IF5 – Algorithmes de tri

L'objectif de ce chapitre est de mettre en place des algorithmes qui réalisent un tri ordonné d'une liste aléatoire. Nous allons voir dans un premier temps des tris par comparaison puis des tris par comptage.

Evidemment, un algorithme de tri est efficace si, lorsqu'il retourne la liste triée, il a effectué ce tri en un temps le plus court possible. Le caractère aléatoire de la liste à trier va nous conduire à discuter des complexités dans le meilleur et le pire des cas.

Objectifs :
- Être capable de programmer des tris par sélection, par insertion, par fusion ou « rapides ».
- Retenir le coût des différents tris.
- Reconnaître un tri par comptage et estimer son coût.

I – Introduction

I-1) Contexte

On considère des données numériques. Trier ces données consiste à les ranger en ordre croissant ou décroissant. Une opération de tri consomme un temps de calcul important sur un ordinateur et il donc nécessaire d'étudier la complexité temporelle des différents algorithmes de tri. On peut évaluer cette complexité dans le "pire des cas", ou dans le "meilleur des cas" ou enfin "en moyenne" et ensuite utiliser l'algorithme qui convient le mieux suivant la situation.

Les algorithmes étudiés, dans un premier temps, sont basés sur des comparaisons successives entre les données et éventuellement une permutation des éléments comparés. Le nombre de permutations est toujours inférieur au nombre de comparaisons. Ainsi la complexité d'un algorithme est du même ordre de grandeur que le nombre de comparaisons effectuées par cet algorithme.

Il y a $n!$ manières de ranger n données ($n!$ permutations). La première comparaison concerne deux des données a et b de la liste et consiste à poser la question : a<b ? La réponse permet de diviser les $n!$ manières en deux parties égales. Donc après k comparaisons, il restera $\frac{n!}{2^k}$ permutations à envisager. Le tri sera terminé lorsqu'il ne restera plus qu'une permutation, soit si :

$$\frac{n!}{2^k} \leq 1 \Rightarrow 2^k \geq n! \Rightarrow kln(2) \geq \ln(n!)$$

Dès que n est grand on peut utiliser la formule de Stirling :

$$\ln(n!) \sim nln(n)$$
$$\Rightarrow k \gtrsim \frac{nln(n)}{\ln(2)}$$

Conclusion : Le minimum de comparaisons est donc de l'ordre de $n\,log_2(n)$.

I-2) Fonctions natives de python

Introduisons deux fonctions que python propose nativement « .sort » et « sorted(L) ». Par exemple :

```
import random
n=20
L=[random.randint(1,n) for i in range (n)]
LL=sorted(L)
#L.sort() #Même rôle sur sorted sauf qu'on perd la liste initiale
print(L)
print(LL)
```

```
[16, 18, 17, 20, 12, 14, 6, 10, 12, 15, 6, 12, 19, 4, 9, 9, 5, 4, 4, 8]
[4, 4, 4, 5, 6, 6, 8, 9, 9, 10, 12, 12, 12, 14, 15, 16, 17, 18, 19, 20]
```

L'algorithme de tri utilisé est le « Timsort », du nom de son inventeur Tim Peters, en 2002. C'est un algorithme performant, dérivé de l'algorithme du tri fusion que nous étudierons par la suite.

I-3) Type de tris

- Clefs : c'est ce qui est utilisé pour trier des éléments. Par exemple :
  - On peut trier des mots à l'aide de leur première lettre. La clé est ainsi la première lettre.

- On peut trier des couples (4,5) (1,2) (1,3) (2,3) (3,1) selon leur premier terme, ce sera donc la clé, ou selon leur deuxième terme…

- Tri comparatif : tri fondé sur la comparaison entre les « clefs » des éléments pour les trier.
- Tri itératif : tri basé sur un ou plusieurs parcours itératifs de la liste à trier
- Tri récursif : Tri basé sur une méthode récursive
- Tri en place : Un tri est dit en place s'il modifie directement la structure qu'il est en train de trier. Il n'utilise pas de liste auxiliaire.
- Tri stable : Tri qui conserve l'ordre initial des éléments de même clef. Deux éléments avec des clefs égales apparaîtront dans le même ordre dans la liste triée et non triée.

