ALGORITHMES DE TRI

B. Landelle

Table des matières

I Introduction **2**
1 Définitions 2
2 Enjeu 3

II Tri par insertion **4**
1 Principe 4
2 Étude 5

III Tri rapide **7**
1 Principe 7
2 Étude 11

IV Tri fusion **13**
1 Principe 13
2 Étude 17

V Notions d'optimalité **18**
1 Présentation 18
2 Complexité temporelle 18

Dans ce chapitre, on s'intéresse à des techniques de tri sur des listes de nombres entiers ou flottants. Le cadre usuel pour cette étude est celui des tris sur tableaux mais les listes présentent certaines caractéristiques semblables comme l'accès à un élément en lecture/écriture en temps constant. On peut étendre ces techniques de tri à des éléments d'un ensemble muni d'une relation d'ordre total.

Rappels

La partie entière notée $\lfloor \cdot \rfloor$ et la partie entière supérieure notée $\lceil \cdot \rceil$ sont définies par

$$\forall x \in \mathbb{R} \qquad \lfloor x \rfloor = \text{Max}\ \{n \in \mathbb{Z},\ n \leqslant x\} \qquad \lceil x \rceil = \text{Min}\ \{n \in \mathbb{Z},\ n \geqslant x\}$$

Pour x réel, la partie entière $\lfloor x \rfloor$ et la partie entière supérieure $\lceil x \rceil$ sont les uniques entiers relatifs vérifiant

$$\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1 \qquad \lceil x \rceil - 1 < x \leqslant \lceil x \rceil$$

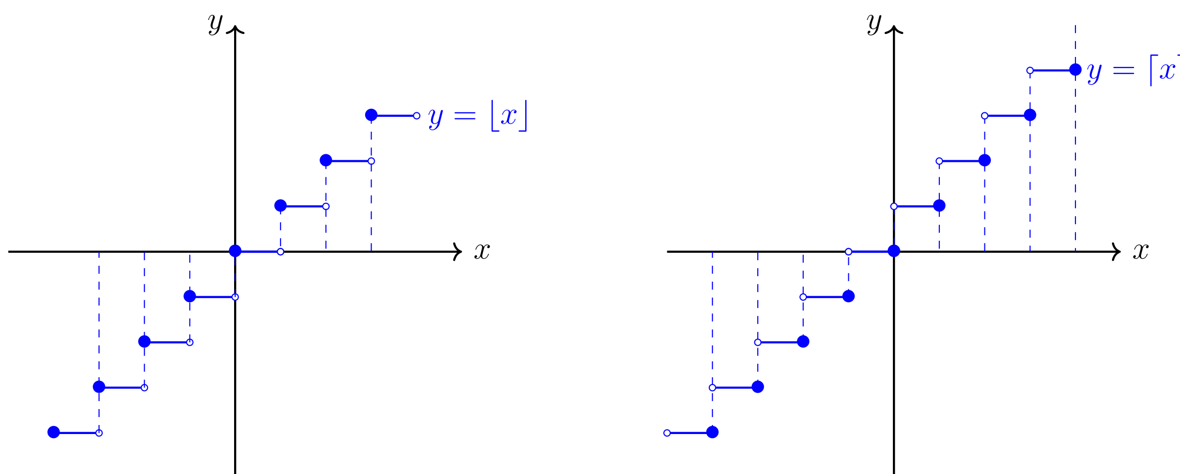

Figure 1 – Graphe de la partie entière et partie entière supérieure

En particulier, on a la propriété

$$\forall n \in \mathbb{N} \qquad n = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil$$

I Introduction

1 Définitions

Définition 1. *Un* algorithme de tri *a pour objet d'ordonner une liste de nombres.*

Définition 2. *Un algorithme de tri est dit* en place *s'il modifie directement la structure qu'il est en train de trier sans créer de variable de taille de même ordre que la liste à trier.*

Ce caractère en place est important lorsqu'on procède à un tri sur une liste de grande taille.

En python, la méthode `sort` et l'instruction `sorted` permettent d'ordonner des listes :

```
>>> a=[4,3,6,1]
>>> sorted(a)
[1, 3, 4, 6]
>>> a.sort()
>>> a
[1, 3, 4, 6]
```

L'instruction `sorted` produit une nouvelle liste. Il ne s'agit donc pas d'un tri en place tandis que c'est le cas de la méthode `sort` qui ordonne directement la variable `a`.

La méthode `sort` et l'instruction `sorted` utilisent un algorithme intitulé *Timsort* inventé en 2002 par Tim Peters [1]. C'est un algorithme hybride dérivé des algorithmes *tri fusion* et *tri par insertion* que nous étudierons dans la suite de ce chapitre. La complexité temporelle de Timsort est en $\mathrm{O}(n \log(n))$ dans le pire des cas et en $\mathrm{O}(n)$ dans le meilleur des cas.

2 Enjeu

La question du tri est un problème majeur en informatique. Par exemple, si l'on veut afficher le contenu d'un répertoire en ordonnant les fichiers par nom, par type, si l'on veut afficher le résultat d'une recherche sur internet par pertinence, par date, etc. ... on fait appel à une technique de tri. La performance d'un algorithme de tri est donc un élément clé pour le choix d'une méthode.

Le *tri par bulles* (*bubble sort* en anglais) est un tri simple à programmer. Il consiste à faire remonter les éléments les plus grands dans le haut de la liste comme des bulles d'air dans l'eau.

3	7	2	6	5	1	4
3	2	6	5	1	4	7
2	3	5	1	4	6	7
2	3	1	4	5	6	7
2	1	3	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

Figure 2 – Réalisation du tri par bulles

```
def tri_bulles(T):
    """tri_bulles(T:list)->None
    Réalise le tri par bulles de la liste T"""
    n=len(T)
    for i in range(n-1):
        for j in range(n-(i+1)):
            if T[j]>T[j+1]:
                T[j],T[j+1]=T[j+1],T[j]
```

La complexité temporelle du tri par bulles est en $O(n^2)$ même si la liste est déjà triée ! C'est un algorithme lent dont l'intérêt est essentiellement pédagogique.

Dans ce qui suit, on va se concentrer sur des algorithmes présentant de meilleures caractéristiques que ce premier exemple naïf.

II Tri par insertion

1 Principe

Le mécanisme du *tri par insertion* (*insertion sort* en anglais) est assez simple et intuitif. C'est souvent le procédé de tri que l'on réalise quand on doit classer des copies par notes croissantes par exemple. Dans la pile des copies, on prend les deux premières puis on les classe, puis on prend la troisième et on la classe avec les deux premières, puis la quatrième et on la classe avec les trois premières, etc. ...

