Tris

Le tri d'une liste est une opération courante en informatique. Il existe de nombreux algorithmes de tri : on en étudie ici six. Ils se distinguent les uns des autres par leur complexité asymptotique (en fonction de la longueur n de la liste) ainsi que par les caractéristiques suivantes.

On dit qu'un tri se fait **en place** s'il ne nécessite pas de création de nouvelle liste, il ne modifie que la liste à trier.

On dit qu'un tri est **stable** si, lorsqu'on trie une liste selon un paramètre, les éléments ayant la même valeur pour ce paramètre restent rangés dans l'ordre initial. Par exemple, si on dispose d'une liste d'élèves rangés par ordre alphabétique et de leurs notes à un devoir et qu'on les classe selon cette note, les élèves ayant la même note restent rangés par ordre alphabétique après le tri.

Noter que la méthode `sort` et la fonction `sorted` de Python utilisent une technique appelée tri par tas dont la complexité asymptotique est $O(n \ln n)$.

Tri par sélection

Le principe du tri par sélection est le suivant : on cherche le plus petit élément de la liste et on l'échange avec l'élément d'indice 0, puis on cherche le deuxième plus petit élément de la liste et on l'échange avec l'élément d'indice 1, et ainsi de suite.

Par exemple, avec la liste `[3, 7, 2, 6, 1, 3]` :

```
L = [3, 7, 2, 6, 1, 3]
Le minimum est L[4], on échange L[0] et L[4] : [1, 7, 2, 6, 3, 3]
Le minimum est L[2], on échange L[1] et L[2] : [1, 2, 7, 6, 3, 3]
Le minimum est L[4], on échange L[2] et L[4] : [1, 2, 3, 6, 7, 3]
Le minimum est L[5], on échange L[3] et L[5] : [1, 2, 3, 3, 7, 6]
Le minimum est L[4], on échange L[4] et L[5] : [1, 2, 3, 3, 6, 7]
```

On peut programmer ce tri en place de la manière suivante :

```
def tri_selection(L):
    n = len(L)
    for i in range(n):
        # On recherche l'indice du minimum à partir de i
        ind_min = i
        m = L[i]
        for j in range(i+1, n):
            if L[j] < m:
                ind_min = j
                m = L[j]
        # On échange L[i] et L[ind_min]
        L[ind_min], L[i] = L[i], L[ind_min]
```

Noter que cette fonction modifie la liste et ne renvoie rien.

La complexité asymptotique du tri par sélection est $O(n^2)$ dans tous les cas car on fait toujours $(n-1)+(n-2)+\ldots+1 = \dfrac{n(n-1)}{2}$ comparaisons. Le nombre d'affectations dépend de l'ordre initial des éléments mais ne modifie pas la complexité.

Tri par insertion

L'idée est de prendre chaque élément de la liste l'un après l'autre et de le mettre à sa place dans la partie de la liste qui le précède.

Par exemple, avec la liste `[3, 7, 2, 6, 1, 3]` :

```
L = [3, 7, 2, 6, 1, 3]
On met L[1] = 7 à sa place :
[3, 7, 2, 6, 1, 3]  (pas de changement car 3 <= 7)
On met L[2] = 2 à sa place :
[3, 7, 2, 6, 1, 3] -> [3, 2, 7, 6, 1, 3] -> [2, 3, 7, 6, 1, 3]
On met L[3] = 6 à sa place :
[2, 3, 7, 6, 1, 3] -> [2, 3, 6, 7, 1, 3]
On met L[4] = 1 à sa place :
[2, 3, 6, 7, 1, 3] -> [2, 3, 6, 1, 7, 3] -> [2, 3, 1, 6, 7, 3]
-> [2, 1, 3, 6, 7, 3] -> [1, 2, 3, 6, 7, 3]
On met L[5] = 3 à sa place :
[1, 2, 3, 6, 7, 3] -> [1, 2, 3, 6, 3, 7] -> [1, 2, 3, 3, 6, 7]
```

