Le tri rapide

méthode Sedgewick

Assistants interactif animé : 

C'est le plus performant des tris en table qui est certainement celui qui est le plus employé dans les programmes. Ce tri a été trouvé par C.A.Hoare, nous nous référons à Robert Sedgewick qui a travaillé dans les années 70 sur ce tri et l'a amélioré et nous renvoyons à son ouvrage pour une étude complète de ce tri. Nous donnons les principes de ce tri et sa complexité en moyenne et au pire.

A) Spécification abstraite

Son principe est de parcourir la liste L = (**a**1, **a**2, ... , **a**n) en la divisant systématiquement en deux sous-listes L1 et L2. L'une est telle que tous ses éléments sont inférieurs à tous ceux de l'autre liste et en travaillant séparément sur chacune des deux sous-listes en réappliquant la même division à chacune des deux sous-listes jusqu'à obtenir uniquement des sous-listes à un seul élément.

C'est un algorithme dichotomique qui divise donc le problème en deux sous-problèmes dont les résultats sont réutilisés par recombinaison, il est donc de complexité **O(n.log(n))**.

Pour partitionner une liste L en deux sous-listes L1 et L2 :

- on choisit une valeur quelconque dans la liste L (la dernière par exemple) que l'on dénomme pivot,
- puis on construit la sous-liste L1 comme comprenant tous les éléments de L dont la valeur est inférieure ou égale au pivot,
- et l'on construit la sous-liste L2 comme constituée de tous les éléments dont la valeur est supérieure au pivot.

L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]
prenons comme pivot la dernière valeur pivot = 16

Nous obtenons par exemple :
L1 = [4, 14, 3, 2]
L2 = [23, 45, 18, 38, 42]

A cette étape voici l'arrangement de L :
L = L1 + pivot + L2 = [4, 14, 3, 2, 16, **23, 45, 18, 38, 42**]

En effet, en travaillant sur la table elle-même par réarrangement des valeurs, le pivot **16** est placé au bon endroit directement :
[**4<16, 14<16, 3<16, 2<16**, 16, **23>16, 45>16, 18>16, 38>16, 42>16**]

En appliquant la même démarche au deux sous-listes : L1 (pivot=2) et L2 (pivot=42)
[4, 14, 3, 2, 16, **23, 45, 18, 38, 42**] nous obtenons :

L11=[ ] liste vide
L12=[3, 4, 14]
L1=L11 + pivot + L12 = (2,3, 4, 14)

L21=[**23, 38, 18**]
L22=[**45**]
L2=L21 + pivot + L22 = (**23, 38, 18, 42, 45**)

A cette étape voici le nouvel arrangement de L :
L = [(2,3, 4, 14), 16, (**23, 38, 18, 42, 45**)]
etc...

Ainsi de proche en proche en subdivisant le problème en deux sous-problèmes, à chaque étape nous obtenons un pivot bien placé.

B) Spécification concrète

La suite (**a**1, **a**2, ... , **a**n) est rangée dans un tableau T[...] en mémoire centrale.
Le processus de partionnement décrit ci-haut (appelé aussi segmentation) est le point central du tri rapide, nous contruisons une fonction **Partition** réalisant cette action .Comme l'on ré-applique la même action sur les deux sous-listes obtenues après partion, la méthode est donc récursive, le tri rapide est alors une procédure récursive.

B-1) Voici une spécification générale de la procédure de tri rapide :

Tri Rapide sur [a..b]
Partition [a..b] renvoie **pivot** & [a..b] = **[x .. pivot']+[pivot]+[pivot'' .. y]**
Tri Rapide sur **[pivot'' .. y]**
Tri Rapide sur **[x .. pivot']**

B-2) Voici une spécification générale de la fonction de partionnement :

La fonction de partionnement d'une liste [a..b] doit répondre aux deux conditions suivantes :

- renvoyer la valeur de l'indice noté **i** d'un élément appelé pivot qui est bien placé définitivement : pivot = T[i],
- établir un réarrangement de la liste [a..b] autour du pivot telque :