Définition 3. *Le* tri par insertion *consiste à itérer le procédé suivant : si les k premiers éléments sont triés, on vient insérer le $k+1$-ème à sa place avec les k premiers.*

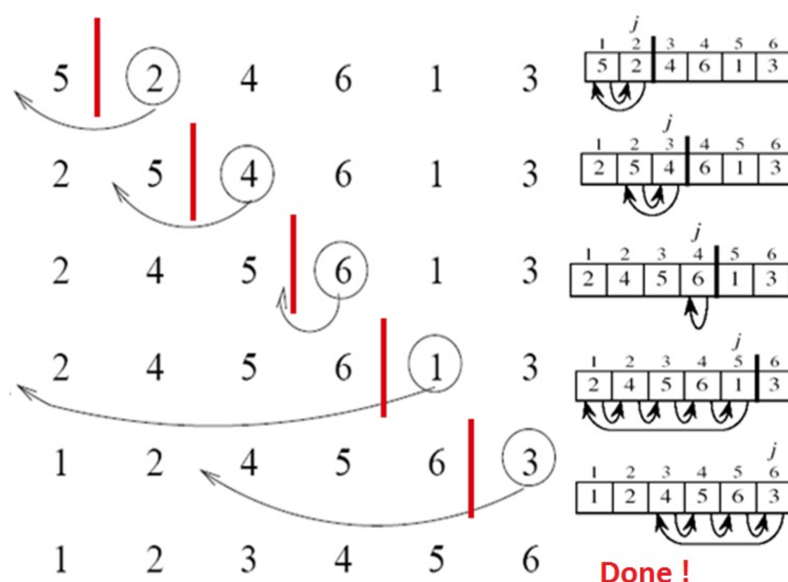

FIGURE 3 – Principe du tri par insertion

On va donc parcourir la liste à partir du deuxième élément puis classer l'élément courant avec les éléments précédents (à gauche du trait rouge sur la figure 2). Par construction, les éléments précédents sont classés par ordre croissant et il suffit par conséquent de détecter le premier strictement plus grand que l'élément courant pour s'arrêter. Si aucun de ces éléments précédents n'est plus grand que l'élément courant, ce dernier vient en première position.

```
def tri_ins(T):
    """tri_ins(T:list)->None
    Réalise le tri par insertion de la liste T"""
    n=len(T)
    for i in range(1,n):
        j=i
        while j>0 and T[j-1]>T[j]:
            T[j],T[j-1]=T[j-1],T[j]
            j-=1
```

3	7	2	6	5	1	4

3	7	2	6	5	1	4

2	3	7	6	5	1	4

2	3	6	7	5	1	4

2	3	5	6	7	1	4

1	2	3	5	6	7	4

1	2	3	4	5	6	7

Figure 4 – Réalisation du tri par insertion

Code commenté :

```
def tri_ins(T):
    n=len(T)
    for i in range(1,n):                # on parcourt la liste
        j=i                             # j indice élément courant
        while j>0 and T[j-1]>T[j]:      # tant que élément courant mal placé
            T[j],T[j-1]=T[j-1],T[j]     # on permute
            j-=1
```

2 Étude

Proposition 1. *Le tri par insertion est un tri en place.*

Démonstration. Le tri par insertion modifie directement la liste à trier et ne crée aucune variable de type composé. □

Pour la complexité temporelle, la présence d'une boucle conditionnelle `while` nous amène à considérer le meilleur et le pire des cas.

Théorème 1. *La complexité temporelle du tri par insertion est :*
- *dans le meilleur des cas en $O(n)$;*
- *dans le pire des cas en $O(n^2)$.*

Démonstration. Identifions les situations correspondant au meilleur puis au pire des cas.

Meilleur cas : Le meilleur des cas est réalisé si on ne rentre jamais dans la boucle `while`, situation qui arrive si la liste fournie en argument est déjà triée. Comme on a $n-1$ passages dans la boucle `for`, on en déduit une complexité temporelle en $\mathrm{O}(n)$ dans le meilleur des cas.

Pire cas : Le pire des cas est réalisé si, à chaque passage dans la boucle `for`, on rentre le nombre maximal de fois possibles dans la boucle `while`. Autrement dit, c'est la condition `j>0` qui provoque la sortie de `while`. On en déduit une complexité temporelle en

$$\sum_{i=1}^{n-1}\sum_{j=1}^{i}\mathrm{O}(1)=\mathrm{O}(n^2)$$

Cette situation la plus défavorable se produit si la liste est triée par ordre décroissant. □

Exercice : Quelle est la complexité spatiale du tri par insertion ?

Corrigé : Le tri par insertion utilise un nombre fixe de variables de tailles fixées d'où une complexité spatiale en $\mathrm{O}(1)$.

Exercice : On propose l'implémentation du tri par insertion suivante :

```
def tri_ins(T):
    n=len(T)
    for i in range(1,n):
        c=T[i]
        j=i-1
        while j>=0 and T[j]>c:
            T[j+1]=T[j]
            j-=1
        T[j+1]=c
```

Comprendre cette implémentation puis la comparer à la précédente.

Corrigé : Le code commenté permet de constater qu'il s'agit bien d'une implémentation du tri par insertion.

```
def tri_ins(T):
    n=len(T)
    for i in range(1,n):        # on parcourt la liste
        c=T[i]                  # c est l'élément courant
        j=i-1                   # j balaie les indices des éléments précédents
        while j>=0 and T[j]>c:  # tant que c n'est pas bien placé
            T[j+1]=T[j]         # on décale les éléments précédents
            j-=1
        T[j+1]=c
```

Au lieu d'effectuer des échanges successifs entre élément courant et un élément d'indice inférieur, on décale tous les éléments d'indice inférieur plus grand que l'élément courant et seulement ensuite, on déplace l'élément courant. Au lieu des échanges, on a donc des affectations simples qui sont moins couteuses. Cette version est donc un peu plus performante que la précédente mais la classe de complexité temporelle est inchangée.

III Tri rapide

1 Principe

L'algorithme du *tri rapide* (*quicksort* en anglais) a été inventé en 1961 par le britannique Tony Hoare.

Il repose sur le paradigme « diviser pour régner ». Considérons une liste de nombres entiers ou flottants. On choisit un élément de la liste (par exemple le premier) qu'on appelle pivot.