II – Tris quadratiques

II-1) Tri par sélection

a) Principe

On dispose de n données. On cherche la plus petite donnée et on la place en première position, puis on cherche la plus petite donnée parmi les données restantes et on la place en deuxième position, et ainsi de suite...Si les données sont les éléments d'une liste, l'algorithme consiste donc à faire varier un indice i de 0 à n−2.

Pour chaque valeur de i, on cherche dans la tranche liste[i:n] le plus petit élément et on l'échange avec liste[i]. On répète la recherche d'un minimum.

L	L'	L''	L'''	L''''
7	7	7	7	8
3	3	3	8	7
8	8	8	3	3
1	2	2	2	2
2	1	1	1	1

```
def tri_selection(liste):
    n=len(liste)
    for i in range(0,n-1): #On choisit un élement de la liste (on peut mettre n mais le dernier élément est déjà trié)
        i_mini=i #On va stocker le premier élément comme grandeur à comparer
        for j in range(i+1,n): #Je compare l'élément choisi aux autres
            if liste[j]<liste[i_mini]:
                i_mini=j
        liste[i],liste[i_mini]=liste[i_mini],liste[i] #si la donnée j est plus petite, on permute
    return liste

LLL=tri_selection(L)
print(LLL)
```

b) Type de tri

Le tri par sélection est un tri comparatif, en place mais n'est pas stable.

- Tri comparatif : on compare les éléments de la liste (liste[j]<liste[i_mini]).
- Tri itératif : on parcourt plusieurs fois la liste.
- Tri en place : on modifie la liste directement, on n'en crée pas d'autres.
- Tri non stable :
  - Prenons la liste suivante [4,4*,2,1].
  - Pour i égal à 0 : [1,4*,2,4]
  - Pour i égal à 1 : [1,2,4*,4]
  - Pour i égale à 2 : [1,2,4*,4]

  On remarque que la position des deux 4 a été inversée d'où le tri non stable.

c) Complexité

Pour chaque valeur de i, la seconde boucle effectue exactement n-i-1 comparaisons. Comme i varie de 0 à n-2, nous obtenons : $(n-1)+(n-2)+\ldots+2+1$ comparaisons, soit $\frac{n(n-1)}{2}$ comparaisons.

Le coût est donc de l'ordre de n^2 quelle que soit la liste, même si elle est déjà triée. Cela signifie que le tri par sélection n'est pas efficace. Il est cependant simple à programmer. L'algorithme de tri par sélection sur une liste de n éléments a un coût quadratique en fonction de n. Le nombre de comparaisons est de l'ordre de n^2.
Dans le pire des cas, le meilleur des cas et en moyenne :

$$C_n(sélection) \sim n^2$$

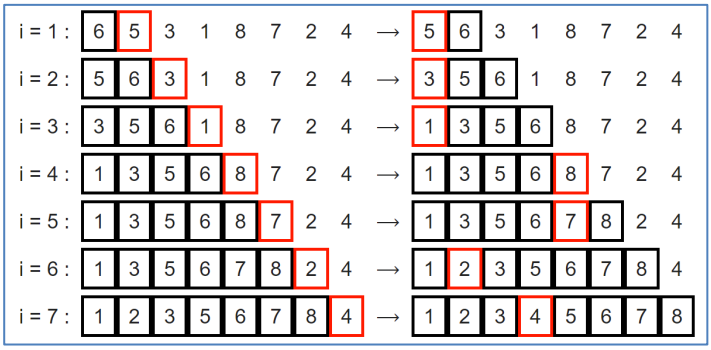
II-2) Tri par insertion

a) Principe

Le tri par insertion consiste à insérer les éléments d'une partie de la liste non triée dans la liste triée.

