Ch 13. Algorithmes de tri.

1 Introduction

Un algorithme de tri est un algorithme permettant d'organiser une liste d'éléments selon un ordre fixé. On peut dire que les éléments à trier feront partie d'un ensemble E muni d'une relation d'ordre total noté $\leq$.

Exemple

Les ensembles $\mathbb{N}$, $\mathbb{R}$... sont munis de l'ordre $\leq$.
L'ensemble des chaînes de caractères peut être muni de l'ordre lexicographique (ordre du dictionnaire). Ainsi en Python, `'a'<'b'`, `'aa'<'b'`, `'A'<'a'` (les lettres majuscules sont avant les lettres minuscules).

Définition :

Effet de bord – On dit qu'une fonction est à effet de bord lorsqu'elle modifie une variable en dehors de son environnement local. C'est par exemple le cas lorsqu'on donne une liste (objet mutable, passé par référénce) comme argument d'une fonction.
Conséquence sur les algorithmes de tri : une fonction de tri ne renvoie rien la plupart du temps (on peut donc l'appeler une procédure). La liste passée en argument en entrée sera triée et cela, même en dehors du scope de la fonction.

Définition :

Tri en place – Un tri est effectué en place lorsque la liste à trier est modifiée jusqu'à devenir triée. Dans le cas contraire, la fonction de tri pourra renvoyer une novelle liste contenent les mêmes éléments, mais triés.

Définition :

Tri stable – Un algorithme est dit stable si les positions relatives de deux éléments égaux ne sont pas modifiées par l'algorithme : c'est-à-dire que si deux éléments x et y égaux se trouvent aux positions i_x et i_y de la liste avant l'algorithme, avec $i_x < i_y$, alors c'est également le cas de leurs positions après l'algorithme.

Définition :

Tri comparatif – Un tri est dit comparatif lorsqu'il s'appuie uniqument sur la comparaison deux à deux des éléménts de la liste et pas sur la valeur ce ces éléments.

2 Le tri par sélection

Supposons qu'on demande de trier une liste de n (disons 10) élements. À un instant donné, supposons que les k premiers (disons 4) soient triés. On sélectionne alors le plus petit des $n-k$ éléments restants (soient 6) qu'on place alors à la $k+1^{\text{e}}$position (5^{e}). Les $k+1$ premiers éléments sont donc triés.

```
def tri_selection(L):
    n=len(L)
```

```
    for i in range(n-1):
        #Inv(i): L[0:i] triée, ses éléments sont plus petits que les autres éléments de L.
        imin=i
        for j in range(i+1,n):
            #Inv2(j): imin est l'indice du plus petit élément de L[i:j]
            if L[j]<L[imin]:
                imin=j
            #Inv2(j+1)
        if i!=imin:
            L[i],L[imin]=L[imin],L[i]
        #Inv(i+1)
```

3 Le tri par insertion

Supposons qu'on demande de trier une liste de n (disons 10) élements. À un instant donné, supposons que les k premiers (disons 4) soient triés. On s'intéresse alors au $k+1$e élément (le 5e). On va l'insérer dans la liste des k premiers éléments de telle sorte que les $k+1$ éléments soient triés.
Pour cela on stocke la valeur $k+1$ dans une variable temporaire `tmp`. Si `tmp` est plus petite que $k$evaleur (4e), on décale cette dernière en $k+1$eplace (5e). On s'intéresse alors à la 3evaleur. Si elle est supérieure à `tmp`, on la décale en 4eposition. Sinon on positionne `tmp` en 4eposition.

```
def tri_insertion(L):
    n=len(L)
    for i in range(1,n):
        #Inv(i): L[0:i] est trié
        j=i
        x=L[i]
        while j>0 and L[j-1]>x:
            #Inv: Pour tout k vérifiant j<k<=i, L[k]>x
            L[j]=L[j-1]
            j-=1
            #Inv: Pour tout k vérifiant j<k<=i, L[k]>x
        L[j]=x
        #Inv(i+1): L[0:i+1] est trié
```

4 Le tri fusion

Pour ce tri on utilise la stratégie de diviser pour régner :

1°) on coupe en 2 la liste à trier ;

2°) on trie les deux sous listes ;

3°) on fusionne les deux sous listes triées.

Pour trier les sous-listes on réapplique, par récursivite, la méthode précédente jusqu'à avoir des sous listes d'un seul élément.
La difficulté est alors de fusionner deux listes triées. Pour cela, on parcours les deux listes « simultanément » en ajoutant à chaque instant le plus petit élément dans une troisième liste (qui contiendra tous les éléments triés).

```
def fusion(L1,L2):
    L=[]
    i1,i2 = 0,0
    n1,n2 =len(L1),len(L2)
    for i in range(n1+n2):
        #Inv(i): L est triée dans l'ordre croissant et ses éléments sont ceux de L1[0:i1] et L2[0:i2].
```

```
        if i2==n2 or i1<n1 and L1[i1]<=L2[i2]:
            L.append(L1[i1])
            i1+=1
        else:
            L.append(L2[i2])
            i2+=1
        #Inv(i+1)
    return L

def tri_fusion(L):
    n=len(L)
    m=len(L)//2
    if n<=1:
        return L[:]
    else:
        return fusion(tri_fusion(L[:m]),tri_fusion(L[m:]))
```

5 Le tri rapide – Quicksort – Tri de Hoare

Pour ce tri on utilise la stratégie de diviser pour régner :

1°) on choisit aléatoirement un nombre de la liste à trier, par exemple le premier. Il est appelé pivot ;

2°) on positionne à gauche de ce pivot tous les nombres de la liste qui lui sont inférieurs et à sa droite tous les nombres supérieurs. On remarque donc qu'à ce stade le pivot est à sa place définitive dans la liste triée ;

3°) on réapplique ces deux étapes aux deux sous listes (à gauche et à droite du pivot).

La difficulté est ici de réaliser la partition des deux sous listes.

```
def partition(L,g,d): #g inclus, d exclus.
    pivot=L[g]
    m=g
    for i in range(g+1,d):
        #Inv(i): L[g+1:m+1] a ses éléments < pivot, ceux de L[m+1:i] sont >= pivot, avec i>=m+1.
        if L[i]<pivot:
            m+=1
        if i>m:
            L[i],L[m]=L[m],L[i]
        #Inv(i+1)
    if m>g:
        L[m],L[g]=L[g],L[m]
    return m

def tri_rapide(L,g,d):
        if g<d-1: #sinon il n'y a rien a faire.
            m=partition(L,g,d)
            tri_rapide(L,g,m)
            tri_rapide(L,m+1,d)

# Exécution
tri_rapide(L,0,len(L))
```