Ch 13. Les algorithmes de tris.

Objectifs :

- Comprendre et analyser un algorithme de tri ;
- Évaluer la complexité d'un algorithme de tri ;
- Comparer différents algorithmes de tri.

1 Présentation

Un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon un ordre déterminé. Le tri permet notamment de faciliter les recherches ultérieures d'un élément dans une liste (recherche dichotomique). On s'intéresse ici à des méthodes de tri d'une liste de valeurs numériques. Celle-ci est implémentée sous la forme d'un tableau à une dimension.

2 Complexité d'un algorithme de tri

2.a Définitions

Ordre de grandeur : Soient f et g deux fonctions positives d'une même variable entière n. La fonction f est dite avoir un ordre de grandeur au plus égal à celui de la fonction g s'il existe un entier strictement positif k et un entier N tels que, pour tout $n \geq N$, on ait $f(n) \leq k \times g(n)$. On écrira $f = O(g)$. Par exemple, les fonctions $f(n) = 3n^2 - 5n + 4$ et $g(n) = n^2$ ont même ordre de grandeur.

Complexité : On considère un algorithme A. On appelle complexité de A tout ordre de grandeur du nombre d'opérations élémentaires effectuées pendant le déroulement de l'algorithme. On exprime ce nombre d'opérations en fonction de paramètres entiers associés aux instances à traiter ; on pourra par exemple exprimer la complexité d'un tri en fonction du nombre de données à trier.

Néanmoins, il se peut qu'avec deux jeux de données différents correspondant aux mêmes paramètres, le nombre d'opérations effectuées ne soit pas le même. Par exemple, un algorithme de tri pourra être plus rapide s'il s'agit de trier des données déjà triées que s'il s'agit de données très désordonnées. On peut alors s'intéresser à :

— la complexité **dans le pire des cas** : les paramètres avec lesquels on exprime la complexité étant fixés, on considère le plus grand nombre d'opérations élémentaires effectuées sur l'ensemble des

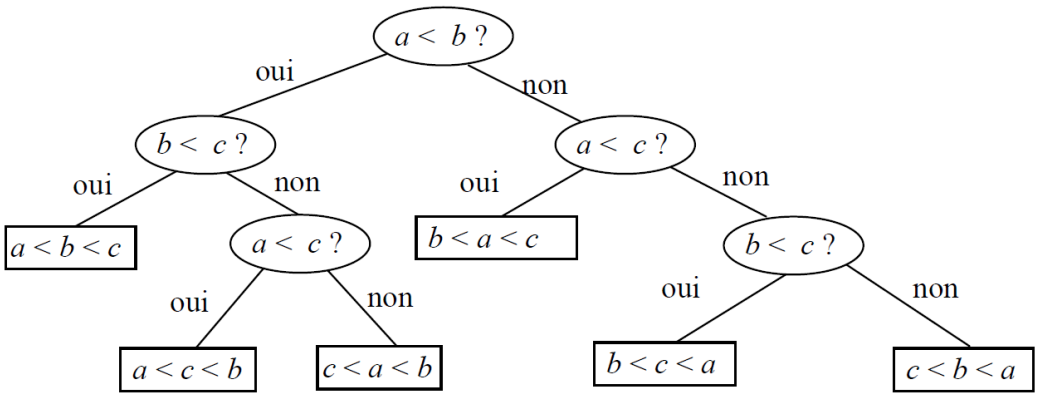
instances correspondant à ces paramètres ; on cherche ainsi un majorant de la complexité qui puisse être atteint dans certains cas ; c'est ce qu'on fait le plus généralement ;

— la complexité **dans le meilleur des cas** : les paramètres avec lesquels on exprime la complexité étant fixés, on considère le plus petit nombre d'opérations élémentaires effectuées sur l'ensemble des instances correspondant à ces paramètres ; cette complexité peut venir compléter la précédente mais ne sera jamais suffisante pour l'utilisateur ;

— la complexité **en moyenne** *(pour information)* : les paramètres avec lesquels on exprime la complexité étant fixés, il faut alors faire la moyenne des nombres d'opérations élémentaires effectuées, moyenne portant sur tous les jeux de données correspondant à ces paramètres. Ce calcul est généralement difficile et souvent même délicat à formuler car il faut connaître la probabilité de chacun des jeux de données pour effectuer un calcul pertinent de cette moyenne.

2.b Illustration

Il existe plusieurs types d'algorithmes de tri que l'on peut classer en comparatifs (ce que l'on verra dans ce cours) et non comparatifs. Un algorithme comparatif est basé sur des comparaisons successives entre les données pour déterminer la permutation correspondant à l'ordre croissant des données. On va chercher ici à évaluer la complexité théorique des tris comparatifs en se basant sur le nombre de comparaisons. Prenons un exemple : on considère un algorithme pour trier trois données a, b et c décrit par l'arbre ci-dessous ; cet algorithme correspond au fonctionnement du tri insertion que nous verrons plus loin :



Cet arbre signifie : la première comparaison faite est « a < b ? ». Si la réponse est oui, la comparaison suivante est « b < c ? », si la réponse est non c'est « a < c ? ». Lorsqu'une permutation est déterminée, on est dans ce que l'on appelle une feuille de l'arbre. On voit qu'on peut avoir plus ou moins de chance ; pour deux des permutations, on fait deux comparaisons, pour les quatre autres, on fait trois comparaisons. Le plus grand nombre de comparaisons est 3, le meilleur est 2 et le nombre moyen de comparaisons est : $(2 \times 2 + 4 \times 3)/6 \approx 2,67$.

