Corrigé du TP Python sur les Tris par comparaison

852 - M.Lalauze - M.Junier

1 Sitographie

Quelques sites internet avec des applications de simulations d'algorithmes de tris et des outils de visualisation :

- Sorting algorithms
- Algorithmes de tris sur Interstices
- Une application de visualisation sans nombres

2 Quelques outils de tests

```
from random import randint

#un tableau contenant 3 tableaux d'entiers aléatoires de tailles 10, 100, 1000
BENCH2 = [[randint(1, 2*10**i) for _ in range(10**i)] for i in range(1,4)]

#idem mais avec 3 tableaux d'entiers aléatoires de tailles impaires
BENCH1 = [[randint(1, 2*10**i) for _ in range(10**i + 1)] for i in range(1,4)]

"""
In [7]: BENCH2[0]
Out[7]: [14, 14, 9, 14, 10, 2, 1, 20, 1, 20]

In [8]: [len(BENCH2[k]) for k in range(len(BENCH2))]
Out[8]: [10, 100, 1000]

#dix derniers éléments de BENCH2[2] (varie selon l'exécution)
In [9]: BENCH2[2][-10:]
Out[9]: [1411, 1942, 1716, 1924, 280, 584, 949, 823, 1770, 1232]
"""

def bontri(t):
    """Détermine si un tableau est trié dans l'ordre croissant"""
    for k in range(len(t)-1):
```

```
        if t[k] > t[k+1]:
            return False
    return True

"""
In [22]: bontri([1,2,3])
Out[22]: True

In [23]: bontri([1,3,2])
Out[23]: False
"""

def procedure_to_fonction(f):
    """Remplace la fonction f qui est une procedure ne retournant rien
    par une fonction fbis qui exécute f sur ses arguments puis retourne
    ses arguments"""

    def fbis(*args):
        f(*args)
        return args

    return fbis

"""

In [11]: t1 = BENCH1[0]

In [12]: t1
Out[12]: [2, 7, 8, 9, 10, 13, 13, 14, 16, 17, 17]

In [13]: procedure_to_fonction(tri_insertion)(t1)
Out[13]: ([2, 7, 8, 9, 10, 13, 13, 14, 16, 17, 17],)

"""


def test_tri(tri, BENCH):
    #copie profonde de BENCH qui est un tableau de tableaux
    COPIE = [t[:] for t in BENCH]
    return [bontri(procedure_to_fonction(tri)(t)) for t in COPIE]
```

3 Exercice 1

```
def tri_insertion(t):
    """Tri d'insertion en place par échanges d'éléments adjacents"""
    for i in range(1, len(t)):
        j = i
```

```
        while j >= 1 and t[j] < t[j - 1]:
            t[j - 1], t[j] = t[j], t[j - 1]
            j = j - 1

def tri_insertion_echangeV2(t):
    """Tri d'insertion en place par échanges d'éléments adjacents"""
    #Première boucle sur l'index du premier élément pas trié
    for indexpatri in range(1, len(t)):
        j = indexpatri - 1
        #seconde boucle sur les éléments déjà triés
        #pour insérer le premier élément pas trié parmi eux
        while j >= 0 and t[j] > t[j+1]:
            t[j], t[j+1] = t[j+1], t[j]
            j = j - 1

def tri_insertion_echangeV3(t):
    """Tri d'insertion en place"""
    #Première boucle sur l'index du premier élément pas trié
    for indexpatri in range(1, len(t)):
        element = t[indexpatri]
        j = indexpatri - 1
        #seconde boucle sur les éléments déjà triés
        #on recherche l'index où il faut insérer l'élément
        while j >= 0 and t[j] > element:
            t[j+1] = t[j]
            j -= 1
        t[j+1] = element



"""
In [1]: BENCH1[0]
Out[1]: [2, 7, 8, 6, 7, 16, 6, 4, 18, 14, 20]

In [2]: test_tri(tri_insertion, BENCH1)
Out[2]: [True, True, True]

In [3]: BENCH1[0]  #les tableaux du banc d'essai n'ont pas été modifiés
Out[3]: [2, 7, 8, 6, 7, 16, 6, 4, 18, 14, 20]

In [4]: test_tri(tri_insertion_echangeV2, BENCH1)
Out[4]: [True, True, True]

In [5]: test_tri(tri_insertion_echangeV3, BENCH1)
Out[5]: [True, True, True]
"""
```

4 Exercice 2 : tri par insertion avec l'outil de slicing

On recherche la place où il faut insérer le nouvel élément dans le tableau déjà trié et on l'insère avec l'outil de slicing.