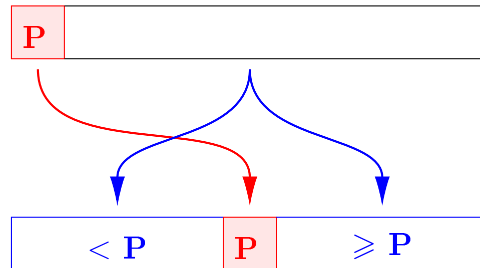

FIGURE 5 – Placement vis-à-vis du pivot

On place les éléments vis-à-vis du pivot puis on effectue récursivement le tri de la liste des éléments à droite du pivot puis à gauche du pivot (pour chaque sous-liste, choix du pivot puis placement à droite et à gauche, etc. . . .)

Définition 4. *Le* tri rapide *consiste en le procédé suivant : choix d'un pivot, placement des éléments de la liste vis-à-vis du pivot puis tri récursif à droite et à gauche du pivot.*

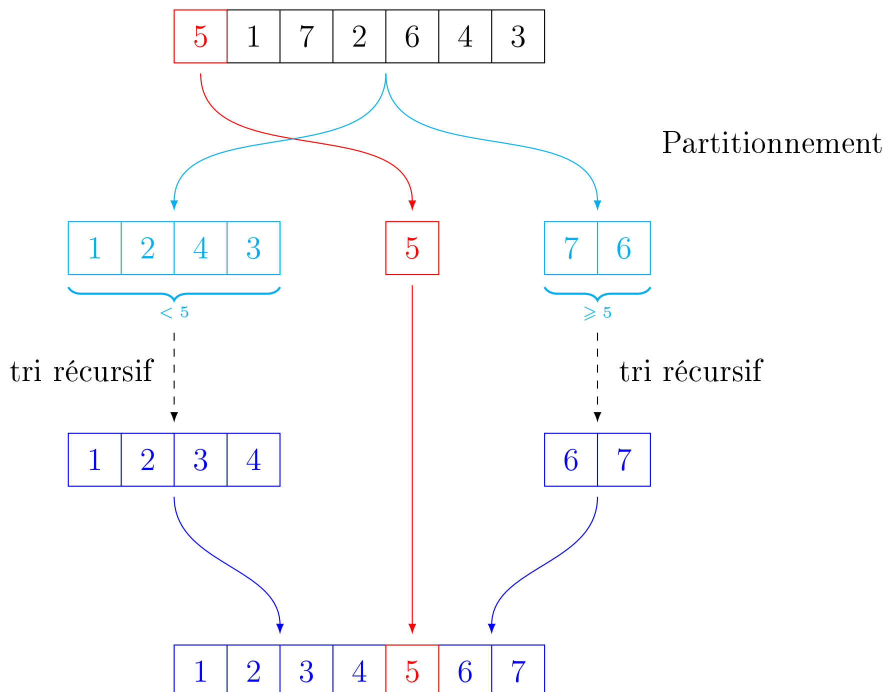

FIGURE 6 – Diviser pour régner - tri des listes à gauche et droite du pivot

On propose une première implémentation de ce tri :

```
def tri_rapide(T):
    """tri_rapide(T:list)->list
    Réalise le tri rapide de la liste T"""
    if T==[]:
        return []
    else:
        pivot=T[0]
        T1,T2=[],[]
        for x in T[1:]:
            if x<pivot:
                T1.append(x)
            else:
                T2.append(x)
        return tri_rapide(T1)+[pivot]+tri_rapide(T2)
```

Code commenté :

```
def tri_rapide(T):
    if T==[]:
        return []
    else:
        pivot=T[0]             # choix du pivot à gauche
        T1,T2=[],[]
        for x in T[1:]:        # placement des éléments vis-à-vis du pivot
            if x<pivot:
                T1.append(x)
            else:
                T2.append(x)
                               # après placement, tri récursif
        return tri_rapide(T1)+[pivot]+tri_rapide(T2)
```

Cette version très simple suit la démarche présentée précédemment. Sa simplicité en fait clairement la version à apprendre parmi les différentes versions du tri rapide. Cependant, elle présente de multiples défauts dont, en particulier, le fait de ne pas réaliser un tri en place. On propose une implémentation théoriquement plus efficace :

```
def tri_rapide_rec(T,g,d):
    """tri_rapide_rec(T:list,g:int,d:int)->None
    Réalise récursivement le tri rapide en place de T[g:d]"""
    if g<d:
        p=partition(T,g,d)
        tri_rapide_rec(T,g,p)
        tri_rapide_rec(T,p+1,d)

def tri_rapide(T):
    """tri_rapide(T:list)->None
    Réalise le tri rapide en place de la liste T"""
    tri_rapide_rec(T,0,len(T))
```

```
def echange(T,i,j):
    """echange(T:list,i:int,j:int)->None
    Échange les valeurs d'indices i et j de la liste T"""
    T[i],T[j]=T[j],T[i]

def partition(T,g,d):
    """partition(T:list,g:int,d:int)->int
    Réalise le placement des éléments de T[g:d] vis-à-vis du pivot
    et renvoie la position finale du pivot après placement"""
    pivot=T[g]
    pos=g
    for i in range(g+1,d):
        if T[i]<pivot:
            pos+=1
            echange(T,i,pos)
    if pos!=g:
        echange(T,g,pos)
    return pos
```

On observe que c'est toujours la même variable `T` qui est transmise en argument, que ce soit dans la fonction `partition` ou dans les appels récursifs de `tri_rapide_rec`. Il n'y a donc pas de créations de nouvelles listes au cours des récursions. Le travail de tri se fait en délimitant une plage d'indices : `g` pour gauche et `d` pour droite.

Code commenté :

```
def echange(T,i,j):
    T[i],T[j]=T[j],T[i]         # échange T[i] et T[j]

def partition(T,g,d):           # place les éléments vis-à-vis d'un pivot
    pivot=T[g]                  # pivot : 1er terme de la liste entre g et d
    pos=g                       # pos : position future du pivot
                                # dans boucle, pos est la position du
                                # dernier elt<pivot
    for i in range(g+1,d):      # on parcourt la liste après pivot
        if T[i]<pivot:          # si elt courant plus petit
            pos+=1              # on incrémente pos
                                # i est la position du dernier elt<pivot
            echange(T,i,pos)    # échange valeurs d'indices pos/i
                                # elt<pivot mis à gauche de pos
                                # (pos : position future du pivot)
    if pos!=g:                  # si pos a été modifié
                                # alors pos=position du dernier elt<pivot
        echange(T,g,pos)        # échange valeurs d'indices g/pos
    return pos                  # renvoie position du pivot
```

```
def tri_rapide_rec(T,g,d):
    if g<d:
        p=partition(T,g,d)      # position du pivot et placement des elts
                                # vis-à-vis du pivot
        tri_rapide_rec(T,g,p)   # on trie à gauche du pivot
                                # (sous liste stricte)
        tri_rapide_rec(T,p+1,d) # on trie à droite du pivot
                                # (sous liste stricte)

def tri_rapide(T):
    tri_rapide_rec(T,0,len(T))  # amorce le tri des éléments
                                # du premier au dernier
```