On peut procéder par échanges successifs comme ci-dessus, ou mémoriser l'élément à placer et décaler les précédents jusqu'à ce qu'on ait trouvé l'endroit où on doit l'insérer. On peut ainsi programmer ce tri en place d'une des manières suivantes :

```
def tri_insertion(L):
    for i in range(1, len(L)):
        x = L[i]
        j = i
        while j > 0 and x < L[j-1]:
            L[j] = L[j-1]
            j = j - 1
        L[j] = x
```

```
def tri_insertion(L):
    for i in range(1, len(L)):
        j = i
        while j > 0 and L[j] < L[j-1]:
            L[j], L[j-1] = L[j-1], L[j]
            j = j - 1
```

Dans le meilleur des cas, la liste est déjà triée : il n'y a alors aucun décalage à faire et l'algorithme est en $O(n)$. Dans le pire des cas la liste est strictement décroissante et il y a $1+2+\ldots+(n-1) = \dfrac{n(n-1)}{2}$ décalages à effectuer : l'algorithme est en $O(n^2)$.

Le tri par insertion est particulièrement intéressant lorsque la liste est presque triée.

Tri à bulles

L'idée est de parcourir plusieurs fois la liste et de permuter deux éléments consécutifs s'ils ne sont pas dans le bon ordre.

Par exemple, avec la liste `[3, 7, 2, 6, 1, 3]` :

```
Premier passage :
On intervertit le 7 et le 2 : [3, 2, 7, 6, 1, 3]
On intervertit le 7 et le 6 : [3, 2, 6, 7, 1, 3]
On intervertit le 7 et le 1 : [3, 2, 6, 1, 7, 3]
On intervertit le 7 et le 3 : [3, 2, 6, 1, 3, 7]
Deuxième passage :
On intervertit le 3 et le 2 : [2, 3, 6, 1, 3, 7]
On intervertit le 6 et le 1 : [2, 3, 1, 6, 3, 7]
On intervertit le 6 et le 3 : [2, 3, 1, 3, 6, 7]
Troisième passage :
On intervertit le 3 et le 1 : [2, 1, 3, 3, 6, 7]
Quatrième passage :
On intervertit le 2 et le 1 : [1, 2, 3, 3, 6, 7]
```

On peut programmer le tri à bulles de manière très simple comme suit :

```
def tri_bulles(L):
    n = len(L)
    for i in range(n):
        for j in range(n-1):
            if L[j] > L[j+1]:
                L[j+1], L[j] = L[j], L[j+1]
```

Sa complexité asymptotique est alors clairement $O(n^2)$ dans tous les cas.

On peut améliorer l'efficacité de la fonction en remarquant qu'après le premier passage le plus grand élément se retrouve toujours en dernière position, après le deuxième passage le deuxième plus grand élément se retrouve toujours en avant-dernière position, etc. On peut aussi remarquer que si, lors d'un passage, rien n'a été modifié, c'est que la liste est triée donc on peut s'arrêter. On obtient ainsi la fonction suivante :

```
def tri_bulles(L):
    for i in range(len(L)-1, 0, -1):
        est_trie = True
        for j in range(i):
            if L[j] > L[j+1]:
                L[j+1], L[j] = L[j], L[j+1]
                est_trie = False
        if est_trie:
            break
```

Dans le meilleur des cas (liste déjà triée) on ne fait alors que $n-1$ comparaisons et aucun échange donc la complexité est $O(n)$. Dans le pire des cas (liste strictement décroissante) on fait $(n-1)+(n-2)+\ldots+1=\dfrac{n(n-1)}{2}$ comparaisons et autant d'échanges donc la complexité asymptotique est $O(n^2)$.

Tri fusion

Le tri fusion est un algorithme récursif qui s'appuie sur une stratégie de type **diviser pour régner**. Si la liste est vide ou de longueur 1, il n'y rien à faire. Sinon, on partitionne la liste en deux sous-listes de même longueur (à une unité près), on trie (récursivement) chacune des deux sous-listes et on les fusionne.

