Quizz_2022_12

May 9, 2022

(A) Écrire une fonction qui prend en paramètre une liste triée d'entiers (ordre croissant) et un élément, et renvoie l'indice de la position où cet élément devrait être inséré dans la liste pour qu'elle reste triée. Par exemple, pour la liste `[2,4,6,9,12]`, la fonction renvoie 2 pour l'élément 5, et 0 pour l'élément 1.

[1]:
```python
# Version sans rupture de contrôle
def indice_bonne_place(liste, element):
    i = 0
    while i < len(liste) and liste[i] < element:
        i += 1
    return i
```

[2]:
```python
# Variante avec rupture de contrôle
def indice_bonne_place_break(liste, element):
    for i in range(len(liste)):
        if liste[i] >= element:
            return i
    return len(liste)
```

[3]:
```python
# Variante qui exploite le tri: recherche dichotomique
# implémentation récursive
def indice_bp_dicho(liste, element, debut, fin):
    m = debut + (fin - debut) // 2
    if liste[m] >= element and liste[m-1] < element:
        return m
    if liste[m] >= element:
        return indice_bp_dicho(liste, element, debut, m)
    else:
        return indice_bp_dicho(liste, element, m+1, fin)

def indice_bonne_place_dicho(liste, element):
    return indice_bp_dicho(liste, element, 0, len(liste))
```

[4]:
```
# Procédure intensive de test
import random

# Génération d'une liste aléatoire triée
def genere_liste_triee():
    liste = random.sample(range(50),20)
    liste.sort()
    return liste

# Création d'une liste de listes de test
liste_de_listes = []
for i in range(20):
    liste_de_listes.append(genere_liste_triee())

# Boucle de test et comparaison des 3 implémentations
for L in liste_de_listes:
    for k in range(10,20):
        valeurs = []
        for f in [indice_bonne_place, indice_bonne_place_break,␣
 ↪indice_bonne_place_dicho]:
            c = f(L,k)
            valeurs.append(c)
        v = valeurs[0]
        for i in range(1,3):
            if valeurs[i] != v:
                print("Problème")
```

(B) Considérons la fonction suivante, qui utilise la fonction précédente pour faire un tri. Quel est l'ordre de grandeur de sa complexité ? Expliquez brièvement comment vous parvenez à ce résultat.

[5]:
```
def nouveau_tri(liste):
    newl = []
    newl.append(liste[0])
    for element in liste[1:]:
        newl.insert(indice_bonne_place(newl,element), element)
    return newl
```

[6]:
```
nouveau_tri([9,5,2,8,6,1,12,3,2])
```

[6]:
```
[1, 2, 2, 3, 5, 6, 8, 9, 12]
```

La boucle principale de ce tri fait autant de tours qu'il y a d'éléments dans la liste (moins un), donc $n-1$. Chaque opération d'insertion fait un appel à la fonction `indice_bonne_place()`, le coût total est donc de $n-1$ fois le coût de la fonction `indice_bonne_place()` plus le coût de l'insertion dans la liste.

- La fonction `indice_bonne_place()` (versions non dichotomiques) fait un nombre de comparaisons qui dépend de la place de l'élément dans la liste: en moyenne on peut dire que c'est de l'ordre la moitié de la longueur de la liste courante (qui croît de 1 à n). Dans le pire des cas, on parcourt toute la liste en construction avant de trouver la bonne place: c'est le cas où les éléments de la liste initiale sont déjà bien rangés (!). La complexité globale est en $O(n)$ dans les deux cas. On peut noter que dans le meilleur des cas, quand la liste est rangée à l'envers, chaque nouvel élément est inséré en première position, et le nombre de tours est d'ordre n.
- La fonction d'insertion dans une liste (ailleurs qu'en position 0) est en général de coût linéaire (proportionnel à la longueur de la liste). Il est possible d'implémenter une liste dans laquelle l'insertion est à coût constant, mais cela demande de la place et une structure de données complexe (il faut une implémentation chaînée, pour ne pas avoir à faire de transferts, et indexée, pour ne pas avoir à faire de parcours), et python ne garantit pas que l'implémentation de la méthode `insert()` soit à coût constant. Le coût total est donc d'ordre n^3 dans le cas moyen et dans le pire des cas. On peut noter que dans le meilleur des cas, avec une insertion toujours en position 0, le coût total devient linéaire.

Avec l'implémentation dichotomique de la fonction `indice_bonne_place()`, on fait baisser le coût moyen: la recherche coûte $\log_2 n$ opérations en moyenne (même si dans le pire des cas on revient à n.) Par conséquent, on obtient un coût total moyen de $O(n^2 \log_2 n)$ (avec l'inconvénient habituel d'une plus grande complication algorithmique).

Si on ajoute que ce tri demande deux fois plus de place qu'un tri *in situ* puisqu'on crée une nouvelle liste, on comprend que cet algorithme ne soit pas considéré comme particulièrement plus intéressant que les algorithmes quadratiques déjà vus en cours.