Expérimentation : Rendons le code bavard et voyons son exécution sur un exemple :

```
...
def tri_rapide_rec(T,g,d):
    if g<d:
        print("liste=",T[g:d])
        p=partition(T,g,d)
        print("pivot=",T[p]," partition=",T[g:d])
        tri_rapide_rec(T,g,p)
        tri_rapide_rec(T,p+1,d)
...
```

Le programme affiche désormais, au cours de ses appels récursifs, la liste qu'il traite, le pivot puis le placement (partition) vis-à-vis du pivot.

```
>>> tab=[5,1,7,2,6,4,3]
>>> tri_rapide(tab)
liste= [5, 1, 7, 2, 6, 4, 3]
pivot= 5  partition= [3, 1, 2, 4, 5, 7, 6]
liste= [3, 1, 2, 4]
pivot= 3  partition= [2, 1, 3, 4]
liste= [2, 1]
pivot= 2  partition= [1, 2]
liste= [1]
pivot= 1  partition= [1]
liste= [4]
pivot= 4  partition= [4]
liste= [7, 6]
pivot= 7  partition= [6, 7]
liste= [6]
pivot= 6  partition= [6]
```

La fonction `partition` qui réalise le placement vis-à-vis du pivot est la seule à modifier le contenu de la liste `T`.

```
def partition(T,g,d):
    pivot=T[g]
    pos=g
    for i in range(g+1,d):
        if T[i]<pivot:
            pos+=1
            echange(T,i,pos)
    if pos!=g:
        echange(T,g,pos)
    return pos
```

La variable locale `pos` est, au cours de la boucle, la position du dernier élément rencontré strictement inférieur au pivot (s'il n'existe pas de tel élément, la variable `pos` stationne à la valeur `g` ce qui est conforme).

Si on ne rencontre dans la boucle que des éléments strictement inférieurs au pivot, alors la variable `pos` prend les mêmes valeurs que la variable `i` et et l'opération d'échange est sans effet.

En revanche, si on rencontre un élément supérieur ou égal au pivot, alors la variable `pos` décroche en restant inchangée, autrement dit en stationnant sur le dernier élément rencontré strictement inférieur au pivot. Si après cette configuration, on rencontre de nouveau un élément strictement inférieur au pivot, alors la variable `pos` est incrémentée et elle indice donc un élément supérieur ou égal au pivot qui est échangé avec l'élément courant d'indice `i` et la variable `pos` redevient la position du dernier élément rencontré strictement inférieur au pivot.

En sortie de boucle, on regarde si les variables `pos` et `g` diffèrent ce qui équivaut à dire qu'il y a des éléments strictement inférieurs au pivot. Dans ce cas, le dernier élément rencontré strictement plus petit que le pivot est échangé avec le pivot et ainsi, on a bien des éléments à gauche du pivot strictement plus petits que le pivot et des éléments à droite supérieurs ou égaux au pivot.

```
i :  0   1   2   3   4   5   6         pos= 0
   [ 5 | 1 | 7   2   6   4   3 ]  i= 1 pos= 1
   [ 5 | 1 | 7   2   6   4   3 ]  i= 2 pos= 1  -> décrochage
   [ 5 | 1   2 | 7   6   4   3 ]  i= 3 pos= 2
   [ 5 | 1   2 | 7   6   4   3 ]  i= 4 pos= 2  -> décrochage
   [ 5 | 1   2   4 | 6   7   3 ]  i= 5 pos= 3
   [ 5 | 1   2   4   3 | 7   6 ]  i= 6 pos= 4
   [ 3   1   2   4 | 5 | 7   6 ]  i= 6 pos= 4  -> échange final
```

2 Étude

On considère la dernière implémentation du tri rapide.

Proposition 2. *Le tri rapide est un tri en place.*

Démonstration. Le tri rapide modifie directement la liste à trier. Les modifications sont effectuées par la fonction de partition qui échange des positions d'éléments pour positionner les éléments vis-à-vis du pivot. □

Théorème 2. *La complexité temporelle du tri rapide est :*
- *dans le meilleur des cas en $O(n \log(n))$;*
- *dans le pire des cas en $O(n^2)$.*

Démonstration. Lors d'un appel de `tri_rapide_rec` sur une liste de taille n, la fonction `partition` réalise une boucle de taille $n-1$ avec des opérations à coût constant puis on appelle récursivement `tri_rapide_rec` sur deux listes, une de taille k et l'autre $n-1-k$ avec $k \in [\![0\,;\, n-1]\!]$ qui dépend de la liste. La complexité temporelle vérifie une relation de la forme

$$\mathrm{T}(n) = \mathrm{O}(n) + \mathrm{T}(k) + \mathrm{T}(n-1-k)$$

Pire cas : La situation la pire correspond à celle de la plus longue branche d'appels récursifs ce qui équivaut à la plus faible décroissance de l'argument, autrement dit une configuration où $k=0$ ou $k=n-1$ à chaque récursion ce qui implique que le long d'une branche, on passera seulement d'un argument de taille i à $i-1$. Dans ce cas, la complexité temporelle suit une relation de la forme

$$\mathrm{T}(n) = \mathrm{O}(n) + \underbrace{\mathrm{T}(0)}_{=\mathrm{O}(1)} + \mathrm{T}(n-1) = \mathrm{O}(n) + \mathrm{T}(n-1)$$

Par suite $$\mathrm{T}(n) = \mathrm{T}(0) + \sum_{i=1}^{n} [\mathrm{T}(i) - \mathrm{T}(i-1)] = \mathrm{T}(0) + \sum_{i=1}^{n} \mathrm{O}(i) = \mathrm{O}(n^2)$$

