Algorithmes de tri

1. Introduction

On considère une liste de N nombres $x_0, x_1, x_2...x_{N-1}$. Le tri est une opération de permutation de ces N nombres qui donne une liste de nombres rangés dans l'ordre croissant :

$$x'_0 \leq x'_1 \leq x'_2 \cdots \leq x'_{N-1} \tag{1}$$

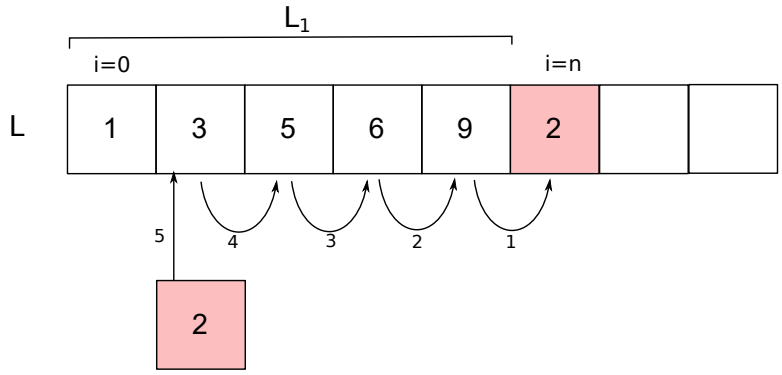
2. Tri par insertion

2.a. Algorithme

Soit L la liste de nombres à trier. Le tri par insertion consiste à prendre les éléments de L un par un, dans l'ordre de rangement dans la liste, et à les insérer dans une liste L_1 au bon emplacement.

Supposons que l'on ait déjà trié les n nombres d'indices $i = 0$ à $i = n - 1$ de L. Ces nombres se trouvent dans la liste L_1 dans l'ordre croissant. Le nombre $L[n]$ doit être inséré dans la liste L_1 juste après le nombre de L_1 le plus grand qui lui est inférieur ou égal, ou bien juste avant le nombre le plus petit qui lui est supérieur.

Pour éviter d'avoir à créer une seconde liste L_1, il est commode de travailler entièrement sur la liste L : les n premiers nombres de L sont alors les nombres déjà triés (c.a.d. la liste L_1). Voyons comment insérer le nombre $L[n]$. La figure suivante montre un exemple. La liste des nombres déjà triés est $(1, 3, 5, 6, 9)$ et le nombre à insérer est 2.



Une manière efficace de le faire est de mémoriser le nombre à insérer $L[n]$ puis de décaler vers la droite les nombres (par ordre d'indice décroissant) tant qu'ils sont supérieurs au nombre à insérer. Pour finir, le nombre à insérer est placé à l'emplacement du dernier nombre décalé.

2.b. Implémentation en python

Les nombres à trier est dans une liste $L[k]$ ($k = 0..N - 1$). Il s'agit de trier cette liste sans utiliser d'autre tableau. Le seul élément à mémoriser en plus est celui qui doit être inséré. On l'appelle la *clé*. La fonction de tri comporte une boucle principale d'indice n qui désigne l'élément à insérer. On commence à $n = 1$ car l'élément d'indice $n = 0$ est déjà en place au démarrage.

La clé doit être insérée dans la partie du tableau allant des indices 0 à $n-1$. Pour cela, il faut décaler d'un rang vers la droite les éléments qui sont supérieurs à la clé, en procédant par indice décroissant, à commencer par l'élément d'indice $n-1$. Il faut donc écrire une boucle pour décaler ces éléments. On note j l'indice de cette boucle. La boucle s'arrête lorsqu'on rencontre un nombre inférieur ou égal à la clé, ou lorsque l'indice $j=0$ est atteint, auquel cas la clé se retrouve en première position. Lorsque le décalage est terminé, on place la clé à sa position finale.

```
def tri_insertion(L):
    N = len(L)
    for n in range(1,N):
        cle = L[n]
        j = n-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j] # decalage
            j = j-1
        L[j+1] = cle
```