Pour chaque valeur de i, on cherche dans la liste liste $[0: i+1]$ à quelle place doit être inséré l'élément $liste[i+1]$ qu'on appelle la clé. Pour cela on compare la clé successivement aux données précédentes, jusqu' à trouver la bonne place, c'est-à-dire entre deux données successives, l'une étant plus petite et l'autre plus grande que la clé. Pour ce faire, on décale d'une place vers le haut les données plus grandes que la clé après chaque comparaison. Voici un exemple illustré avec la clé en orange :

L	L'	L''	L''
7	7	7	7
3	3	3	8
8	8	8	3
1	2	2	2
2	1	1	1



D'où l'algorithme suivant :

```
import random
n=10
L=[random.randint(1,100) for i in range(n)]
print(L)

def tri_insertion(liste):
    n=len(liste)
    for i in range(1,n):
        cle=liste[i] #élement à tester dans la liste déjà triée.
        k=i-1 #On va tester l'élément L[k] de la liste déjà triée.
        while k>=0 and cle<liste[k]:#La condition k>=0 empêche de rentrer dans une boucle infinie.
            liste[k+1]=liste[k] #On fait remonter les éléments afin de préparer la place pour la clé.
            k=k-1 #On descend dans la partie de liste déjà triée, et on teste l'élement suivant.
        liste[k+1]=cle #Boucle finie, On insère la clé à la bonne position.
    return liste

LLL=tri_insertion(L)
print(LLL)
```

```
[66, 87, 73, 81, 21, 13, 39, 46, 18, 36]
[13, 18, 21, 36, 39, 46, 66, 73, 81, 87]
```

b) Type de tri

Le tri par insertion est :

- Un tri par comparaison ;
- Un tri en place : on modifie la liste existante ;
- Un tri stable : deux éléments de même valeur placés dans un certain ordre au départ restent dans le même ordre après le tri.

c) Terminaison

La suite des valeurs de « k » dans la boucle « while » est à valeurs entières et strictement décroissante. Cette boucle termine donc forcément grâce à la condition $k \geq 0$.

d) Correction

On a pour invariant de boucle que « la liste formée des éléments de L d'indice inférieur à k est triée »

e) Complexité

Nous avons deux boucles imbriquées. Pour une liste de longueur n, le nombre de comparaisons peut être différent suivant la liste.

- Si la liste est déjà triée, pour chaque valeur de i, k prend la valeur de $i-1$ et il y a une seule comparaison, le test $clé < liste[k]$. La variable i prenant n-1 valeurs, cela nous fait un total de n-1 comparaisons.

Dans le meilleur des cas, la complexité de l'algorithme est donc de l'ordre de n.

- Si par contre les éléments de la liste sont rangés dans l'ordre décroissant, alors pour chaque valeur de i, k prend les valeurs de i-1 à 0 soit i valeurs et donc i comparaisons. Au total, nous avons donc : $1+2+\ldots+(n-2)+(n-1)$ comparaisons.

$$\Rightarrow C_n = \frac{(n-1)(n-2)}{2} \sim n^2$$

Dans le pire des cas la complexité est donc de l'ordre de n^2 comparaisons.

- On peut démontrer qu'en moyenne, le coût est de l'ordre de n^2 comparaisons, comme pour le tri par sélection.

$$\Rightarrow C_n(insertion) \sim n^2$$

Le tri par insertion est très intéressant si la liste est "presque triée". Dans le pire des cas, et en moyenne, l'algorithme de tri par insertion sur une liste de n éléments a un coût quadratique en fonction de n. Le nombre de comparaisons est de l'ordre de n^2.

II-3) Tri à bulles

Le tri a bulles consiste à comparer les deux premiers éléments d'une liste L et à les échanger s'ils ne sont pas triés par ordre croissant. On recommence ensuite avec le deuxième et le troisième élément de la liste, et ainsi de suite…

Après chaque parcours complet de la liste, on recommence l'opération, ainsi les plus grands éléments remontent de proche en proche vers la droite comme des bulles vers la surface.

Lorsqu'aucun échange a lieu pendant un parcours, cela signifie que la liste est triée et on arrête l'algorithme.

Illustration sur la liste $L = [5,1,4,2,8]$

5	1	4	2	8	⇒	1	5	4	2	8
1	5	4	2	8	⇒	1	4	5	2	8
1	4	5	2	8	⇒	1	4	2	5	8
1	4	2	5	8	⇒	1	4	2	5	8
1	4	2	5	8	⇒	1	4	2	5	8
1	4	2	5	8	⇒	1	2	4	5	8
1	2	4	5	8	⇒	1	2	4	5	8
1	2	4	5	8	⇒	1	2	4	5	8

- Les éléments 5 et 1 sont comparés puis inversés.
- De même pour 5 et 4
- De même pour 5 et 2
- Par contre pour 5 et 8, cela reste inchangé.
- On reparcourt le tableau et on inverse 4 et 2
- La liste est triée mais l'algorithme doit le vérifier
- D'où un dernier test.

