Tri rapide corrige

December 20, 2020

1 Principe

Le tri rapide est basée sur un élément de la liste appelé pivot. Cet élément pivot a deux fonctions : - il est placé à sa place finale dans la liste à chaque appel. - il divise la liste en deux sous listes qui sont triées de manière récursive par la même fonction.

On commence par choisir au hasard un élément pivot p par exemple le premier élément de la liste initiale L avec p = L[0].

On a donc pour une liste de taille n, une liste à trier L = [p, L[1], L[2], ... , L[n-1]].

Le pivot se trouve à sa place finale quand tous les éléments qui le précédent lui sont inférieurs, et tous les éléments qui le succèdent lui sont supérieurs. Pour arriver à ce résultat, on compare p avec tous les autres éléments L[i] allant de L[1] jusqu'à L[n-1]. Si L[i] est inférieur à p, alors on le déplace avant p, si L[i] est supérieur à p, alors on le laisse après p.

On a donc une liste L = [L[a], L[b], ... , p, L[α], L[β], ...] avec tous les éléments L[a], L[b], ... $\leq$ p $\leq$ L[α], L[β], ...

Cette liste peut être séparée en deux autour du pivot comme L = [L_1, p, L_2], comme tous les éléments de L_1 sont inférieur à p qui est inférieur à tous les éléments de L_2, il n'y a pas de fusion à faire L_1, p, et L_2 sont bien ordonnées. Par contre il faut trier L_1 et L_2, pour cela la fonction de tri rapide s'appelle elle même pour trier L_1 et L_2.

On réalise un appel récursif, il faut donc prévoir un cas d'arrêt. Ici si L est de taille 1 ou 0, on a rien à faire la liste est déjà triée ou vide.

Il faut aussi prévoir la terminaison de l'algorithme. Ici la taille de la liste à trier décroit d'au moins 1 à chaque appel récursif jusqu'à atteindre le cas d'arrêt 0 ou 1. En effet le pivot est retiré de la liste à trier à chaque appel ce qui diminue la taille de la liste restante d'au moins 1. Et la taille n d'une liste est un entier positif donc on atteint bien le cas d'arrêt et en un nombre fini d'appel.

2 Exemple de tri rapide

Déroulons les opérations sucessives du tri rapide sur un exemple $L = [14,\ 1,\ 4,\ 3]$:

- on choisit le pivot p = L[0] = 14
- on compare p et L[1] = 1, 1 < 14, donc on réarrangera L en [1, 14, ...]
- on compare p et L[2] = 4, 4 < 14 donc on réarrangera L en [1, 4, 14, ...]
- on compare p et L[3] = 3, 3 < 14 donc on réarrangera L en [1, 4, 3, 14]
- on obtient L = [1, 4, 3, 14], on divise L en [L_1, p, L_2] avec L_1 = [1, 4, 3] et L_2 = []
- L_2 est vide donc est déjà triée c'est le cas d'arrêt
- on choisit le pivot p_1 = L_1[0] = 1

- on compare p_1 et L_1[1] = 4, 1 < 4 donc on laissera [1, 4, . . .]
- on compare p_1 et L_1[2] = 3, 1 < 3 donc on laissera [1, 4, 3]
- on obtient L_1 = [1, 4, 3], on divise L_1 en [L_{11}, p_1, L_{12}] avec L_{11} = [] et L_{12} = [4, 3]
- L_{11} est vide donc déjà triée c'est le cas d'arrêt
- on choisit le pivot p_{12} = L_{12}[0] = 4
- on compare p_{12} et L_{12}[1] = 3, 3 < 4 donc on réangerra L_{12} en [3, 4]
- on obtient L_{12} = [3, 4], on divise L_{12} en [L_{121}, p_{12}, L_{122}] avec L_{121} = [3] et L_{122} = []
- L_{122} est vide c'est le cas d'arrêt et L_{121} est déjà triée donc c'est le deuxième cas d'arrêt.
- on dépile la pile d'appel récursif:
  - L_{121} = [3], p_{12} = 4, L_{122} = [] donc L_{12} = [3, 4]
  - L_{11} = [], p_1 = 1, L_{12} = [3, 4] donc L_1 = [1, 3, 4]
  - L_1 = [1, 3, 4], p = 14, L_2 = [] donc L = [1, 3, 4, 14]

3 Réalisation

Ecrivons l'algorithme de tri en le décomposant en chaque fonction élémentaire.