Meilleur cas : le meilleur cas est réalisé si on a la plus petite hauteur d'arbre d'appels récursifs. Lors d'une étape, on passe d'un argument de taille n à deux arguments de taille k et $n-1-k$. L'un des deux est nécessairement de taille $\geqslant \lfloor n/2 \rfloor$. En effet, si $k \leqslant \lfloor n/2 \rfloor - 1$ et $n-k-1 \leqslant \lfloor n/2 \rfloor - 1$, alors $n-1 \leqslant 2\lfloor n/2 \rfloor - 2$ ce qui est faux. La taille de l'écriture binaire pour le passage de n à $\lfloor n/2 \rfloor = n//2$ décroît de un. Donc, notant p la taille de l'écriture binaire de n, il y aura au moins une branche de taille p dans l'arbre d'appels. La hauteur de l'arbre d'appel est de l'ordre de p si, à chaque étape, la taille de l'argument se répartit en $\left\lfloor \frac{n-1}{2} \right\rfloor$ et $\left\lceil \frac{n-1}{2} \right\rceil$. Ainsi, dans le meilleur cas, la complexité temporelle vérifie une relation de la forme

$$\mathrm{T}(n) = \mathrm{O}(n) + \mathrm{T}\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + \mathrm{T}\left(\left\lceil \frac{n-1}{2} \right\rceil\right)$$

Cette relation est loin d'être simple à résoudre. Supposons la complexité temporelle croissante avec la taille de l'argument (hypothèse simplificatrice mais somme toute assez intuitive). Pour p entier, on a

$$\mathrm{T}(2^p) = \mathrm{O}(2^p) + \underbrace{\mathrm{T}(2^{p-2})}_{\leqslant \mathrm{T}(2^{p-1})} + \mathrm{T}(2^{p-1}) \leqslant \mathrm{O}(2^p) + 2\mathrm{T}(2^{p-1})$$

et par conséquent $$\frac{\mathrm{T}(2^p)}{2^p} - \frac{\mathrm{T}(2^{p-1})}{2^{p-1}} \leqslant \mathrm{O}(1)$$

d'où $$\frac{\mathrm{T}(2^p)}{2^p} = \mathrm{T}(1) + \sum_{k=1}^{p} \left[\frac{\mathrm{T}(2^k)}{2^k} - \frac{\mathrm{T}(2^{k-1})}{2^{k-1}}\right] \leqslant \mathrm{T}(1) + \sum_{k=1}^{p} \mathrm{O}(1) = \mathrm{O}(p)$$

On en déduit $\mathrm{T}(2^p) = \mathrm{O}(p2^p)$. Pour n entier non nul, on a $2^{p-1} \leqslant n \leqslant 2^p$ avec $p = \lfloor \log_2(n) \rfloor + 1$ puis

$$\mathrm{T}(n) \leqslant \mathrm{T}(2^p) = \mathrm{O}(p2^p) = \mathrm{O}(n \log(n))$$

□

Remarque : On peut démontrer que la complexité spatiale du tri rapide est :

— dans le meilleur des cas en $\mathrm{O}(\log(n))$;
— dans le pire des cas en $\mathrm{O}(n)$.

Exercice : Identifier la situation amenant à la pire complexité temporelle. Proposer une alternative simple qui rende cette situation très peu probable.

Après l'importation `import numpy.random as rd`, on pourra utiliser `rd.randint(a,b)` qui permet de générer un nombre aléatoire dans $[\![a\,; b-1]\!]$ avec a et b entiers.

Corrigé : Le pire cas correspond à la situation où la fonction de partition renvoie un pivot sur une des extrémités de la liste. Cette situation arrive par exemple sur une liste triée par ordre croissant ou par ordre décroissant. Pour éviter ce cas, il suffit d'insérer une ligne afin que la fonction `partition` choisisse un pivot au hasard et non plus le premier terme de la liste.

```
def partition(T,g,d):
    echange(T,g,rd.randint(g,d))  # pivot pris au hasard
    pivot=T[g]
    pos=g
    ...
```

Le pire cas correspond donc à une situation bien particulière et on peut raisonnablement considérer que le meilleur cas est le cas le plus probable en pratique.

IV Tri fusion

1 Principe

Le *tri fusion* (*merge sort* en anglais) est également basé sur le paradigme « diviser pour régner ». Le principe de l'algorithme est le suivant : pour trier une liste, on partitionne celle-ci en deux listes de même taille (ou presque selon la parité) que l'on trie récursivement puis que l'on fusionne.

Définition 5. *Le* tri fusion *consiste en le procédé suivant : coupure de la liste de taille n en deux listes de tailles $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$, tri récursif de ces listes puis fusion de celles-ci.*

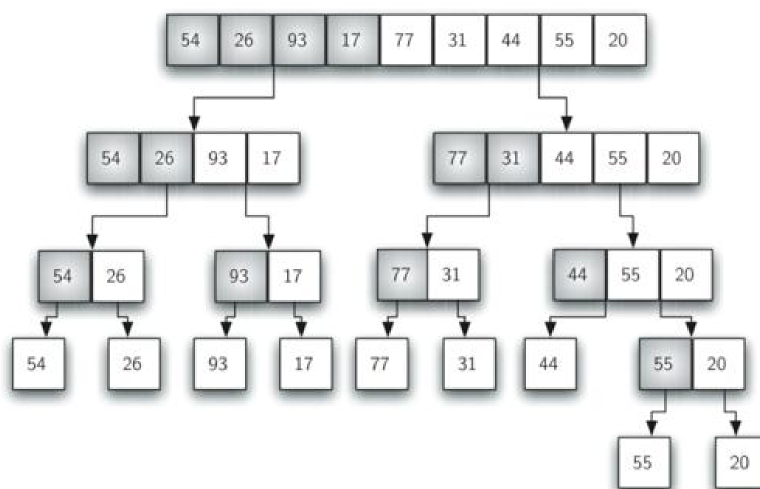

FIGURE 7 – Diviser pour régner - Division en sous-listes

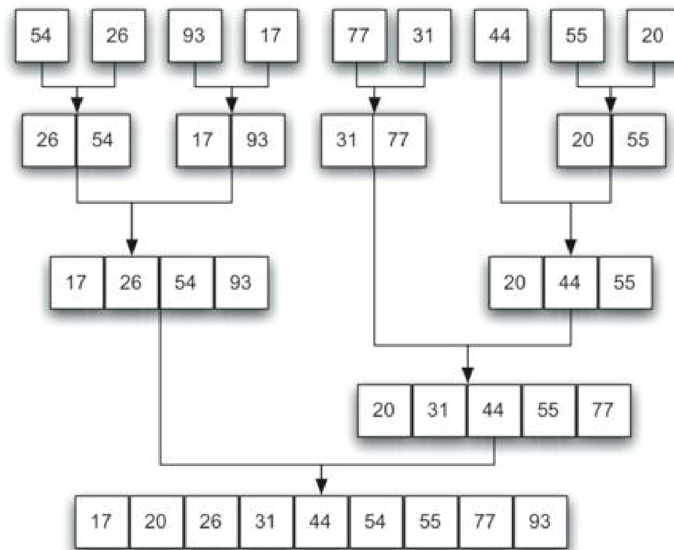

FIGURE 8 – Fusion des listes ordonnées

On propose une première implémentation de ce tri :

```
def fusion(A,B):
    """fusion(A:list,B:list)->list
    Fusionne deux listes triées en une seule triée"""
    a,b=len(A),len(B)
    i,j,tab=0,0,[]
    while i<a and j<b:
        if A[i]<B[j]:
            tab.append(A[i])
            i+=1
        else:
            tab.append(B[j])
            j+=1
    if i==a:
        tab=tab+B[j:]
    else:
        tab=tab+A[i:]
    return tab

def tri_fusion(T):
    """tri_fusion(T:list)->list
    Réalise le tri fusion de la liste T"""
    n=len(T)
    if n<=1:
        return T
    else:
        k=n//2
        A,B=T[:k],T[k:]
        return fusion(tri_fusion(A),tri_fusion(B))
```

