TP : Algorithmes de tri
Informatique commune, 2ème année

Toutes les fonctions doivent bien sûr être testées.

1. Écrire une fonction `croissant` permettant de savoir si une liste est triée par ordre croissant.
2. Écrire une fonction récursive `appartient(e, L, i)` déterminant si `e` apparaît dans `L` à partir de l'indice `i`. Par exemple, si `L = [1, 5, 3, 7, 4]`, `appartient(3, L, 1)` doit renvoyer `True` mais `appartient(3, L, 3)` doit renvoyer `False`.
3. En déduire une fonction `doublon(L)` déterminant si une liste `L` contient plusieurs fois le même élément. Quelle est sa complexité dans le pire des case?

On veut un algorithme plus rapide pour savoir si une liste contient plusieurs fois le même élément, en utilisant un tri.
On rappelle que le tri fusion sur une liste `L` consiste à:

- Séparer `L` en deux listes `L1` et `L2` de même taille.
- Trier récursivement `L1` et `L2`.
- Fusionner `L1` et `L2` pour avoir un tri de `L`.

On donne la fonction `fusion` du cours permettant de fusionner deux listes triées (pour l'étape (iii)):

```python
def fusion(L1, L2):
    if len(L1) == 0: return L2
    if len(L2) == 0: return L1
    if L1[-1] > L2[-1]: m = L1.pop()
    else: m = L2.pop()
    L = fusion(L1, L2)
    L.append(m)
    return L
```

4. Écrire une fonction `tri_fusion(L)` renvoyant un tri de `L`. Montrer que sa complexité est $O(n \log(n))$ où n est la taille de `L`.
5. Écrire une fonction `doublon_triee(L)` déterminant si une liste **triée** `L` contient plusieurs fois le même élément, en complexité O(`len(L)`).
6. En déduire une fonction `doublon2` déterminant si une liste (non triée à priori) de taille n possède un doublon, en complexité $\mathrm{O}(n \log(n))$. Comparer avec la question 2.

On considère une deuxième application: la recherche de l'élément le plus fréquent dans une liste.

7. Écrire une fonction naïve `frequent(L)` renvoyant l'élément apparaissant le plus souvent de `L` (en cas d'égalité, on en renverra un arbitrairement). Quelle est sa complexité?
8. Écrire une fonction `frequent_triee(L)`, en complexité linéaire, renvoyant l'élément apparaissant le plus souvent de la liste **triée** `L`.
9. En déduire une fonction plus rapide que `frequent` pour faire la même chose. Quelle est sa complexité?
10. (pour cette dernière question, il y a juste besoin d'utiliser `fusion`) Écrire une fonction `fusion_all` telle que, si `LL` est une liste de k listes triées, `fusion_all(LL)` renvoie, en $\mathrm{O}(n \log_2(k))$, leur fusion triée de taille n.
    Par exemple, `fusion_all([[1, 3, 8], [2, 5, 10], [4, 7]])` devra renvoyer
    `[1, 2, 3, 4, 5, 7, 8, 10]`.

Exercice supplémentaire

1. Écrire une fonction `somme2` telle que, si `L` est une liste d'entiers triée de taille `n` et `p` un entier, `somme2(L, p)` renvoie en $\mathrm{O}(n)$ un couple de deux indices i et j tels que `L[i] + L[j] = p`.
   Si un tel couple n'existe pas, on renverra $(-1, -1)$.
   Indice : initialiser `i = 0` et `j = len(L) - 1`. Que peut-on faire si `L[i] + L[j] > p` ? Et si `L[i] + L[j] < p` ?

2. Prouver que `somme2` fonctionne en trouvant un invariant de boucle (hypothèse de récurrence sur le nombre d'itérations/nombre d'appels récursifs).

3. Écrire une fonction `somme_nulle` telle que, si `L` est une liste d'entiers, `somme_nulle(L)` détermine si `L` contient une somme consécutive nulle, c'est à dire s'il existe i et j tels que $\displaystyle\sum_{k=i}^{j} L[k] = 0$.
   On pourra d'abord écrire une fonction en complexité $\mathrm{O}(n^3)$, puis l'améliorer pour obtenir $\mathrm{O}(n^2)$, et enfin $\mathrm{O}(n \log(n))$.
   Pour obtenir une complexité $\mathrm{O}(n \log(n))$, on pourra remarquer que :

$$\sum_{k=i}^{j} L[k] = 0 \iff \sum_{k=0}^{i-1} L[k] = \sum_{k=0}^{j} L[k]$$

   On pourra donc calculer toutes les sommes partielles $\displaystyle\sum_{k=0}^{i} L[k]$ et chercher si il y a un doublon.