tab_est_trie(tab):
    """ [int] -> bool
    Détermine si le tableau tab est trié par ordre croissant. """
    pass
```

Code

```
print(tab_est_trie([0, 1, 2]))
print(tab_est_trie([0, 2, 1]))
```

Résultat

```
True
False
```

1.2 Vérifier qu'une liste est triée

Écrire une fonction récursive `liste_est_trié` qui étant donné une liste `l` renvoie `True` si et seulement si ses éléments sont triés par ordre croissant.

Code

```
def liste_est_triee(l):
    """ Liste -> bool
    Détermine si la liste l est triée """
    pass
```

Code

```
affiche(l1)
print(liste_est_triee(l1))
```

Résultat

```
1 - 6 - 8 - 9 - x
True
```

2 Tri par insertion

2.1 Méthode itérative

On se propose dans cette partie de donner une implémentation itérative de l'algorithme du tri par sélection. La description itérative de cet algorithme est la suivante (`n` est la taille du tableau):

- à chaque étape, les `i` premières valeurs du tableau sont triées. Initialement, `i` vaut donc `0`.
- tant que `i` n'est pas égal à `n - 1`, on insère l'élément d'indice `i` dans le sous-tableau trié constitué des éléments d'indice `0` (inclu) à `i` (exclu).

  Pour cela on opère par décalages successifs : en partant de l'élément d'indice `i` jusqu'à l'élément d'indice `1` on décale tous les éléments de la partie triée du tableau qui sont supérieurs à l'élément à insérer.

Jouer au jeu https://www.advanced-ict.info/interactive/insertion_sort.html, puis compléter le code de la fonction `tri_insertion_iter` ci-dessous pour que celle-ci corresponde à l'algorithme décrit dans l'énoncé.

Code

```
def tri_insertion_iter(tab):
    """ [int] -> [int]
    Trie en place le tableau tab (tri par insertion) """
    n = len(tab)
    for i in range(...):
        # les éléments d'indice 0..i du tableau t
        # sont triés dans l'ordre croissant
        element = tab[i]
        trou = i
        # on cherche à insérer l'élément dans le tableau
        # constitué des éléments d'indice 0..i
        while ... > ... and trou > 0:
            tab[trou] = ...          # on décale l'élément précédent de tab
            trou = ...               # mise à jour de l'indice du trou
        # À la fin de la boucle soit
        # trou = 0 et tous les éléments de t sont supérieurs à élément
        # soit trou > 0, t[trou - 1] < element < t[i] pour tout i >= trou
        tab[trou] = ...
        # le tableau est alors trié
    return tab # que se passe-t-il si on ne le met pas ?
```

Code

```
print(tri_insertion_iter([3]))
print(tri_insertion_iter([1, 2, 3, 4, 5, 6]))
print(tri_insertion_iter([5, 1, 9, 3, 4, 1]))
print(tri_insertion_iter([9, 5, 4, 3, 2, 1]))
```

Résultat

```
[3]
[1, 2, 3, 4, 5, 6]
[1, 1, 3, 4, 5, 9]
[1, 2, 3, 4, 5, 9]
```

2.2 Méthode récursive

L'objectif de cette partie est d'implémenter l'algorithme du tri par insertion, décrit ci-après.

- si la liste d'éléments à trier est vide, la liste triée est la liste vide ;
- si la liste d'éléments à trier n'est pas vide, on trie récursivement tous les éléments de la liste sauf le premier. Puis, on insère au bon endroit dans cette liste triée l'élément manquant, de telle sorte que la liste résultante soit également triée.

Question 1. Soit $l = (9, 42, 8, 12, 16)$.

1. Quelle est la liste de tout les éléments de l sauf le premier ?
2. Quelle est la liste triée de tous les éléments de l sauf le premier ? On appelle l_1 cette liste.
3. On souhaite insérer un élément e dans une liste l_1 supposée triée. Décrire *en français* les listes solution des questions suivantes. Vous devrez faire des phrases commençant par "C'est la liste constituée de", et en utilisant des expressions parmi "l'élément e", "la tête de l_1", "la queue de l_1", "suivit de", "les éléments de ...", "les éléments de ... où on a inséré l'élément ...".
   a. Quelle est la liste attendue lorsque l_1 est vide, et que e est un élément quelconque ?
   b. Lorsque l_1 n'est pas vide :
      i. Quelle est la liste attendue lorsque e est plus petit que le premier élément de l_1 ? (par exemple $l_1 = (12, 16)$ et $e = 9$)
      ii. Quelle est la liste attendue lorsque e est plus grand que le premier élément de l_1 ? (par exemple $l_1 = (8, 12, 16)$ et $e = 9$)

2.2.1 Insérer dans une liste triée

Afin d'implémenter l'algorithme du tri par insertion, on a besoin d'une fonction auxiliaire. La fonction `insere_trie` prend comme argument un élément `e` et une liste `l` d'éléments **supposée triée**, insère l'élément `e` dans la liste `l` de telle sorte que la liste résultante soit triée. En vous aidant des réponses à la question **??**, écrire le code python d'une fonction récursive `insere_trie` qui réponde à l'énoncé. On rappelle que la fonction `ajoute` admet deux arguments `l` et `e` et renvoie la liste constituée de `e` suivi des éléments de `l`.

Code

