Chapitre 2 : Algorithmes de tri

1. Présentation

Un tri est une opération de classement d'éléments d'une liste selon un ordre total défini. Sur le plan pratique, on considère généralement deux domaines d'application des tris : les tris internes et les tris externes.

Que se passe-t-il dans un tri ? On suppose qu'on se donne une suite de nombres entiers (ex: 5, 8, -3 ,6 ,42, 2,101, -8, 42, 6) et l'on souhaite les classer par ordre croissant (relation d'ordre au sens large). La suite précédente devient alors après le tri (classement) : (-8, -3, 2, 5, 6, 6, 8, 42, 42, 101). Il s'agit en fait d'une nouvelle suite obtenue par une permutation des éléments de la première liste de telle façon que les éléments résultants soient classés par ordre croissant au sens large selon la relation d'ordre totale " ≤ " : (-8 ≤ -3 ≤ 2 ≤ 5 ≤ 6 ≤ 6 ≤ 8 ≤ 42 ≤ 42 ≤ 101).

Cette opération se retrouve très souvent en informatique dans beaucoup de structures de données. Par exemple, il faut établir le classement de certains élèves, mettre en ordre un dictionnaire, trier l'index d'un livre, etc...

1.1. Tri interne, tri externe

Un tri interne s'effectue sur des données stockées dans une table en mémoire centrale, un tri externe est relatif à une structure de données non contenue entièrement dans la mémoire centrale (comme un fichier sur disque par exemple).

Dans certains cas les données peuvent être stockées sur disque (mémoire secondaire) mais structurées de telle façon à ce que chacune d'entre elles soit représentée en mémoire centrale par une clef associée à un pointeur. Le pointeur lié à la clef permet alors d'atteindre l'élément sur le disque (n° d'enregistrement...). Dans ce cas seules les clefs sont triées (en table ou en arbre) en mémoire centrale et il s'agit là d'un tri interne. Nous réservons le vocable tri externe uniquement aux manipulations de tris directement sur les données stockées en mémoire secondaire.

1.2. Des algorithmes classiques de tri interne

Dans les algorithmes référencés ci-dessous, nous notons (a1, a2, ... , an) la liste à trier. Etant donné le mode d'accès en mémoire centrale (accès direct aux données) une telle liste est généralement implantée selon un tableau à une dimension de n éléments (cas le plus courant).

Nous attachons dans les algorithmes présentés, à expliciter des tris majoritairement sur des tables, certains algorithmes utiliserons des structures d'arbres en mémoire centrale pour représenter notre liste à trier (a1, a2, ... , an).

Les opérations élémentaires principales les plus courantes permettant les calculs de complexité sur les tris, sont les suivantes :

- la comparaison de deux éléments de la liste ai et ak , (si ai > ak, si ai < ak,....)
- l'échange des places de deux éléments de la liste ai et ak , (place(ai) <=> place(ak)).

Ce sont ces opérations qui seront utilisées pour fournir une mesure de comparaison des tris entre eux.

2. Tri à bulles

C'est le moins performant de la catégorie des tris par échange ou sélection, mais comme c'est un algorithme simple, il est intéressant à utiliser pédagogiquement.

2.1. Spécification abstraite

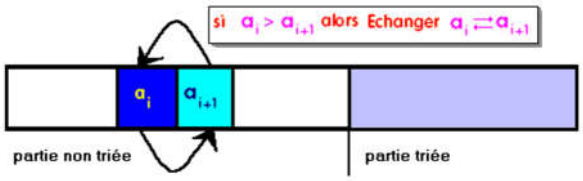
Son principe est de parcourir la liste (a1, a2, ... , an) en intervertissant toute paire d'éléments consécutifs (ai-1, ai) non ordonnés. Ainsi après le premier parcours, l'élément maximum se retrouve en an. On suppose que l'ordre s'écrit de gauche à droite (à gauche le plus petit élément, à droite le plus grand élément).

On recommence l'opération avec la nouvelle sous-suite (a1, a2, ... , an-1) , et ainsi de suite jusqu'à épuisement de toutes les sous-suites (la dernière est un couple).

Le nom de tri à bulle vient donc de ce qu'à la fin de chaque itération interne, les plus grands nombres de chaque sous-suite se déplacent vers la droite successivement comme des bulles de la gauche vers la droite.