On donne ci-dessous un exemple d'insertion d'élément en position 1 (sans suppression de l'élément qui était auparavant en position 1).

```
"""
In [25]: t = [3, 1, 5]

In [26]: t[1:1] = [7]

In [27]: t
Out[27]: [3, 7, 1, 5]
"""

def place(x, t):
    '''Retourne l'index de la place de x dans le tableau t trié dans l'ordre croissant'''
    k = len(t) - 1
    while k >= 0 and t[k] > x:
        k -= 1
    return k + 1

def tri_insertion_2(t):
    tri = []
    for x in t:
        k = place(x, tri)
        tri[k : k] = [x]
    return tri

"""
In [22]: test_tri(tri_insertionbis, BENCH1)
Out[22]: [True, True, True]
"""
```

Analyse de complexité du tri par insertion : (hors programme)

- Si le tableau est *déjà trié*, on effectue alors une seule comparaison à chaque insertion soit $n-1$ comparaisons au total, **la complexité est alors linéaire**. Cette bonne propriété que ne possède pas le tri par sélection fait du tri par insertion un tri efficace lorsque le tableau est déjà trié ou lorsqu'il y a peu de comparaisons.
- Si le tableau est *trié dans l'ordre contraire* on effectue i comparaisons lors de l'insertion d'indice i pour i variant de 1 à $n-1$ (tableaux indexés de 0 à $n-1$) , soit $1+2+\cdots+n-1 = n(n-1)/2$ comparaisons. **La complexité est alors quadratique**.

5 Exercice 3 : Médiane d'une série

```
def mediane(t):
    """Retourne la médiane d'un tableau de nombres
    sans modifier le tableau"""
    tcroi = tri_insertion_2(t) #tableau t trié dans l'ordre croissant
    m = len(t)//2
    if len(t)%2 == 0:
        return (tcroi[m -1] + tcroi[m])/2
    return tcroi[m]


def medianebis(tab):
    """Détermine la médiane d'un tableau de nombres en triant et modifiant
    le tableau sur place"""
    #tri du tableau dans l'ordre croissant
    tri_insertion(tab)
    n = len(tab)
    # on applique la formule de la médiane vue au lycée
    # attention les indices commencent à 0
    # donc si le nombre d'éléments dans la liste est impaire
    # sa taille est paire et vice-versa
    if n%2==1:
        return tab[n//2]
    return 1/2*(tab[(n-1)//2]+tab[(n-1)//2+1])

"""
In [16]: [medianebis(t) for t in BENCH1]
Out[16]: [11, 98, 983]

In [17]: [mediane(t) for t in BENCH1]
Out[17]: [11, 98, 983]
"""
```

6 Exercice 4 : recherche d'un élément

6.1 Question 1 : Recherche séquentielle

```
def appartient(liste, element):
    '''Recherche séquentielle d'un élément dans une liste'''
    for x in liste:
        if x == element:
            return True
    return False
```

Lors d'une recherche séquentielle, la complexité est linéaire. Si l'élément cherchée se trouve en dernière position ou est absent dans la liste, alors il faut comparer l'élément cherché avec tous les éléments de la liste.

La **complexité temporelle est linéaire**, de l'ordre de $O(n)$ où n est la taille de la liste.

6.2 Question 2 : Recherche par dichotomie

```
def recherche_dicho(liste, element):
    '''Retourne l'index d'element dans liste triée dans l'ordre croissant
    si element dans liste, sinon retourne None'''
    n = len(liste)
    a = 0
    b = n - 1
    while a <= b:
        m = (a + b) // 2
        if liste[m] < element:
            a = m + 1
        elif liste[m] > element:
            b = m - 1
        else:
            #l'élément cherché est en position m
            return m
    #l'élément cherché n'est pas dans la liste
    return None
```

L'algorithme de recherche dichotomique ne peut s'appliquer qu'à un tableau trié.

- Commençons par prouver la terminaison et la correction de l'algorithme.
  - **Premier cas :** Si `element` appartient à `t`, la boucle maintient **l'invariant** `a <= element <= b`. `element` est nécessairement atteint avant la fin de la boucle, lorsque le test `liste[m] == element` retourne `True`, ce qui finit par se produire puisque la boucle s'exécute tant que `a <= b`.
    La fonction retourne alors `True` avant la fin de la boucle ce qui est correct dans ce cas.
  - **Deuxième cas :** Si `element` n'appartient à `t`, la boucle se termine car la structure conditionnelle incrémente ou décrémente `m` à chaque tour et on atteint nécessairement l'état `a < b` ou `a > b`. La boucle se terminant sans que `element` soit trouvé, la fonction retourne `False` ce qui est correct dans ce cas.
- Analysons la **complexité temporelle** de cet algorithme :
  - À chaque tour de boucle, la longueur de la zone de recherche est divisée par 2 environ. Initialement la zone de recherche a pour longueur $b-a$ donc la boucle se termine dans le pire des cas au bout de n tours avec n vérifiant $\frac{b-a}{2^n} < 1 \iff 2^{n-1} \leqslant b-a < 2^n$.