Par exemple, avec la liste `[3, 7, 2, 6, 1, 3]` :

```
On trie [3, 7, 2, 6, 1, 3]
On partitionne : [3, 7, 2] [6, 1, 3]
On trie [3, 7, 2]
On partitionne : [3] [7, 2]
On trie [3]
On trie [7, 2]
On partitionne : [7] [2]
On trie [7]
On trie [2]
On fusionne [7] et [2] -> [2, 7]
On fusionne [3] et [2, 7] -> [2, 3, 7]
On trie [6, 1, 3]
On partitionne : [6] [1, 3]
On trie [6]
On trie [1, 3]
On partitionne : [1] [3]
On trie [1]
On trie [3]
On fusionne [1] et [3] -> [1, 3]
On fusionne [6] et [1, 3] -> [1, 3, 6]
On fusionne [2, 3, 7] et [1, 3, 6] -> [1, 2, 3, 3, 6, 7]
```

La fonction `fusion`, qui reçoit deux listes triées et renvoie leur fusion (i.e. la liste triée des éléments des deux listes), peut se programmer comme suit :

```
def fusion(L1, L2):
    L = []
    i1 = i2 = 0
    while True:
        if i1 < len(L1) and (i2 >= len(L2) or L1[i1] < L2[i2]):
            L.append(L1[i1])
            i1 += 1
        elif i2 < len(L2):
            L.append(L2[i2])
            i2 += 1
        else:
            return L
```

La fonction `tri_fusion` est alors très simple à écrire. Noter que cette fonction ne modifie pas la liste initiale, elle renvoie une nouvelle liste.

```
def tri_fusion(L):
    if len(L) <= 1:
        return L
    m = len(L) // 2
    return fusion(tri_fusion(L[:m]), tri_fusion(L[m:]))
```

On peut montrer que la complexité asymptotique du tri fusion est $O(n \ln n)$.

Tri rapide

Le tri rapide est également un algorithme récursif qui s'appuie sur une stratégie de type **diviser pour régner**. Soit `L` une liste non vide. On prend `p = L[0]` (le pivot), on crée la liste `L1` des éléments de `L` plus petits (au sens large) que le pivot (sans le pivot lui-même) et la liste `L2` des éléments de `L` plus grands (au sens strict) que le pivot, on les trie (récursivement) et on renvoie la concaténation de `L1` triée, `[p]` et `L2` triée.

Par exemple, avec la liste `[3, 7, 2, 6, 1, 3]` :

```
On trie [3, 7, 2, 6, 1, 3]
Pivot 3, première liste [2, 1, 3], deuxième liste [7, 6]
On trie [2, 1, 3]
Pivot 2, première liste [1], deuxième liste [3]
On trie [1]
On trie [3]
Résultat du tri : [1] + [2] + [3] -> [1, 2, 3]
On trie [7, 6]
Pivot 7, première liste [6], deuxième liste []
On trie [6]
On trie []
Résultat du tri : [6] + [7] + [] -> [6, 7]
Résultat du tri : [1, 2, 3] + [3] + [6, 7] -> [1, 2, 3, 3, 6, 7]
```

On peut programmer le tri rapide de la manière suivante (cette fonction ne modifie pas la liste initiale, elle renvoie une nouvelle liste) :

```
def tri_rapide(L):
    if len(L) <= 1:
        return L
    pivot = L[0]
    L1, L2 = [], []
    for x in L[1:]:
        if x <= pivot:
            L1.append(x)
        else:
            L2.append(x)
    return tri_rapide(L1) + [pivot] + tri_rapide(L2)
```

On peut montrer que la complexité asymptotique du tri rapide est $O(n \ln n)$ en moyenne, mais dans le pire des cas (liste triée ou strictement décroissante) ce tri est en $O(n^2)$. La fonction précédente peut même faire déborder la pile des appels récursifs :

```
>>> L = list(range(1000))
>>> tri_rapide(L)
RecursionError: maximum recursion depth exceeded in comparison
```

Il est possible d'améliorer les performances en choisissant aléatoirement le pivot plutôt que de prendre systématiquement le premier élément de la liste.