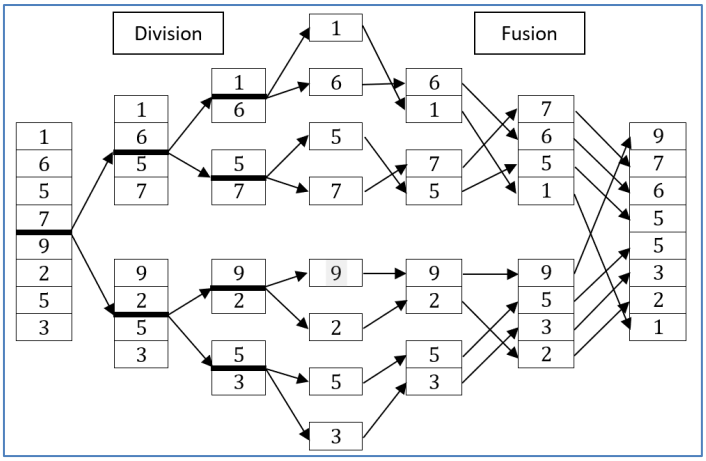
L'algorithme sera étudié en TD.

III – Stratégie « diviser pour régner »

III-1) Tri fusion

a) Principe

Le tri par fusion consiste à réunir deux sous-listes triées en une seule liste. Il s'appuie sur la méthode ≪ diviser pour régner ≫. On partage la liste initiale en deux sous-listes de longueurs quasi-égales que l'on trie de façon récursive. Il ne reste plus qu'à fusionner ces deux sous-listes triées.



D'où l'algorithme suivant en deux parties :

- La fonction fusion :

Comme les deux listes : liste1 et liste2 sont déjà triées, le premier élément de la fusion est le plus petit élément entre *liste*1[0] et *liste*2[0] puis on passe à l'élément suivant de la liste qui a déjà eu un élément trié et ainsi de suite jusqu'à avoir vérifié les éléments des deux listes.

```
def fusion(liste1, liste2):
    liste_fus=[]
    i,j = 0,0
    while i<len(liste1) and j<len(liste2):
        if liste1[i]<=liste2[j]: #On compare les éléments des deux sous-listes triées
            liste_fus.append(liste1[i]) #On ajoute le plus petit des éléments à la liste en train d'être triée.
            i=i+1 #On incrémente pour passer à l'élément suivant de la liste qui a <perdu> un élément
        else:
            liste_fus.append(liste2[j]) #Sinon on fait de même sur l'autre liste.
            j=j+1
    for k in range(i,len(liste1)): #s'il reste des éléments dans liste1
        liste_fus.append(liste1[k])
    for k in range(j,len(liste2)): #s'il reste des éléments dans liste2
        liste_fus.append(liste2[k])
    return liste_fus
```

Ou :

```
def fusion(liste1, liste2):
    liste_fus=[]
    i,j = 0,0
    while i<len(liste1) and j<len(liste2):
        if liste1[i]<=liste2[j]: #On compare les éléments des deux sous-listes triées
            liste_fus.append(liste1[i]) #On ajoute le plus petit des éléments à la liste en train d'être triée.
            i=i+1 #On incrémente pour passer à l'élément suivant de la liste qui a <perdu> un élément
        else:
            liste_fus.append(liste2[j]) #Sinon on fait de même sur l'autre liste.
            j=j+1
    return liste_fus+liste1[i:]+liste2[j:] #On peut aussi rajouter les éléments finaux de cette manière
```

- La fonction tri-fusion :

Si la liste contient un seul élément, elle est déjà triée : c'est la condition d'arrêt de la fonction récursive. Sinon, on partage la liste en deux sous-listes liste1 et liste2 que l'on trie récursivement en utilisant la fonction «tri-fusion». Il ne reste plus qu'à fusionner les deux sous-listes triées.