La première fonction consiste à choisir et placer le pivot dans la portion de liste entre les indices debut et fin.

Cette fonction renvoie un tuple avec la nouvelle position du pivot et la liste avec le pivot positionné.

[1]:
```
def positionnement_pivot(L,debut,fin):
    p = L[debut]
    j = debut
    for i in range(debut,fin):
        if p > L[i] :
            L = L[:j]+[L[i]]+L[j:i]+L[i+1:]
            j +=1

    return j,L
```

On peut par exemple positionner le pivot de $L = [14,\ 1,\ 4,\ 3]$ avec:

[2]:
```
L = [14, 1, 4, 3]
positionnement_pivot(L,0,len(L))
```

[2]:
```
(3, [1, 4, 3, 14])
```

La deuxième fonction constitue l'appel récursif du tri rapide.

Elle effectue le tri rapide complet sur la portion de tableau allant de debut à fin.

[3]:
```
def tri_rapide_rec(L,debut,fin):
    if debut+1>=fin :
        return L
    else :
        j,L = positionnement_pivot(L,debut,fin)
        debut_1 = debut
        fin_1 = j
```

```
        L = tri_rapide_rec(L,debut_1,fin_1)
        debut_2 = j+1
        fin_2 = fin
        L = tri_rapide_rec(L,debut_2,fin_2)
        return L
```

On peut par exemple l'appeler pour [14, 1, 4, 3] entre 0 et 4

[4]:
```
L = [14, 1, 4, 3]
tri_rapide_rec(L,0,4)
```

[4]:
```
[1, 3, 4, 14]
```

La fonction tri rapide est donc simplement l'exécution de l'appel récursif sur l'ensemble du tableau

[5]:
```
def tri_rapide(L) :
    L = tri_rapide_rec(L,0,len(L))
    return L
```

[6]:
```
L = [14, 1, 4, 3]
tri_rapide(L)
```

[6]:
```
[1, 3, 4, 14]
```

par exemple on peut visualiser les différentes étapes du tri de la liste [14 1 4 3] avec les lignes de commandes suivantes

[7]:
```
def positionnement_pivot_visualisation(L,debut,fin):
    p = L[debut]
    j = debut
    for i in range(debut,fin):
        if p > L[i] :
            L = L[:j]+[L[i]]+L[j:i]+L[i+1:]
            j +=1
        print(L)
    return j,L

def tri_rapide_rec_visualisation(L,debut,fin):
    if debut+1>=fin :
        return L
    else :
        j,L = positionnement_pivot_visualisation(L,debut,fin)
        debut_1 = debut
        fin_1 = j
        L = tri_rapide_rec_visualisation(L,debut_1,fin_1)
        debut_2 = j+1
        fin_2 = fin
        L = tri_rapide_rec_visualisation(L,debut_2,fin_2)
        return L

def tri_rapide_visualisation(L) :
```

```
    L = tri_rapide_rec_visualisation(L,0,len(L))
    return L

L = [14, 1, 4, 3]
tri_rapide_visualisation(L)
```

```
[14, 1, 4, 3]
[1, 14, 4, 3]
[1, 4, 14, 3]
[1, 4, 3, 14]
[1, 4, 3, 14]
[1, 4, 3, 14]
[1, 4, 3, 14]
[1, 4, 3, 14]
[1, 3, 4, 14]
```

[7]: `[1, 3, 4, 14]`

4 Complexité temporelle

Le positionnement du pivot dépend de la liste initiale, on se retrouve à nouveau dans le cas d'une étude de la complexité qui peut varier selon que l'on est dans le pire des cas ou le meilleur des cas.

4.1 Calcul de la complexité dans le meilleur des cas

Le meilleur des cas est une liste pour laquelle le pivot se place au centre de la liste de taille n et la divise en deux listes de taille n/2.

Le nombre d'opérations à effectuer est alors de $n-1$ pour le positionnement du pivot puis toutes les opérations pour trier les deux listes de taille n/2, soit $2 \times C(n/2)$.

On a donc une relation de récurrence $C(n) = 2C(n/2) + n - 1$, comme pour le tri par fusion.