Tri par comptage

Si la liste est formée d'éléments appartenant tous à un ensemble fini connu et pas trop gros (par exemple des entiers compris entre deux valeurs a et b fixées) on peut la trier très rapidement en comptant simplement le nombre d'occurrences de chaque valeur. Par exemple, la liste `[3, 7, 2, 6, 1, 3]` contient un 1, un 2, deux 3, un 6 et un 7, ce qui donne `[1, 2, 3, 3, 6, 7]`.

Ce tri par comptage (ou par dénombrement) se programme efficacement avec un tableau (ou un dictionnaire) d'occurrences. La fonction suivante reçoit une liste d'entiers et les valeurs minimale et maximale que peuvent prendre ces entiers :

```
def tri_comptage(L, a, b):
    occurrences = [0] * (b-a+1)
    for k in L:
        occurrences[k-a] += 1
    res = []
    for k in range(a, b+1):
        for i in range(occurrences[k-a]):
            res.append(k)
    return res
```

La fonction est en $O(n)$ dans tous les cas. On peut aussi calculer le minimum et le maximum de la liste au début plutôt que de donner a et b.

Comparaisons des différents algorithmes

Avec la fonction `time` du module du même nom, on peut tester les fonctions précédentes sur différents types de listes.

```
from time import time

fonctions = [tri_selection, tri_insertion, tri_bulles, tri_fusion, tri_rapide,
             tri_comptage, sorted]
noms = ['Tri sélection', 'Tri insertion', 'Tri à bulles', 'Tri fusion', 'Tri rapide',
        'Tri par comptage', 'Fonction sorted de Python']

def test(L):
    for i in range(7):
        f = fonctions[i]
        copie = L.copy()
        t = time()
        try:
            f(copie)
            print(noms[i], time()-t)
        except:
            print(noms[i], 'échec')
```

`try ... except ...` permet d'éviter d'arrêter le programme en cas d'erreur.

Avec une liste aléatoire de 10000 entiers compris entre 0 et 10000 :

```
>>> from random import randint
>>> L = [randint(0, 10000) for i in range(10000)]
>>> test(L)
Tri sélection 3.4049994945526123
Tri insertion 6.138937950134277
Tri à bulles 11.13805603981018
Tri fusion 0.06251978874206543
Tri rapide 0.031012535095214844
Tri par comptage 0.015619516372680664
Fonction sorted de Python 0.0
```

Avec la liste (triée) des entiers de 0 à 9999 :

```
>>> L = list(range(10000))
>>> test(L)
Tri sélection 3.2491981983184814
Tri insertion 0.0
Tri à bulles 0.0
Tri fusion 0.06249189376831055
Tri rapide échec
Tri par comptage 0.0
Fonction sorted de Python 0.0
```

Avec la liste triée par ordre décroissant des entiers de 0 à 9999 :

```
>>> L = list(range(10000))[::-1]
>>> test(L)
Tri sélection 4.6239118576049805
Tri insertion 12.215895414352417
Tri à bulles 16.105352878570557
Tri fusion 0.06226205825805664
Tri rapide échec
Tri par comptage 0.015662670135498047
Fonction sorted de Python 0.0
```

On peut aussi regarder ce qui se passe quand on applique ces algorithmes à des listes de petite taille. Par exemple en triant 100000 fois la liste `[3, 7, 2, 6, 1, 3]` on obtient :

```
Tri sélection 0.499922513961792
Tri insertion 0.3593306541442871
Tri à bulles 0.5779068470001221
Tri fusion 1.1557579040527344
Tri rapide 0.49979209899902344
Tri par comptage 0.5155041217803955
Fonction sorted de Python 0.06252193450927734
```

En triant 10000 fois une liste aléatoire de 100 entiers compris entre 0 et 10000 :

```
Tri sélection 3.4522736072540283
Tri insertion 5.967258453369141
Tri à bulles 11.528481721878052
Tri fusion 3.874093770980835
Tri rapide 1.4994158744812012
Tri par comptage 29.258481979370117
Fonction sorted de Python 0.12496352195739746
```

Exercices

1) Parmi tous ces algorithmes de tri, lesquels sont stables ?

2) Démontrer la terminaison et la correction des différents algorithmes étudiés.