

Tris

March 3, 2022

1 Préliminaires

Dans ce TP on va revenir sur le tri par sélection et le compléter avec trois autres manières de trier des éléments : le tri par insertion, le tri fusion (merge sort) et le tri rapide (quicksort). Vous pouvez voir ces tris *en action* en allant sur le site : <https://visualgo.net/en/sorting>.

```
[3]: def swap(t,i,j):  
    '''Échange les valeurs dans le tableau t situées aux indices i et j'''  
    # BEGIN SOLUTION  
    t[i], t[j] = t[j], t[i]  
    # END SOLUTION
```

```
[2]: from civ import *  
init()  
t = [1, 2, 3]  
test(swap, t, 0, 2, None)  
assert(t == [3,2,1])  
valide()
```

OK!: swap([1, 2, 3], 0, 2) vaut None

Tous les tests sont valides

Conseil : n'hésitez pas à faire des print(t) dans les boucles pour observer l'effet d'une itération sur le tableau t.

2 Tri par sélection

On rappelle le principe du tri par selection : on cherche le minimum des non triés, on le place a la bonne position, on itère.

```
[3]: def indice_minimum_suffixe(t, i):  
    '''Renvoie le plus petit indice j >= i tel que  
    t[j] soit le minimum de t[i], t[i+1], ..., t[n-1] où len(t) = n'''  
    # BEGIN SOLUTION  
    m = i  
    for j in range(i+1, len(t)):  
        if t[j] < t[m]:  
            m = j
```

```
return m
# END SOLUTION
```

```
[4]: from civ import *
init()
test(indice_minimum_suffixe, [6,1,1,1,3], 0, 1)
test(indice_minimum_suffixe, [6,1,1,1,3], 1, 1)
test(indice_minimum_suffixe, [6,1,1,1,3], 2, 2)
test(indice_minimum_suffixe, [6,1,1,1,3], 4, 4)
test(indice_minimum_suffixe, ['b','a','c'], 0, 1)
valide()
```

```
OK!: indice_minimum_suffixe([6, 1, 1, 1, 3], 0) vaut 1
OK!: indice_minimum_suffixe([6, 1, 1, 1, 3], 1) vaut 1
OK!: indice_minimum_suffixe([6, 1, 1, 1, 3], 2) vaut 2
OK!: indice_minimum_suffixe([6, 1, 1, 1, 3], 4) vaut 4
OK!: indice_minimum_suffixe(['b', 'a', 'c'], 0) vaut 1
Tous les tests sont valides
```

```
[5]: def tri_par_selection(t):
# BEGIN SOLUTION
for i in range(len(t)):
    j = indice_minimum_suffixe(t, i)
    swap(t, i, j)
# END SOLUTION
```

```
[6]: from civ import *
init()
t = [6,2,1,4,6]
test_condition(tri_par_selection, condition_trie, t)
valide()
```

```
OK!: tri_par_selection([6, 2, 1, 4, 6]) a fait ce qu'on attendait
Tous les tests sont valides
```

3 Tri par insertion

Le tri par insertion d'un tableau t de n éléments consiste à rechercher itérativement la place d'insertion de l'élément d'indice i dans les éléments d'indice 0 à i sachant que les éléments d'indice 0 à $i - 1$ sont triés. Une fois cette place déterminée, on procède par échanges successifs pour que le tableau des éléments d'indice 0 à i soit trié.

On commence en considérant que l'élément d'indice 0 est, à lui tout seul, un tableau trié, et on procède ainsi pour insérer les éléments d'indice 1 à $n-1$.

Optionnel Pour la prochaine fonction on pourra faire en sorte d'avoir une complexité logarithmique en ayant une approche dichotomique : on prend la case du milieu entre 0 et $i-1$, si elle est $< x$, on va chercher la position à gauche, si elle est $\geq x$ on va chercher la position à droite. En procédant ainsi,

on peut faire en sorte que cette recherche de la position d'insertion soit en complexité logarithmique, plutôt que linéaire, sur le nombre de **comparaisons**. Cette méthode sera alors programmée avec deux indices $a \leq b$ déterminant le sous-tableau dans lequel s'effectue la recherche. Quand $a = b$, on a trouvé la place d'insertion.

```
[2]: def position_insertion(t, i, x):
    '''renvoie la position d'insertion de x dans le tableau t
    considéré trié entre les indices 0 et i-1.
    Si x est plus grand que toutes les autres valeurs, on renverra i'''
    # BEGIN SOLUTION
    a = 0
    b = i
    while b > a:
        m = (a+b)//2
        if t[m] < x:
            a = m+1
        else:
            b = m
    return a
    # END SOLUTION
```

```
[4]: from civ import *
init()
test(position_insertion, [1,1,1], 3, 1, 0)
t = [1,2,5,7,8,9,10]
test(position_insertion, t, 4, 3, 2)
test(position_insertion, t, 4, 0, 0)
test(position_insertion, t, 4, 1, 0)
test(position_insertion, t, 4, 9, 4)
valide()
```