Voici un exemple, avec une liste initiale générée aléatoirement :

```
import random
liste = []
for k in range(10):
    liste.append(random.randint(0,20))
tri_insertion(liste)


print(liste)
--> [1, 1, 2, 5, 9, 9, 9, 11, 19, 19]
```

On remarque que cette fonction trie la liste fournie en argument. En effet, le nom `L` fait référence à une liste en mémoire. Lorsqu'on modifie un élément de cette liste, on ne change pas la liste elle-même. Si l'on ne veut pas modifier la liste transmise en argument de la fonctioon `tri_insertion`, il faut faire une copie de la liste et trier cette copie :

```
def tri_insertion(liste):
    L = list(liste) # copie de la liste
    N = len(L)
    for n in range(1,N):
        cle = L[n]
        j = n-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j] # decalage
            j = j-1
        L[j+1] = cle
    return L

liste = []
for k in range(10):
    liste.append(random.randint(0,20))
liste_triee = tri_insertion(liste)
```

```
print(liste)
--> [4, 4, 1, 14, 4, 3, 11, 19, 18, 18]

print(liste_triee)
--> [1, 3, 4, 4, 4, 11, 14, 18, 18, 19]
```

2.c. Analyse du temps d'exécution

On cherche à savoir comment le temps d'exécution de la fonction de tri par insertion dépend du nombre N d'éléments à trier. Cette dépendance est aussi appelée *complexité temporelle*. Le temps d'exécution dépend non seulement de la longueur de la liste à trier, mais aussi de l'ordre des nombres. Si les nombres sont déjà dans l'ordre croissant dans la liste de départ, la boucle de décalage n'est jamais exécutée. Comme il y a $N-1$ itérations dans la boucle principale, le temps d'exécution dans ce cas (cas le plus favorable) est de la forme :

$$t_1(N) = a(N-1) \tag{2}$$

La constante a dépend du temps qu'il faut pour exécuter les instructions dans la boucle, mais pas de N.

Le cas le plus défavorable est celui où les nombres de la liste initiale sont en ordre décroissant. Dans ce cas, la boucle de décalage est toujours exécutée, mais avec un nombre de décalages croissant. Pour la première clé, il y a 1 décalage, pour la clé suivante 2 décalages, jusqu'à la clé $N-1$, qui nécessite $N-1$ décalages. Le temps d'exécution a donc la forme suivante :

$$t_2(N) = b(1+2+\cdots+N-2+N-1) \tag{3}$$

Pour évaluer cette somme, on utilise le résultat suivant :

$$\sum_{k=1}^{N} k = \frac{1}{2}N(N+1) \tag{4}$$

ce qui donne :

$$t_2(N) = \frac{b}{2}(N-1)N \tag{5}$$

Dans le cas le plus défavorable, il existe deux constantes c_1 et c_2 tells que, pour N assez grand :

$$c_1 N^2 \leq t_2(N) \leq c_2 N^2$$

On écrira donc que le temps d'exécution dans le cas défavorable est $\Theta(N^2)$ (voir annexe). Dans le cas général, tout ce qu'on peut dire est qu'il existe une constante c telle que pour N assez grand :

$$t(N) \leq cN^2$$

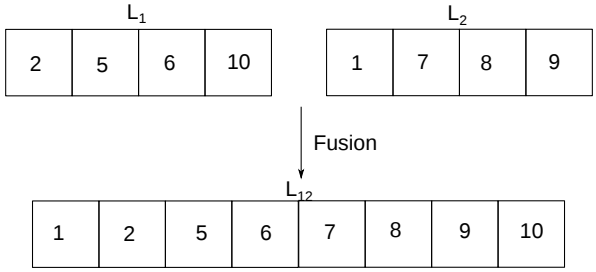
On écrira donc que le temps de calcul est en général $O(N^2)$ (ordre de grandeur maximal).

