Chapitre VIII

Tris

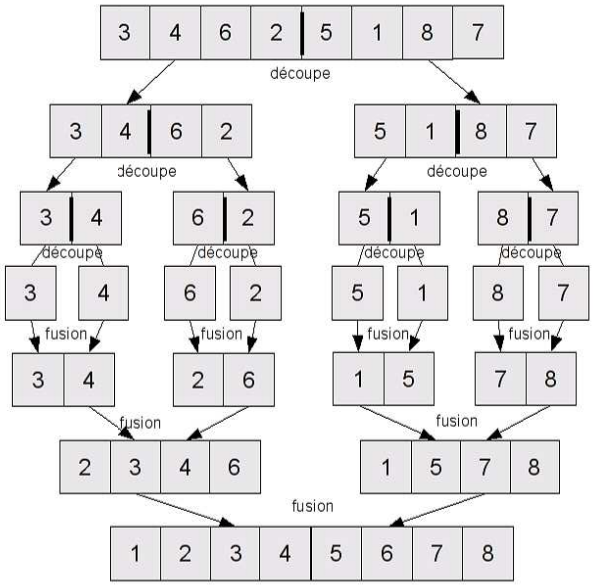

Figure VIII.1 – Exemple de tri : le tri fusion ou «Merge Sort».

Sommaire

1 Introduction

1.1 Pourquoi trier les données

L'avènement du traitement informatisé des données a conduit à manipuler des masses d'informations de plus en plus volumineuses : analyse de cohortes statistiques à l'échelle d'un pays, décodage du génôme, acquisition d'échantillons de données sur des phénomènes temporellement ou spatialement très étendus comme en astronomie par exemple etc....

Par exemple le génôme humain est composé d'environ 3,3 milliards de paires de bases azotées, qui codent l'information d'environ 60000 gènes. L'organisation correcte des données acquises selon des critères pertinents est alors un enjeu crucial afin que la recherche d'une information précise dans un tel volume de données puisse se faire dans des délais raisonnables. Ainsi, l'un des premiers traitements que subissent les données est le **tri**, mise en forme préalable à toute recherche selon critère.

Ce chapitre vise à présenter quelques algorithmes classiques de tris que nous appliquerons à des tableaux d'entiers sans perte de généralités. Nous comparerons en particulier leurs complexités temporelles respectives, car leurs performances varient de manière très significative.

1.2 Première approche : recherche d'un entier dans une liste - intérêt des tris

Considérons une liste $L[0], L[1], \ldots, L[n-1]$ de n entiers et x un entier donné. Une méthode de recherche de l'indice de la première occurence de x dans `L` consiste à parcourir tous les éléments de la liste afin de les comparer à x. Cet algorithme simple est le plus naturel et donne en Python :

Listing VIII.1 –

```
def Trouve(x,L):
    i=0
    while i<len(L):
        if x==L[i]:
            return i
        else:
            i+=1
    return False #valeur renvoyée si l'élément n'est pas présent dans la liste
```

La complexité moyenne de cet algorithme est en $\mathcal{O}(n)$, et il n'est pas possible de faire mieux sans traitement préalable de la liste.

Supposons maintenant que la liste soit préalablement triée. La recherche de la première occurrence de l'élément x peut se faire par l'algorithme de dichotomie :

Listing VIII.2 –

```
def dicho(e,T):
    g, d = 0, len(T)-1
    while g <= d:
        m = (g + d) // 2
        if T[m] == e:
            return True
        if T[m] < e: #e est dans le tableau de droite
            g=m+1
        else: #e est dans le tableau de gauche
            d=m-1
    return False
```

On montre que la complexité de cet algorithme est en $\mathcal{O}(ln_2 n)$; le progrès de la recherche est donc significatif une fois le tri effectué.

Remarque - (1.2) - 1:

- En outre, cet algorithme trouve la valeur x dans la liste, mais non nécessairement sa première occurrence.
- Le temps de traitement des données doit naturellement intégrer **le temps imparti au tri** qui peut s'avérer plus ou moins long suivant l'algorithme adopté ; cependant celui-ci est effectué une fois pour toutes, alors que les requêtes de recherche seront à priori très nombreuses. Le gain est donc assuré.

2 Quelques algorithmes de tris

Préambule : la complexité de chaque algorithme sera évaluée par le calcul ; on ajoutera également une évaluation «de terrain» en mesurant le temps de tri d'une même liste par chaque algorithme.

La liste d'entiers aléatoires (avec répétition possible) à trier sera générée par le code suivant :

Listing VIII.3 –

```
import random as rd
def genliste(n):
        L=[]
        for i in range(n):
            elt=rd.randint(0,1000)
            L.append(elt)
        return L
```

2.1 Les tris par comparaison

a - Tri par sélection

- Principe

  - ▶ On balaie la liste $L[0], \ldots, L[n-1]$ afin de répérer le plus petit élément.
  - ▶ Un fois trouvé, on permute sa position avec celle du premier élément de la liste.
  - ▶ On reprend cette démarche avec la liste $L[1], \ldots, L[n-1]$ et ainsi de suite.

- Implémentation

  Le plus simple est l'emploi de boucles inconditionnelles :

  Listing VIII.4 –

```
def Tri_Select(L):
    n=len(L)
    for i in range(n-1): #On balaie l'élément à comparer d'indice 0 à n-2
        for j in range(i+1,len(L)): #on balaie les suivants de la liste repérés par j
     pour comparaison avec l'élément d'indice i
            if L[j]<L[i]: # on fait la comparaison
                L[j],L[i]=L[i],L[j] # et on permute si l'élément d'indice j est plus
    petit que celui d'indice i en tête de sous liste.
    return L
```

- Complexité

EXERCICE N°1: *Evaluer la complexité asymptotique de cet algorithme (complexité "en gros").*

b - Tri par insertion version impérative

- PRINCIPE

Ce tri est celui adopté lorsque l'on classe un paquet de copies initialement non trié. La première copie est laissé au dessus puis la seconde est comparée à la première afin d'être laissée en place ou bien être placée au dessus de la première.

- ▶ On examine l'élément $L[1]$ que l'on compare à $L[0]$ et on l'insère en première position si $L[1] < L[0]$.
- ▶ On réitère l'opération en insérant l'élément $L[2]$ dans la liste $L[0:2]$ et ainsi de suite i.e. $L[i]$ dans la liste $L[0:i]$

Ce schéma d'algorithme est illustré "sommairement" dans l'exemple ci-dessous dans lequel on trie une liste de 5 entiers :

$$\begin{array}{ccccc} 24 & 12 & 11 & 29 & 4 \\ 12 & 24 & 11 & 29 & 4 \\ 11 & 12 & 24 & 29 & 4 \\ 11 & 12 & 24 & 29 & 4 \\ 4 & 11 & 12 & 24 & 29 \end{array}$$

- IMPLÉMENTATION

Le code python pour la version itérative de cet algorithme est

Listing VIII.5 –

```
def Tri_Insert(L):
    for i in range(1,len(L)):
        j=i
        elt=L[i]
        while j>0 and elt<L[j-1]:
            L[j]=L[j-1]
            j=j-1
        L[j]=elt
    return L
```

- COMPLEXITÉ

La boucle `for` exécute de toute façon $n-1$ itérations pour le compteur i, puis la boucle conditionnelle `while` exécute :

- ▶ $j = i$ itérations dans le pire des cas, soit une complexité de :

$$C_{pire} = \sum_{i=1}^{n-1}\sum_{j=1}^{n-1} 1 = (n-1)+(n-2)+,\ldots,+1 = (n-1)\times\frac{n}{2} \longrightarrow \frac{n^2}{2}$$

- ▶ $j = 0$ itération dans le meilleur des cas, donc :

$$C_{meilleur} = \sum_{i=1}^{n-1} 1 = (n-1) \longrightarrow n$$

EXERCICE N°2: *Faire tourner à la main l'algorithme de tri par insertion sur la liste suivante :* $L = [13, 4, 3, 45, 22, 63, 26]$

c - Tri par insertion en version récursive

On peut implémenter cet algorithme en version récursive en scindant le fonctionnement en deux étapes :

- Fonction récursive d'insertion à la bonne place de l'élément d'indice i dans la partie gauche de la liste

Listing VIII.6 –

```
def Insert_i_rec(L,i):
    if i>0 and L[i]<L[i-1]:
        L[i-1],L[i]=L[i],L[i-1]
        Insert_i_rec(L,i-1)
```

- Fonction de tri

Listing VIII.7 –

```
def Tri_Insert(L):
    for i in range(1,len(L)):
        Insert_i_rec(L,i)
    return L
```

d - Tri bulle

- PRINCIPE

Le tri bulle consiste à examiner tous les éléments consécutifs d'un tableau et de les permuter si nécessaire. Le principe algorithmique est le suivant :

- ▶ on examine les éléments $L[0]$ et $L[1]$; on les permute si nécessaire.
- ▶ on examine les éléments $L[1]$ et $L[2]$; on les permute si nécessaire.
- ▶ on itère jusqu'à $L[i-1]$ et $L[i]$
- ▶ à la fin du premier parcours de liste, l'élément le plus grand se trouve en fin de liste.
- ▶ on itère ce processus pour la liste privée du dernier élément.

- IMPLÉMENTATION :

Listing VIII.8 – Tri bulle itératif

```
def Tri_bulle(L):
    for i in range(0,len(L)):
        for j in range(1,len(L)-i):
            if L[j-1]>L[j]:
                L[j-1],L[j]=L[j],L[j-1]
```

- COMPLEXITÉ :

Sans suprise, la complexité est quadratique puisqu'il s'agit de deux boucles imbriquées :

$$C = \mathcal{O}(n^2)$$

EXERCICE N°3: *Proposer une implémentation récursive de cet algorithme.*

2.2 Les tris dichotomiques

a - Tri fusion («Merge Sort»)

Ce tri fait partie de la catégorie des tris rapides type `diviser pour règner`.

- PRINCIPE :
  - la liste initiale à trier est coupée en deux sous-listes que l'on trie récursivement
  - les deux sous-listes triées sont ensuite **fusionnées** en faisant appel à une fonction auxiliaire `fusion`.
  - l'algorithme de fusion consiste à comparer les éléments de tête des deux sous-listes et à sélectionner le plus petit d'entre-eux afin de l'ajouter à une liste auxilaire. Dès qu'une liste est épuisée, on concatène la liste auxilaire avec les éléments restants de la liste non épuisée.

- IMPLÉMENTATION :

Listing VIII.9 –

```
def fusion(L1,L2):
    auxil=[]
    i=0
    j=0
    while i<len(L1) and j<len(L2): #tant qu'aucune liste n'est épuisé on poursuit
        if L1[i]<L2[j]:
            auxil.append(L1[i])
            i=i+1
        else:
            auxil.append(L2[j])
            j=j+1
    #maintenant, une des deux listes est forcément vide et on ajoute les éléments de
    l'autre liste à auxil
    if i=len(L1):  #on teste si la liste L1 est épuisé
        auxil+=L2[j:]
    else:
        auxil+=L1[i:]
    return auxil
```

On écrit ensuite la fonction permettant de trier les sous-listes récursivement

Listing VIII.10 –

```
def trifusion(L):
    n=len(L)
    if n<=1: #cas de base
        return L
    m=n//2 #on créé l'indice médian qui coupe la liste en 2
    return fusion(trifusion(L[0:m]),trifusion(L[m:n]))
```

- COMPLEXITÉ :
  Choisissons pour simplifier une liste initiale dont le nombre d'éléments est une puissance de 2, soit $N = 2^p$.

  Appelons $T(N)$ le nombre de comparaisons effectuées par la fonction `Tri_fusion` et $F(N)$ le nombre de comparaisons effectuées par `fusion` pour trier un tableau de longueur totale N.
  On a la récurrence suivante :

$$T(N) = T(N/2) + T(N/2) + F(N) = 2T(N/2) + F(N)$$

puisque les deux appels récursifs se font sur deux listes de même dimension.

▶ **Meilleur des cas :**
Dans le meilleur des cas, `fusion` n'examine que les éléments d'une seule des deux sous listes et donc $F(N) = N/2$.
Finalement :

$$T(N) = 2T(N/2) + N/2$$

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

soit en divisant par 2^p :

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

En posant $u_p = \dfrac{T(2^p)}{2^p}$, on peut définir la suite $u_p = u_{p-1} + \dfrac{1}{2}$, suite arithmétique de raison $\dfrac{1}{2}$. Le terme général de cette suite est :

$$u_p = \underbrace{u_0}_{=1} + \frac{p}{2}$$

soit :

$$T(2^p) \simeq \frac{p}{2} \times 2^p$$

et comme $N = 2^p$ on a finalement : $T(N) \simeq \dfrac{p}{2}N = \dfrac{N}{2} \times \log_2 N$
Ainsi la complexité dans le meilleur des cas est $\boxed{C(N) = \mathcal{O}(N \log N)}$.

NB : si N n'est pas une puissance de deux, le raisonnement reste le même mais par encadrement de N ($2^p < N < 2^{p+1}$).

▶ **Pire des cas :**
Cette fois, tous les éléments des deux listes sont examinés, soit : $F(N) = N - 1$.

La démarche est identique et le résultat également !

b - Tri rapide ou «Quick Sort»

Le tri rapide s'appuie comme le tri fusion sur une méthode de type diviser pour régner. La liste à trier est encore une fois scindée en deux sous-listes ; la liste "d'en bas" contenant des éléments plus petits que ceux de la liste "d'en haut". L'élément `p` situé à **l'indice de séparation** des deux sous-listes est choisi arbitrairement et est appelé **pivot**. Les éléments plus petits que `p` sont placés dans la sous-liste "d'en bas" et les éléments plus grand dans celle "du haut".

- PRINCIPE

▶ On choisit au hasard dans la liste un élément **pivot** `p`.
▶ On compare les éléments des deux sous listes au pivot `p` afin de les classer dans la bonne sous-liste.
▶ On itère le processus en réalisant récursivement le tri de chaque sous-liste

- IMPLÉMENTATION :
Pour simplifier la procédure, nous choisirons le pivot en tête de la sous-liste à trier, soit la position gauche (indicée g plus bas).

On écrit d'abord la procédure d'échange permettant la permutation de deux éléments $L[i]$ et $L[j]$ de la liste :

Listing VIII.11 –

```
def echange(L,i,j):
    L[i],L[j]=L[j],L[i]
```

On écrit ensuite la fonction de partition permettant d'organiser la sous-liste comprise entre les indices g et d par rapport au pivot :

Listing VIII.12 –

```
def partition(L,g,d):
    assert g<d #on vérifie que les indices sont bien organisés
......
```

On prend comme pivot l'élément à gauche de la sous liste L :

Listing VIII.13 – choix pivot

```
....
    pivot=L[g]
....
```

On parcourt ensuite la liste par une boucle inconditionnelle afin de placer les éléments par rapport au pivot (appel à `echange(L,i,j)` si nécessaire) :

Listing VIII.14 – parcours de la liste

```
....
    m=g #on initialise un indice courant à l'indice de gauche de la liste
    for i in range(g+1,d):
        if L[i]<pivot:
            m=m+1 #on incrémente l'indice courant de la position à laquelle il faudra
     replacer le pivot
            echange(L,i,m) #on permute les éléments d'indice i et m
    if m!=g: # si le pivot a bougé, on va devoir le remettre à sa place m connue
        echange(L,g,m)
    return m # on renvoie finalement la position du pivot
```

A titre d'exemple, faisons tourner ce code manuellement sur la liste `L` ci-dessous. On notera m l'indice courant du pivot, i l'indice de la boucle avec le point noir ($\bullet$) indiquant sa position dans la liste en traitement :
NB : chaque ligne du tableau ci-dessous indique la situation de la liste une fois l'itération de rang i effectuée.

$$L = [7^g_m, 6, 3, 8, 4, 1, 9^d]$$

$i = g+1$	7^g	$6_m\bullet$	3	8	4	1	9^d
$i = g+2$	7^g	6	$3_m\bullet$	8	4	1	9^d
$i = g+3$	7^g	6	3_m	$8\bullet$	4	1	9^d
$i = g+4$	7^g	6	3	4_m	$8\bullet$	1	9^d
$i = g+5$	7^g	6	3	4	1_m	$8\bullet$	9^d

– Fin boucle –
– Puis replacement du pivot à sa position définitive :

1^g	6	3	4	7_m	8	9^d

On finit par la partie récursive du tri qui consiste à trier la liste `L` contenue entre les indices g et d :

Listing VIII.15 –

```
def Tri_rapide_rec(L,g,d):
    if g>=d-1: #teste si la liste contient un seul élément dans ce cas ne rien faire
        return
    m=partition(L,g,d) #on coupe la liste en deux, on l'organise, et on renvoie la position m définitive du pivot
    #on procède ensuite au tri récursif des sous-listes à partir de cette césure de liste
    Tri_rapide_rec(L,g,m)
    Tri_rapide_rec(L,m+1,d)
```

L'appel au tri pour une liste `L` doit se faire en appelant la totalité de la liste :

```
Tri_rapide_rec(L,0,len(L))
```

- Complexité :

La fonction `partition` réalise la comparaison avec le pivot des éléments de $g+1$ à d exclu (donc d-1) ; ce qui correspond donc à $d-g-1$ comparaisons. Si N est le nombre d'éléments dans la liste, alors `partition` fait N-1 comparaisons lors de son premier appel. Puis les 2 appels récursifs dont la complexité va dépendre du "cas" :

▶ Pire des cas :
le pire des cas correspond à un pivot qui reste toujours en position extrême tout le temps donc l'une des sous-listes est vide et l'autre comporte $N-1$ éléments. On a alors pour les appels récursifs $N-1$ comparaisons donc :

$$T(N) = N - 1 + T(N-1)$$

soit en définissant la suite récurrente $u_{N+1} = u_N + N$
dont la somme des termes donne : $C(N) = \sum_{i=0}^{N} u_N = N\dfrac{1+N}{2} \longrightarrow C(N) \sim \mathcal{O}(N^2)$

▶ Meilleur des cas :
Le meilleur des cas correspond à un pivot exactement au milieu de la liste à chaque récurrence. Dans ces conditions les appels récursifs trient chacun $N/2$ éléments :

$$T(N) = N - 1 + 2T(N/2)$$

Cette récurrence est la même que celle développée plus haut pour le tri fusion. Elle conduit à une complexité en :

$$\boxed{C(N) = \mathcal{O}(N \log N)}$$