2.2. Spécification concrète

La suite (a1, a2, ... , an) est rangée dans un tableau T[...] en mémoire centrale. Le tableau contient une partie triée (en violet à droite) et une partie non triée (en blanc à gauche). On effectue plusieurs fois le parcours du tableau à trier; le principe de base étant de ré-ordonner les couples (ai-1, ai) non classés (en inversion de rang soit ai-1 > ai) dans la partie non triée du tableau, puis à déplacer la frontière (le maximum de la sous-suite (a1, a2, ... , an-1)) d'une position :



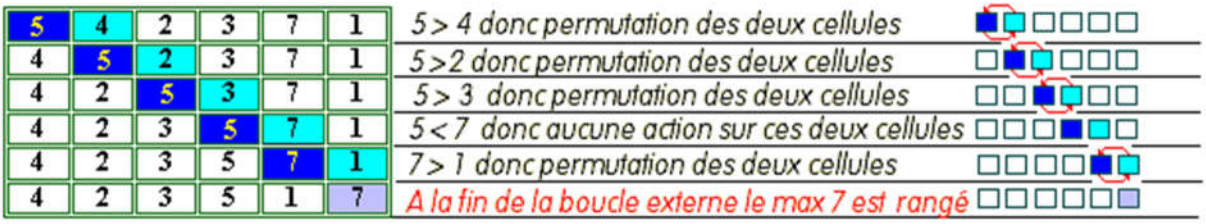
Tant que la partie non triée n'est pas vide, on permute les couples non ordonnés ((ai-1, ai) tels que ai-1 > ai)) pour obtenir le maximum de celle-ci à l'élément frontière. C'est à dire qu'au premier passage c'est l'extremum global qui est bien classé, au second passage le second extremum etc...

2.3. Algorithme

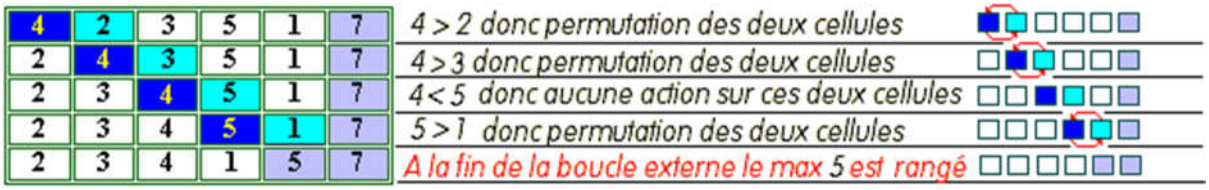
```
Algorithme Tri_a_Bulles
 local:  i , j , n, temp sont des  entiers naturels
 Entrée : Tab est un  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab est un  Tableau d'Entiers naturels de 1 à n éléments
Début
 pour i de n jusqu'à 1 faire // recommence une sous-suite (a1, a2, ... , ai)
  pour j de 2 jusqu'à i faire // échange des couples non classés de la sous-suite
   si Tab[ j-1 ] > Tab[ j ] alors // aj-1et aj non ordonnés
     temp ← Tab[ j-1 ] ;
     Tab[ j-1 ] ← Tab[ j ] ;
     Tab[ j ]  ← temp //on échange les positions de aj-1et aj
   Fsi
  fpour
 fpour
Fin Tri_a_Bulles
```

Exemple : soit la liste (5 , 4 , 2 , 3 , 7 , 1), appliquons le tri à bulles sur cette liste d'entiers. Visualisons les différents états de la liste pour chaque itération externe contôlée par l'indice i :

