6| Algorithmes de tri

1 Introduction

De manière formelle, le problème du tri consiste, à partir d'une séquence de n éléments <a1,a2,…an>, à trouver une permutation σ de l'ensemble des indices {1,2…,n} telle que aσ(1)≤aσ(2)≤…aσ(n). Les ai ne sont pas forcément des nombres, il suffit qu'il existe une relation d'ordre entre ces éléments.

Le problème de tri intervient dans beaucoup d'application, soit directement sous la forme d'un classement des données, soit indirectement par le fonctionnement interne de l'application.

2 Tri par insertion

C'est le tri du joueur de carte. Au cours de la distribution, le joueur ramasse une carte qu'il place au bon endroit dans son paquet déjà trié. Au début, il n'a qu'une carte, puis il ramasse une deuxième qu'il place avant ou après la première en fonction de sa valeur, puis la troisième qu'il insère à la bonne place… Ainsi, le paquet est toujours trié.

La programmation est donc la suivante :

- La liste initiale est parcourue terme à terme.
- L'élément à placer est inséré au bon endroit dans la liste résultat qui doit toujours être triée.

Exemple :

liste :	[**8**, 3, 4, 6, 1, 2, 5, 7]
étape 1 :	[**3, 8**, 4, 6, 1, 2, 5, 7]
étape 2 :	[**3, 4, 8**, 6, 1, 2, 5, 7]
étape 3 :	[**3, 4, 6, 8**, 1, 2, 5, 7]
étape 4 :	[**1, 3, 4, 6, 8**, 2, 5, 7]
étape 5 :	[**1, 2, 3, 4, 6, 8**, 5, 7]
étape 6 :	[**1, 2, 3, 4, 5, 6, 8**, 7]
étape 7 :	[**1, 2, 3, 4, 5, 6, 7, 8**]

```
>>> def tri_Insertion(l) :
>>>     n=len(l)
>>>     for i in range (1,n) :      # On commence à 1 car la liste du premier élément est triée.
>>>             elem=l[i]           # Elément à insérer
>>>             j=i-1               # Indice de l'élément précédent
>>>             while j>= 0 and l[j] > elem     # On reste dans la liste et elem est plus petit
>>>                     l[j+1]=l[j]             # On décale l'élément vers la droite
>>>                     j=j-1                   # On se déplace d'un cran vers la gauche
>>>             l[j+1]=elem                     # On place l'élément à insérer au bon endroit
>>>     return l
```

Complexité temporelle : Le pire des cas est celui où la liste est triée en sens inverse et chaque élément à insérer doit être placé en début de liste. Supposons que l'élément à trier soit au rang i, dans le pire des cas, il y aura 2i comparaisons ($j > 0$ et $l[j] > elem$). Le nombre d'opérations élémentaires pour le tri par insertion est donc de $\sum_{i=1}^{n-1} 2i = n(n-1) = O(n^2)$ avec $n = len(l)$.

3 Tri par sélection

Ce tri est également conceptuellement assez basique. Il faut chercher la plus petite donnée pour la placer en première position, puis chercher à nouveau la plus petite donnée parmi celles restantes et la placer à la suite de la précédente déjà repositionnée. On réitère cette opération jusqu'à ce qu'il ne reste plus aucune donnée non traitée.

La programmation est donc la suivante :

```
>>> def tri_Selection(l) :
>>>     n=len(l)
>>>     for i in range (0,n) :
>>>             i_mini=i                        # Initialisation de l'indice du minimum
>>>             mini=l[i]                       # Initialisation de la valeur du minimum
>>>             for j in range (i+1,n)          # On balaye le reste de la liste
>>>                     if l[j]<mini
>>>                             i_mini=j        # Nouvel indice du minimum
>>>                             mini=l[j]       # Nouvelle valeur du minimum
>>>             l[i],l[i_mini]=l[i_mini],l[i],  # On intervertit les éléments
>>>     return l
```

Complexité temporelle : Dans tous les cas, l'algorithme de tri par sélection sur une liste de n éléments à un cout quadratique en fonction de n, c'est-à-dire en $O(n^2)$.

4 Tri par fusion

Le tri par fusion permet d'optimiser la complexité par rapport au tri par insertion qui a une complexité quadratique.

