ALGORITHMES DE TRIS

Exercice n° 1 : Tri par dénombrement

On a vu en cours que la complexité minimale d'un tri par comparaison, dans le pire des cas, est semi-logarithmique (c'est même vrai en moyenne).
Par contre, si l'on a des informations sur le tableau L, il se peut que le coût puisse être ramené à un ordre linéaire.
C'est le cas lorsque le tableau T de longueur n à ses éléments dans l'intervalle entier $[\![0, k-1]\!]$. En un parcours du tableau T, il est possible de dénombrer le nombre d'apparitions de chacune des k valeurs possibles, au prix d'un espace mémoire proportionnel à k.

1. Programmer les fonctions initiées en cours.
2. Les tester sur la liste $L = [4, 1, 4, 1, 3, 4, 1, 0, 0, 1]$.

Dans toute la suite, on travaille avec des tableaux non vides de réels distincts, de longueur $n \geqslant 1$.
Le but est de le trier *par ordre croissant*.

Exercice n° 2 : Identificaton d'un tri

1. On considère un tableau `Lt` déjà triée par ordre croissant.
   Expliquer le rôle de la fonction **récursive** `IR(L,e)` suivante :

```
def IR(Lt, e) :
    n = len(Lt)
    if Lt==[] or e>= Lt[n-1] :
        return Lt + [e]
    else :
        return IR(Lt[:n-1], e) + [Lt[n-1]]
```

   Cette fonction modifie-t-elle `Lt` ?

2. À l'aide de la fonction précédente, et en vous inspirant de la structure de programme du tri par insertion, écrire maintenant une fonction `tri(T)` de paramètre un tableau Tet qui retourne le tableau trié.
3. Quelle est la différence notable de ce programme avec le tri par insertion vu en classe ?
4. Tester cette fonction sur la liste décroissante des entiers de 999 à 0.
   On rappelle qu'on peut modifier la profondeur de la pile d'appel à n appels, grâce à :

```
import sys
sys.setrecursionlimit(n)
```

5. *Application* :
   On rappelle que la médiane d'une liste ordonnée L est :
   – l'élément centré si n = len(L) est impair.
   – la moyenne des 2 éléments centrés si n = len(L) est pair.

À l'aide de l'algorithme de tri, coder une fonction `mediane` de paramètre une liste L de scalaires, et qui retourne la médiane de ces valeurs.

La tester sur les listes décroissantes d'entiers de 100 à 0, puis de 100 à 0.

6. *Complexité dans le pire des cas*

a) Rappeler pour quel type de listes l'algorithme de tri par insertion se trouve dans le pire des cas.

b) Modifier l'algorithme `tri` de sorte qu'il retourne , en plus de L, le nombre d'appels de la fonction `IR`.

c) Créer une fonction `complexite_pire(n)` qui retourne le nombre d'itérations pour trier par insertion la liste $[n-1,\dots,1,0]$.

d) Représenter la fonction `complexite_pire(n)` en fonction de `n`.

e) Conjecturer alors l'ordre de complexité dans le pire des cas. Vérifier avec la preuve vue en cours.

Exercice n° 3 : Tri rapide

Deux programmes de tri rapide vous sont données dans le répertoire de TP. Le deuxième est juste une compactification du premier, utilisant les listes par compréhension.
À la suite de ces algorithmes, vous trouverez un module permettant d'évaluer et de représenter la complexité du tri rapide :
– dans le cas d'une liste triée (c'est le pire des cas !) ;
– dans le cas d'une liste triée, mais en modifiant la clé du tri rapide par une clé aléatoire, afin d'améliorer le coût (sensiblement) ;
– pour une liste aléatoire d'entiers (non forcément distincts et on remarquera d'ailleurs que l'algorithme de tri fonctionne dans ce cas-là, même s'il n'est pas stable).

1. Compléter les lignes 30, 43 et 56 (on pourrra consulter l'aide de `r.randint`).

2. Compléter les algorithmes de tri rapide, en rajoutant une compteur c du nombre de comparaisons effectuées.

3. Lancer alors le programme, qui représentera graphiquement le nombre de comparaisons nécessaires, en fonction de n, pour les 2 cas proposés.

4. Analyser les courbes.

5. Comment peut-on "lisser" la courbe de la complexité sur liste aléatoire ?

6. *Application* : écrire un programme similaire `tri_couple(L)` de paramètre une liste L de couples de scalaires, et qui retourne la liste de ces couples ordonnés suivant leur premère composante.

Exercice n° 4 : Application des tris dans un fichier de mesures

On trouvera dans le dossier de TP un fichier "CampDatas.txt". Celui-ci contient une liste de vitesses et de couples, chaque ligne (à partir de la 2e) se présentant sous la forme :

chaine_vitesse_entiere ; chaine_couple_flottant\n

Extraire les données de ce fichier, puis les trier suivant les vitesses croissantes, puis représenter le couple en fonction de la vitesse.

On utilisera une fonction de tri rapide agissant sur une liste de couples.

Exercice n° 5 : **Le tri gnome (*gnome sort*)**

« Gnome » car il paraît que c'est la méthode utilisée par les nains de jardin hollandais pour trier une rangée de pots de fleurs de la plus petite fleur à la plus grande...

1. Voici ci-derrière une implémentation du tri en Python. Expliquer son fonctionnement :

```
def tri_gnome(L) :
    i, n = 0, len(L)
    while i < n - 1 :
        print(L, i)
        if L[i+1] >= L[i] :
            i += 1
        else :
            L[i+1], L[i] = L[i], L[i+1]
            if i > 0:
                i -= 1
    return L
```

2. Tester cet algorithme sur un exemple (*on poura rajouter des "print" intermédiaires, pour mieux voir les étapes de tri*).
3. Quelle est la complexité dans le meilleur des cas de cet algorithme ?

REMARQUE *: on montre que la complexité temporelle moyenne du tri gnome reste quadratique... Encore aucun gain, donc.*

Exercice n° 6 : **Le tri à bulles (*bubble sort*)**

Le tri à bulles consiste à comparer deux à deux et successivement tous les couples consécutifs d'éléments d'une liste donnée et à les permuter dans le cas où le deuxième est plus petit que le premier. On répète l'opération jusqu'à ce que la liste soit triée.
Vous pouvez aller voir sur le site : lwh.free.fr/pages/algo/tri/tri.htm

1. Tester cet algorithme sur : L=[8, 7,9,6].
2. Écrire une fonction `tri_bulle(n)` renvoyant la liste triée, en suivant l'ébauche :

```
def tri_bulle(L) :
    n = len(L)
    fini = False
    while not fini :
        c=0   # le but de ce compteur est de compter le nombre d'échanges effectués à chaque balayage complet de la liste
              # Dès qu'il reste égal à 0, c'est que c'est fini !
        (...)
```

3. Combien d'échanges sont nécessaires pour trier une liste de taille n dans le meilleur des cas ? dans le pire des cas ?
   Combien de comparaisons sont nécessaires pour trier une liste donnée dans le meilleur des cas ? dans le pire des cas ?
   Qu'en déduire sur la complexité de cet algorithme ?