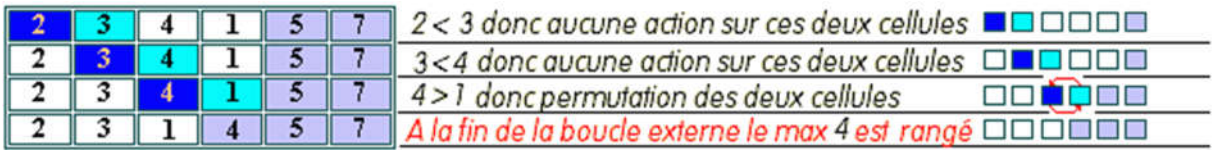
i = 6 / pour j de 2 jusquà 6 faire



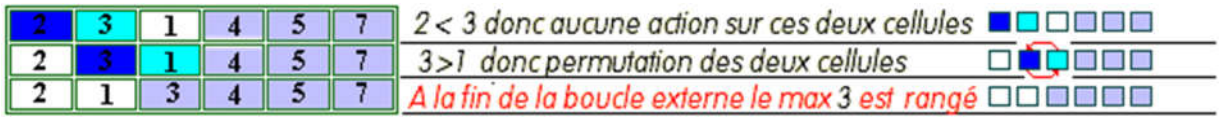
i = 5 / pour j de 2 jusquà 5 faire



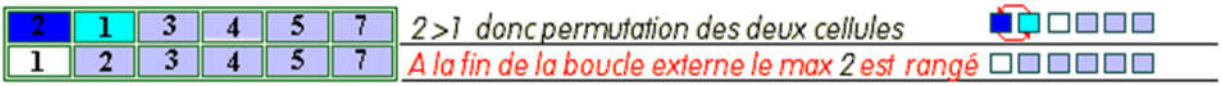
i = 4 / pour j de 2 jusquà 4 faire



i = 3 / pour j de 2 jusquà 3 faire



i = 2 / pour j de 2 jusquà 2 faire



i = 1 / pour j **de** 2 **jusquà** 1 **faire** ***(boucle vide)***

2.4. Complexité

Choisissons comme opération élémentaire la comparaison de deux cellules du tableau.

Le nombre de comparaisons "si Tab[j-1] > Tab[j] alors" est une valeur qui ne dépend que de la longueur n de la liste (n est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle "pour i de n jusquà 1 faire" s'exécute n fois (donc une somme de n termes) et qu'à chaque fois la boucle "pour j de 2 jusquà i faire" exécute (i-2)+1 fois la comparaison "si Tab[j-1] > Tab[j] alors".

La complexité en nombre de comparaisons est égale à la somme des n termes suivants (i = n, i = n-1,....)

C = (n-2)+1 + ([n-1]-2)+1 +.....+1+0 = (n-1)+(n-2)+...+1 = n(n-1)/2 (c'est la somme des n-1 premiers entiers).

La complexité en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choisissons maintenant comme opération élémentaire l'échange de deux cellules du tableau.

Calculons par dénombrement le nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échanges). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse et donc chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.

La complexité au pire en nombre d'échanges est de l'ordre de n^2, que l'on écrit $O(n^2)$.

3. Tri par sélection

C'est une version volontairement inefficace de la catégorie des tris par sélection, l'amélioration est apportée dans un autre feuillet de cours.

3.1. Spécification abstraite

La liste (**a**1, **a**2, ... , **a**n) est décomposée en deux parties : une partie triée (**a**1, **a**2, ... , **ak**) et une partie non-triée (**a**k+1, **a**k+2, ... , **a**n); l'élément **a**k+1 est appelé élément frontière (c'est le premier élément non trié).

Le principe est de parcourir la partie non-triée de la liste (**a**k+1, **a**k+2, ... , **a**n) en cherchant l'élément minimum, puis en l'échangeant avec l'élément frontière **a**k+1, puis à déplacer la frontière d'une position. Il s'agit d'une récurrence sur les minima successifs. On suppose que l'ordre s'écrit de gauche à droite (à gauche le plus petit élément, à droite le plus grand élément).

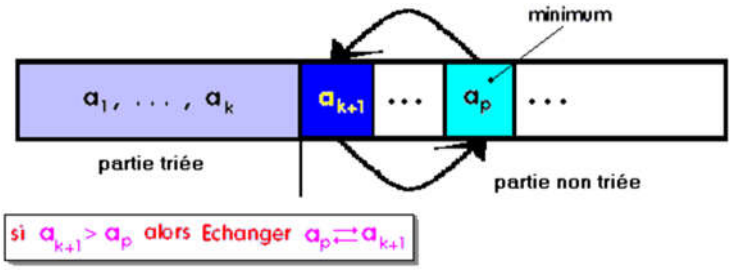
On recommence l'opération avec la nouvelle sous-suite (**a**k+2, ... , **a**n) , et ainsi de suite jusqu'à ce que la dernière soit vide.