Code commenté :

```
def fusion(A,B):
    a,b=len(A),len(B)
    i,j,tab=0,0,[]              # i indice pour A, j indice pour B
    while i<a and j<b:          # tant que A ou B pas complètement parcourus
        if A[i]<B[j]:           # on choisit le plus petit entre A[i] et B[j]
            tab.append(A[i])    # on le place dans tab
            i+=1                # et on passe au suivant dans A
            i+=1
        else:
            tab.append(B[j])
            j+=1                # ou dans B selon les cas
    if i==a:                    # si A parcouru intégralement
        tab=tab+B[j:]           # on complète avec les restants de B
    else:
        tab=tab+A[i:]           # sinon on complète avec les restants de A
    return tab

def tri_fusion(T):
    n=len(T)
    if n<=1:
        return T
    else:
        k=n//2
        A,B=T[:k],T[k:]                              # coupure en 2 listes
        return fusion(tri_fusion(A),tri_fusion(B))   # tri récursif et fusion
```

Cette version simple à rédiger et à mémoriser est très lourde. La fonction `fusion` crée une variable de taille la somme des tailles des sous-listes `A` et `B` mais surtout, ces variables `A` et `B` sont créées dans la fonction `tri_fusion` avant chaque récursion et demeurent conservées tout au long des appels récursifs.

On propose une version qui améliore ce dernier point, sans toutefois réussir à pallier le défaut de la fonction `fusion`. Néanmoins, la version qui suit utilise une construction « dos-à-dos » assez confortable pour la gestion des indices.

```
def fusion(T,g,m,d):
    """fusion(T:list,g:int,m:int,d:int)->None
    Réalise la fusion des sous-listes triées T[g:m] et T[m:d]
    directement sur la plage T[g:d]"""
    tab=T[g:m]+T[d-1:m-1:-1]
    i=0
    j=-1
    for k in range(g,d):
        if tab[i]<tab[j]:
            T[k]=tab[i]
            i+=1
        else:
```

```
            T[k]=tab[j]
            j-=1

def tri_fusion_rec(T,g,d):
    """tri_fusion_rec(T:list,g:int,d:int)->None
    Réalise récursivement le tri fusion de T[g:d]"""
    if d-g>1:
        m=(g+d)//2
        tri_fusion_rec(T,g,m)
        tri_fusion_rec(T,m,d)
        fusion(T,g,m,d)

def tri_fusion(T):
    """tri_fusion(T:list)->None
    Réalise le tri fusion de la liste T"""
    tri_fusion_rec(T,0,len(T))
```

Code commenté :

```
def fusion(T,g,m,d):       # fusionne les listes triées T[g:m] et T[m:d]
    tab=T[g:m]+T[d-1:m-1:-1]
                           # construit une liste "dos-à-dos"
                           # [T[g],...,T[m-1],T[d-1],...T[m]]
    i=0                    # l'indice i parcourt tab en montant (donc T[g:m])
    j=-1                   # l'indice j parcourt tab en descendant (donc T[m:d])
    for k in range(g,d):   # on remplit chaque case
        if tab[i]<tab[j]:  # choix de la liste pour l'élément courant
            T[k]=tab[i]    # si dans première moitié
            i+=1           # on incrémente i
        else:
            T[k]=tab[j]    # sinon on décrémente j
            j-=1

def tri_fusion_rec(T,g,d):
    if d-g>1:
        m=(g+d)//2
        tri_fusion_rec(T,g,m)   # on trie la moitié gauche
        tri_fusion_rec(T,m,d)   # on trie la moitié droite
        fusion(T,g,m,d)         # on fusionne chaque moitié triée


def tri_fusion(T):
    tri_fusion_rec(T,0,len(T))  # amorce le tri fusion
```

Expérimentation : Rendons le code bavard et voyons son exécution sur un exemple :

```
def fusion(T,g,m,d):
    tab=T[g:m]+T[d-1:m-1:-1]
    print("liste dos-à-dos=",tab)
```

```
    i=0
    ...

def tri_fusion_rec(T,g,d):
    if d-g>1:
        m=(g+d)//2
        print("sous-listes=",T[g:m],T[m:d])
        tri_fusion_rec(T,g,m)
        tri_fusion_rec(T,m,d)
        fusion(T,g,m,d)
        print("fusion=",T[g:d])
```

Le programme affiche désormais les différentes sous-listes à trier, les listes dos-à-dos qui sont constituées avant fusion puis les fusions elles-mêmes.

```
>>> a=[5,2,1,0,3,4,6]
>>> tri_fusion(a)
sous-listes= [5, 2, 1] [0, 3, 4, 6]
sous-listes= [5] [2, 1]
sous-listes= [2] [1]
liste dos-à-dos= [2, 1]
fusion= [1, 2]
liste dos-à-dos= [5, 2, 1]
fusion= [1, 2, 5]
sous-listes= [0, 3] [4, 6]
sous-listes= [0] [3]
liste dos-à-dos= [0, 3]
fusion= [0, 3]
sous-listes= [4] [6]
liste dos-à-dos= [4, 6]
fusion= [4, 6]
liste dos-à-dos= [0, 3, 6, 4]
fusion= [0, 3, 4, 6]
liste dos-à-dos= [1, 2, 5, 6, 4, 3, 0]
fusion= [0, 1, 2, 3, 4, 5, 6]
```

2 Étude

Proposition 3. *Le tri fusion n'est pas un tri en place.*

Démonstration. La fonction de fusion crée une liste de même taille que la liste à trier. La modification n'est pas directement faite sur la liste d'entrée. □

Remarque : En fait, il est possible d'implémenter un tri fusion en place mais c'est vraiment très difficile...

