Livret - 5

Complexité, tri, recherche

Complexité d'algorithmes, tris classiques, recherches en table.

Outil utilisé : C#

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

5 : Complexité, tri, recherche

Plan du chapitre:

1. Complexité d'un algorithme et performance

Nous faisons la distinction entre les méthodes (algorithmes) de tri d'un grand nombre d'éléments (plusieurs milliers ou plus), et le tri de quelques éléments (quelques dizaines, voir quelques centaines). Pour de très petits nombres d'éléments, la méthode importe peu. Il est intéressant de pouvoir comparer différents algorithmes de tris afin de savoir quand les utiliser. Ce que nous énonçons dans ce paragraphe s'applique en général à tous les algorithmes et en particulier aux algorithmes de tris qui en sont une excellente illustration.

1.1 Notions de complexité temporelle et spatiale

L'efficacité d'un algorithme est directement liée au programme à implémenter sur un ordinateur. Le programme va s'exécuter en un **temps fini** et va mobiliser des **ressources mémoires** pendant son exécution; ces deux paramètres se dénomment **complexité temporelle** et **complexité spatiale**.

Dès le début de l'informatique les deux paramètres "temps d'exécution" et "place mémoire" ont eu une importance à peu près égale pour comparer l'efficacité relative des algorithmes. Il est clair que depuis que l'on peut, à coût très réduit avoir des mémoires centrales d'environ 1 Giga octets dans une machine personnelle, les soucis de place en mémoire centrale qui s'étaient fait jour lorsque l'on travaillait avec des mémoires centrales de 128 Kilo octets (pour des gros matériels de recherche des années 70) sont repoussés psychologiquement plus loin pour un utilisateur normal. Comme c'est le système d'exploitation qui gère la mémoire disponible (RAM, cache, virtuelle etc...), les analyses de performances de gestion de la mémoire peuvent varier pour le même programme.

Le facteur temps d'exécution reste l'élément qualitatif le plus perceptible par l'utilisateur d'un programme ne serait ce que parce qu'il attend derrière son écran le résultat d'un travail qui représente l'exécution d'un algorithme.

L'informatique reste une science de l'ingénieur ce qui signifie ici, que malgré toutes les études ou les critères théoriques permettant de comparer l'efficacité de deux algorithmes dans l'absolu, dans la pratique nous ne pourrons pas dire qu'il y a un **meilleur** algorithme pour résoudre tel type de problème. Une méthode pouvant être lente pour certaines configurations de données et dans une autre application qui travaille systématiquement sur une configuration de données favorables la méthode peut s'avérer être la "meilleure".
La recherche de la performance à tout prix est aussi inefficace que l'attitude contraire.

Prenons à notre compte les recommandations de R.Sedgewick :

> Quel que soit le problème mettez d'abord en œuvre l'algorithme le plus simple, solution du problème, car le temps nécessaire à l'implantation et à la mise au point d'un algorithme "optimisé" peut être bien plus important que le temps requis pour simplement faire fonctionner un programme légèrement moins rapide.

Il nous faut donc un outil permettant de comparer l'efficacité ou complexité d'un algorithme à celle d'un autre algorithme résolvant le même problème.

1.2 Mesure de la complexité temporelle d'un algorithme

Nous prenons le parti de nous intéresser uniquement au temps théorique d'exécution d'un algorithme. Pourquoi théorique et non pratique ? Parce que le temps pratique d'exécution d'un programme, comme nous l'avons signalé plus haut dépend :

- de la machine (par exemple processeur travaillant avec des jeux d'instructions optimisées ou non),
- du système d'exploitation (par exemple dans la gestion multi-tâche des processus),
- du compilateur du langage dans lequel l'algorithme sera traduit (compilateur natif pour un processeur donné ou non),
- des données utilisées par le programme (nature et/ou taille),
- d'un facteur intrinsèque à l'algorithme.