3.2. Spécification concrète

La suite (**a**1, **a**2, ... , **a**n) est rangée dans un tableau T[...] en mémoire centrale. Le tableau contient une partie triée (en violet à gauche) et une partie non triée (en blanc à droite). On recopie le minimum de la partie non-triée du tableau dans la cellule frontière (le premier élément de cette partie).

si ak+1 > ap alors ak+1 <--- ap Fsi

et l'on obtient ainsi à la fin de l'examen de la sous-liste (**a**k+1, **a**k+2, ... , **a**n) la valeur min(**a**k+1, **a**k+2, ... , **a**n) stockée dans la cellule **a**k+1. La sous-suite (**a**1, **a**2, ... , **a**k, **a**k+1) est maintenant triée et l'on recommence la boucle de recherche du minimum sur la nouvelle sous-liste (**a**k+2, **a**k+3, ... , **a**n) etc...



Tant que la partie non triée n'est pas vide, on range le minimum de la partie non-triée dans l'élément frontière.

3.3. Algorithme

Une version maladroite de l'algorithme mais exacte a été fournie par un groupe d'étudiants elle est dénommée /version 1/. Elle échange physiquement et systématiquement l'élément frontière Tab[i] avec un élément Tab[j] dont la valeur est plus petite (la suite (**a**1, **a**2, ... , **a**i) est triée) :

```
Algorithme Tri_Selection /Version 1/
 local:  m, i , j , n, temp sont des  Entiers naturels
 Entrée : Tab est un Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab est un Tableau d'Entiers naturels de 1 à n éléments

début
 pour i de 1 jusqu'à n-1faire // recommence une sous-suite
  m ← i ; // i est l'indice de l'élément frontière Tab[ i ]
  pour j de i+1 jusqu'à n faire// liste non-triée : (ai+1, a2, ... , an)
   si Tab[ j ] <  Tab[ m ] alors // aj est le nouveau minimum partiel
     m ← j ;
     temp ←  Tab[ m ] ;
     Tab[ m ] ←  Tab[ i ] ;
     Tab[ i ]  ←  temp //on échange les positions de ai et de aj
     m ←  i ;
   Fsi
  fpour
 fpour
Fin Tri_Selection
```

Voici une version correcte et améliorée du précédent (nous allons voir avec la notion de complexité comment appuyer cette intuition d'amélioration), dans laquelle l'on sort l'échange *ai et aj* de la boucle interne "**pour** j **de** i+1 **jusqà n faire**" pour le déporter à la fin de cette boucle.

Au lieu de travailler sur les contenus des cellules de la table, nous travaillons sur les indices, ainsi lorsque *aj* est plus petit que *ai* nous mémorisons l'indice "j" du minimum dans une variable " m ← j ; " plutôt que le minimum lui-même. A la fin de la boucle interne "**pour** j **de** i+1 **jusqà n faire**" la variable m contient l'indice de min(ai+1, ak+2, ... , an) et l'on permute l'élément concerné (d'indice m) avec l'élément frontière *ai* :

```
Algorithme Tri_Selection  /Version 2/
 local:  m, i , j , n, temp sont des  Entiers naturels
 Entrée : Tab est un  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab est un  Tableau d'Entiers naturels de 1 à n éléments

début
 pour i de 1 jusqu'à n-1faire // recommence une sous-suite
  m ← i ; // i est l'indice de l'élément frontière ai = Tab[ i ]
  pour j de i+1 jusqu'à n faire// (ai+1, a2, ... , an)
   si Tab[ j ] <  Tab[ m ] alors // aj est le nouveau minimum partiel
     m ←  j ; // indice mémorisé
   Fsi
  fpour;
  temp ←  Tab[ m ] ;
```

```
 Tab[ m ] ← Tab[ i ] ;
 Tab[ i ] ← temp //on échange les positions de ai et de aj
 fpour
Fin Tri_Selection
```

3.4. Complexité

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Pour les deux versions 1 et 2 :

Le nombre de comparaisons "**si** Tab[j] < Tab[m] **alors**" est une valeur qui ne dépend que de la longueur **n** de la liste (**n** est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle "**pour** i **de** 1 **jusquà n-1 faire**" s'exécute n-1 fois (donc une somme de n-1 termes) et qu'à chaque fois la boucle "**pour** j **de** i+1 **jusquà n faire**" exécute (n-(i+1)+1 fois la comparaison "**si** Tab[j] < Tab[m] **alors**".