  [a..b] = **[x .. pivot']+[pivot]+[pivot'' .. y]**

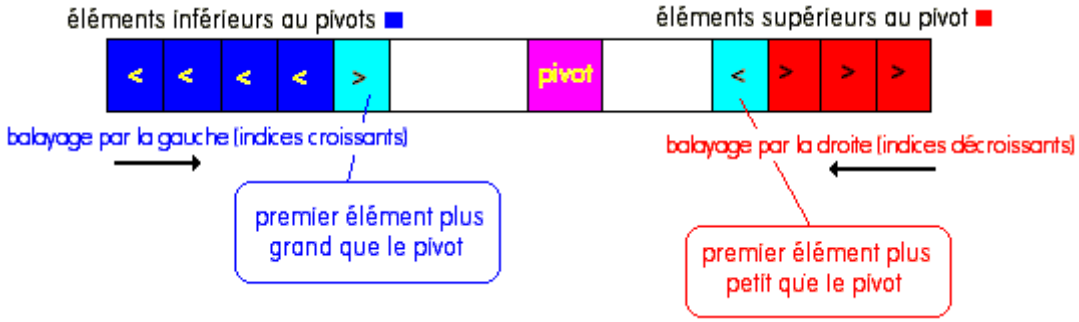
  [x .. pivot'] = T[G] , .. , T[i-1] (où : x = T[G] et pivot' = T[i-1]) tels que les T[G] , .. , T[i-1] sont tous inférieurs à T[i] ,

  [pivot'' .. y] = T[i+1] , .. , T[D] (où : y = T[D] et pivot'' = T[i+1]) tels que les T[i+1] , .. , T[D] sont tous supérieurs à T[i] ,

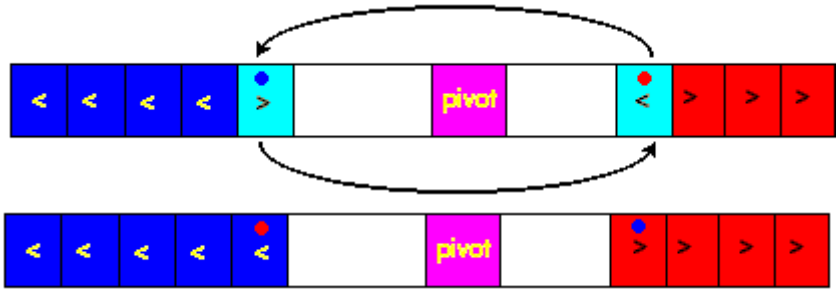
Il est proposé de **choisir arbitrairement le pivot** que l'on cherche à placer, puis ensuite de balayer la liste à réarranger dans les deux sens (par la gauche et par la droite) en construisant une sous-liste à gauche dont les éléments ont une valeur inférieure à celle du pivot et une sous-liste à droite dont les éléments ont une valeur supérieure à celle du pivot .

1) Dans le balayage par la gauche, on ne touche pas à un élément si sa valeur est inférieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous-liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus grande que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous-liste mais plutôt dans l'autre sous-liste.

2) Dans le balayage par la droite, on ne touche pas à un élément si sa valeur est supérieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous-liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus petite que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous-liste mais plutôt dans l'autre sous-liste.

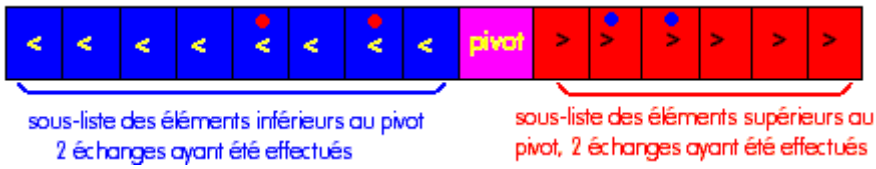


3) on procède à l'échange des deux éléments mal placés dans chacune des sous-listes :



4) On continue le bayage par la gauche et le balayage par la droite tant que les éléments sont bien placés (valeur inférieure par la gauche et valeur supérieure par la droite), en échangeant à chaque fois les éléments mal placés.

5) La construction des deux sous-listes est terminée dès que l'on atteint (ou dépasse) le pivot.