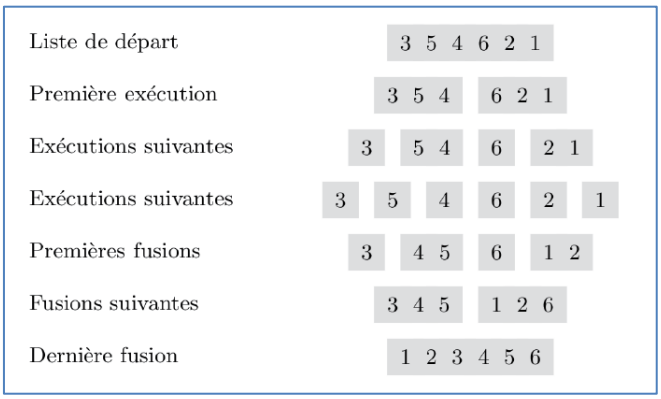
```
def tri_fusion(liste):
    if len(liste)<=1: #Notre condition d'arrêt
        return liste
    else: #Ici c'est la partie division
        milieu = len(liste)//2 #On prépare notre découpe de manière quasi-égale.
        liste1 = tri_fusion(liste[:milieu]) #Récursivité, on découpe la liste en 2 parties (ici on prend les premiers éléments)
        liste2 = tri_fusion(liste[milieu:]) #jusqu'à avoir un élément dans chaque liste (ici on prend les derniers éléments)
                                            #que l'on trie/merge avec la fonction fusion
    return fusion(liste1, liste2) #Ici c'est la partie fusion
```

Puis on teste notre algorithme :

```
#On teste notre algorithme
import random
n=10
L=[random.randint(1,100) for i in range(n)]
print(L)
a=tri_fusion(L)
print(a)
```

```
[100, 26, 99, 88, 49, 52, 46, 93, 11, 16]
[11, 16, 26, 46, 49, 52, 88, 93, 99, 100]
```

Voici, un exemple de fonctionnement de la fonction fusion sur deux listes triées :



b) Type de tri

Le tri fusion est un :

- Tri récursif.
- Tri non en place : en effet il y a création de sous-listes.
- Tri stable.

c) Complexité

Supposons que la taille du tableau initial est $n = 2^p$ où $p \geq 1$. On a donc une complexité $C(n)$ tel que :

$$C(n) = 2C\left(\frac{n}{2}\right) + n_{fusion}$$

Or n_{fusion} égale à :

- $2^0 = 1$ fusion de 2^1 listes de taille $\frac{n}{2^1}$.
- 2^1 fusions de 2^2 listes de taille $\frac{n}{2^2}$.
- $\frac{2^p}{2} = 2^{p-1}$ fusions de 2^p listes de taille $\frac{n}{2^p}$.

Ainsi $n_{fusion} = 1 + 2 + \cdots + 2^{p-1} = \frac{1-2^p}{1-2} = 2^p - 1 \sim n$

$$\Rightarrow C(n) = 2C\left(\frac{n}{2}\right) + n$$

En effet il y a deux appels récursifs avec des tableaux de taille divisée par deux puis n fusions. D'où :

$$\frac{C(n)}{n} = \frac{2}{n} \times C\left(\frac{n}{2}\right) + 1 \Leftrightarrow \frac{C(2^p)}{2^p} = \frac{C(2^{p-1})}{2^{p-1}} + 1$$

$$On\ pose\ u_p = \frac{C(2^p)}{2^p} \Rightarrow u_p = u_{p-1} + 1$$

On reconnaît une suite arithmétique de raison 1 avec $u_0 = \frac{C(2^0)}{2^0} = 1$.

$$\Rightarrow u_p = u_0 + p \times r = 1 + p$$

$$\Rightarrow \frac{C(2^p)}{2^p} = (1 + p)$$

$$\Rightarrow C(2^p) = \left(1 + \underbrace{p}_{\log_2 n}\right) \times \underbrace{2^p}_{n}$$

On suppose p grand :

$$\Rightarrow C(n) \sim n \log_2 n$$

De manière générale, le tri fusion d'une liste de taille n, a une complexité de l'ordre de $n \log_2 n$.