3 Tri par Insertion

3.a Principe

Le tri par insertion est le tri que l'on effectue naturellement, par exemple pour trier un jeu de cartes. Soit un jeu de n cartes. On prend la première dans une main. On saisit la seconde carte et on l'insère avant ou après la première selon le cas. A l'étape « i », la $i^{ème}$ carte est insérée à sa place dans le paquet déjà trié.
Pour cela, on peut :

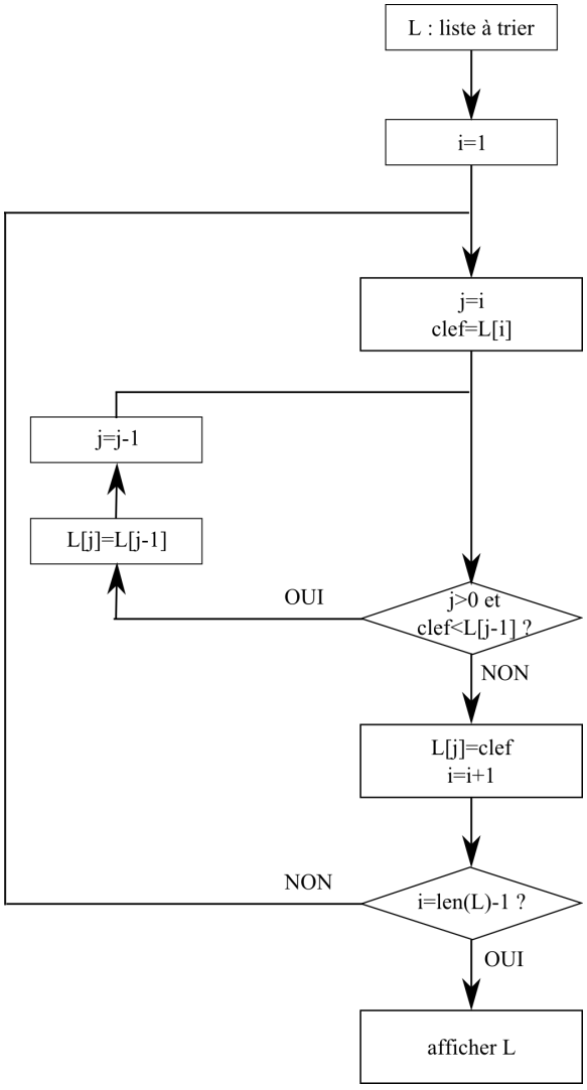
- soit partir de la fin du tas déjà trié et s'arrêter si on rencontre une carte plus petite que la $i^{ème}$ (méthode 1) ;
- soit partir du début du tas déjà trié et s'arrêter lorsqu'on rencontre une carte plus grande que la $i^{ème}$ (méthode 2).

Le paquet contient alors « i » cartes triées. On procède ainsi de suite jusqu'à la dernière carte.
Ce tri s'effectue en place, c'est-à-dire qu'il ne demande pas d'autre tableau que celui que l'on trie. Son coût en mémoire est donc constant si on ne compte pas la place occupée par les données.

`Exemple` : L'objectif de cet exemple est de trier "à la main" la liste L=`[7,6,3,5,4,2,1,8]` par insertion. On choisira comme clef (nouvel élément à trier) le premier élément non trié.

3.b Algorigramme du tri par insertion

L'algorithme proposé présente une comparaison avec, en premier, l'élément se situant à droite de la partie triée (méthode 1).




```
Commentaires
```

3.c Implémentation en python

```
Tri par insertion :
def tri_Insertion(t:list)-> None :
    ''' Trie la liste t
    Entrée : Une liste t
    Sortie : La liste est modifiée mais n'est pas renvoyée '''
    for i in range (1 , len(t)) :
        cle = t[i]
        k = i-1
        while ( k >= 0 and t[k] > cle ) :
            t[k+1]=t[k]
            k=k-1
        t[k+1] = cle
```

