Algorithmes de tri (suite)

Commencer par lancer spyder, et téléchargez à l'adresse `http://cahier-de-prepa.fr/pcsi/.....` le fichier `TPtris2.py` et sauvegardez celui-ci dans votre répertoire personnel.

1 Tri rapide (quicksort)

Là où le tri fusion s'appuie sur un algorithme qui réalise la fusion de deux listes triées, le tri rapide s'appuie sur un algorithme qui réalise un partitionnement d'une liste autour d'une valeur pivot, choisie parmi les valeurs de la liste à trier.

Partons par exemple de la liste `L` :

7	2	1	8	6	5	11	9	10	2	6

On va partir de la dernière valeur de `L` qui va servir de pivot : on va réorganiser les 11-1 premiers termes de `L` pour que figurent en tête tous les termes inférieurs (ou égaux) à celui-ci, puis que suivent tous ceux strictement supérieurs au pivot. Un seul parcours, de gauche à droite, de ces 11-1 termes suffit :

Le premier terme de la liste 7 est strictement supérieur au pivot 6 (il fera partie de la seconde partie de notre liste à la fin de notre travail).

Le second 2 est lui inférieur à notre pivot, donc on le déplace en tête (on échange pour ce faire les deux premiers termes) et notre liste devient :

2	7	1	8	6	5	11	9	10	2	6

Les termes grisés ne bougeront plus !

le terme suivant 1 est lui aussi inférieur à notre pivot, donc il vient compléter notre tête de liste en échangeant deux valeurs :

2	1	7	8	6	5	11	9	10	2	6

le suivant 8 est supérieur à notre pivot, donc on le laisse en place, puis le terme suivant 6 est inférieur ou égal à notre pivot donc il rejoint la tête de liste :

2	1	6	8	7	5	11	9	10	2	6

on fait de même pour le terme suivant 5, on laisse en place les trois suivants et on déplace encore le terme 2 en tête de liste pour parvenir à :

2	1	6	5	2	8	11	9	10	7	6

Garder notre valeur pivot en queue de liste n'a plus guère de sens, et on le déplace à sa position définitive en échangeant encore deux valeurs dans notre liste :

2	1	6	5	2	6	11	9	10	7	8

On le voit alors, notre valeur pivot est désormais bien placée, et ce qu'il reste à faire, c'est à trier la tête de liste (jusqu'au pivot non compris) et la queue de liste (à partir du pivot non compris)

Question 1. Compléter la fonction suivante, qui prend en argument une liste `L`, deux entiers `deb<fin` et qui modifie en place `L` pour partitionner `L[deb:fin]` selon la démarche proposée ci-dessus. Comme ci-dessus, on prendra comme pivot la dernière valeur de `L[deb:fin]` (c'est-à-dire `L[fin-1]`). Notre fonction renverra l'indice de la position finale du pivot, laquelle nous permet de connaître quelles sont les parties obtenues par ce partitionnement.

```
def partition(L, deb, fin):
    v = L[fin-1] # plus commode de nommer la valeur du pivot
    pos = deb # à tout moment la tête de liste sera formée de L[deb:pos]
    for i in range(deb, fin-1):
        if L[i] <= v:
            ...
```

```
    # on pense ici à placer le pivot à sa place définitive
    return pos
```

Question 2. Ecrire alors une fonction récursive, d'entête `def triRapideRec(L, deb, fin):` qui modifie en place `L` pour trier les termes d'indices `i` tels que `deb<=i<fin` à l'aide de l'algorithme décrit précédemment,

puis en déduire une fonction d'entête `def triRapide(L):` qui en modifiant en place `L` en réalise le tri.

Question 3. Qu'advient-il si on essaye de trier avec le tri rapide tel qu'implémenté une liste telle que `list(range(5000,0,-1))` ?

Comment expliquer ce phénomène, et comment pourrait-on le corriger ?

Remarque 1. La démonstration est trop compliquée pour la détailler ici, mais on peut établir qu'en moyenne, la complexité du tri rapide pour trier une liste de n termes est proportionnelle à $n \ln n$, comme pour le tri fusion. (On parle de complexité quasi-linéaire). Petit souci, comme vu à la question précédente, dans le pire des cas, la complexité peut-être quadratique, ce qui est beaucoup moins bon.