III-2) Tri rapide

a) Principe

Appelé quick sort en anglais, ce tri adopte lui aussi une démarche de type « diviser pour régner ».

Le tri rapide s'appuie sur la méthode ≪ diviser pour régner ≫ comme le tri par fusion.

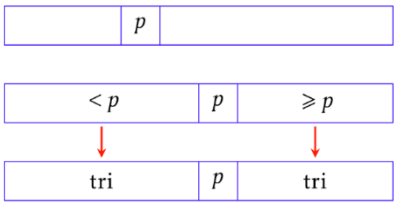
- On choisit dans la liste arbitrairement un pivot p (souvent le premier ou dernier élément de la liste). On enlève le pivot p de la liste.
- On segmente la liste en deux sous-listes, l'une contenant les éléments inférieurs ou égaux à p, l'autre les éléments restants strictement supérieurs à p.

On recommence récursivement avec les listes situées à gauche et à droite du pivot pour rassembler le tout.

a. Choix d'un pivot

b. Partitionnage

c. Appels récursifs



L'algorithme est aussi formé de deux fonctions principales :

- La fonction pivot :

La liste de nombres à trier est partagée en deux parties à l'aide d'une valeur choisie nommée le pivot. Une partie contient les valeurs plus petites que le pivot et l'autre les valeurs plus grandes. Voici la méthode décrite par *Anthony Hoare en 1960* :

- On choisit un élément de la liste qui est le pivot ;
- On parcourt les éléments de la liste à l'aide deux indices appelés respectivement indice gauche et indice droit, notés g et d ;
- L'indice gauche commence à 1 et on se déplace vers la droite en l'augmentant d'une unité tant que l'on rencontre des valeurs inférieures ou égales au pivot ;
- L'indice droit commence à n et on se déplace vers la gauche en le diminuant d'une unité tant que l'on rencontre des valeurs supérieures ou égales au pivot limite ;
- Les deux valeurs où s'arrêtent les indices g et d sont du mauvais côté du pivot donc elles sont échangées ;
- Chaque indice est incrémenté ou décrémenté d'une unité dans la direction respective de déplacement et l'indice gauche recommence son déplacement ;
- Ce processus est reproduit jusqu'à ce que les indices g et d se croisent ;
- On place le pivot à la bonne place, par exemple en l'échangeant avec l'élément d'indice d si le pivot qui a été choisie est le premier élément de la liste.

Finalement, la liste contient au début des éléments inférieurs ou égaux au pivot, puis le pivot, puis des éléments supérieurs ou égaux au pivot. Le pivot est à la bonne place. On peut alors reproduire le processus sur la partie gauche et sur la partie droite de la liste.

```
def pivot(liste,debut,fin):
    pivot=liste[debut] #On choisit le pivot comme le premier terme de la liste
    g=debut+1
    d=fin
    while g<=d: #Tant que les deux indices ne se croisent pas
        while g<=fin and liste[g] <= pivot: # On monte dans la liste
            g=g+1                           # tant que liste[g]<pivot
        while d>debut and liste[d]>=pivot:  # On descend dans la liste
            d=d-1                           # tant que liste[d]>pivot
        if g < d: #On évite le croisement des valeurs
            liste[g], liste[d] = liste[d], liste[g] # On permute les deux valeurs mal positionnées
            g=g+1                                   # On continue pour les autres valeurs
            d=d-1
    liste[d], liste[debut]= liste[debut],liste[d]  #On remet le pivot à sa bonne position
    return d #On garde la position du pivot pour la prochaine fonction
```

- Fonction tri-rapide :

