tri_insertion_v2

May 28, 2020

0.1 Tri par insertion

1 Principe

Voir le principe à l'adresse: http://lwh.free.fr/pages/algo/tri/tri_insertion.html

L'idée directrice est de **parcourir le tableau de la gauche vers la droite, en maintenant la partie déjà triée sur sa gauche**. Concrètement, on va décaler d'une case vers la droite tous les éléments déjà triés, qui sont plus grands que l'élément à classer, puis déposer ce dernier dans la case libérée.

2 Implémentation en python

[ ]:
```
def tri_insertion(t):
    """
    Trie le tableau t par ordre croissant
    """
    for i in range(1,len(t)):
        elt_a_classer = t[i]
        j = i
        #décalage des éléments du tableau pour trouver la place de t[i]
        while j>0 and t[j - 1] > elt_a_classer:
            t[j] = t[j - 1]
            j = j - 1
        #on insère l'élément à sa place
        t[j] = elt_a_classer
```

3 Validité de l'algorithme

Au début de chaque itération d'index `i`, le sous tableau `t[0..i-1]` est trié (attention, on fait référence aux éléments d'index 0 à `i-1` **inclus**, ce n'est pas une notation python). Cette propriété est appelée **invariant de boucle**. L'invariant doit respecter les deux propriétés suivantes:

3.1 être vérifié avant la première itération (initialisation)

Lorsqu'on est en ligne 5 la première fois, `i = 1` et cette propriété se traduit par: la tableau `t[0]` est trié. Ce qui est vrai évidemment car on n'a qu'un seul élément.

3.2 être conservé (conservation)

Si l'invariant est vrai avant une itération, il l'est aussi avant la prochaine.

Ici pour une itération d'index `i` donné, on va décaler les `t[i-1]`, `t[i-2]`, .. etc, vers la droite pour trouver la place de `t[i]`. On place alors `t[i]`. Ces éléments sont triés et comme on incrémente `i` au début de la boucle `for` de la prochaine itération, l'invariant est conservé.

Par ailleurs, la combinaison de l'invariant avec la terminaison de la boucle permet de conclure à la validité (on dit *correction*) de l'algorithme.

Dans notre cas, à la fin de la boucle, `i = len(t)`. Si on substitue cette valeur dans l'expression de l'invariant, on a :
à l'itération `i = len(t)`, le sous tableau `t[0..len(t)-1]` est trié, soit:

le tableau t est trié, l'algorithme est correct.

4 Efficacité: complexité de l'algorithme

Dans le pire des cas, pour un tableau de taille `n`, il faudra effectuer:

$$1 + 2 + \cdots + (n-2) + (n-1) = \frac{n}{2} \times (n-1)$$

comparaisons et échanges. Le coût (on dit aussi *complexité*) est **quadratique**. On écrit souvent que la complexité est en $\Theta(n^2)$.

[ ]: