IF3 – 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 la complexité d'un algorithme a le 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. Les comparaisons suivantes, dans le pire des cas, permettront aussi de partager chaque partie 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 :

$$\frac{n!}{2^k} < 2 \Rightarrow 2^k > \frac{n!}{2} \Rightarrow kln(2) > \ln(n!) - \ln(2)$$

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 : dans le pire des cas, le nombre de comparaisons est 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 n'utilise qu'un nombre très limité de variables et qu'il modifie directement la structure qu'il est en train de trier
- Tri stable : Tri qui conserve l'ordre relatif des éléments de même clef.

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(n):
   i_mini=i
   for j in range(i+1,n): #On cherche le plus petit élément de la liste [i+1:n]
      if liste[j]<liste[i_mini]:
        i_mini=j
   liste[i],liste[i_mini]=liste[i_mini],liste[i] #On inverse les positions à la fin de la boucle
  return liste
LLL=tri_selection(L)
print(LLL)
```

```
[2, 3, 5, 6, 6, 9, 10, 11, 13, 14, 14, 14, 15, 16, 16, 17, 18, 18, 19, 20]
```

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 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)+...+ 2+1 comparaisons, soit n(n-1)/2 comparaisons.

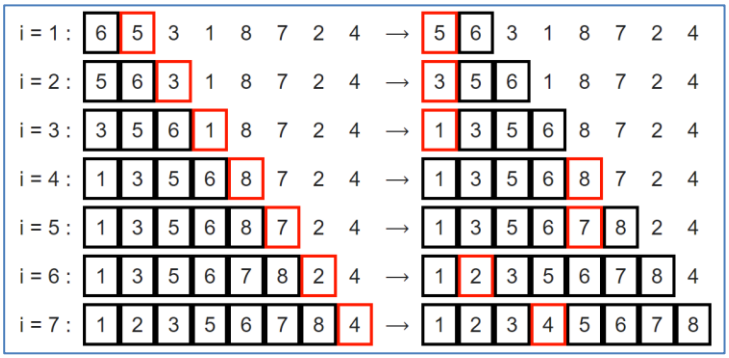
Le coût est donc de l'ordre de n^2 quelle que soit la liste, même si elle est déjà triée. Cela signifie que le tri par sélection n'est pas efficace. Il est cependant simple à programmer. L'algorithme de tri par sélection sur une liste de n éléments a un coût quadratique en fonction de n. Le nombre de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

II-2) Tri par insertion

a) Principe

On dispose de n données. A chaque étape, on suppose que les k premières données sont triées et on insère une donnée supplémentaire à la bonne place parmi ces k données. 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 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, en commençant par la donnée d'indice i puis en remontant dans la liste 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 :

```
def tri_insertion(liste):
  n=len(liste)
  for i in range(n-1):
    k=i+1 #indice de la clé que l'on prend à i+1
    cle=liste[k] #élement testé pour l'insertion
    while k>0 and cle<liste[k-1]: #La condition k>0 empêche de rentrer dans une boucle infinie.
      liste[k]=liste[k-1] #Si liste[k-1] est plus grand, il monte dans la liste.
      k=k-1 #On descend dans la partie de liste déjà triée, et on teste l'élement suivant/à la clé.
    liste[k]=cle #Boucle finie, on insère la clef à la bonne position.
  return liste
LLL=tri_insertion(L)
print(LLL)
```