La complexité en nombre de comparaison est égale à la somme des n-1 termes suivants (i = 1, ...i = n-1)

C = (n-2)+1 + (n-3)+1 +.....+1+0 = (n-1)+(n-2)+...+1 = n.(n-1)/2 (c'est la somme des n-1 premiers entiers).

La complexité en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choisissons maintenant comme opération élémentaire **l'échange de deux cellules** du tableau.

Calculons par dénombrement du nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échanges). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse.

Pour la version 1

Au pire chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.

La complexité au pire en nombre d'échanges de la version 1 est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Pour la version 2

L'échange a lieu systématiquement dans la boucle principale "**pour** i **de** 1 **jusquà n-1 faire**" qui s'exécute n-1 fois :

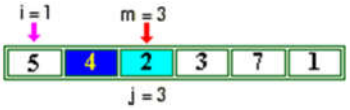
La complexité en nombre d'échanges de cellules de la version 2 est de l'ordre de n, que l'on écrit O(n).

Un échange valant 3 transferts (affectation) la complexité en transfert est O(3n) = O(n)

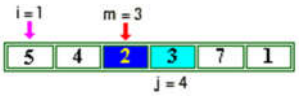
Toutefois cette complexité en nombre d'échanges de cellules n'apparaît pas comme significative du tri, outre le nombre de comparaison, c'est le nombre d'affectations d'indice qui représente une opération fondamentale et là les deux versions ont exactement la même complexité **O(n²)**.

Exemple : soit la liste à 6 éléments (**5 , 4 , 2 , 3 , 7 ,1**), appliquons la version 2 du tri par sélection sur cette liste d'entiers. Visualisons les différents états de la liste pour la première itération externe contrôlée par i (i = 1) et pour les itérations internes contôlées par l'indice j (de j = 2 ... à ... j = 6) :

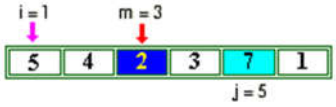
comme 5>4 on mémorise dans m



comme 4>2 on mémorise dans m

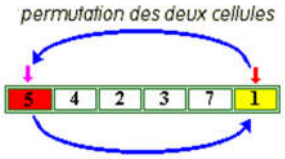


comme 2<3 on ne mémorise pas



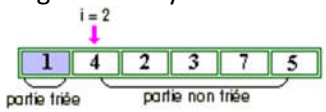
comme 2<7 on ne mémorise pas

comme 2>1 on mémorise dans m



on échange T[1] et T[6]

L'algorithme ayant terminé l'échange de T[1] et de T[6], il passe à l'itération externe suivante (i = 2) :



etc....

4. Tri par insertion

C'est un tri en général un peu plus coûteux en particulier en nombre de transfert à effectuer qu'un tri par sélection cf. complexité.

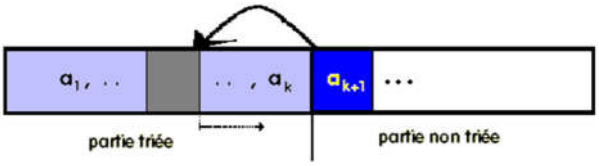
4.1. Spécification abstraite

Son principe est de parcourir la liste non triée (**a**1, **a**2, ... , **a**n) en la décomposant en deux parties une partie t déjà triée et une partie non triée. La méthode est identique à celle que l'on utilise pour ranger des cartes que l'on tient dans sa main : on insère dans le paquet de cartes déjà rangées une nouvelle carte au bon endroit. L'opération de base consiste à prendre l'élément frontière dans la partie non triée, puis à l'insérer à sa place dans la partie triée (place que l'on recherchera séquentiellement), puis à déplacer la frontière d'une position vers la droite. Ces insertions s'effectuent tant qu'il reste un élément à ranger dans la partie non triée.. L'insertion de l'élément frontière est effectuée par décalages successifs d'une cellule.

La liste (**a**1, **a**2, ... , **a**n) est décomposée en deux parties : une partie triée (**a**1, **a**2, ... , **ak**) et une partie non-triée (**a**k+1, **a**k+2, ... , **a**n); l'élément **a**k+1 est appelé élément frontière (c'est le premier élément non trié).

