TP en autonomie

Le sujet

But du TP

On vous demande ici de programmer avec plein de **petites fonctions** qui effectuent une tâche unique dont le nom permet de comprendre la fonction. On essayera ainsi de décomposer les algorithmes en tâches que l'on pense insécables.

Par exemple pour le tri par insertion vu en TD :

```
def insertion(tableau):
    """Tri par insertion.
    """

    for etape in range(len(tableau)):
        valeur_a_placer = tableau[etape]
        indice_insertion = cree_place_pour_inserer(valeur_a_placer, etape,
tableau)

        tableau[indice_insertion] = valeur_a_placer

def cree_place_pour_inserer(valeur_a_inserer, indice_max_tableau, tableau):
    position = indice_max_tableau
    while position > 0 and ordre_non_respecte(tableau[position - 1],
valeur_a_inserer):
        decale_a_droite(tableau, position)
        position = position - 1

    return position

def ordre_non_respecte(valeur_petite, valeur_grande):
    return valeur_grande < valeur_petite

def decale_a_droite(tableau, indice):
    tableau[indice] = tableau[indice - 1]
```

L'idée est de pouvoir lire l'algorithme comme on le ferait en pseudo-code.

Tests

On vous demande de coder des tests (voir Les tests) qui vous permettront de vérifier expérimentalement que le code du tri par insertion ci-dessus fonctionne. On vous demande donc de :

- copier-coller le code du tri par insertion dans un fichier que vous nommerez `tri_insertion.py`

- de créer un fichier `test_tri.py` où vous testerez que le tri par insertion fonctionne. Il doit permettre de trier:
    - un tableau déjà trié,
    - un tableau trié à l'envers,
    - un tableau de nombres placés aléatoirement.

Vous utiliserez ces tests pour valider les implémentations des différents tris que vous coderez dans ce tp.

Le code ci-dessous vous donne la matrice du fichier `test_tri.py`. Vous devez combler les trous et utiliser le TP1 pour l'executer avec pycharm.

```
from tri_insertion import insertion

def test_deja_trie:
    tableau_a_trier = [1, 3, 5, 6]
    insertion(tableau_a_trier)
    assert tableau_a_trier == [1, 3, 5, 6]
def test_a_l_envers:
    # faire le test

    assert True == False

def test_aleatoire:
    # faire le test
    assert True == False
```

Tri Rapide (Quicksort)

Le tri rapide est un algorithme de tri ayant une complexité maximale différente de sa complexité moyenne. On vous demande de le vérifier expérimentalement.

Ainsi :

- codez l'algorithme du tri rapide et testez le (voir tp1) sur un tableau trié de façon croissante, un tableau trié de façon décroissante et un tableau non trié.
- vérifiez expérimentalement que le temps mis pour trier un tableau trié est de l'ordre de $\mathcal{O}(n^2)$. Le rapport entre temps pris et complexité théorique devant tendre vers une constante.

Affichage

Adaptez le code de la partie Affichage de courbes du TP1 pour tracer les temps pris par le tri rapide et le tri fusion pour des tableaux trié par ordre croissant pour des tailles allant de 2 à 200.

Complexité moyenne

Utilisez le module random de python (par exemple la méthode shuffle) pour mesurer la complexité en moyenne du tri rapide (on pourra faire la moyenne du temps pris sur 10 tableau aléatoires de talle donnée). Vérifiez expérimentalement que le temps moyen pour trier un tableau est de l'ordre de $\mathcal{O}(n \log((n))$. Le rapport entre temps pris et complexité théorique devant tendre vers une constante.

Afficher la courbe de temps.

Tri Fusion (Mergesort)

Le tri fusion est un tri de complexité maximale minimale. Codez ce tri et testez le de la même manière que le tri rapide.

Vérifiez expérimentalement que le rapport entre temps pris pour trier des tableaux déjà tris avec MergeSort et Quicksort tend vers 0 lorsque la taille augmente.

From:
https://wiki.centrale-med.fr/informatique/ - **WiKi informatique**

Permanent link:
https://wiki.centrale-med.fr/informatique/tc_info:tpa2

Last update: **2019/10/22 12:53**