Le tri d'une liste est exécuté à l'aide d'appels récursifs sur les parties successivement déterminées par la fonction pivot. Aucune nouvelle liste n'est créée. Les parties sont délimitées par les indices des extrémités. On peut définir une fonction tri plus simple d'utilisation par la fonction Quicksort.

```
def tri_rapide(liste,g,d):
    if g < d:
        p = pivot(liste,g,d) #Dans l'indice p se trouve l'indice du pivot choisi.
        tri_rapide(liste,g,p-1)  #Tri récursif gauche
        tri_rapide(liste,p+1,d)  #Tri récursif droit

def quicksort(liste):
    tri_rapide(liste,0,len(liste)-1) #On veut avoir que L comme paramètre.
    return(liste)

print(L)
print(quicksort(L))
```

```
[2, 4, 8, 9, 9, 6, 4, 3, 7, 1]
[1, 2, 3, 4, 4, 6, 7, 8, 9, 9]
```

b) Type de tri

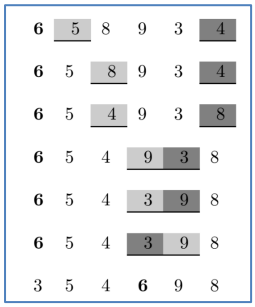
Le tri rapide est :

- Un tri récursif
- Un tri en place : pas de création de sous-listes. (Cependant il existe aussi des versions de tri rapide non en place)
- Un tri non stable. On tri directement la liste.

c) Exemple

Voici un exemple où en gris clair on représente la valeur « g » et en gris foncé la valeur « d ».

Le pivot étant marqué en gras.



- On choisit 6 comme pivot
- 5 est bien placé (g=1), on monte d'un cran jusqu'à 8 (g=2). 4 est mal placé on s'arrête (d=5).
- On inverse alors 4 et 8, on monte g (3) d'un cran et on descend d (4) d'un cran.
- Ils sont mal placés on inverse on monte g (4) et on descend d(3). La boucle while principale s'arrête
- On replace le pivot à la place de d.

d) Exemple de tri rapide « non en place » (Hoare 1962)

L'idée consiste à partitionner d'abord le tableau t à trier autour d'un pivot : on choisit l'une des valeurs du tableau (ledit pivot), par exemple L[0] et l'on construit deux tableaux avec les L[i] pour i > 0 :

- Le premier L1 avec les valeurs correspondant aux indices i tels que L[i]<pivot ;
- Le second L2 avec les valeurs correspondant aux indices i tels que L[i]≥pivot.

Il n'y a plus qu'à trier récursivement L1 et L2 et à renvoyer les valeurs triées de L1, suivies de la valeur du pivot et des valeurs triées de L2. On obtient ainsi les valeurs de L triées.

- Algorithme du tri rapide 2 :

```
def QS2(L):
    if len(L)<=1: return L #Notre condition d'arrêt de la récursivité
    pivot=L[0] #Choix du pivot
    L1,L2=[],[] #On crée nos deux sous-listes vides
    for i in L[1:]: #Le choix est lié au fait qu'on a choisi le pivot au début : L[0]
        if i<pivot: L1.append(i) #On crée la liste inférieure au pivot
        else: L2.append(i) #Puis la supérieure
    return QS2(L1)+[pivot]+QS2(L2) #Et la magie de la récursivité entraîne notre liste triée.

L=[4,1,2,5,6,0,3]
print(L)
print(QS2(L))
```

```
[4, 1, 2, 5, 6, 0, 3]
[0, 1, 2, 3, 4, 5, 6]
```

e) Terminaison

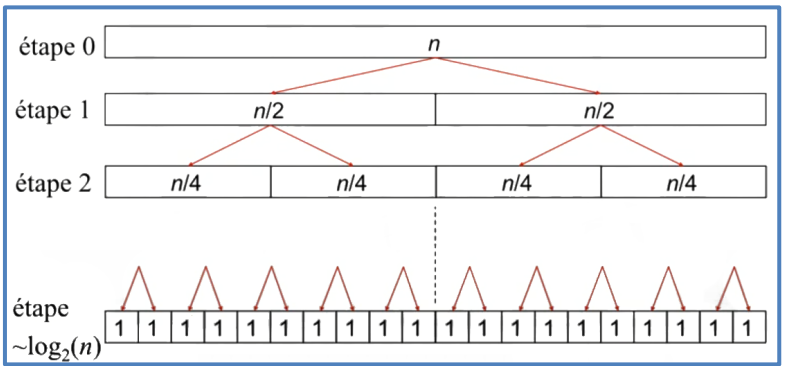
L'algorithme se termine puisque les appels récursifs ont lieu pour arguments des listes de longueurs strictement plus petites que len(L). La condition d'arrêt sera remplie dans tous les cas en un temps fini lorsque la taille de la sous-liste atteint la valeur 0 ou 1. Ceci assure la terminaison.