OK!: position_insertion([1, 1, 1], 3, 1) vaut 0
 OK!: position_insertion([1, 2, 5, 7, 8, 9, 10], 4, 3) vaut 2
 OK!: position_insertion([1, 2, 5, 7, 8, 9, 10], 4, 0) vaut 0
 OK!: position_insertion([1, 2, 5, 7, 8, 9, 10], 4, 1) vaut 0
 OK!: position_insertion([1, 2, 5, 7, 8, 9, 10], 4, 9) vaut 4
 Tous les tests sont valides

```
[6]: def decale(t, i, j):
    ''' place la valeur à l'indice i à l'indice i + 1, celle
    d'indice i + 1 à l'indice i+2, .., celle d'indice j à l'indice j+1.
    Cela revient à décaler vers la droite les valeurs entre les indices i et j
    ce qui permet de libérer la place i'''
    # BEGIN SOLUTION
    for k in reversed(range(i,j+1)):
        t[k+1] = t[k]
    # END SOLUTION
```

```
[7]: from civ import *
init()
test_condition(decale, lambda t,i,j: t == [0,1,2,2,3,4,6],
               [0,1,2,3,4,5,6], 2, 4)
valide()
```

OK!: `decale([0, 1, 2, 3, 4, 5, 6], 2, 4)` a fait ce qu'on attendait
Tous les tests sont valides

```
[11]: def tri_par_insertion(t):
        # BEGIN SOLUTION
        for i in range(1,len(t)):
            v = t[i]
            j = position_insertion(t, i, v)
            decale(t, j, i-1)
            t[j] = v
        # END SOLUTION
```

```
[12]: from civ import *
init()
t = [6,2,1,4,6]
test_condition(tri_par_insertion, condition_trie, t)
valide()
```

OK!: `tri_par_insertion([6, 2, 1, 4, 6])` a fait ce qu'on attendait
Tous les tests sont valides

4 Tri fusion

L'idée du tri fusion est de procéder récursivement pour trier un tableau `t`. Ce tri n'est pas **en place** c'est-à-dire que le fait de trier va renvoyer un **nouveau** tableau `t`. On procède ainsi :

- si `len(t) < 2`, alors `t` est déjà trié
- sinon, on considère `t0` et `t1` les sous-tableaux des éléments d'indices respectivement pairs et impairs, on trie ces sous-tableaux récursivement et on obtient `t0'` et `t1'`. On crée alors le tableau trié déduit de `t` en **fusionnant** les deux sous-tableaux triés. Pour cela, on peut se servir du fait qu'ils sont triés. Par exemple, le premier élément de `t'` est soit le premier élément de `t0'` soit le premier élément de `t1'`.

Note : faites un exemple à la main pour fusionner `[1,4,5]` et `[2,3,6]`.

```
[4]: def separe(t):
        '''Prend en entrée un tableau t et renvoie
        deux tableaux t1 = [t[0], t[2], t[4], ...]
        et t2 = [t[1], t[3], t[5], ...]
        '''
        # BEGIN SOLUTION
        t1 = []
```

```

t2 = []
for i in range(len(t)):
    if i % 2 == 0:
        t1.append(t[i])
    else:
        t2.append(t[i])
return t1, t2
# END SOLUTION

```

```

[5]: from civ import *
init()
test(separe, [1,'a',2,'b',3], ([1,2,3], ['a','b']))
valide()

```

OK!: separe([1, 'a', 2, 'b', 3]) vaut ([1, 2, 3], ['a', 'b'])
Tous les tests sont valides

Pour la question suivante. Idée : on pourra faire un for i entre 0 et n1+n2-1 (où n1 et n2 sont les tailles resp. de t1 et t2) et garder deux indices a et b sur le plus petit element de t1 et de t2 non encore ajoutés dans t. A chaque i, on choisit l'élément voulu qu'on place dans t.

```

[8]: def fusionne(t1, t2):
    '''Prend en entrée deux tableaux t1 et t2 **triés**
    et renvoie le tableau t fusion de ces deux tableaux :
    donc t est trié et il contient les mêmes éléments que t1 et t2
    réunis'''
    # BEGIN SOLUTION
    t = []
    a, b = 0, 0
    n1, n2 = len(t1), len(t2)
    for i in range(n1+n2):
        if a < n1 and (b == n2 or t2[b] > t1[a]):
            t.append(t1[a])
            a = a + 1
        else:
            t.append(t2[b])
            b = b + 1
    return t
    # END SOLUTION

```

```

[10]: from civ import *
init()
test(fusionne, [1,5,6], [1,2,3,6,7],[1,1,2,3,5,6,6,7])
valide()