```
def insere_trie(l, e):
    """ Liste, int -> Liste
    l est triée par ordre croissant """
    pass
```

Code

```
l = ajoute(ajoute(None, 16), 12)
l = insere_trie(l, 8)
affiche(l)
l = insere_trie(l, 9)
affiche(l)
```

Résultat

```
8 - 12 - 16 - x
8 - 9 - 12 - 16 - x
```

2.2.2 Tri par insertion

À l'aide de la fonction `insere_trie`, écrire une fonction `trie_insertion_rec` qui étant donnée une liste d'éléments `l` renvoie la liste des éléments de `l` triée dans l'ordre croissant. Vous utiliserez l'algorithme décrit dans l'introduction de cette section.

Code

```
def trie_insertion_rec(l):
    """ Liste -> Liste
    Trie la liste l à l'aide de l'algorithme du tri par insertion """
    pass
```

Code

```
l = ajoute(ajoute(ajoute(ajoute(None, 4), 1), 9), 0)
affiche(l)
affiche(trie_insertion_rec(l))
```

Résultat

```
0 - 9 - 1 - 4 - x
0 - 1 - 4 - 9 - x
```

3 Tri par sélection

3.1 Méthode itérative

On se propose dans cette partie de donner une implémentation itérative de l'algorithme du tri par sélection. La description itérative de cet algorithme est la suivante (`n` est la taille du tableau et on prend comme exemple le tri du tableau `[7, 5, 2, 8, 6, 4]`):

- On commence par rechercher le plus petit élément du tableau, puis on l'échange avec l'élément d'indice 0.

  Exemple. Le plus petit élément du tableau à partir de l'indice 0 est 2 que l'on échange avec 7. Le tableau devient `[2, 5, 7, 8, 6, 4]`
- On recherche le plus petit élément du tableau à partir de l'élément d'indice `1`, puis on l'échange avec l'élément d'indice `1`.

  Exemple. Le plus petit élément du tableau à partir de l'indice `1` est `4` que l'on échange avec `5`. Le tableau devient `[2, 4, 7, 8, 6, 5]`
- etc.
- On recherche le plus petit élément du tableau à partir de l'élément d'indice `n  1`, puis on l'échange avec l'élément d'indice `n  1`.

3.1.1 Minimum à partir d'un indice donné

Écrire une fonction `mini_a_partir` qui étant donné un tableau `tab` supposé non vide et un indice `i` renvoie l'indice du plus petit élément du tableau `tab` parmi les éléments d'indice compris entre `i` (inclu) et `len(tab)` (exclu).

Code

```
def mini_a_partir(tab, i):
    """ [int], int -> int
    Renvoie l'indice du plus petit élément de tab à partir de l'indice i
    """
    pass
```

Code

```
print(mini_a_partir([1, 5, 2, 8, 6, 4], 0))
print(mini_a_partir([1, 5, 2, 8, 6, 4], 1))
```

Résultat

```
0
2
```

3.1.2 Tri par sélection

En déduire une fonction `tri_selection_iter` qui étant donné un tableau `tab`, renvoie un tableau constitué des éléments de `tab` triés par ordre croissant. On pourra modifier au passage le tableau initial `tab`.

Code

