1 Introduction

Les exercices peuvent être résolus indépendamment. La documentation complète des fonctions (*docstring*) n'est pas exigée, cependant les contraintes d'utilisation doivent être explicitées.

Indiquez votre numéro de groupe sur votre copie.

2 Tris des étudiants inattentifs.

0. Avez-vous lu l'introduction ?

3 Algorithmes de tris

Soit `t` la séquence `[3, 4, 1, 7, 12, 0, 5, 41, 42]`.

1. On souhaite appliquer l'algorithme du tri par insertion vu en cours sur `t`. Indiquez l'état de `t` à la fin de chacune des itérations de la boucle principale de cet algorithme.
2. Combien de comparaisons d'éléments de la séquence `t` ont été effectuées pour répondre à la question 1. ?
3. On souhaite appliquer l'algorithme du tri par sélection vu en cours sur `t`. Indiquez l'état de `t` à la fin de chacune des itérations de la boucle principale de cet algorithme.
4. Combien de comparaisons d'éléments de la séquence `t` ont été effectuées pour répondre à la question 3. ?

4 Tri de fichiers

Dans cet exercice, on souhaite trier de différentes façons une liste de chaînes contenant des noms de fichiers .

On ne demande pas de réimplanter les tris par insertion et sélection vus en cours, mais d'utiliser correctement des fonctions déjà définies en Python, et d'éventuellement définir des fonctions de comparaison appropriées.

5. Définissez une fonction `tri_par_nom` paramétrée par une liste de chaînes et renvoyant une copie de cette liste triée dans l'ordre lexicographique unicode décroissant (l'ordre lexicographique unicode est l'ordre usuel sur les chaînes de caractères).

```
>>> tri_par_nom(["correction.txt", "liste_etus.csv", "cigale.txt"])
['liste_etus.csv', 'correction.txt', 'cigale.txt']
>>> tri_par_nom(['Zorro.S03.E02.srt', 'asterix.s05.e01.srt'])
['asterix.s05.e01.srt', 'Zorro.S03.E02.srt']
```

On suppose avoir défini la fonction `compare` et importé la fonction `cmp_to_key`. Voici un exemple d'utilisation de ces fonctions :

```
>>> from functools import cmp_to_key
>>> def cmp_by_len(s1, s2):
...     return compare(len(s1), len(s2))
...
>>> seq = [ "aaaa", "bb", "ccc", "", "d"]
>>> sorted(seq, key=cmp_to_key(cmp_by_len))
['', 'd', 'bb', 'ccc', 'aaaa']
```

6. Définissez une fonction `nombre_cars` paramétrée par une chaîne et renvoyant le nombre de caractères dans le fichier dont le nom est passé en paramètre.

```
>>> with open('toto.txt', 'w') as f:
...     f.write('Timoleon')
8
>>> nombre_cars('toto.txt')
8
```

7. Déduisez-en une fonction `cmp_par_taille` paramétrée par deux chaînes de caractères `s1` et `s2` et qui renvoie :

- +1 si :
  - soit le fichier désigné par `s1` contient strictement plus de caractères que celui désigné par `s2`,

  - soit les fichiers ont le même nombre de caractères et `s1` est après `s2` dans l'ordre lexicographique ;
- -1 si :
  - soit le fichier désigné par `s1` contient strictement moins de caractères que celui désigné par `s2` ,
  - soit les fichiers ont le même nombre de caractères et `s1` est avant `s2` dans l'ordre lexicographique ;
- 0 si `s1` et `s2` désignent le même fichier.

8. Déduisez-en une fonction `tri_par_nb_cars` paramétrée par une liste de noms de fichiers et renvoyant une copie de cette liste, triée dans l'ordre croissant par nombre de caractères dans le fichier, puis par nom de fichier.

9. Définissez une procédure `sauvegarder_infos` paramétrée par une liste de noms de fichiers `liste_fichiers`, et qui crée un nouveau fichier `ls_lr.csv`, dont le contenu est un tableau au format CSV décrit ci-dessous.

Chacune des lignes du fichier `ls_lr.csv` contient le nom d'un fichier dont le nom est dans la liste `liste_fichiers`, suivi d'un point-virgule, suivi du nombre de caractères contenus dans le fichier, supposé existant, portant ce nom.