Théorème 3. *La complexité temporelle du tri fusion est en $O(n \log(n))$.*

Démonstration. Lors d'un appel de `tri_fusion_rec` sur une liste de taille n, on appelle récursivement `tri_fusion_rec` sur deux listes de tailles respectives $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ puis on effectue la fusion qui construit la liste dos-à-dos de taille n et parcourt celle-ci. Ainsi, la complexité temporelle vérifie une relation de la forme

$$\mathrm{T}(n) = \mathrm{T}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \mathrm{T}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathrm{O}(n)$$

On est dans une situation analogue à celle du meilleur cas pour le tri rapide. La relation générale est très difficile à résoudre et on fait l'hypothèse simplificatrice (mais naturelle) de la croissance de T. Pour p entier, on a

$$\mathrm{T}(2^p) = 2\mathrm{T}(2^{p-1}) + \mathrm{O}(2^p)$$

C'est exactement la configuration observée pour le meilleur cas du tri rapide. On conclut

$$\mathrm{T}(n) = \mathrm{O}(n \log(n))$$

□

Remarque : On peut démontrer que la complexité spatiale du tri fusion est $\mathrm{O}(n)$. C'est le même raisonnement que celui vu lors de l'étude de complexité spatiale de la FFT.

V Notions d'optimalité

1 Présentation

Résumons les performances de chacun des algorithmes précédemment étudiés :

	Complexité temporelle	Complexité spatiale
tri par insertion	$\mathrm{O}(n) - \mathrm{O}(n^2)$	$\mathrm{O}(1)$
tri rapide	$\mathrm{O}(n \log(n)) - \mathrm{O}(n^2)$	$\mathrm{O}(\log(n)) - \mathrm{O}(n)$
tri fusion	$\mathrm{O}(n \log(n))$	$\mathrm{O}(n)$

Schéma - Tableau comparatif des complexités

On observe des différences notables de comportement. Le choix d'un algorithme sera donc déterminé, entre autre, par ses caractéristiques de complexité. Précisons toutefois qu'il s'agit de comportements asymptotiques, pour n grand. Pour de petites listes, les classes de complexité ne sont plus des outils aussi pertinents pour la décision. C'est d'ailleurs ce qui explique que les fonctions de tri implémentées dans les langages de programmation soient, la plupart du temps, des algorithmes hybrides.

2 Complexité temporelle

Exceptée la situation la plus favorable (celle d'une liste déjà triée) du tri par insertion, la meilleur complexité temporelle de ces algorithmes de tri est en $\mathrm{O}(n \log(n))$. On constate même que le tri fusion est $\mathrm{O}(n \log(n))$ en toutes circonstances. Une question naturelle serait : cette complexité temporelle quasi-linéaire est-elle optimale pour un algorithme de tri ?

Étant donnée une liste d'entiers $[a_1, \ldots, a_n]$, un algorithme de tri renvoie la liste $\left[a_{\sigma(1)}, \ldots, a_{\sigma(n)}\right]$ avec $a_{\sigma(1)} \leqslant a_{\sigma(2)} \leqslant \ldots \leqslant a_{\sigma(n)}$. Ainsi, l'algorithme détermine d'une certaine manière une permutation σ parmi l'ensemble de toute les permutations S_n de cardinal $n!$. Si on représente

l'arbre des comparaisons où chaque comparaison est un nœud de l'arbre, cet arbre possède au final $n!$ feuilles. Or, un arbre de hauteur minimale h qui contient N feuilles vérifie

$$2^{h-1} < \mathrm{N} \leqslant 2^h$$

puisque à chaque incrément de 1 en hauteur, le nombre de feuilles double. On en déduit $h = \lceil \log_2(\mathrm{N}) \rceil$. Ainsi, pour un nombre de feuilles égal à $n!$, on a une hauteur $h_n = \lceil \log_2(n!) \rceil$.

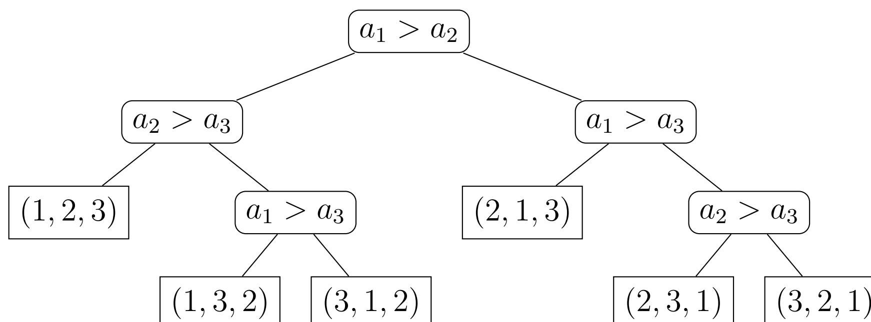

FIGURE 9 – Arbre de décision d'un algorithme de tri

Il faut donc $\mathrm{O}(\log(n!))$ comparaisons pour aboutir à la feuille correspondant à la permutation cherchée. Rappelons l'équivalent de Stirling

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

d'où

$$\log(n!) \sim n \log(n)$$

Ainsi, une complexité temporelle en $\mathrm{O}(n \log(n))$ est optimale.

En réalité, le critère mathématique ne fait pas tout. En pratique, le tri rapide est préféré au tri fusion pour des raisons de localité de données : chaque appel récursif du tri rapide travaille sur des données proches de celle de départ. En raison des caches (type de mémoire très rapides) qui permettent des optimisations considérables quand les données ont une forte localité, c'est le tri rapide qui réalise les meilleures performances.

Le lecteur curieux pourra compléter cette étude en consultant les ouvrages de référence [2], [3] et [4].

Annexe

On détaille le calcul de la complexité temporelle pour l'algorithme du tri rapide. On rappelle son principe :

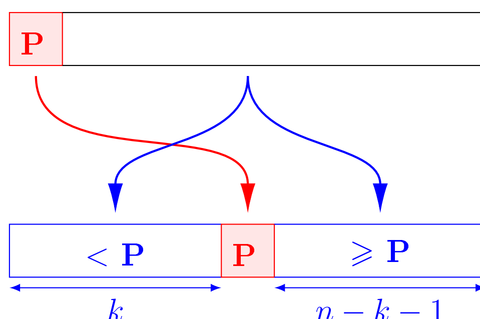

FIGURE 10 – Placement vis-à-vis du pivot

La complexité temporelle vérifie la relation

$$\mathrm{T}(n) = \mathrm{O}(n) + \mathrm{T}(k) + \mathrm{T}(n-1-k) \quad \text{avec} \quad k \in [\![0\,;\, n-1]\!]$$

Soit n entier non nul. On note L_n la plus longue branche dans l'arbre des récursions avec l'argument de taille initiale n et on pose $\ell_n = \lfloor \log_2(n) \rfloor + 1$. On a

$$\mathrm{L}_n = 1 + \max(\mathrm{L}_k, \mathrm{L}_{n-k-1}) \quad \text{avec} \quad k \in [\![0\,;\, n-1]\!]$$

Par convention, on pose $\mathrm{L}_0 = \ell_0 = 0$.