Le tri par fusion est un tri qui utilise la technique « diviser pour régner » qui agit de la manière suivante :

- **Diviser** : diviser la suite de n éléments en deux sous-suites de n/2 éléments (à un près) ;
- **Régner** : trier les deux sous-suites obtenues de manière récursive en utilisant le tri par fusion ;
- **Fusionner** : combiner les deux sous-suites triées pour obtenir une suite complètement triée.

Exemple :

liste1	liste2	résultat
[3,5,8,9]	[1,2,6,10]	[ ]
[3,5,8,9]	[2,6,10]	[1]
[3,5,8,9]	[6,10]	[1,2]
[5,8,9]	[6,10]	[1,2,3]
[8,9]	[6,10]	[1,2,3,5]
[8,9]	[10]	[1,2,3,5,6]
[9]	[10]	[1,2,3,5,6,8]
[ ]	[10]	[1,2,3,5,6,8,9]
[ ]	[ ]	[1,2,3,5,6,8,9,10]

Chacune de ces opérations va correspondre à une fonction qui sera utilisée pour écrire l'algorithme de tri final.

```
>>> def diviser(l) :
>>>     n=len(l)
>>>     return l[0 : n//2], l[n//2 : n]
```

La difficulté de l'algorithme du tri par fusion repose sur la procédure de fusion de deux sous-listes triées. En reprenant l'exemple du jeu de cartes : supposons que deux tas de cartes sont triés avec la carte la plus faible placée en haut. Parmi les deux cartes visibles, la plus faible est placée sur la pile résultat et ainsi de suite jusqu'à ce que l'une des deux piles soit vide. A ce moment, toutes les cartes de la pile restante peuvent être ajoutées à la pile résultat.

D'où l'algorithme :

```
>>> def fusion (l1 , l2) :
>>>     res = [ ]
>>>     n1, n2 = len (l1) , len (l2)
>>>     i1 , i2 = 0 , 0
>>>     while i1<n1 and i2<n1 :  # Tant que l'on a pas parcouru complétement une des deux listes
>>>             if l[i1] < l[i2] :
>>>                     res.append( l1[i1] )
>>>                     i1+=1
>>>             else :
>>>                     res.append( l2[i2] )
>>>                     i2+=1
>>>     if i1 == n1 and i2 != n2 :          # On n'a pas encore complétement parcouru l2
>>>             for i in range (i2, n2) :
>>>                     res.append( l2[i] )
>>>     if i2 == n2 and i1 != n1 :
>>>             for i in range (i1, n1) :
>>>                     res.append( l1[i] )
>>>     return res
```

Finalement le tri par fusion s'écrit sous forme récursive :

```
>>> def tri_fusion(l) :
>>>     n = len(l)
>>>     if n ==0 or n ==1 :
>>>             return l
>>>     else :
>>>             l1,l2=diviser(l)
>>>             l1= tri_Fusion(l1)
>>>             l2 = tri_Fusion(l2)
>>>             return fusion (l1,l2)
```

Complexité : la complexité en temps de cet algorithme dans le pire des cas est en $O(n^2)$.Cependant, la complexité moyenne est en $O(n\ ln(n))$.

5 Tri rapide

Le tri rapide est basé sur le même principe que le tri par fusion : c'est un algorithme de type « diviser pour régner ». C'est la manière de diviser la liste initiale qui diffère. On ne partage pas le tableau en deux moitiés de même longueur : on choisit un élément appelé pivot (le plus simple est de choisir le premier élément du tableau) on sépare le tableau en deux parties : les éléments inférieurs au pivot, et les éléments supérieurs au pivot.

Pour rassembler les deux morceaux une fois chacun d'eux triés une simple concaténation suffit, puisque les éléments du premier morceau sont tous inférieurs à ceux du deuxième morceau.