```
def tri_selection_iter(tab):
    """ [int] -> [int]
    Trie le tableau tab (tri par sélection) """
    pass
```

Code

```
print(tri_selection_iter([1, 5, 2, 8, 6, 4]))
```

Résultat

```
[1, 2, 4, 5, 6, 8]
```

3.2 Méthode récursive

On se propose dans cette question de donner une implémentation récursive de l'algorithme de tri par sélection. La description récursive de cet algorithme est la suivante :

- Si la liste est vide alors la liste triée correspondante est la liste vide.
- Sinon, la liste triée est composée du plus petit élément `mini` de la liste suivi `l` de la liste triée des éléments de la liste `l` où l'on a supprimé un des éléments `mini`.

Question 2. **1.** Soit $l = (3, 42, 1, 8, 1, 12)$.

a. Quel est le plus petit élément de `l` ?

b. Quelle est la liste des éléments de `l` où l'on a supprimé un des plus petits éléments de `l` ?

2. Écrire les fonctions `minimum`, `supprime` et `tri_selection_rec` permettant d'implémenter une version récursive de l'algorithme de tri par sélection.

Code

```
def minimum(l):
    """ Liste -> int
    l est non vide
    Renvoie le plus petit élément de la liste l """
    pass

def supprime(l, e):
    """ Liste, int -> Liste
    l est non vide, e appartient à l
```

```
    Renvoie la liste l où on a supprimé une occurence de e """
    pass

def tri_selection_rec(l):
    """ Liste -> Liste
    Trie la liste l (tri par sélection) """
    pass
```

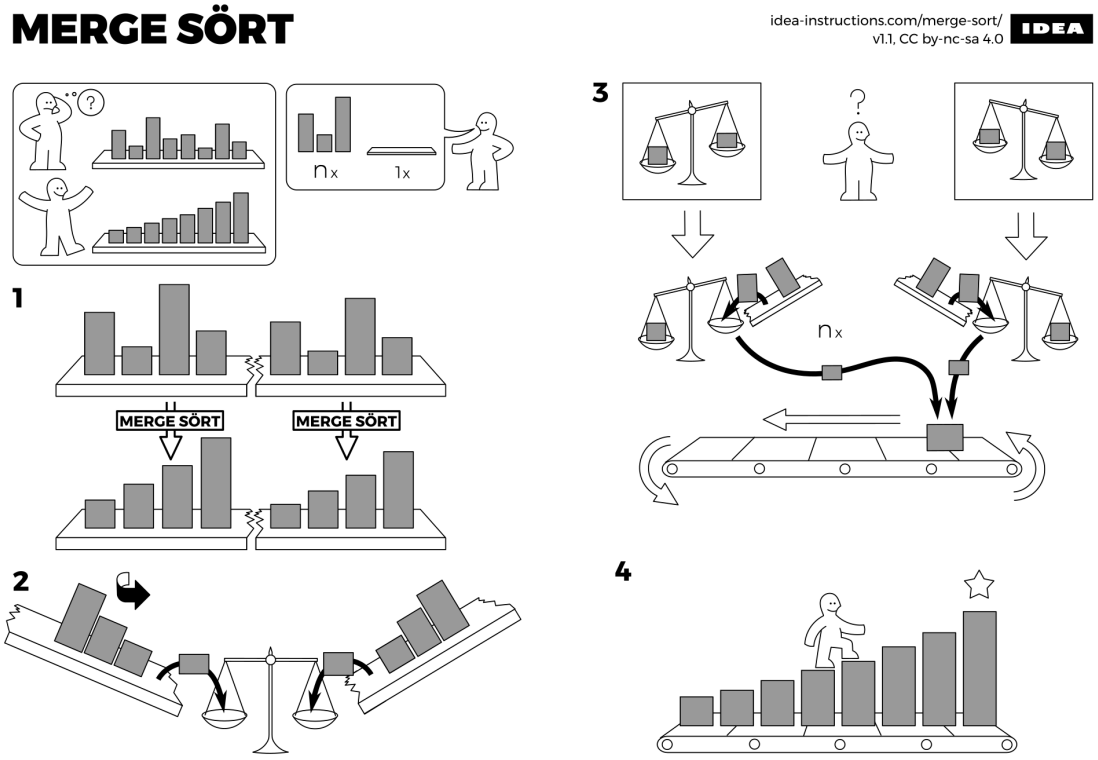
Code