4.2. Spécification concrète itérative

La suite (**a**1, **a**2, ... , **a**n) est rangée dans un tableau T[...] en mémoire centrale. Le tableau contient une partie triée ((**a**1, **a**2, ... , **ak**) en violet à gauche) et une partie non triée ((**a**k+1, **a**k+2, ... , **a**n) en blanc à droite).



En faisant varier j de k jusqu'à 2 , afin de balayer toute la partie (**a**1, **a**2, ... , **a**k) déjà rangée, on décale d'une place les éléments plus grands que l'élément frontière :

tantque **a**j-1 > **a**k+1 **faire**
décaler **a**j-1 en **a**j ;
passer au j précédent
ftant

La boucle s'arrête lorsque **a**j-1 < **a**k+1, ce qui veut dire que l'on vient de trouver au rang j-1 un élément **a**j-1 plus petit que l'élément frontière **a**k+1, donc **a**k+1 doit être placé au rang j.

4.3. Algorithme

```
Algorithme Tri_Insertion
 local:   i , j , n, v sont des  Entiers naturels
 Entrée : Tab est  Tableau d'Entiers naturels de 0 à n éléments
 Sortie : Tab est Tableau d'Entiers naturels de 0 à n éléments
{ dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la
boucle tantque .. faire si l'indice j n'est pas inférieur à 1, elle aura une valeur
inférieure à toute valeur possible de la liste
}
début
pour i de 2 jusqu'à n faire// la partie non encore triée (ai, ai+1, ... , an)
    v ← Tab[ i ] ;  // l'élément frontière : ai
    j ← i ;       // le rang de l'élément frontière
     Tantque Tab[ j-1 ] > v faire//on travaille sur la partie déjà triée (a1, a2, ... , ai)
           Tab[ j ]  ← Tab[ j-1 ]; // on décale l'élément
            j ← j-1;              // on passe au rang précédent
     FinTant ;
     Tab[ j ]  ← v //on recopie ai dans la place libérée
  fpour
Fin Tri_Insertion
```

Sans la sentinelle en T[0] nous aurions une comparaison sur j à l'intérieur de la boucle :

```
Tantque Tab[ j-1 ] > v faire//on travaille sur la partie déjà triée (a1, a2, ... , ai)
          Tab[ j ]  ← Tab[ j-1 ]; // on décale l'élément
           j ← j-1;              // on passe au rang précédent
        si j = 0 alors Sortir de la boucle fsi
FinTant ;
```

4.4. Complexité

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Dans le pire des cas le nombre de comparaisons "**Tantque** Tab[j-1] > v **faire**" est une valeur qui ne dépend que de la longueur **i** de la partie (**a**1, **a**2, ... , **a**i) déjà rangée. Il y a donc au pire i comparaisons pour chaque i variant de 2 à n :

La complexité au pire en nombre de comparaison est donc égale à la somme des n termes suivants (i = 2, i = 3,.... i = n)

C = 2 + 3 + 4 +...+ n = n(n+1)/2 -1 comparaisons au maximum. (c'est la somme des n premiers entiers moins 1).

La complexité au pire en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choisissons maintenant comme opération élémentaire **le transfert d'une cellule** du tableau.

Calculons par dénombrement du nombre de transferts dans le pire des cas. Il y a autant de transferts dans la boucle "**Tantque** Tab[j-1] > v **faire**" qu'il y a de comparaisons il faut ajouter 2 transferts par boucle "**pour** i **de** 2 **jusqu'à n faire**", soit au total dans le pire des cas :

$C = n(n+1)/2 + 2(n-1) = (n^2 + 5n - 4)/2$

La complexité au pire en nombre de transferts est de l'ordre de n^2, que l'on écrit $O(n^2)$.

5. Tri fusion

5.1. Spécification

On sépare le tableau en deux sous-tableaux de même taille, que l'on trie récursivement ; ensuite, on fusionne les deux sous-tableaux en gardant l'ordre (application du principe de *diviser pour régner*). Le cas de base correspond au tri d'un seul élément, qui ne nécessite aucune opération.

Pour la fusion des tableaux, on prend trois paramètres tels que $p \le q < r$ $p \le q < r$, en sachant que les deux sous-tableaux à fusionner (A[p..q] et A[q+1..r]) sont déjà triés.