Nous souhaitons donc pouvoir utiliser un instrument mathématique de mesure qui rende compte de l'efficacité spécifique d'un algorithme indépendamment de son implantation en langage évolué sur une machine. Tout en sachant bien que certains algorithmes ne pourront pas être analysés ainsi soit parce que mathématiquement cela est impossible, soit parce que les configurations de données ne sont pas spécifiées d'un manière précise, soit parce que le temps mis à analyser correctement l'algorithme dépasserait le temps de loisir et de travail disponible du développeur !

Notre instrument, la complexité temporelle, est fondé sur des outils abstraits (qui ont leur correspondance concrète dans un langage de programmation). L'outil le plus connu est l'opération élémentaire (quantité abstraite définie intuitivement ou d'une manière évidente par le développeur).

Notion d'opération élémentaire

Une opération élémentaire est une opération fondamentale d'un algorithme si le temps d'exécution est directement lié (par une formule mathématique ou empirique) au nombre de ces opérations élémentaires. Il peut y avoir plusieurs opérations élémentaires dans un même algorithme.

Nous pourrons ainsi comparer deux algorithmes résolvant le même problème en comparant ce nombre d'opérations élémentaires effectuées par chacun des deux algorithmes.

1.2.1 La complexité temporelle : notation

C'est le décompte du nombre d'opérations élémentaires effectuées par un algorithme donné.

Il n'existe pas de méthodologie systématique (art de l'ingénieur) permettant pour un algorithme quelconque de compter les opérations élémentaires. Toutefois des règles usuelles sont communément admises par la communauté des informaticiens qui les utilisent pour évaluer la complexité temporelle.

Soient $\mathbf{i_1}$, $\mathbf{i_2}$, ... , $\mathbf{i_k}$ des instructions algorithmiques (affectation, itération, condition,...)
Soit une opération élémentaire dénotée **OpElem**, supposons qu'elle apparaisse n_1 fois dans l'instruction $\mathbf{i_1}$, n_2 fois dans l'instruction $\mathbf{i_2}$, ... n_k fois dans l'instruction $\mathbf{i_k}$. Nous noterons $Nb(i_1)$ le nombre n_1, $Nb(i_2)$ le nombre n_2 etc.

Nous définissons ainsi la fonction Nb (i_k) indiquant le nombre d'opérations élémentaires dénoté **OpElem** contenu dans l'instruction algorithmique i_k :

Nb( ) : Instruction à Entier .

1.2.2 Complexité temporelle d'une séquence d'instructions

Soit **S** la séquence d'exécution des instructions $\mathbf{i_1 ; i_2 ; ... ; i_k}$, soit n_k =Nb (i_k) le nombre d'opérations élémentaires de l'instruction $\mathbf{i_k}$.

Le nombre d'opérations élémentaires **OpElem** de S, Nb(**S**) est égal par définition à la somme des nombres: $n_1 + n_2 + ... + n_k$:

S =
début
$\mathbf{i_1}$;
$\mathbf{i_2}$;
... ;
$\mathbf{i_k}$
fin

$$Nb(\mathbf{S}) = \sum Nb(i_p) = n_1 + n_2 + ... + n_k$$

1.2.3 Complexité temporelle d'une instruction conditionnelle

Dans les instructions conditionnelles étant donné qu'il n'est pas possible d'une manière générale de déterminer systématiquement quelle partie de l'instruction est exécutée (le **alors** ou le **sinon**), on prend donc un majorant :

Cond =
si Expr **alors** E1
sinon E2
fsi

$$Nb(\mathbf{Cond}) < Nb(Expr) + \max(Nb(E1), Nb(E2))$$

1.2.4 Complexité temporelle d'une itération finie bornée

Dans le cas d'une boucle finie bornée (comme pour...fpour) contrôlée par une variable d'indice "i", l'on connaît le nombre exact d'itérations noté ***Nbr_d'itérations*** de l'ensemble des instructions composant le corps de la boucle dénotées S (où S est une séquence d'instructions), l'arrêt étant assuré par la condition de sortie Expr(i) dépendant de la variable d'indice de boucle i.