Dans le meilleur des cas, nous avons de même que pour le tri fusion une complexité $C(n) = O(n \log n)$, soit une complexité quasi-linéaire.

4.2 Calcul de la complexité dans le pire des cas

Le pire des cas est une liste pour laquelle le pivot se place en début ou en fin de la liste initiale et la divise en deux listes de taille 0 et n.

Le nombre d'opération à effectuer est alors de $n-1$ pour le positionnement du pivot puis toutes les opérations pour trier une liste de taille n-1, soit $n-1$.

On a donc une relation de récurrence $C(n) = n - 1 + C(n-1)$. Si on développe le calcul de la relation de récurrence $C(n) = n - 1 + C(n-1) = n - 1 + n - 2 + C(n-2) = n - 1 + n - 2 + n - 3 + ... + 1 + C(1) = \sum_{i=n-1}^{1} i = \frac{(n-1)(n-2)}{2}$.

Dans le pire des cas nous avons de même que pour le tri par insertion une complexité $C(n) = O(n^2)$, soit une complexité quadratique.

5 Exercices

5.1 exercice : meilleur et pire des cas

- Dérouler chaque étape de l'algorithme de tri rapide sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$

[8]:
```
L = [15, 4, 2, 9, 55, 16, 0, 1]
tri_rapide_visualisation(L)
```

```
[15, 4, 2, 9, 55, 16, 0, 1]
[4, 15, 2, 9, 55, 16, 0, 1]
[4, 2, 15, 9, 55, 16, 0, 1]
[4, 2, 9, 15, 55, 16, 0, 1]
[4, 2, 9, 15, 55, 16, 0, 1]
[4, 2, 9, 15, 55, 16, 0, 1]
[4, 2, 9, 0, 15, 55, 16, 1]
[4, 2, 9, 0, 1, 15, 55, 16]
[4, 2, 9, 0, 1, 15, 55, 16]
[2, 4, 9, 0, 1, 15, 55, 16]
[2, 4, 9, 0, 1, 15, 55, 16]
[2, 0, 4, 9, 1, 15, 55, 16]
[2, 0, 1, 4, 9, 15, 55, 16]
[2, 0, 1, 4, 9, 15, 55, 16]
[0, 2, 1, 4, 9, 15, 55, 16]
[0, 1, 2, 4, 9, 15, 55, 16]
[0, 1, 2, 4, 9, 15, 55, 16]
[0, 1, 2, 4, 9, 15, 55, 16]
[0, 1, 2, 4, 9, 15, 55, 16]
[0, 1, 2, 4, 9, 15, 16, 55]
```

[8]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

- Ecrire cette même liste où les éléments sont ordonnés dans le meilleur des cas et dérouler chaque étape de son tri

[9]:
```
L = [9, 2, 0, 1, 4, 16, 15, 55]
tri_rapide_visualisation(L)
```

```
[9, 2, 0, 1, 4, 16, 15, 55]
[2, 9, 0, 1, 4, 16, 15, 55]
[2, 0, 9, 1, 4, 16, 15, 55]
[2, 0, 1, 9, 4, 16, 15, 55]
[2, 0, 1, 4, 9, 16, 15, 55]
[2, 0, 1, 4, 9, 16, 15, 55]
[2, 0, 1, 4, 9, 16, 15, 55]
[2, 0, 1, 4, 9, 16, 15, 55]
[2, 0, 1, 4, 9, 16, 15, 55]
```

```
[0, 2, 1, 4, 9, 16, 15, 55]
[0, 1, 2, 4, 9, 16, 15, 55]
[0, 1, 2, 4, 9, 16, 15, 55]
[0, 1, 2, 4, 9, 16, 15, 55]
[0, 1, 2, 4, 9, 16, 15, 55]
[0, 1, 2, 4, 9, 16, 15, 55]
[0, 1, 2, 4, 9, 15, 16, 55]
[0, 1, 2, 4, 9, 15, 16, 55]
```

[9]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

- Ecrire cette même liste où les éléments sont ordonnés dans le pire des cas et dérouler chaque étape de son tri