En pratique, il est intéressant de connaître le comportement asymptotique du temps de calcul dans le cas moyen, celui de N nombres tirés aléatoirement. Le cas moyen a une complexité $\Theta(N^2)$ comme le cas le plus défavorable, car le nombre de décalages est simplement divisé par deux par rapport au cas le plus défavorable.

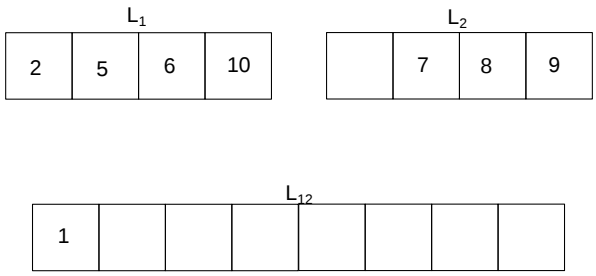
3. Tri par fusion

3.a. Fusion de deux listes

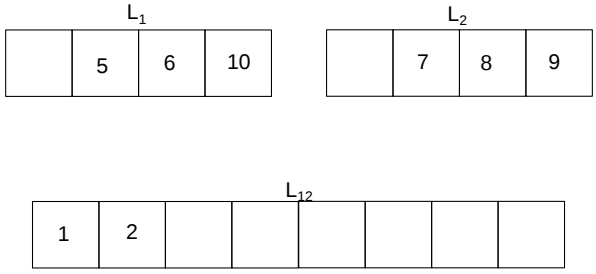
Soient deux listes triées L_1 et L_2. La fusion consiste à obtenir, à partir de ces deux listes, la liste triée contenant tous les éléments. La figure suivante montre un exemple de fusion, où deux listes de 4 nombres sont fusionnées.



L'algorithme de fusion des listes L_1 et L_2 consiste à créer la liste L_{12} en prenant les éléments de L_1 et L_2 dans l'ordre. On commence par comparer les deux premiers éléments de L_1 et L_2 (2 et 1 sur l'exemple). Le plus petit est placé en premier dans la liste L_{12}. On peut imaginer que l'élément placé dans la liste a été enlevé de la liste L_2, ce qui donne la figure suivante :



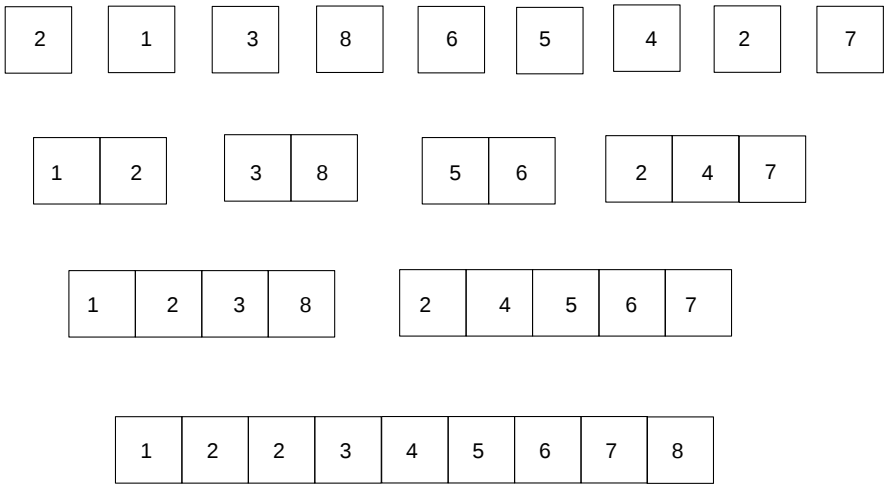
On compare les deux premiers éléments des deux listes (ici 2 et 7) pour placer le plus petit en deuxième position :



L'opération est répétée jusqu'à épuisement d'une des deux listes. Lorsqu'une des listes est vide, on doit placer les éléments de l'autre liste dans l'ordre.

3.b. Tri par fusion

Dans la liste à trier, on commence par fusionner les éléments deux à deux pour obtenir des listes triées de deux éléments. Si le nombre d'éléments est impair, la dernière liste en comporte trois. Voici un exemple :



La deuxième étape consiste à fusionner les listes de 2 (ou 3) éléments pour obtenir des listes de 4 (ou 5) éléments. La dernière étape fusionne une liste de 4 éléments avec une liste de 5 éléments.

3.c. Tri récursif

L'algorithme de tri par fusion peut être formulé de manière récursive, ce qui en facilite l'implémentation.

Divisons la liste initiale en deux listes, la première allant de l'indice 0 à la partie entière de $N/2$. Les deux sous-listes ont la même taille à une unité près. L'étape suivante consiste à trier ces deux sous-listes avant de les fusionner. Ces deux sous-listes sont elles-mêmes triées par fusion de deux sous listes. On obtient ainsi un algorithme récursif. La récursion s'arrête lorsqu'on obtient une liste d'un seul élément.

Cet algorithme fait partie de la classe des algorithmes de type *diviser pour régner*, pour lesquels la tâche à accomplir peut être scindée en deux tâches similaires à la première.

3.d. Implémentation en python

On commence par implémenter la fusion de deux listes :

```
def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0
    i2 = 0
    i = 0
    while i1<n1 and i2<n2:
        if L1[i1] < L2[i2]:
            L12[i] = L1[i1]
            i1 += 1
```

```
        else:
            L12[i] = L2[i2]
            i2 += 1
        i += 1
    while i1<n1:
        L12[i] = L1[i1]
        i1 += 1
        i += 1
    while i2<n2:
        L12[i] = L2[i2]
        i2 += 1
        i += 1
    return L12
```

Voici un test :

```
L = fusion([1,6,10],[0,7,8,9])

print(L)
--> [0, 1, 6, 7, 8, 9, 10]
```

Pour l'implémentation du tri par fusion, on a le choix entre une implémentation récursive et une implémentation itérative. En effet, l'état final de la récursion est connu, puisqu'il s'agit de la liste de départ non triée. Nous avons vu dans le cours Piles et récursion un exemple d'algorithme (la triangulation d'un triangle), qui est nécessairement implémentée de manière récursive. Ce n'est pas le cas ici, mais l'implémentation récursive est probablement plus simple à faire, et donnera un code plus clair.

On écrit une fonction récursive qui modifie la liste L fournie en argument.

```
def tri_fusion_recursif(L):
    n = len(L)
    if n > 1:
        p = int(n/2)
        L1 = L[0:p]
        L2 = L[p:n]
        tri_fusion_recursif(L1)
        tri_fusion_recursif(L2)
        L[:] = fusion(L1,L2)

def tri_fusion(L):
    M = list(L)
    tri_fusion_recursif(M)
    return M


liste = []
for k in range(11):
    liste.append(random.randint(0,20))
liste_triee = tri_fusion(liste)

print(liste)
--> [8, 8, 10, 18, 10, 11, 11, 20, 17, 13, 17]
print(liste_triee)
--> [8, 8, 10, 10, 11, 11, 13, 17, 17, 18, 20]
```

3.e. Analyse du temps d'exécution

Le temps d'exécution du tri par fusion ne dépend que de la taille N de la liste et pas du contenu de cette liste, ce qui rend aisé la détermination de la complexité temporelle.

Voyons tout d'abord le temps d'exécution des fusions à un niveau de la récursion, par exemple les fusions des listes de 2 éléments. Le temps d'exécution de la fonction `fusion` est proportionnel à la taille de la liste `L12` obtenue. Le temps d'exécution de toutes les fusions à un niveau quelconque est donc proportionnel à N.

Il faut aussi compter le nombre de récursions. Pour simplifier, supposons que N soit une puissance de 2, c'est-à-dire $N = 2^r$, qui s'écrit aussi :

$$\ln(N) = r\ \ln(2) \tag{6}$$

Le nombre de récursions est r. Il est donc proportionnel au logarithme de N. Finalement le temps d'exécution du tri par fusion s'écrit :

$$t(N) = a\,N\ \ln(N) \tag{7}$$

L'ordre de grandeur du temps d'exécution est donc $\Theta(N\ \ln(N))$.

On peut maintenant comparer avec le tri par insertion, dont le temps d'exécution est quadratique (dans le pire des cas et dans le cas moyen) :

$$\frac{t_{fus}}{t_{ins}} = \frac{a_{fus}}{a_{ins}}\frac{\ln N}{N} \tag{8}$$

Lorsque N tend vers l'infini, ce rapport tend vers 0. Pour les très grandes listes, on a donc intérêt à utiliser le tri par fusion. Cela ne signifie pas que le tri par fusion soit toujours plus rapide, car ce rapport dépend aussi des constantes, qui sont déterminées par le détail de l'implémentation (nombre de tests, de lecture et écriture en mémoire, etc.).

Pour avoir plus d'informations, le plus simple est de faire une comparaison expérimentale. La fonction suivante compare les temps d'exécution en effectant un grand nombre de tris :

```
import time
import numpy

def rapport_temps(N):
    n = 100
    t1 = time.time()
    for k in range(n):
        liste = numpy.random.randint(0,N,size=N)
        tri_insertion(liste)
    t1 = time.time()-t1
    t2 = time.time()
    for k in range(n):
        liste = numpy.random.randint(0,N,size=N)
        tri_fusion(liste)
    t2 = time.time()-t2
    return t2/t1


print(rapport_temps(10))
--> 1.4182534025727183

print(rapport_temps(50))
--> 1.050666196952731
```

```
print(rapport_temps(100))
--> 0.6850070257611242

print(rapport_temps(1000))
--> 0.07128893390526313
```

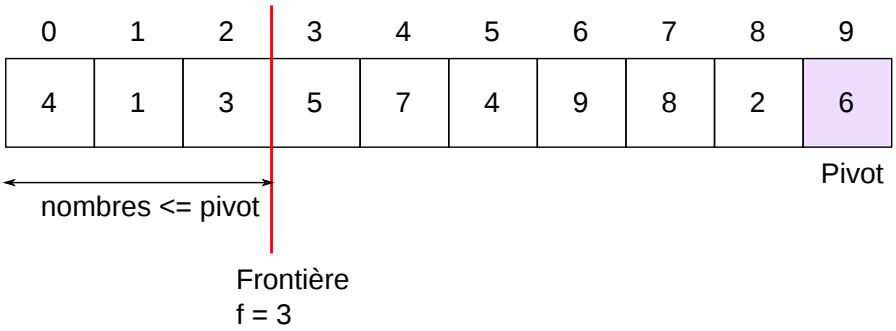
Le gain de temps est en faveur du tri par fusion dès que $N > 50$. Pour les petites listes (de l'ordre de la dizaine), on a ici intérêt à utiliser le tri par insertion (mais la conclusion peut être différente avec une autre implémentation ou un autre langage).

4. Tri par partitionnement (tri rapide)

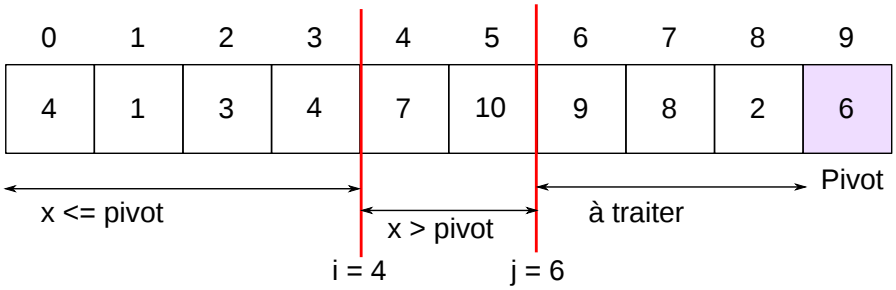
4.a. Algorithme

Le tri par fusion fonctionne sur le principe *diviser pour régner*, qui consite à diviser la tâche initiale en deux tâches similaires plus petites. Une autre manière de diviser la tâche est le partitionnement.

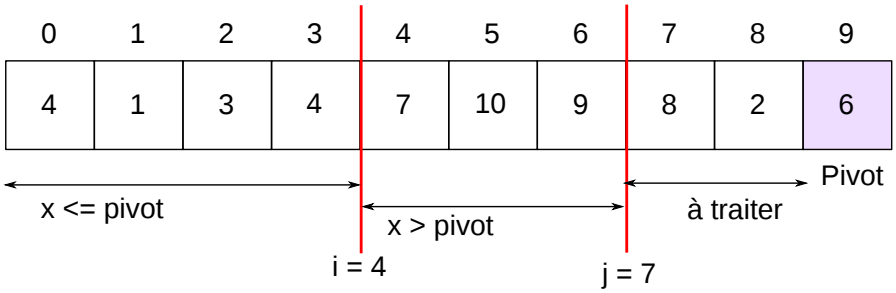
Le partitionnement consiste à choisir un élément dans la liste, appelé le pivot, puis à réarranger la liste pour que tous les éléments inférieurs ou égaux au pivot soient situés avant le pivot et tous les éléments supérieurs au pivot soient situés après le pivot. On peut choisir le dernier élément de la liste comme pivot (cela facilite la suite). Il s'agit de constituer deux listes (dont on ne connaît pas *a priori* les tailles) : la première constituée des nombres inférieurs ou égaux au pivot, la seconde des nombres supérieurs au pivot. Il est possible de construire ces deux listes dans le tableau initial, sans utiliser de tableau de stockage auxiliaire. On utilise pour cela un indice, noté i, qui marque la frontière entre les deux listes. Plus précisément, cet indice correspond au nombre d'éléments avant la frontière. Un deuxième indice, noté j, marque la fin de la seconde liste, celle dont les nombres sont supérieurs au pivot. La figure suivante montre un exemple, avec un partitionnement en cours de réalisation.



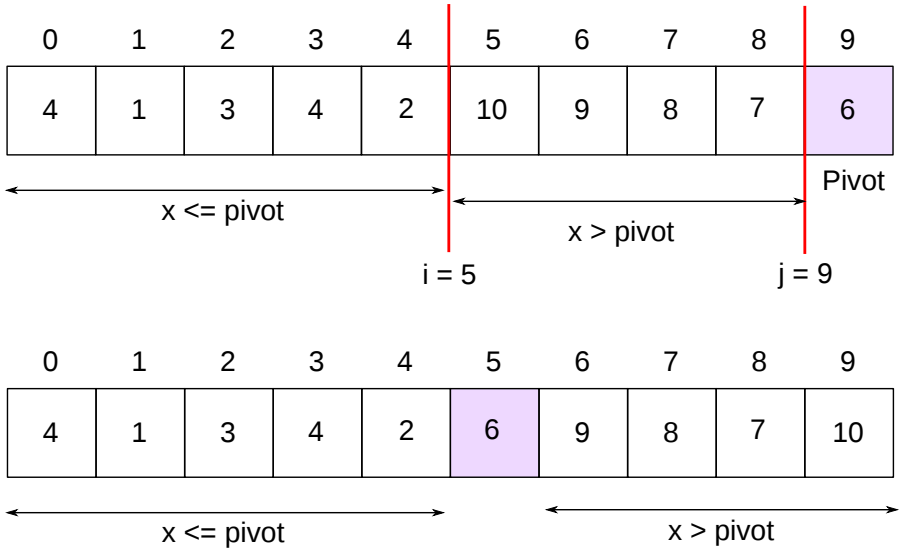
L'élément suivant à traiter est celui d'indice $j = 5$. Il est inférieur au pivot. Il faut donc l'échanger avec l'élément d'indice i et faire avancer les deux frontières d'une unité, ce qui conduit à :



Le nombre suivant (d'indice 6) est supérieur au pivot. Dans ce cas, il suffit de déplacer la frontière j :



Lorsque tous les éléments ont été traités, il faut échanger le pivot avec l'élément d'indice i :



4.b. Tri récursif

La liste initiale est partitionnée autour de son dernier élément. Les deux listes obtenues de part et d'autre du pivot sont à leur tour partitionnées selon la même méthode, ce qui conduit à un algorithme récursif. La récursion s'arrête lorsque la liste ne contient plus qu'un élément.

4.c. Implémentation en python

On implémente tout d'abord le partitionnement :

```
def partition(L):
    n = len(L)
    pivot = L[n-1]
    i = 0
    j = 0
    while j < n-1:
```

```
        if L[j] <= pivot:
            L[i],L[j] = L[j],L[i]
            i += 1
        j += 1
    L[n-1],L[i] = L[i],L[n-1]
```

Voici un test :

```
liste = []
for k in range(11):
    liste.append(random.randint(0,40))


print(liste)
--> [15, 28, 10, 35, 28, 5, 0, 33, 39, 5, 15]

partition(liste)

print(liste)
--> [15, 10, 5, 0, 5, 15, 35, 33, 39, 28, 28]
```

Pour l'implémentation récursive, il est plus simple de disposer d'une fonction de partitionnement qui opère directement sur la liste à N éléments. Nous allons donc modifier la fonction de partitionnement précédente pour qu'elle opère sur une partie de la liste. `debut` est le premier indice et `fin` le dernier indice de la sous-liste à partitionner. La fonction renvoie l'indice du pivot à la fin du partitionnement.

```
def partition(L,debut,fin):
    pivot = L[fin]
    i = debut
    j = debut
    while j < fin:
        if L[j] <= pivot:
            L[i],L[j] = L[j],L[i]
            i += 1
        j += 1
    L[fin],L[i] = L[i],L[fin]
    return i
```

Voici le tri récursif. La récursion s'arrête lorsque debut=fin.

```
def tri_partition_recursif(L,debut,fin):
    if debut < fin:
        i = partition(L,debut,fin)
        tri_partition_recursif(L,debut,i-1)
        tri_partition_recursif(L,i+1,fin)
```

Voici la fonction de démarrage :

```
def tri_partition(liste):
    L = list(liste)
    tri_partition_recursif(L,0,len(L)-1)
    return L
```

Voici un exemple :

```
liste = []
for k in range(11):
    liste.append(random.randint(0,40))
liste_triee = tri_partition(liste)


print(liste)
--> [23, 27, 15, 13, 4, 35, 38, 27, 12, 34, 36]

print(liste_triee)
--> [4, 12, 13, 15, 23, 27, 27, 34, 35, 36, 38]
```

4.d. Analyse du temps d'exécution

Faisons tout d'abord une comparaison expérimentale avec le tri par fusion :

```
def rapport_temps(N):
    n = 100
    t1 = time.time()
    for k in range(n):
        liste = numpy.random.randint(0,N,size=N)
        tri_fusion(liste)
    t1 = time.time()-t1
    t2 = time.time()
    for k in range(n):
        liste = numpy.random.randint(0,N,size=N)
        tri_partition(liste)
    t2 = time.time()-t2
    return t2/t1


print(rapport_temps(10))
--> 0.6665338645418327

print(rapport_temps(50))
--> 0.68748412544262

print(rapport_temps(100))
--> 0.6060575327977514

print(rapport_temps(1000))
--> 0.7008155323445183
```

Le tri par partitionnement, appelé aussi tri rapide, est ici plus rapide que le tri par fusion, mais le rapport ne semble pas dépendre de N, ce qui suggère une dépendance du temps d'exécution par rapport à N similaire, en $N\ ln(N)$.

Le temps d'exécution du tri par partitionnement dépend de l'ordre des nombres dans la liste de départ. En effet, le partitionnement d'une liste peut produire deux listes de tailles voisines, ou au contraire deux listes de tailles très différentes.