La complexité est égale au produit du nombre d'itérations par la somme de la complexité de la séquence d'instructions du corps et de celle de l'évaluation de la condition d'arrêt Expr(i).

Iter =
Itération Expr(i)
S
finItér

$$Nb(\textbf{ Iter }) = [\ Nb(\ S) + Nb(Expr(i))\ ] \times \textbf{\textit{Nbr\_d'itérations}}$$

*Exemple dans le cas d'une boucle **pour**...**fpour** :*

Iter =
pour i<-- a **jusquà** b **faire**
$\mathbf{i}_1$ **;**
$\mathbf{i}_2$ **;**
... ;
$\mathbf{i}_k$
fpour

La complexité de la condition d'arrêt est par définition de **1** (<= le temps d'exécution de l'opération effectuée en l'occurence un test, ne dépend ni de la taille des données ni de leurs valeurs), en **notant |b-a| le nombre exact d'itérations exécutées** (lorsque les bornes sont des entiers **|b-a|** vaut exactement la valeur absolue de la différence des bornes) nous avons :
$Nb(\textbf{ Iter }) = (\ \sum Nb(\mathbf{i}_p) + 1\ ) \ .\ |b\text{-}a|$

Lorsque le nombre d'itérations n'est pas connu mais seulement majoré (nombre noté ***Majorant_Nbr_d'itérations***), alors on obtient un majorant de la complexité de la boucle (le majorant correspond à la complexité dans le pire des cas).

Complexité temporelle au pire :

$$Majorant\_Nb(\textbf{ Iter }) = [\ Nb(\ S) + Nb(Expr(i))\ ] \times \textbf{\textit{Majorant\_Nbr\_d'itérations}}$$

1.3 Notation de Landau O(n)

Nous avons admis l'hypothèse qu'en règle générale **la complexité en temps** dépend de la taille **n** des données (plus le nombre de données est grand plus le temps d'exécution est long).

Cette remarque est d'autant plus proche de la réalité que nous étudierons essentiellement des algorithmes de tri dans lesquels les **n** données sont représentées par une liste à **n** éléments.

Afin que notre instrument de mesure et de comparaison d'algorithmes ne dépende pas de la machine physique, nous n'exprimons pas le temps d'exécution en unités de temps (millisecondes etc..) mais en unité de taille des données.

Nous ne souhaitons pas ici rentrer dans le détail mathématique des notations O(f(**n**)) de Landau sur les infiniment grands équivalents, nous donnons seulement une utilisation pratique de cette notation.

Pour une fonction f (n) dépendant de la variable n, on écrit :
f est O(g(n)) g(n) où g est elle-même une fonction de la variable entière n, et l'on lit **f est de l'ordre de grandeur de g(n)** ou plus succinctement **f est d'ordre g(n)**, lorsque :

f est d'ordre g(n) :

Pour tout valeur entière de **n**, il existe deux constantes a et b positives telles que : $a.g(\mathbf{n}) < f(\mathbf{n}) < b.g(\mathbf{n})$

Ce qui signifie que lorsque **n** tend vers l'infini (**n** devient très grand en informatique) le rapport f(**n**)/g(**n**) reste borné.

f est d'ordre g(n) :

$a < f(\mathbf{n})/g(\mathbf{n}) < b$ quand $\mathbf{n} \rightarrow \infty$
Lorsque **n** tend vers l'infini, le rapport f(**n**)/g(**n**) reste borné.

Exemple :

Supposons que f et g soient les polynômes suivants :
$f(n) = 3\mathbf{n}^2 - 7\mathbf{n} + 4$
$g(n) = \mathbf{n}^2$;
Lorsque **n** tend vers l'infini le rapport f(**n**)/g(**n**) tend vers 3:
$f(\mathbf{n})/g(\mathbf{n}) \rightarrow 3$ quand $\mathbf{n} \rightarrow \infty$
ce rapport est donc borné.

donc **f est d'ordre $\mathbf{n}^2$** ou encore **f est $\mathbf{O(n^2)}$**

C'est cette notation que nous utiliserons pour mesurer la complexité temporelle C en nombre d'opérations élémentaires d'un algorithme fixé. Il suffit pour pouvoir comparer des complexités temporelles différentes, d'utiliser les mêmes fonctions g(n) de base.