Lemme 1. *On a* $$\forall n \in \mathbb{N} \qquad \ell_n \leqslant \mathrm{L}_n \leqslant n$$

De plus $$\forall n \geqslant 1 \qquad \mathrm{L}_n = 1 + \max(\mathrm{L}_0, \mathrm{L}_{n-1}) \implies \forall n \in \mathbb{N} \qquad \mathrm{L}_n = n$$

et $$\forall n \geqslant 1 \qquad \mathrm{L}_n = 1 + \max\left(\mathrm{L}_{\lfloor \frac{n-1}{2} \rfloor}, \mathrm{L}_{\lceil \frac{n-1}{2} \rceil}\right) \implies n \in \mathbb{N} \qquad \mathrm{L}_n = \ell_n$$

Démonstration. L'encadrement est vrai pour $n = 0$. On le suppose vrai jusqu'au rang n entier non nul. On a

$$\mathrm{L}_n = 1 + \max(\mathrm{L}_k, \mathrm{L}_{n-k-1}) \quad \text{avec} \quad k \in [\![0\,;\, n-1]\!]$$

Ainsi $$\mathrm{L}_n \geqslant 1 + \max(\ell_k, \ell_{n-k-1})$$

Or, on a $k \geqslant \lfloor n/2 \rfloor$ ou $n-k-1 \geqslant \lfloor n/2 \rfloor$. En effet, sinon, on aurait $k \leqslant \lfloor n/2 \rfloor - 1$ et $n-k-1 \leqslant \lfloor n/2 \rfloor - 1$ d'où $n - 1 \leqslant 2\lfloor n/2 \rfloor - 2$ ce qui est faux. Il vient

$$\mathrm{L}_n \geqslant 1 + \ell_{\lfloor n/2 \rfloor} = 1 + \ell_{n//2} = 1 + \ell_n - 1 = \ell_n$$

Puis $$\mathrm{L}_n \leqslant 1 + \max(k, n-k-1) \leqslant 1 + n - 1 = n$$

ce qui clôt la récurrence. Puis

$$\forall n \geqslant 1 \qquad \mathrm{L}_n = 1 + \max(\mathrm{L}_0, \mathrm{L}_{n-1}) \iff \forall n \geqslant 1 \qquad \mathrm{L}_n = 1 + \mathrm{L}_{n-1}$$

et il en résulte clairement $\mathrm{L}_n = n$ pour tout n entier. Puis, on suppose

$$\forall n \geqslant 1 \qquad \mathrm{L}_n = 1 + \max\left(\mathrm{L}_{\lfloor \frac{n-1}{2} \rfloor}, \mathrm{L}_{\lceil \frac{n-1}{2} \rceil}\right)$$

On procède à nouveau par récurrence pour établir $\mathrm{L}_n = \ell_n$ pour tout n entier. Le résultat est vrai pour $n = 0$ et on le suppose vrai jusqu'au rang $n-1 \geqslant 1$. On a

$$\mathrm{L}_n = 1 + \max\left(\ell_{\lfloor \frac{n-1}{2} \rfloor}, \ell_{\lceil \frac{n-1}{2} \rceil}\right) = 1 + \ell_{\lceil \frac{n-1}{2} \rceil}$$

En distinguant les cas pair/impair, on établit

$$\ell_{\lceil \frac{n-1}{2} \rceil} = \ell_{n//2}$$

d'où
$$\mathrm{L}_n = 1 + \ell_{n//2} = \ell_n$$

ce qui clôt la récurrence. □

Théorème 4. *La complexité temporelle du tri rapide est en $O(n\mathrm{L}_n)$ et en particulier :*
- *en $O(n\log(n))$ dans le meilleur des cas qui correspond à un pivot équilibré ;*
- *en $O(n^2)$ dans le pire des cas qui correspond à un pivot exclusif.*

Démonstration. La complexité temporelle vérifie la relation pour n entier non nul

$$\mathrm{T}(n) = \mathrm{O}(n) + \mathrm{T}(k) + \mathrm{T}(n-1-k) \quad \text{avec} \quad k \in [\![0\,;\, n-1]\!]$$

On dispose de $\mathrm{C} \geqslant 0$ tel que

$$\mathrm{T}(n) \leqslant \mathrm{C}n + \mathrm{T}(k) + \mathrm{T}(n-1-k)$$

On montre $\mathrm{T}(n) \leqslant \mathrm{C}(n+1)(\mathrm{L}_n+1)$ par récurrence. L'inégalité est vraie pour $n = 0$ (on choisit $\mathrm{C} \geqslant \mathrm{T}(0)$). Les décalages en $+1$ sont d'ailleurs là pour l'initialisation car le coût d'un tri sur une liste vide n'est pas nul (il faut tester la vacuité de la liste). Supposons la propriété vraie jusqu'au rang $n-1$. Il vient

$$\begin{aligned}\mathrm{T}(n) &\leqslant \mathrm{C}n + \mathrm{C}(k+1)(\mathrm{L}_k+1) + \mathrm{C}(n-1-k+1)(\mathrm{L}_{n-1-k}+1)\\ &\leqslant \mathrm{C}n + \mathrm{C}(k+1)\mathrm{L}_n + \mathrm{C}(n-k)\mathrm{L}_n \leqslant \mathrm{C}(n+1) + \mathrm{C}(n+1)\mathrm{L}_n \leqslant \mathrm{C}(n+1)(\mathrm{L}_n+1)\end{aligned}$$

ce qui clôt la récurrence. Ceci prouve $\mathrm{T}(n) = \mathrm{O}(n\mathrm{L}_n)$ et le résultat suit d'après le lemme précédent. □

Références

[1] Tim Peters, Timsort description, accessed june 2015, `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`

[2] Donald Knuth, *The Art of Computer Programming, Volume 3 : Sorting and Searching*, Second Edition, Addison-Wesley, 1998

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition MIT Press and McGraw-Hill, 2009

[4] Robert Sedgewick, Philippe Flajolet, *An Introduction to the Analysis of Algorithms*, Second Edition, Addison-Wesley, 2013