4 Tri rapide (quicksort par Tony Hoare - 1960)

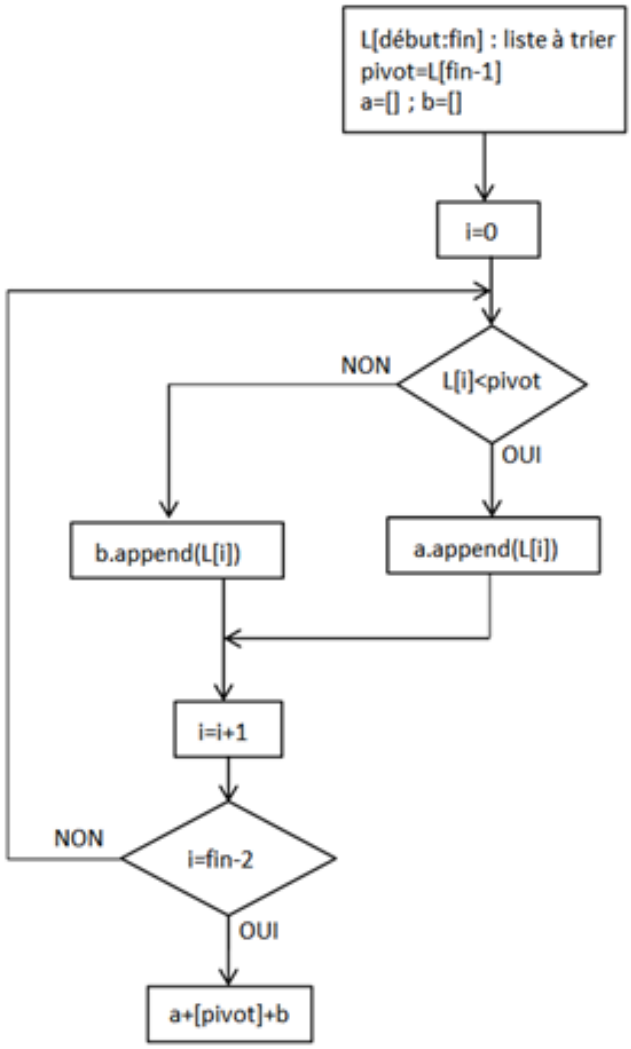
4.a Principe

L'algorithme fait partie de la catégorie des algorithmes « diviser pour régner ».
Á chaque appel de la fonction tri, on choisit une valeur "pivot", par exemple le dernier élément. On effectue ensuite une partition des éléments à trier :

- un premier groupe est constitué de valeurs inférieures au pivot ;
- un deuxième avec les valeurs supérieures.

Ainsi à chaque appel de la fonction, le nombre de données à traiter est diminué de un. C'est-à-dire que l'on ne traite plus l'élément appelé « pivot » dans les appels ultérieurs de fonction, il est placé à sa place définitive dans le tableau.

4.b Algorigramme du tri rapide




```
Commentaires
```

4.c Implémentations en python

```
Tri rapide (méthode avec l'utilisation de deux listes de stockage) :
def tri_rapide(t:list)-> list :
     ''' Trie la liste t par une méthode récursive
     Entrée : Une liste t
     Sortie : La liste est modifiée '''
     if len(t) < 2 :
          return (t)
     else :
          x = t[-1]
          a=[]
          b=[]
          for i in range (0,len(t)-1) :
               if t[i] < x :
                    a.append(t[i])
               else :
                    b.append(t[i])
          return (tri_rapide(a) + [x] + tri_rapide(b))
```

Remarque :
Certains algorithmes de tri rapide prennent pour « pivot » le premier élément, la valeur moyenne du premier et du dernier, ou un positionnement aléatoire dans le tableau. Pour se placer dans le meilleur des cas pour chaque segment de tableau, il faut prendre pour pivot la valeur médiane du tableau de valeurs. Le problème est que cette recherche de pivot idéal a aussi un « coût ».

5 Méthodes de Python

5.a sort et sorted

Les listes Python ont une méthode native `list.sort()` qui modifie les listes elles-mêmes et renvoie **None**.

Python shell

```
>>> a=[25,94,89,113,67]
>>> a.sort()
>>> a
[25,67,89,94,113]
```

Il y a également une fonction native `sorted()` qui construit une nouvelle liste triée depuis un itérable (liste de listes, tuple, dictionnaires).

Python shell

```
>>> sorted([25,94,89,113,67])
[25,67,89,94,113]
```

On utilise l'indice de la "colonne" à trier en utilisant la fonction `lambda` associée à `key` :

Python shell

```
>>> etudiants=[('Julie','PTS1',15),('Elio','PTS2',14),('Jules','PTS1',17),
('Adam','PTS2',16)]
>>> sorted(etudiants, key=lambda etudiants : etudiants[2])
[('Elio','PTS2',14),('Julie','PTS1',15),('Adam','PTS2',16),('Jules','PTS1',17)]

>>> etudiants      # etudiants est inchangé
[('Julie','PTS1',15),('Elio','PTS2',14),('Jules','PTS1',17),('Adam','PTS2',16)]
```

Sans indication particulière, le tri se fait sur la première valeur puis sur la suivante dans le cas où les premières valeurs sont identiques :

Python shell

```
>>> couple=[(3,3),(3,6),(3,1)]
>>> sorted(couple)
[(3, 1), (3, 3), (3, 6)]
# le tri est dit stable
```