[10]:
```
L = [55, 16, 15, 9, 4, 2, 1, 0]
tri_rapide_visualisation(L)
```

```
[55, 16, 15, 9, 4, 2, 1, 0]
[16, 55, 15, 9, 4, 2, 1, 0]
[16, 15, 55, 9, 4, 2, 1, 0]
[16, 15, 9, 55, 4, 2, 1, 0]
[16, 15, 9, 4, 55, 2, 1, 0]
[16, 15, 9, 4, 2, 55, 1, 0]
[16, 15, 9, 4, 2, 1, 55, 0]
[16, 15, 9, 4, 2, 1, 0, 55]
[16, 15, 9, 4, 2, 1, 0, 55]
[15, 16, 9, 4, 2, 1, 0, 55]
[15, 9, 16, 4, 2, 1, 0, 55]
[15, 9, 4, 16, 2, 1, 0, 55]
[15, 9, 4, 2, 16, 1, 0, 55]
[15, 9, 4, 2, 1, 16, 0, 55]
[15, 9, 4, 2, 1, 0, 16, 55]
[15, 9, 4, 2, 1, 0, 16, 55]
[9, 15, 4, 2, 1, 0, 16, 55]
[9, 4, 15, 2, 1, 0, 16, 55]
[9, 4, 2, 15, 1, 0, 16, 55]
[9, 4, 2, 1, 15, 0, 16, 55]
[9, 4, 2, 1, 0, 15, 16, 55]
[9, 4, 2, 1, 0, 15, 16, 55]
[4, 9, 2, 1, 0, 15, 16, 55]
[4, 2, 9, 1, 0, 15, 16, 55]
[4, 2, 1, 9, 0, 15, 16, 55]
[4, 2, 1, 0, 9, 15, 16, 55]
[4, 2, 1, 0, 9, 15, 16, 55]
[2, 4, 1, 0, 9, 15, 16, 55]
[2, 1, 4, 0, 9, 15, 16, 55]
[2, 1, 0, 4, 9, 15, 16, 55]
```

```
[2, 1, 0, 4, 9, 15, 16, 55]
[1, 2, 0, 4, 9, 15, 16, 55]
[1, 0, 2, 4, 9, 15, 16, 55]
[1, 0, 2, 4, 9, 15, 16, 55]
[0, 1, 2, 4, 9, 15, 16, 55]
```

[10]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

5.2 exercice : calcul numérique de la complexité temporelle

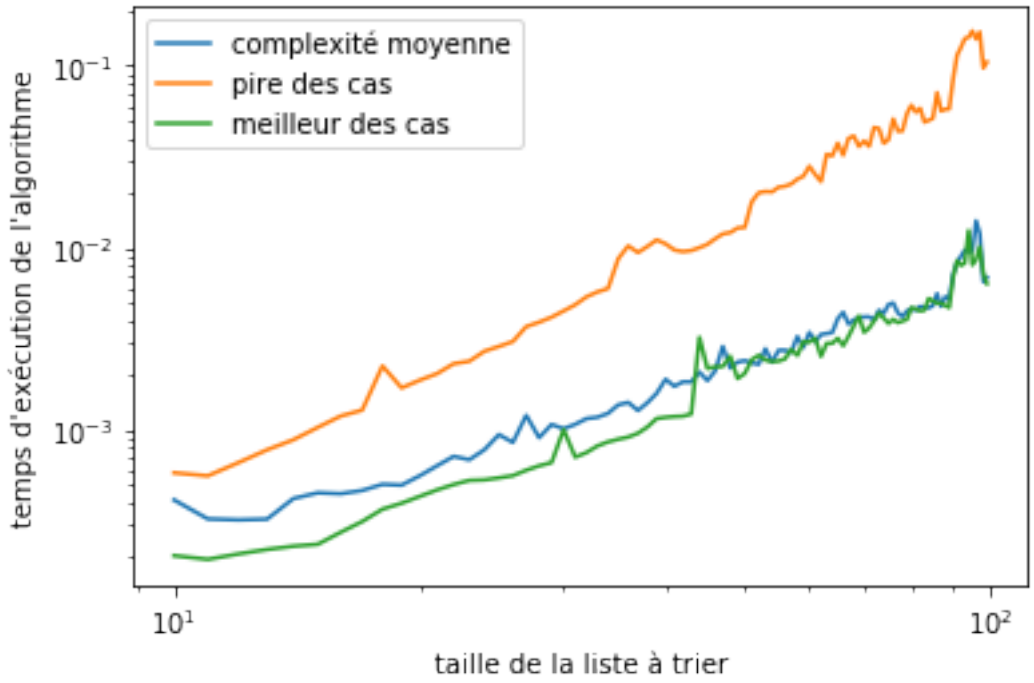
- à l'aide de tirage au sort successif de liste de différentes tailles, mesurer numériquement le temps mis pour l'exécution du tri rapide sur ces listes et tracer le temps d'éxécution en fonction de la taille des listes à trier.