```
affiche(l2)
mini = minimum(l2)
reste = supprime(l2, mini)
print(mini)
affiche(reste)
affiche(tri_selection_rec(l2))
```

Résultat

```
3 - 6 - 1 - 5 - x
1
3 - 6 - 5 - x
1 - 3 - 5 - 6 - x
```

4 Tri fusion



Le tri fusion est un tri récursif qui met en place une technique dite de "diviser pour régner". Pour trier une liste `l` d'éléments, on peut :

- **Diviser** : séparer la liste `l` en deux listes `l1` et `l2` de mêmes tailles (à un élément près) ;
- **Régner** : trier récursivement les listes `l1` et `l2` en des listes `l1_triée` et `l2_triée` ;
- **Combiner** : fusionner les deux listes `l1_triée` et `l2_trée` en une liste triée `l_triée`

Question 3. Soit $l = (3, 42, 1, 8, 1, 12)$.

1. Donner deux listes `l1` et `l2`, dont la longueur diffère d'au plus 1, et dont l'ensemble des éléments soient l'ensemble des éléments de la liste `l`.
2. Donner, sans justification, les listes `l1_triée` et `l2_triée`.
3. Expliquer comment obtenir à l'aide des deux listes `l1_triée` et `l2_triée` la liste `l_triée`.

4.1 Diviser

Écrire le code d'une fonction `diviser` qui étant donné une liste `l` renvoie deux listes de même taille (à un élément près), et dont l'ensemble des éléments soient l'ensemble des éléments de la liste `l`. On pourra s'inspirer de la manière de distribuer un jeu de carte (la liste) à deux joueurs :

- si il n'y a aucune carte, les deux joueurs ne reçoivent aucune carte ;
- s'il n'y a qu'une seule carte, le premier joueur reçoit une carte, le second aucune carte ;
- dans le cas général (il y a au moins deux cartes), on donne la première carte au premier joueur, la seconde carte au second joueur. Puis, on distribue les cartes restantes aux deux joueurs.

Code

```
def diviser(l):
    """ Liste -> Liste, Liste
    Divise la liste en deux listes """
    pass
```

Code

```
l11, l12 = diviser(l1)
affiche(l11)
affiche(l12)
```

Résultat

```
1 - 8 - x
6 - 9 - x
```

4.2 Fusionner deux listes triées

Écrire le code python d'une fonction `fusionner` qui fusionne deux listes `l1` et `l2` **supposées triées** en une liste `l` **triée** dont les éléments sont les éléments de `l1` et de `l2`. On pourra écrire une fonction itérative (en empilant successivement dans une pile auxiliaire la plus petite valeur entre les valeurs au sommet des piles `l1` et `l2`) ou bien récursive.

Code

```
def fusionner(l1, l2):
    """ Liste, Liste
    Renvoie la liste des éléments de l1 et l2 triés """
    pass
```

Code

```
affiche(l3)
affiche(l4)
affiche(fusionner(l3, l4))
```

Résultat

```
1 - 7 - 9 - x
2 - 3 - 8 - x
1 - 2 - 3 - 7 - 8 - 9 - x
```

4.3 Trier récursivement

Question 4. **1.** Écrire une fonction `tri_fusion` qui trie récursivement la liste `l` de la manière suivante :

- on séparer la liste `l` en deux listes `l1` et `l2` de mêmes tailles (à un élément près) ;
- on trier récursivement les listes `l1` et `l2` en des listes `l1_triée` et `l2_triée` ;
- on fusionne fusionner les deux listes `l1_triée` et `l2_trée` en une liste triée `l_triée`

2. Dresser l'arbre d'appel de l'instruction `tri_fusion(l)` où `l` est la liste $(3, 42, 1, 8, 1, 12)$.

Code

```
def tri_fusion(l):
    """ Liste -> Liste
    Trie la liste l (tri fusion) """
    pass
```

Code

```
affiche(l2)
affiche(tri_fusion(l2))
```

Résultat

```
3 - 6 - 1 - 5 - x
1 - 3 - 5 - 6 - x
```