Les informaticiens ont répertorié des situations courantes et ont calculé l'ordre de complexité associé à ce genre de situation.

Les fonctions g(n) classiquement et pratiquement les plus utilisées sont les suivantes :

$g(n) = 1$
$g(n) = \log_k (n)$
$g(n) = n$
$g(n) = n.\log_k(n)$
$g(n) = n^2$

Ordre de complexité C	Cas d'utilisation courant
$g(n) = 1 \Rightarrow$ **C est O(1)**	Algorithme ne dépendant pas des données
$g(n) = \log_k (n) \Rightarrow$ **C est O($\log_k$(n))**	Algorithme divisant le problème par une quantité constante (base k du logarithme)
$g(n) = n \Rightarrow$ **C est O(n)**	Algorithme travaillant directement sur chacune des n données
$g(n) = n.\log_k(n) \Rightarrow$ **C est O(n.$\log_k$(n))**	Algorithme divisant le problème en nombre de sous-problèmes constants (base k du logarithme), dont les résultats sont réutilisés par recombinaison
$g(n) = n^2 \Rightarrow$ **C est O(n^2)**	Algorithme traitant généralement des couples de données (dans deux boucles imbriquées).

2. Trier des tableaux en mémoire centrale

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 " $\leq$ " : ($-8 \leq -3 \leq 2 \leq 5 \leq 6 \leq 6 \leq 8 \leq 42 \leq 42 \leq 101$).

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...

2.1 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 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.

2.2 Des algorithmes classiques de tri interne

Dans les algorithmes référencés ci-dessous, nous notons ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) 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 ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$).

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

Deux opérations élémentaires

La comparaison de deux éléments de la liste $\mathbf{a}_i$ et $\mathbf{a}_k$, (**si** $\mathbf{a}_i > \mathbf{a}_k$, **si** ai < ak,....) .
L'échange des places de deux éléments de la liste $\mathbf{a}_i$ et $\mathbf{a}_k$, (place ($\mathbf{a}_i$) $\leftrightarrow$ place ($\mathbf{a}_k$)).

Ces deux opérations seront utilisées afin de fournir une mesure de comparaison des tris entre eux. Nous proposons dans les pages suivantes cinq tris classiques, quatre concerne le tri de données dans un tableau, le cinquième est un tri de données situées dans un arbre binaire, ce dernier pourra en première lecture être ignoré, si le lecteur n'est pas familiarisé avec la notion d'arbre binaire

Tris sur des tables :

- Tri itératif à bulles
- Tri itératif par sélection
- Tri itératif par insertion
- Tri récursif rapide QuickSort

Tris sur un arbre binaire :

- Le Tri par tas / HeapSort

Le tri à bulle

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.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri à bulle est de parcourir la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en intervertissant toute paire d'éléments consécutifs ($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) non ordonnés.

Ainsi après le premier parcours, l'élément maximum se retrouve en $\mathbf{a}_n$. 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 ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_{n-1}$), 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.

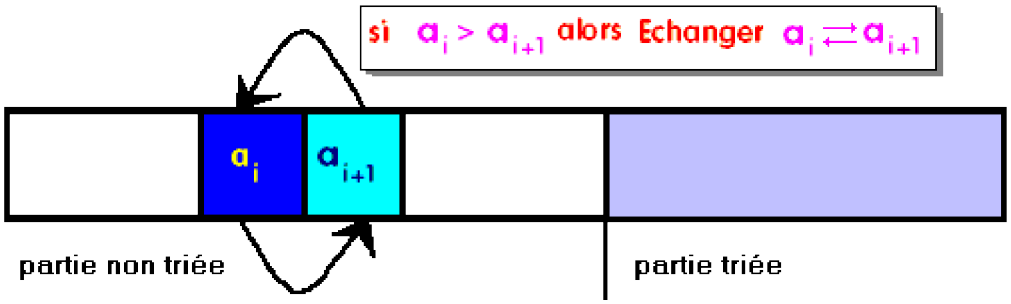
B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimension T[...] en mémoire centrale.