[14]:
```
import time
import matplotlib.pyplot as plt
import numpy as np
import random as rd

def complexite_moyenne(n_min,n_max,pas,nbre_moyenne,algorithme):
    duree = []
    taille = []
    i = 0
    for n in range(n_min,n_max,pas):
        taille.append(n)
        duree.append(0)
        L=list(range(n))
        for m in range(nbre_moyenne):
            rd.shuffle(L)
            depart = time.clock()
            algorithme(L)
            arrive = time.clock()
            duree[i]=duree[i]+arrive-depart
        i += 1
    return taille, duree

def complexite_pire(n_min,n_max,pas,nbre_moyenne,algorithme):
    duree = []
    taille = []
    i = 0
    for n in range(n_min,n_max,pas):
        taille.append(n)
        duree.append(0)
        L=list(range(n))
        L = list(np.flip(L))
        for m in range(nbre_moyenne):
            depart = time.clock()
            algorithme(L)
```

```
            arrive = time.clock()
            duree[i]=duree[i]+arrive-depart
        i += 1
    return taille, duree

def meilleur_cas(L,L_prime,debut) : # algorithme à récursivité terminale pour construire ordonner
                                    #une liste depuis un ordre croissant vers le meilleur des cas
    if len(L) == 1:
        L_prime[debut] = L[0]
    elif len(L) > 1:
        n = len(L)
        L_prime[debut] = L[n//2]
        L1 = L[:n//2]
        L2 = L[n//2+1:]
        meilleur_cas(L1,L_prime,debut+1)
        meilleur_cas(L2,L_prime,debut+n//2+1)

def complexite_meilleur(n_min,n_max,pas,nbre_moyenne,algorithme):
    duree = []
    taille = []
    i = 0
    for n in range(n_min,n_max,pas):
        taille.append(n)
        duree.append(0)
        L=list(range(n))
        L_prime = len(L)*[None]
        meilleur_cas(L,L_prime,0)
        for m in range(nbre_moyenne):
            depart = time.clock()
            algorithme(L_prime)
            arrive = time.clock()
            duree[i]=duree[i]+arrive-depart
        i += 1
    return taille, duree

n_min,n_max,pas,nbre_moyenne = 10,100,1,10

taille_moyenne, duree_moyenne = complexite_moyenne(n_min,n_max,pas,nbre_moyenne,tri_rapide)
taille_pire, duree_pire = complexite_pire(n_min,n_max,pas,nbre_moyenne,tri_rapide)
taille_meilleur, duree_meilleur = complexite_meilleur(n_min,n_max,pas,nbre_moyenne,tri_rapide)

plt.loglog(taille_moyenne,duree_moyenne,'-')
```

```
plt.loglog(taille_pire,duree_pire,'-')
plt.loglog(taille_meilleur,duree_meilleur,'-')
plt.xlabel('taille de la liste à trier')
plt.ylabel("temps d'exécution de l'algorithme")
plt.legend(('complexité moyenne','pire des cas','meilleur des cas'), loc='upper␣
 ↪left')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:16: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
  app.launch_new_instance()
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:18: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:33: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:35: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:63: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:65: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



- est-ce cohérent avec le calcul de complexité effectué précédemment ?

La complexité dans le pire des cas est bien quadratique car on remarque une droite de pente 2 en échelle logarithmique pour le temps d'exécution de l'algorithme.

La complexité dans le meilleur des cas est bien quasi-linéaire, on a bien obtenu une droite de pente 1 en échelle logarithmique.

La complexité moyenne est bien comprise entre le pire et le meilleur des cas, et on remarque qu'elle a une complexité plus proche du meilleur des cas que du pire des cas.

5.3 exercice : complexité spatiale

- calculer la complexité spatiale d'un algorithme de tri rapide

Le tri rapide se fait en place, on a pas besoin de créer de nouvelle liste, il faut donc un espace mémoire de taille n pour trier une liste de taille n. La complexité est alors linéaire en O(n).

5.4 exercice : tri rapide avec pivot en sentinelle

Dans cet exercice nous allons programmer une version du tri rapide différent de celui présenté plus haut.

Un pivot est toujours choisit dans la liste, mais nous allons considérer une autre méthode pour trier les deux sous listes d'élément inférieur et supérieur au pivot.

Au cours de l'organisation des éléments autour du pivot la liste est orientée comme suit :

L = [p, L[1], L[2], … , L[i-1], L[i], … , L[j-1], L[j], …]

avec p le pivot,

tous les éléments de L[1] à L[i-1] sont inférieurs à p,
tous les éléments de L[i] à L[j-1] sont encore inconnus,
tous les éléments de L[j] jusqu'à la fin de la liste sont supérieurs à p.
Considérons l'algorithme suivant prenant en entrée 1 une liste.

(0) Choisir pour pivot le premier élément de la liste. Initialiser judicieusement les variables i et j.

(1) Augmenter i tant que ça ne modifie pas l'organisation de la liste.

(2) Diminuer j tant que ça ne brise pas l'organisation de la liste.

(3) Si la zone de points d'interrogation est vide, arrêter l'algorithme et renvoyer i.

(4) Échanger L[i] et L[j], augmenter i de 1 et diminuer j de 1.

(5) Retourner à l'étape (1).

- Écrire une fonction implémentant cet algorithme en Python.

[12]:
```
def organisation_pivot(L):
    pivot = L[0]
    i, j = 1, len(L)-1
    while i < j:
        while L[i]<pivot and i<len(L)-1:
                i +=1
        while L[j]>= pivot and j>0:
                j -= 1
        if i<j :
            L[i], L[j] = L[j], L[i]
        else :
            resultat = i
    return resultat