Appliquons cette démarche à l'exemple précédent : L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]

- Choix arbitraire du pivot : l'élément le plus à droite ici **16**

- Balayage à gauche :

4 < 16 => il est dans la bonne sous-liste, on continue
liste en cours de construction : [4,16]
23 > 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage gauche,
liste en cours de construction :[4,23, 16]

- Balayage à droite :

38 > 16 => il est dans la bonne sous-liste, on continue
liste en cours de construction : [4,23, 16, 38]
18 > 16 => il est dans la bonne sous-liste, on continue
liste en cours de construction : [4,23, 16, 18, 38]
45 > 16 => il est dans la bonne sous-liste, on continue
liste en cours de construction : [4,23, 16, 45, 18, 38]
14 < 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage droit,
liste en cours de construction : [4,23, 16, 14, 45, 18, 38]

- Echange des deux éléments mal placés :

[4, 23, 16, 14, 45, 18, 38] ----> [4,14, 16, 23, 45, 18, 38]

- On reprend le balayage gauche à l'endroit où l'on s'était arrêté :

--------↓----------
[4, 14, 3, 42, 2, 23, 45, 18, 38, 16]

3 < 16 => il est dans la bonne sous-liste, on continue

liste en cours de construction : [4,14, 3 ,16, 23, 45, 18, 38]
42 > 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête de nouveau le balayage gauche,
liste en cours de construction : [4,14, 3, 42, 16, 23, 45, 18, 38]

- On reprend le balayage droit à l'endroit où l'on s'était arrêté :

----------↓----------
[4, 14, 3, 42, 2, 23, 45, 18, 38, 16]

2 < 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage droit,
liste en cours de construction : [4,14, 3, 42, 16, 2, 23, 45, 18, 38]

- On procède à l'échange :

[4, 14, 3, 2, 16, 42, 23, 45, 18, 38]

et l'on arrête la construction puisque nous sommes arrivés au pivot la fonction partition a terminé son travail elle a évalué :

- le pivot : 16
- la sous-liste de gauche : L1 = [4, 14, 3, 2]
- la sous-liste de droite : L2 = [23, 45, 18, 38, 42]
- la liste réarrangée : [4,14, 3, 2, 16, 42, 23, 45, 18, 38]

Il reste à recommencer les mêmes opérations sur les parties L1 et L2 jusqu'à ce que les partitions ne contiennent plus qu'un seul élément.

C) Algorithme :

```
Global :Tab[min..max] tableau d'entier
```

```
fonction Partition( G , D : entier )  résultat : entier
Local : i , j ,  piv , temp : entier
début
    piv  ←  Tab[D];
    i  ← G-1;
    j  ← D;
    repeter
      repeter  i ←  i+1 jusquà Tab[i] >= piv;
      repeter  j ←  j-1 jusquà Tab[j] <= piv;
      temp ←  Tab[i];
      Tab[i] ←  Tab[j];
      Tab[j] ←  temp
    jusquà j <= i;
    Tab[j] ←  Tab[i];
    Tab[i] ←  Tab[d];
    Tab[d] ←  temp;
    résultat ←  i
FinPartition
Algorithme TriRapide( G  , D : entier );
Local :  i : entier
début
 si D > G alors
   i ← Partition( G , D );
  TriRapide( G , i-1 );
  TriRapide( i+1 , D );
 Fsi
FinTRiRapide
```

Nous supposons avoir mis une sentinelle dans le tableau, dans la première cellule la plus à gauche, avec une valeur plus petite que n'importe qu'elle autre valeur du tableau.

Cette sentinelle est utile lorsque le pivot choisi aléatoirement se trouve être le plus petit élément de la table /pivot = min (**a**1, **a**2, ... , **a**n)/ :

Comme nous avons:
∀j , Tab[j] > piv ,

alors la boucle :
"**repeter** j ← j-1 **jusquà** Tab[j] <= piv ;"
pourrait ne pas s'arrêter ou bien s'arrêter sur un message d'erreur.

La sentinelle étant plus petite que tous les éléments y compris le pivot arrêtera la boucle et encore une fois évite de programmer le cas particulier du pivot = min (**a**1, **a**2, ... , **a**n).

D) Complexité :

Nous donnons les résultats classiques et connus mathématiquement (pour les démonstrations nous renvoyons aux ouvrages de R.Sedgewick & Aho-Ullman cités dans la bibliographie).

L'opération élémentaire choisie est **la comparaison de deux cellules** du tableau.

Comme tous les algorithmes qui divisent et traitent le problème en sous-problèmes le nombre moyen de comparaisons est en **O(n.log(n))** que l'on nomme **complexité moyenne.**

L'expérience pratique montre que cette complexité moyenne en **O(n.log(n))** n'est atteinte que lorsque les pivots successifs divisent la liste en deux sous-listes de taille à peu près équivalente.