  En appliquant le logarithme népérien qui est une fonction croissante sur $[0;+\infty[$, on obtient :

$$(n-1)\ln(2) \leqslant \ln(b-a) < n\ln(2)$$
$$n-1 \leqslant \frac{\ln(b-a)}{\ln(2)} < n$$

Ainsi le nombre maximal d'itérations est $\left\lfloor \frac{\ln(b-a)}{\ln(2)} \right\rfloor + 1$, et la **complexité temporelle est logarithmique**, de l'ordre de $O(\ln(n))$ où n est la taille de la liste.

```
def place_dicho(x, t):
    '''Retourne l'index de la place de x dans le tableau t trié dans l'ordre
    croissant. Recherche dichotomique'''
    n = len(t)
    a = 0
    b = n - 1
    if t[b] <= x:
        return n
    elif x <= t[a]:
        return 0
    else:
        while b - a > 1:
            m = (a + b) // 2
            if t[m] >= x:
                b = m
            else:
                a = m
        return b

"""
In [19]: place_dicho(12, [10,11,12])
Out[19]: 3

In [20]: place_dicho(10, [10,11,12])
Out[20]: 0

In [21]: place_dicho(11, [10,11,12])
Out[21]: 1

In [22]: place_dicho(11.5, [10,11,12])
Out[22]: 2
"""
```

7 Exercice 4 : tri par bulles :

```
def tri_bulle(t):
    '''Tri par bulles de t en place'''
    n = len(t)
    #n - 1 passages nécessaires dans le pire des cas
    for i in range(n - 1):
        #on fait remonter les bulles/éléments les plus grand(es)
        #de la première position d'index 0 jusqu'à la dernière position non triée n -1 - i
        #à chaque passage le plus grand élément trié remonte jusqu'à sa position n - 1 - i
        for j in range(n - 1 - i):
```

```
            if t[j] > t[j + 1]:
                t[j], t[j + 1]= t[j+1], t[j]

def tri_bulle2(t):
    '''Tri par bulles de t en place'''
    change = True
    while change:
        change = False
        for i in range(len(t) - 1):
            if t[i] > t[i + 1]:
                t[i], t[i + 1] = t[i + 1], t[i]
                change = True

def tri_bulle3(t):
    '''Tri par bulle amélioré'''
    change = True
    #compteur de passages
    c = 0
    while change:
        change = False
        k = 0
        #à chaque passage on doit balayer les positions de 0 à len(t) - 2
        #moins les c  dernières positions où sont déjà classés les + grands
        #il reste donc len(t) - c  positions à balayer
        while  k < len(t) - c:
            if t[k] > t[k+1]:
                t[k], t[k+1] = t[k+1], t[k]
                # on modifie le booleen change uniquement au premier mouvement
                if not change:
                    change = True
            k += 1
        c += 1

"""
In [8]: test_tri(tri_bulle, BENCH1)
Out[8]: [True, True, True]

In [9]: BENCH1[0]
Out[9]: [2, 7, 8, 6, 7, 16, 6, 4, 18, 14, 20]

In [10]: test_tri(tri_bulle2, BENCH1)
Out[10]: [True, True, True]
"""
```

Analyse Complexité du tri par bulles : (hors programme)

- Si le tableau est déjà *trié dans l'ordre croissant*, il n'y a aucune permutation d'éléments adjacents lors du premier passage. Le booleen `change` reste à `False` et la boucle `while` externe se termine au bout du premier tour. Il y a eu $n-1$ comparaisons, il s'agit d'une **complexité linéaire**.

- Si le tableau est *trié dans l'ordre décroissant*, lors de l'itération k de la boucle externe while (pour k allant de 1 à n), on fait remonter la plus grosse bulle non triée de la position 0 où elle est tombée après remontée des $k-1$ bulles plus grosses, jusqu'à sa position finale $n-k$. Lors de cette remontée, $n-k$ comparaisons/permutations sont effectuées sauf pour le dernier passage. Pour $k=1$ on a $n-1$ comparaisons/permutations , pour $k=2$ on a $n-2$ comparaisons/permutations, pour $k=n-1$ on a 1 comparaison et 1 permutation, pour $k=n$ on a 1 comparaison et 0 permutation.

  Au total, $n-1+n-2+\cdots+1+1 = n(n-1)/2+1$comparaisons sont effectuées. Il s'agit donc d'une **complexité quadratique** dans ce cas.