5 Tri fusion de listes récursives

Dans cet exercice, on travaillera exclusivement avec des listes récursives construites en utilisant `ApLst`. Il est conseillé d'écrire des fonctions récursives. On suppose que le module `ApLst` est importé.

On rappelle les fonctions et méthodes utiles vues en TD et TP.

```
>>> liste_vide = ApLst()
>>> liste_3_1_4 = ApLst(3, ApLst(1, ApLst(4, ApLst())))
>>> liste_vide.is_empty()
True
>>> liste_3_1_4.is_empty()
False
>>> liste_3_1_4.head()
3
>>> liste_3_1_4.tail()
ApLst(1, ApLst(4, ApLst()))
```

10. Définissez un prédicat `zero_ou_un_element` paramétré par une liste récursive et renvoyant `True` si ce paramètre contient exactement 0 ou 1 élément, et `False` s'il en contient 2 ou plus.

    Indication : que peut-on dire du *reste* (de la *queue*) d'une liste récursive à exactement 1 élément ?

11. Définissez une fonction `decoupage` paramétrée par une liste récursive, et renvoyant un couple de listes récursives. La première liste récursive contient les éléments d'indice pair du paramètre, la seconde ceux d'indice impair.

```
>>> (lr1, lr2) = decoupage(liste_3_1_4)
>>> lr1
ApLst(3, ApLst(4, ApLst()))
>>> lr2
ApLst(1, ApLst())
```

    Indication : Il va y avoir deux cas d'arrêt et un appel récursif.

12. Définissez une fonction `fusion` paramétrée par deux listes récursives d'entiers distincts triées dans l'ordre croissant et renvoyant une liste récursive triée dans l'ordre croissant contenant les éléments des deux listes passées en paramètre (et uniquement ceux-ci).

```
>>> lr1 = ApLst(1, ApLst(3, ApLst(4, ApLst())))
>>> lr2 = ApLst(2, ApLst(5, ApLst()))
>>> fusion(lr1, lr2)
ApLst(1, ApLst(2, ApLst(3, ApLst(4, ApLst(5, ApLst())))))
```

Pour effectuer un tri fusion d'une liste récursive on a deux cas :

- soit la liste contient exactement 0 ou 1 élément et elle est donc déjà triée ;
- soit cette liste contient deux éléments ou plus et dans ce cas :
  - on découpe la liste en deux ;
  - on trie récursivement chacune des deux demies listes ;

– on fusionne les deux demies listes triées.

13. Définissez une fonction `tri_fusion` paramétrée par une liste récursive et qui renvoie cette liste triée dans l'ordre croissant en utilisant l'algorithme du tri fusion.

6 Tri par sélection d'une pile

Dans cet exercice, on s'intéresse à l'implantation de l'algorithme du tri par sélection sur une pile.

On rappelle les primitives vues en cours :

- `st = ApStack()` pour créer une pile vide ;
- `st.is_empty()` pour tester la vacuité de la pile `st` ;
- `st.push(v)` pour *empiler* la valeur `v` dans la pile `st` ;
- `v = st.pop()` pour *dépiler* le sommet de la pile `st` et récupérer sa valeur dans la variable dont l'identifiant est `v`.

14. Définissez une fonction `extrait_min` paramétrée par une pile `st` qui renvoie la valeur du plus petit élément de `st` et qui modifie `st` pour lui enlever une occurrence du plus petit élément. L'ordre des éléments dans `st` une fois cette mutation faite n'a pas d'importance.

```
>>> st = ApStack()
>>> st.push(42)
>>> st.push(2)
>>> st.push(13)
>>> extrait_min(st)
2
>>> st.pop()
13
>>> st.pop()
42
>>> st.is_empty()
True
```

Indication : utilisez une seconde pile.

15. Définissez une procédure `tri_par_selection` paramétrée par une pile et qui modifie son paramètre pour que ce dernier soit trié à l'aide de l'algorithme du tri par sélection, avec le plus petit élément au sommet de la pile.

Indication : utilisez une seconde pile.