Le tableau contient une partie triée (en foncé 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 est de ré-ordonner les couples ($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) non classés (en inversion de rang soit $\mathbf{a}_{i-1} > \mathbf{a}_i$) dans la partie non triée du tableau, puis à déplacer la frontière (le maximum de la sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_{n-1}$)) d'une position :



Tant que la partie non triée n'est pas vide, on permute les couples **non ordonnés** (($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) tels que $\mathbf{a}_{i-1} > \mathbf{a}_i$)) 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...

C) Algorithme :

```
Algorithme Tri_a_Bulles
 local:   i , j , n, temp ∈   Entiers naturels
 Entrée : Tab ∈   Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈   Tableau d'Entiers naturels de 1 à n éléments
début
 pour i de n jusquà 1 faire // recommence une sous-suite (a1, a2, ... , ai)
  pour j de 2 jusquà 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-1 et 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**

5	4	2	3	7	1	*5 > 4 donc permutation des deux cellules*
4	5	2	3	7	1	*5 > 2 donc permutation des deux cellules*
4	2	5	3	7	1	*5 > 3 donc permutation des deux cellules*
4	2	3	5	7	1	*5 < 7 donc aucune action sur ces deux cellules*
4	2	3	5	7	1	*7 > 1 donc permutation des deux cellules*
4	2	3	5	1	7	*A la fin de la boucle externe le max 7 est rangé*

i = 5 / pour j **de** 2 **jusquà** 5 **faire**

4	2	3	5	1	7	*4 > 2 donc permutation des deux cellules*
2	4	3	5	1	7	*4 > 3 donc permutation des deux cellules*
2	3	4	5	1	7	*4 < 5 donc aucune action sur ces deux cellules*
2	3	4	5	1	7	*5 > 1 donc permutation des deux cellules*
2	3	4	1	5	7	*A la fin de la boucle externe le max 5 est rangé*

i = 4 / pour j **de** 2 **jusquà** 4 **faire**

2	3	4	1	5	7	*2 < 3 donc aucune action sur ces deux cellules*
2	3	4	1	5	7	*3 < 4 donc aucune action sur ces deux cellules*
2	3	4	1	5	7	*4 > 1 donc permutation des deux cellules*
2	3	1	4	5	7	*A la fin de la boucle externe le max 4 est rangé*

i = 3 / pour j **de** 2 **jusquà** 3 **faire**

2	3	1	4	5	7	*2 < 3 donc aucune action sur ces deux cellules*
2	3	1	4	5	7	*3 > 1 donc permutation des deux cellules*
2	1	3	4	5	7	*A la fin de la boucle externe le max 3 est rangé*

i = 2 / pour j **de** 2 **jusquà** 2 **faire**

2	1	3	4	5	7	*2 > 1 donc permutation des deux cellules*
1	2	3	4	5	7	*A la fin de la boucle externe le max 2 est rangé*

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

1	2	3	4	5	7	*Il ne reste plus d'éléments à comparer !*

D) Complexité :

Choix opération

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 comparaison est égale à la somme des n termes suivants (i = n, i = n-1,.... i = 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 du tri à bulle en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons 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 du tri à bulle au pire en nombre d'échanges est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

E) Programme C# (tableau d'entiers):

```
class ApplicationTriBulle {
   static int[] table;  // le tableau à trier, par exemple 19 éléments de l'index 1 à l'index 19

   static void AfficherTable()
   {
       // Affichage du tableau
       int n = table.Length - 1;
       for (int i = 1; i <= n; i++)
           System.Console.Write(table[i] + " , ");
       System.Console.WriteLine();
   }

   static void InitTable()
   {
       int[] tableau = { 0 ,25, 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 ,
              98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52 };
       table = tableau;
   }

   static void Main(string[] args)
   {
       InitTable();
       System.Console.WriteLine("Tableau initial :");
       AfficherTable();
       TriBulle();
       System.Console.WriteLine("Tableau une fois trié :");
       AfficherTable();
```