```
[8, 2, 8, 5, 6, 10, 1, 4, 7, 2]
[1, 2, 2, 4, 5, 6, 7, 8, 8, 10]
```

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 : on peut vérifier que deux éléments de même valeur placés dans un certain ordre au départ restent dans le même ordre après le tri. (On peut prendre une liste [1,1*] pour le vérifier)

c) 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-1]. 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 1 soit i+1 valeurs et donc i+1 comparaisons. Au total, nous avons donc : 1+2+...+(n-2)+(n-1) comparaisons.

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 ($n^2/4$ exactement), comme pour le tri par sélection.

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.

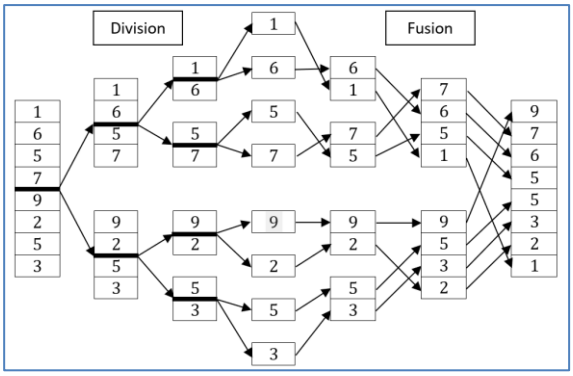
III – Stratégie « diviser pour régner »

III-1) Tri fusion

a) Principe

Le tri fusion fait partie des algorithmes basés sur la stratégie « diviser pour régner » : on ≪ divise ≫ en réduisant un problème en sous-problèmes du même type, puis on ≪ règne ≫ en résolvant ces sous-problèmes.

Le principe du tri fusion, en anglais « merge sort », est simple. La liste à trier est partagée en deux parties de tailles égales à une unité près. Il s'agit d'un tri dichotomique. Un appel récursif est alors réalisé sur chacune des deux parties. Lorsque ces deux parties sont triées, elles sont fusionnées en une liste triée. Le programme repose donc sur l'écriture d'une fonction fusion qui prend deux listes triées en paramètres et renvoie une liste triée composée de la réunion des éléments de chaque liste. Pour effectuer le tri fusion, on fusionne deux par deux des parties contigües de la liste qui ont été triées.



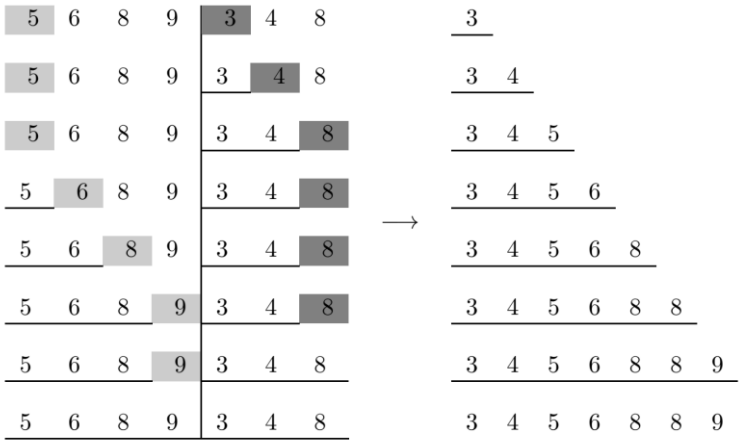
D'où l'algorithme suivant :

```
def fusion(liste1, liste2):
  liste_fus=[]
  i,j = 0,0
  while i<len(liste1) and j<len(liste2):
    if liste1[i]<=liste2[j]:
      liste_fus.append(liste1[i])
      i=i+1
    else:
      liste_fus.append(liste2[j])
      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

def tri_fusion(liste):
  if len(liste)<=1:
    return liste
  else:
    milieu = len(liste)// 2
    liste1 = tri_fusion(liste[:milieu]) #Récursivité, on découpe la liste en deux parties
    liste2 = tri_fusion(liste[milieu:]) #jusqu'à avoir un éléments dans chaque liste
  return fusion(liste1, liste2)         #que l'on trie/merge avec la fonction fusion
print(tri_fusion(L))
```

```
[1, 2, 3, 6, 6, 7, 7, 7, 8, 10]
```

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 :

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

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

$$\Rightarrow u_p = u_0 + p \times r = 1 + p$$
$$\Rightarrow C(2^p) = (1 + p) \times 2^p \sim p\, 2^p$$
$$C(n) \sim n \log_2 n$$

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

$n \log_2 n$.

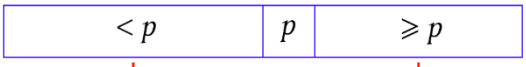
III-2) Tri rapide

a) Principe

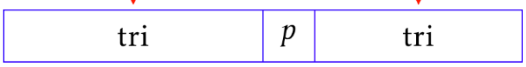
Appelé quick sort en anglais, ce tri adopte lui aussi une démarche de type « diviser pour régner » : on commence par partitionner le tableau autour d'un pivot choisi parmi les éléments du tableau en plaçant les éléments qui lui sont inférieurs à sa gauche, et les éléments qui lui sont supérieurs, à sa droite. À l'issue de cette étape, le pivot se trouve à sa place définitive, et les parties gauche et droite sont triées par l'intermédiaire d'un appel récursif.

a. Choix d'un pivot

b. Partitionnage



c. Appels récursifs



i. Fonction partition

La liste de nombres à trier est partagée en deux parties à l'aide d'une valeur choisie nommée le pivot. Ce pivot peut être n'importe quel élément de la liste. 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 0 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-1 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. Le nombre total d'éléments à trier, séparés en deux parties, est diminué d'une unité. Voici une vidéo explicative d'un algorithme proche de celui choisie :

ii. Fonction tri_rapide

Le tri d'une liste est exécuté à l'aide d'appels récursifs sur les parties successivement déterminées par la fonction partition. 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.

La fonction Quicksort permet d'avoir une fonction tri rapide plus simple d'utilisation.