```

OK!: fusionne([1, 5, 6], [1, 2, 3, 6, 7]) vaut [1, 1, 2, 3, 5, 6, 6, 7]
Tous les tests sont valides

```
[13]: def tri_fusion(t):
    '''Tri en utilisant les deux fonctions précédentes et des appels
    récurifs au tri fusion.
    On renvoie le tableau trié, ON NE MODIFIE PAS t'''
    # BEGIN SOLUTION
    if len(t) == []:
        return []
    if len(t) == 1:
        return [t[0]]
    t1, t2 = separe(t)
    return fusionne(tri_fusion(t1), tri_fusion(t2))
    # END SOLUTION
```

```
[15]: from civ import *
init()
test(tri_fusion, [6,2,1,4,6], [1,2,4,6,6,])
test(tri_fusion, ['b','c','e','f','a','b'], ['a','b','b','c','e','f'])
valide()
```

OK!: tri_fusion([6, 2, 1, 4, 6]) vaut [1, 2, 4, 6, 6]

OK!: tri_fusion(['b', 'c', 'e', 'f', 'a', 'b']) vaut ['a', 'b', 'b', 'c', 'e', 'f']

Tous les tests sont valides

5 Tri rapide

Le tri rapide est un tri également **récurif** mais qui peut se faire naturellement en place. On va gérer la récursivité en considérant des sous-tableaux de t entre les indices i et j . On note $t[i:j+1]$ comme en Python un tel sous-tableau.

Pour simplifier, on suppose ici que t est sans répétitions.

Pour trier $t[i:j+1]$ dans le cas où $j+1-i > 1$, on commence par isoler $p = t[j]$ et on va effectuer des échanges dans $t[i:j+1]$ pour ranger les éléments en trois catégories : $< p$, $= p$, $> p$. Vu qu'on est sans répétitions, la catégorie du milieu sera réduite à un élément. Par exemple, si on a $t = [5, 3, 1, 2, 7, 9, 6, 4]$ alors en partitionnant $t[0:7+1]$ on partitionne t en entier avec $p = 4$, le tableau deviendra par exemple $[3, 1, 2, 4, 5, 7, 9, 6]$. Si on partitionne $t[0:5+1]$ alors $p = 9$ et le sous-tableau est déjà partitionné.

```
[8]: def partitionne(t, i, j):
    '''si p = t[j], modifie le tableau t en place, et renvoie un entier k
    tel que pour i <= a < k => t[a] < p, t[k] = p et k < b <= j : t[b] > p'''
    # BEGIN SOLUTION
    p = t[j]
    a = i
    for k in range(i, j):
        if t[k] < p:
            swap(t, a, k)
```

```

        a = a + 1
    swap(t, a, j)
    return a
# END SOLUTION

```

```

[10]: from civ import *
init()
t = [5,3,1,2,7,9,6,4]
test(partitionne, t, 0, 7, 3)
print(t)
assert(all(t[k] < 4 for k in range(3)))
assert(t[3] == 4)
assert(all(t[k] > 4 for k in range(4,8)))
t = [5,3,1,2,7,9,6,4]
test(partitionne, t, 0, 5, 5)
print(t)
test(partitionne, t, 0, 3, 1)
print(t)
assert(all(t[k] < 2 for k in range(1)))
assert(t[1] == 2)
assert(all(t[k] > 2 for k in range(2,4)))
valide()

```

OK!: partitionne([5, 3, 1, 2, 7, 9, 6, 4], 0, 7) vaut 3

[3, 1, 2, 4, 7, 9, 6, 5]

OK!: partitionne([5, 3, 1, 2, 7, 9, 6, 4], 0, 5) vaut 5

[5, 3, 1, 2, 7, 9, 6, 4]

OK!: partitionne([5, 3, 1, 2, 7, 9, 6, 4], 0, 3) vaut 1

[1, 2, 5, 3, 7, 9, 6, 4]

Tous les tests sont valides

Pour trier avec le tri rapide, on va alors récursivement trier les sous-tableaux des catégories <p et >p après partitionnement du sous-tableau courant. On s'arrête naturellement quand $j + 1 - i \leq 1$ car le sous-tableau est trié (il contient 0 ou une valeur).

```

[36]: def quicksort(t, i, j):
    '''Effectue un tri rapide de t entre les indices i et j'''
    # BEGIN SOLUTION
    if i >= j:
        return
    k = partitionne(t, i, j)
    quicksort(t, i, k-1)
    quicksort(t, k+1, j)
    # END SOLUTION

```

```

[40]: from civ import *
init()
t = [6,2,1,4,7]

```

```
test_condition(quicksort, lambda t,i,j: condition_trie(t), t, 0, len(t)-1)
t = [2,4,6,1,3,7,5,9]
test_condition(quicksort, lambda t,i,j: condition_trie(t), t, 0, len(t)-1)
valide()
```

OK!: quicksort([6, 2, 1, 4, 7], 0, 4) a fait ce qu'on attendait

OK!: quicksort([2, 4, 6, 1, 3, 7, 5, 9], 0, 7) a fait ce qu'on attendait

Tous les tests sont valides