Le cas le plus défavorable se produit lorsque le partionnement de chaque liste à n éléments produit une liste à $n-1$ éléments et une liste à 0 éléments. Il s'agit d'un partitionnement complètement déséquilibré. Dans notre implémentation où le pivot est le dernier élément de la liste, cela se produit si la liste de départ est déjà triée. Le partitionnement d'une liste à n éléments prend un temps proportionnel à n car il comporte une boucle (indice j). Dans le cas le plus défavorable, le temps d'exécution total est donc :

$$t_1(N) = a(N-1+N-2+\cdots+2+1) = \frac{a}{2}N(N+1) \tag{9}$$

ce qui donne un ordre de grandeur $\Theta(N^2)$.

Dans le cas le plus défavorable, le tri par partionnement est donc aussi lent que le tri par insertion. De plus, pour une liste déjà triée, le tri par insertion est en fait meilleur car dans ce cas sa complexité est $\Theta(N)$.

Le cas le plus favorable se produit lorsque tous les partitionnements sont parfaitement équilibrés, c'est-à-dire lorsqu'une liste de taille n est partitionnée en une liste de taille $n/2$ et une autre de taille $n/2-1$. Le temps d'exécution total a alors la forme suivante :

$$t_2(N) = a\left(N + \frac{N}{2} + \frac{N}{2} - 1 + \frac{N}{4} + \frac{N}{4} - 1 + \frac{N}{4} + \frac{N}{4} - 1 + \cdots\right) \tag{10}$$

Le nombre de récursions est proportionnel à $ln(N)$, et non pas à N comme dans le cas défavorable. L'ordre de grandeur du temps d'exécution est alors $\Theta(N\ln(N))$

Dans le cas le plus favorable, le tri par partitionnement a donc la même complexité temporelle que le tri par fusion.

L'analyse du cas moyen est plus difficile, mais on voit bien que le cas le plus défavorable est très improbable (liste déjà triée). En moyenne, le partitionnement sera équilibré à certains niveaux de récursion et déséquilibré à d'autres niveaux. Le comportement moyen est proche du cas le plus favorable (on l'admet). Le tri par partitionnement se comporte donc aussi bien que le tri par fusion, ce que confirme l'étude expérimentale ci-dessus. Le temps d'exécution est même plus faible pour le tri par partitionnement.

5. Annexe : complexité temporelle asymptotique

La complexité temporelle d'un algorithme est la relation entre son temps d'exécution $t(N)$ et le nombre de données N qu'il traite, par exemple la taille de la liste à trier. On s'intéresse généralement à *l'ordre de grandeur de ce temps*, c'est-à-dire son comportement asymptotique lorsque N est très grand.

La notation *Theta* est utilisée lorsqu'il est possible de trouver un encadrement du temps de calcul. On écrit que la complexité (temporelle) est $\Theta(f(N))$ s'il existe deux constantes c_1 et c_2 telle que, pour N assez grand on ait :

$$c_1 f(N) \leq t(N) \leq c_2 f(N)$$

Par exemple, si la complexité est quadratique :

$$c_1 N^2 \leq t(N) \leq c_2 N^2$$

Dans ce cas, on peut aussi dire que l'ordre de grandeur du temps d'exécution est N^2.

Lorsqu'il est seulement possible de majorer le comportement asymptotique du temps de calcul, on utilise la notation *grand O*. On écrit que la complexité est $O(f(N))$ s'il existe une constante c telle que, pour N assez grand on ait :

$$t(N) \leq cf(N)$$

Par exemple, on dira que la complexité est grand $O(N^2)$ s'il existe une constante c telle que pour N assez grand :

$$t(N) \leq cN^2$$

Dans ce cas, il n'est pas exlu que, sous certaines conditions, le temps d'exécution soit par exemple $\Theta(N)$. La notation grand O est utilisée lorsqu'on souhaite exprimer la complexité dans le pire des cas.