Dans le pire des cas (par exemple le pivot choisi est systématiquement à chaque fois la plus grande valeur) on montre que la complexité est en **O(n²)**.

Comme la littérature a montré que ce tri était le meilleur connu en complexité, il a été proposé beaucoup d'améliorations permettant de choisir un pivot le meilleur possible, des combinaisons avec d'autres tris par insertion généralement, si le tableau à trier est trop petit etc...

Ce tri est pour nous un excellent exemple illustrant la récursivité et en outre un exemple de tri en **n.log(n)**.

E) Programme pascal :

```
program TriQuickSort;
const N = 10;  { Limite supérieure de tableau }
type TTab = array [0..N] of integer;   { TTab : Type Tableau }
var Tab : TTab ;

function Partition ( G , D : integer) : integer;
  var i , j : Integer;
     piv, temp  : integer;
begin
    i := G-1;
    j := D;
    piv := Tab[D];
    repeat
       repeat i := i+1 until Tab[i] >= piv;
       repeat j := j-1 until Tab[j] <= piv;
       temp :=Tab[i];
       Tab[i] :=Tab[j];
       Tab[j] :=temp;
    until j <= i;
    Tab[j] := Tab[i];
    Tab[i] := Tab[d];
    Tab[d] := temp;
    result := i;
end;{Partition}

procedure TriRapide( G, D : integer);
var i: Integer;
begin
  if D>G then
  begin
    i := Partition( G , D );
    TriRapide( G , i-1 );
    TriRapide( i+1 , D );
  end
end;{TriRapide}

procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
```

```
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
  Tab[0] := -Maxint ; // la sentinelle
end;
procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
       writeln;
end;
begin
  Initialisation(Tab);
  writeln('TRI RAPIDE');
  writeln;
  Impression(Tab);
  TriRapide( 1 , N );
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI RAPIDE

-------------------------------------------------------------
 17 |  32 |  14 |  45 |  54 |  50 |  60 |  10 |  68 |  12 |
-------------------------------------------------------------
 10 |  12 |  14 |  17 |  32 |  45 |  50 |  54 |  60 |  68 |
-------------------------------------------------------------
```

F) Classe Java (tableau d'entiers)

```
class ApplicationTriQSort
{
 static  int [] table  =  new  int [21]  ;  // le tableau à trier en attribut
 /*  Les cellules [0] et [20] contiennent
     des sentinelles,
     Les cellules utiles vont de 1 è 19.
     (de 1 à table.length-2)
 */

  static void  impression  ( )
 {
   // Affichage sans les sentinelles
   int  n  =  table.length - 2 ;
  for (  int  i  =  1 ;  i <= n ;  i ++ )
    System.out.print ( table[i] + " , ");
   System.out.println ();
 }

 static void  initialisation   ( )
 {
```

```
 // remplissage aléatoire du tableau
 int  n  =  table.length - 2 ;
for( int  i  =  1 ;  i <= n ;  i ++ )
  table[i]  =  ( int )( Math.random () * 100 );
 // Les sentinelles:
 table[0]  = - Integer.MAX_VALUE ;
 table[n + 1]  =  Integer.MAX_VALUE ;
}

// ----> Tri rapide :

 static  int  partition  ( int  G,  int  D  )
{  // partition / Sedgewick /
 int  i, j, piv, temp ;
 piv  =  table[D] ;
 i  =  G - 1 ;
 j  =  D ;
 do
 {
  do
    i ++ ;
  while ( table[i] < piv );
  do
    j -- ;
  while ( table[j] > piv );
   temp  =  table[i] ;
   table[i]  =  table[j] ;
   table[j]  =  temp ;
 }
 while( j > i );
 table[j]  =  table[i] ;
 table[i]  =  table[D] ;
 table[D]  =  temp ;
 return  i ;
}



static void  QSort  ( int  G,  int  D  )
{  // tri rapide, sous-programme récursif
  int  i ;
 if( D > G )
 {
   i  =  partition ( G,D );
   QSort ( G,i - 1 );
   QSort ( i + 1,D );
 }
}

public static void  main ( string[ ] args )
{
 Initialisation  ( );
 int  n  =  table.length - 2 ;
 System.out.println ("Tableau initial :");
 Impression  ( );
 QSort ( 1,n );
 System.out.println ("Tableau une fois trié :");
 Impression  ( );
```

```
        }
    }
```