```python
def partition(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

def tri_rapide(liste,g,d):
    if g < d:
        p = partition(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))
```

```
[10, 1, 6, 6, 2, 10, 1, 9, 3, 6]
[1, 1, 2, 3, 6, 6, 6, 9, 10, 10]
```

b) Type de tri

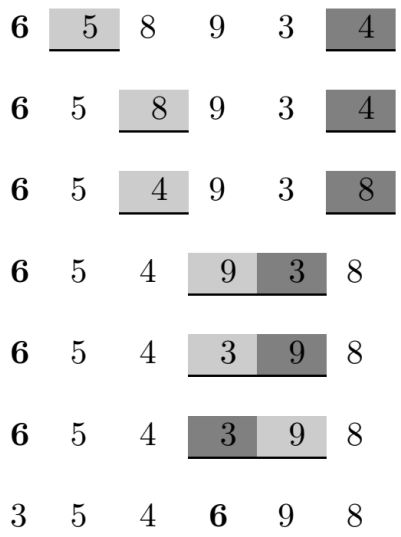
Le tri rapide est :

- 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.



d) Exemple de tri rapide « non en place »

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 t[0] et l'on construit deux tableaux avec les t[i] pour i > 0 :

- Le premier t1 avec les valeurs correspondant aux indices i tels que t[i]<pivot ;
- Le second t2 avec les valeurs correspondant aux indices i tels que t[i]≥pivot.

Il n'y a plus qu'à trier récursivement t1 et t2 et à renvoyer les valeurs triées de t1, suivies de la valeur du pivot et des valeurs triées de t2. On obtient ainsi les valeurs de t triées. Notons toutefois que contrairement à la situation du tri par fusion t1 et t2 peuvent être de tailles déséquilibrées, ce qui fait que la complexité de ce tri peut être quadratique dans le pire des cas. Mais sa complexité en moyenne est en $nlog_2(n)$ et sa mise en œuvre s'avère très rapide, d'où son nom !

Algorithme du tri rapide 2 :

```
def quicksort(liste):
  if liste == []:
    return []
  else:
    pivot = liste[0]
    liste1 = []
    liste2 = []
    for x in liste[1:]: #La valeur 1 est lié au choix du pivot.
      if x<pivot:
        liste1.append(x)
      else:
        liste2.append(x)
    return quicksort(liste1)+[pivot]+quicksort(liste2) #Tri récursif

print(quicksort(L))
```

e) Terminaison

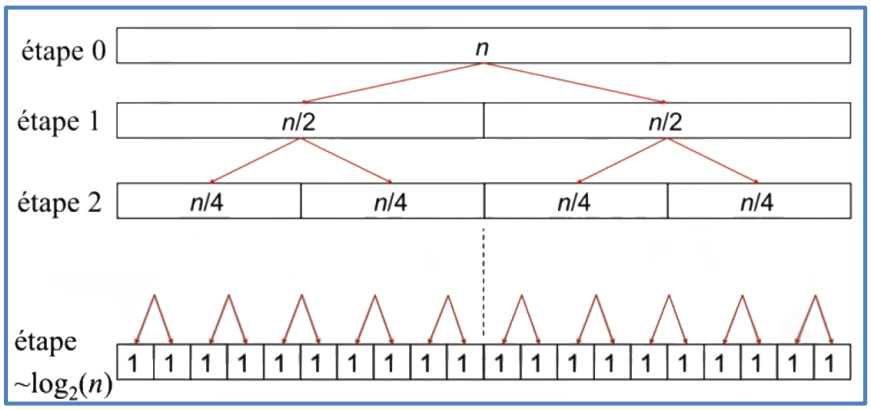
L'algorithme se termine puisque les appels récursifs ont lieu pour arguments des listes de longueurs strictement plus petites que len(L). La condition d'arrêt sera remplie dans tous les cas en un temps fini lorsque la taille de la sous-liste atteint la valeur 0 ou 1. Ceci assure la terminaison.

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

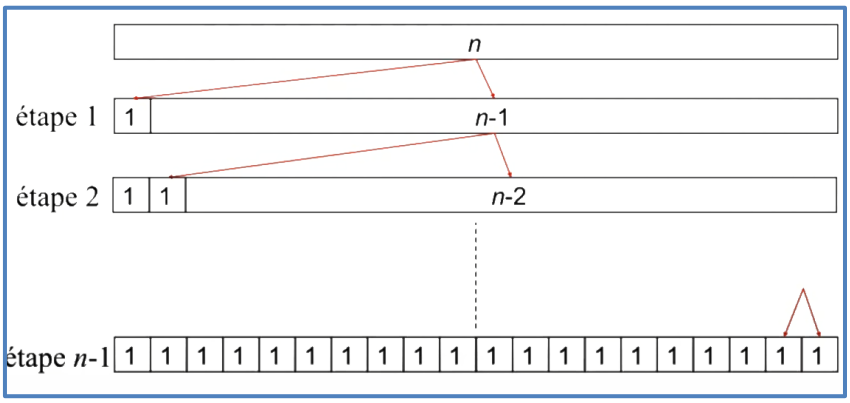


Dans le meilleur des cas on choisit le pivot au milieu de la liste et on se retrouve avec la même complexité que le tri fusion c'est-à-dire :

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

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

ii. Pire des cas



Cette fois il y a (n-1) étapes, 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.

```
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.