L = [15, 4, 2, 9, 55, 16, 0, 1]
i = organisation_pivot(L)
print(L, i)
```

```
[15, 4, 2, 9, 1, 0, 16, 55] 6
```

- Pourquoi l'étape (4) de l'algorithme conserve toujours la bonne organisation de la liste ?

Car on ne fait l'échange que lorsque l'on est sur que L[i] est supérieur au pivot et L[j] inférieur au pivot. Leur place est alors inversé car les indices i sont en principe inférieur au pivot et j supérieur au pivot.

- Quelle est la complexité de cette fonction ?

On part de i = 0 et de j = n-1, avec n-1 la taille de la liste. Puis deux boucles indépendantes incrémentent les indices i et j jusqu'à que i=j. La boucle sur les i fait (i+1) étapes et la boucle sur les j fait (n-1-j) étapes, on a donc au total on a $C(n) = (i+1) + (n-1-j) = n + i - j$, or $i = j$ donc $C(n) = n = O(n)$.

Cette fonction a donc une complexité linéaire.

- Utiliser cette fonction pour écrire une fonction permettant d'implémenter le tri rapide comme vu plus haut.

La fonction que l'on vient d'écrire permet d'organiser en place les deux sous listes avant et après le pivot. On peut reprendre le même algorithme que plus haut en utilisant cette fonction pour orgnaiser les sous listes.

[13]:
```
def positionnement_pivot_sentinelle(L,debut,fin):
    L_prime = L[debut:fin]
    i = organisation_pivot(L_prime)
    L[debut:i-1] = L_prime[1:i]
    L[i-1] = L_prime[debut]
    L[i:fin] = L_prime[i:fin]
    return i,L

def tri_rapide_rec_sentinelle(L,debut,fin):
    if debut+1>=fin :
        return L
    else :
        j,L = positionnement_pivot_sentinelle(L,debut,fin)
        debut_1 = debut
        fin_1 = j
        L = tri_rapide_rec_sentinelle(L,debut_1,fin_1)
        debut_2 = j+1
        fin_2 = fin
        L = tri_rapide_rec_sentinelle(L,debut_2,fin_2)
        return L

def tri_rapide_sentinelle(L) :
    L = tri_rapide_rec_sentinelle(L,0,len(L))
    return L

L = [15, 4, 2, 9, 55, 16, 0, 1]
tri_rapide_sentinelle(L)
```

[13]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```