5.2. Algorithme

```
Tri par fusion : MergeSort(A, p, r)
Si p < r Alors
        q ← round down (p+r)/2
        MergeSort(A, p, q)
        MergeSort(A, q + 1, r)
        Merge(A, p, q, r)
Fusion de tableaux : Merge(A, p, q, r)
L[1..n1 + 1], R[1..n2 + 1] deux tableaux
n1 ← q - p + 1
n2 ← r – q

pour i ← 1 jusqu'à n1 faire
        L[i] ← A[p + i - 1]
pour j ← 1 jusqu'à n2 faire
        R[j] ← A[q + j]

L[n1 + 1] ← 1
```

```
R[n2 + 1] ← 1

i ← 1
j ← 1
pour k ← p jusqu'à r faire
        Si L[i] <= R[j] Alors
                A[k] ← L[i]
                i ← i + 1
        Sinon
                A[k] ← R[j]
                j ← j + 1
```

6. Tri rapide

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.

6.1. 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.

6.2. 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 construisons une fonction **Partition** réalisant cette action .Comme l'on réapplique 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]**

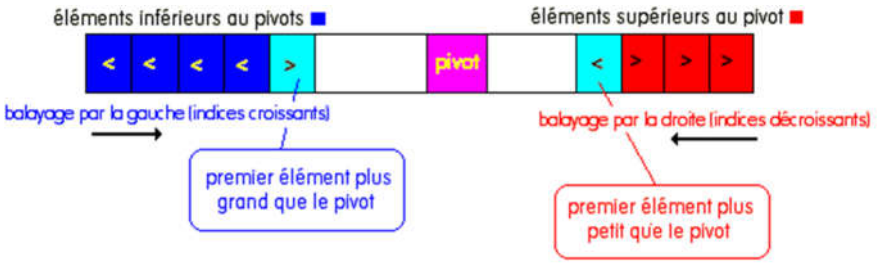
[x .. pivot'] = T[G] , .. , T[i-1] (où : x = T[G] et pivot' = T[i-1]) tels que les T[G] , .. , T[i-1] sont tous inférieurs à T[i] ,

[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] ,

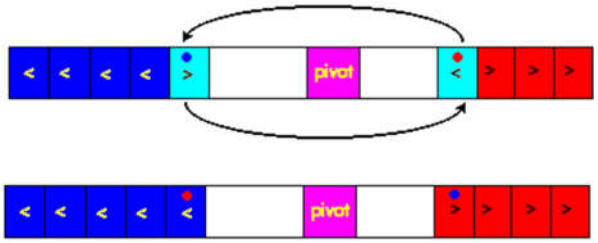
Il est proposé de **choisir arbitrairement le pivot** que l'on cherche à placer, puis ensuite de balayer la liste à réarranger dans les deux sens (par la gauche et par la droite) en construisant une sous-liste à gauche dont les éléments ont une valeur inférieure à celle du pivot et une sous-liste à droite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

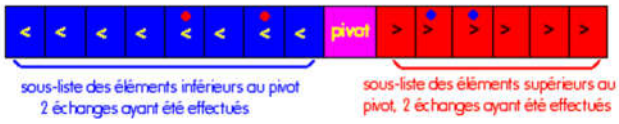


3) on procède à l'échange des deux éléments mal placés dans chacune des sous-listes :



4) On continue le balayage 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.

6.3. 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 jusqu'à Tab[i] >= piv;
      repeter  j ←  j-1 jusqu'à 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).

6.4. Complexité

Nous donnons les résultats classiques et connus mathématiquement.

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 $\mathbf{O(n^2)}$.

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)**.

Série des travaux dirigés N°1

Exercice 1 :

Ecrire un programme en langage C++ permettant d'effectuer l'insertion d'un nouvel élément dans un tableau trié.

Exercice 2 :

Ecrire un programme en langage C++ permettant d'effectuer la fusion de deux tableaux triés.

Exercice 3 :

Ecrire un programme en langage C++ permettant d'effectuer la recherche d'un élément dans un tableau non trié.

Exercice 4 :

Ecrire un programme en langage C++ permettant d'effectuer une recherche dichotomique d'un élément dans un tableau trié.