f) Correction

Une liste de taille 1 ou 0 est triée. Pour une liste de taille supérieure, la fonction partition positionne le pivot à la bonne place, sa place définitive dans la liste triée. De plus les éléments de la sous liste de gauche sont tous inférieurs au pivot et les éléments de la sous liste de droite sont tous supérieurs au pivot.

g) Complexité

i. Meilleur des cas



Dans le meilleur des cas on choisit le pivot au milieu de la liste et on se retrouve avec la même complexité que le tri fusion c'est-à-dire : $C(n) \sim n \log_2 n$

NB : Les « 1 » marquent le fait que l'élément est à la bonne place

ii. Pire des cas

Dans le pire des cas, la partition donne une des listes L1 ou L2 vide et l'autre de longueur (n-1). Si c'est le cas à chaque étape (liste initiale triée dans le mauvais sens) on a alors (n-1) fois la fonction pivot à appliquer qui est de complexité linéaire, ce qui entraîne une complexité quadratique :

$$C(n) \sim n^2$$

iii. Bien choisir le pivot

Si la liste est déjà triée, prendre comme pivot une extrémité n'est pas un bon choix, par conséquent on préfère le prendre de façon aléatoire ou la médiane de trois éléments.

En moyenne la complexité du tri rapide est du type : $C(n) \sim n \log_2 n$.

IV – Tri sans comparaison

IV-1) Tri par comptage

Les tris présentés utilisent des comparaisons. D'autres algorithmes peuvent utiliser la structure des données. C'est le cas des tris par comptage, par base, par paquets. Le tri par comptage est présenté ici. On dispose d'une liste d'entiers naturels qui sont tous inférieurs ou égaux à un entier naturel non nul m qui est de l'ordre de n, la taille de la liste.

On commence par écrire une fonction comptage, d'arguments une liste « entiers » et un entier « m », renvoyant une liste de longueur m+1 telle pour tout k de 0 a m, l'élément d'indice k a pour valeur le nombre d'occurrences de l'entier k dans la liste entiers.

IV-2) Algorithme

Les valeurs des éléments de la liste entiers sont représentées par les indices de la liste compteurs. On en déduit une fonction tri, d'arguments une liste entiers et un entier m, renvoyant la liste triée dans l'ordre croissant.

```python
def comptage(entiers, m):
  compteurs = (m + 1) * [0]
  for k in entiers:
    compteurs[k] = compteurs[k] + 1 #On recherche le nombre d'occurences de k dans la liste
  return compteurs

def tri(entiers, m):
  cpts = comptage(entiers, m)
  triee = []
  for i in range(m+1):
    triee = triee + cpts[i] * [i] #On ajoute l'occurence k le nombre de fois nécessaire...
  return triee

import random
n=10
L=[random.randint(1,n) for i in range(n)]
print(tri(L,n))
```

```
[1, 2, 3, 5, 6, 6, 7, 8, 9, 9]
```

IV-3) Complexité

Le tri par comptage est réservé dans notre cas à un **tri d'entiers** cependant sa complexité est faible. On a deux boucles non imbriquées de longueur n à peu prés.

Donc la complexité est : $C = n + m \sim n$. C'est une bonne complexité pour les entiers mais cela peut devenir beaucoup plus important pour d'autres systèmes de nombres.

V- Conclusion sur les complexités

	Meilleur des cas	Pire des cas	En moyenne
Tri par sélection	$C(n) \sim n^2$	$C(n) \sim n^2$	$C(n) \sim n^2$
Tri par insertion	$C(n) \sim n$	$C(n) \sim n^2$	$C(n) \sim n^2$
Tri à bulles	$C(n) \sim n$	$C(n) \sim n^2$	$C(n) \sim n^2$
Tri fusion	$C(n) \sim n \log_2 n$	$C(n) \sim n \log_2 n$	$C(n) \sim n \log_2 n$
Tri rapide	$C(n) \sim n \log_2 n$	$C(n) \sim n^2$	$C(n) \sim n \log_2 n$