Application sur un exemple :

Les données sont les valeurs qui figurent dans la seconde ligne du tableau suivant, la première ligne précise les indices des cases du tableau :

0	1	2	3	4	5	6	7	8	9
8	4	18	11	15	5	17	1	10	7

On applique la fonction partition avec g = 0 et d = 9. La variable pivot vaut liste[0] = 8. Au premier passage dans la boucle externe, i s'arrête à 2, j reste à 9. On procède à l'échange puisque i<j, et on obtient :

0	1	2	3	4	5	6	7	8	9
8	4	7	11	15	5	17	1	10	18

Ensuite i passe 3 et j à 8. Au deuxième passage dans la boucle, i reste à 3 et j s'arrête à 7. On procède à l'échange et on obtient :

0	1	2	3	4	5	6	7	8	9
8	4	7	1	15	5	17	11	10	18

Ensuite i passe à 4 et j à 6. Au troisième passage dans la boucle, i reste à 4 et j s'arrête à 5. On procède à l'échange et on obtient :

0	1	2	3	4	5	6	7	8	9
8	4	7	1	5	15	17	11	10	18

Ensuite i passe à 5 et j passe à 4. Le test « i < j ? » n'est plus vérifié. On sort de la boucle " tant que i≤j " ; on échange liste[0] avec liste[j], c'est-à-dire liste[4] ; on obtient :

0	1	2	3	4	5	6	7	8	9
5	4	7	1	8	15	17	11	10	18

La fonction retourne l'indice 4. La valeur 8 est bien à sa place définitive, avant on trouve les données inférieures ou égales à 8 et après, les données supérieures à 8.

Programme de la fonction :

```
>>> def partition (l,g, d) :
>>>     pivot=liste[g]
>>>     i=g+1
>>>     j=d
>>>     while i <= j :
>>>             while i < len(l) and l[i] <= pivot :
>>>                     i+=1
>>>             while l[j] > pivot :
>>>                     j - = 1
>>>             if i < j :
>>>                     l[i] , l[j] = l[j] , l[i]          # Echange ligne i et ligne j
>>>                     i + = 1
>>>                     j - = 1
>>>     l[g] , l[j] = l[j] , l[g]                          # Changement de place du pivot
>>>     return j
```

On peut maintenant écrire l'algorithme récursif du tri rapide pour un tableau dont les indices sont compris entre g et d. On appellera tri_rapide(0, n − 1) pour trier des données se trouvant entre les indices 0 et n − 1 d'un tableau à n éléments.

Retour à l'exemple :

Il reste à appliquer le tri rapide entre les indices 0 et 3 puis entre les indices 5 et 9. L'application de tri_rapide(0, 3) appelle partition(0, 3) qui conduit au tableau suivant et renvoie la valeur 2 :

0	1	2	3	4	5	6	7	8	9
4	1	5	7	8	15	17	11	10	18

On applique alors tri_rapide(0, 1), qui appelle partition(0, 1) et conduit au tableau suivant puis retourne la valeur 1 :

0	1	2	3	4	5	6	7	8	9
1	4	5	7	8	15	17	11	10	18

Ensuite, tri_rapide (0, 0) et tri_rapide (3, 3) ne font rien ; le tri rapide entre les indices 0 et 3 est terminé. L'application de tri_rapide(5, 9) fait appel à partition(5, 9) qui conduit au tableau et retourne l'indice 7 :

0	1	2	3	4	5	6	7	8	9
1	4	5	7	8	11	10	15	17	18

Ensuite, tri_rapide(5, 6) puis tri_rapide(8, 9), qui conduisent au tableau :

0	1	2	3	4	5	6	7	8	9
1	4	5	7	8	10	11	15	17	18

Programme :

```
>>> def tri_rapide (l, g ,d) :
>>>     if g<d :
>>>             j = partition (ll,g, d)
>>>             tri_rapide(l,g, j-1)
>>>             tri_rapide(l, j+1, d)
>>>     return l
```

Complexité : la complexité en temps de la fonction partition est linéaire par rapport à la longueur de la liste considérée.

Le pire des cas est celui où le pivot vient toujours à une extrémité de la sous-liste allant de l'indice g à l'indice d pour chaque appel à la fonction partition. C'est le cas si la liste est déjà triée. On fait alors appel à la fonction partition pour des listes de longueur n, puis n − 1, puis n − 2, ... , puis 2 ; la somme des complexités de ces fonctions donne un algorithme en $O(n^2)$.

6 Résumé

Pour schématiser :

- Le tri par fusion est pratique lorsque l'on dispose de listes.
- Le tri rapide (de Hoare) est pratique lorsque l'on utilise des tableaux.
- Les tris par insertion et par sélection seront utiles lorsque le tableau est presque trié (justement quand le tri rapide est moins bon). C'est une situation qui peut arriver assez vite : si on maintient un tableau trié et qu'on le modifie un peu de temps en temps (typiquement en base de donnée).

Table des matières