```
        System.Console.Read();
    }
    static void TriBulle()
    {
        // sous-programme de Tri à bulle : on trie les éléments du n°1 au n°19
        int n = table.Length - 1;
        for (int i = n; i >= 1; i--)
            for (int j = 2; j <= i; j++)
                if (table[j - 1] > table[j])
                {
                    int temp = table[j - 1];
                    table[j - 1] = table[j];
                    table[j] = temp;
                }
       /* Dans le cas où l'on démarre le tableau à l'indice zéro
          on change les bornes des indices i et j:
          for ( int i = n; i >= 0; i--)
             for ( int j = 1; j <= i; j++)
               if ..... reste identique
       */
    }
}
```

Résultat de l'exécution du programme précédent :

Tableau initial (n°1 au n°19):
25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52
Tableau une fois trié (n°1 au n°19) :
7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

Autre version depuis l'indice zéro :
Tableau initial (n°0 au n°19):
0, 25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52
Tableau une fois trié (n°0 au n°19) :
0, 7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

Le tri par sélection

- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme C#

C'est une version de base de la catégorie des tris par sélection.

A) Spécification abstraite

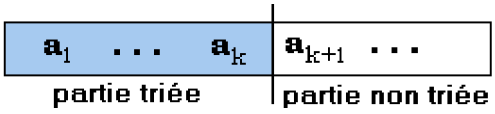
Nous supposons que les données $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ sont mises sous forme d'une liste $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$, la liste $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$ est décomposée en deux parties : une partie liste $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k)$ et une partie non-triée $(\mathbf{a}_{k+1}, \mathbf{a}_{k+2}, \ldots, \mathbf{a}_n)$; l'élément $\mathbf{a}_{k+1}$ 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.

B) Spécification concrète

La suite $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$ est rangée dans un tableau à une dimension T[...] en mémoire centrale.

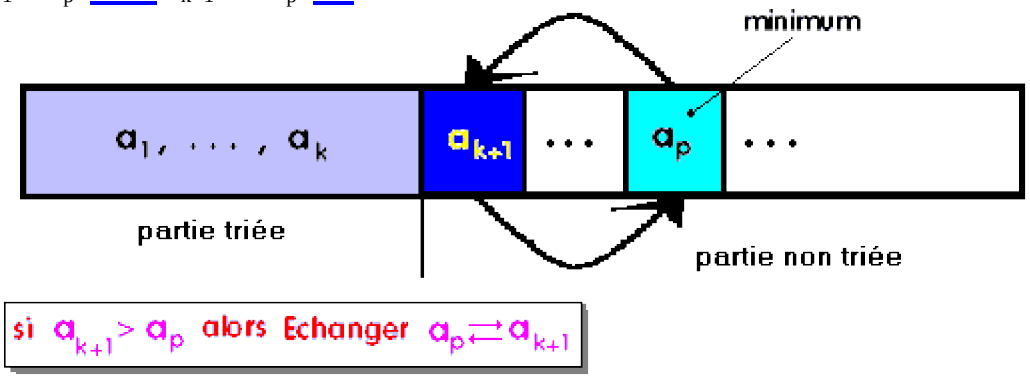
Le tableau contient une partie triée (en foncé à gauche) et une partie non triée (en blanc à droite).



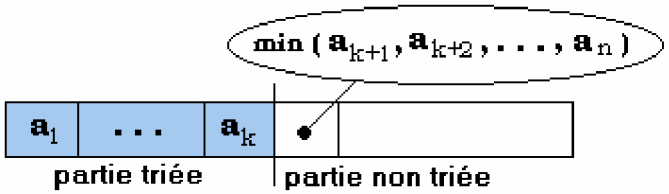
On cherche le minimum de la partie non-triée du tableau et on le recopie dans la cellule frontière (le premier élément de la partie non triée).

Donc pour tout $\mathbf{a}_p$ de la partie non triée on effectue l'action suivante :

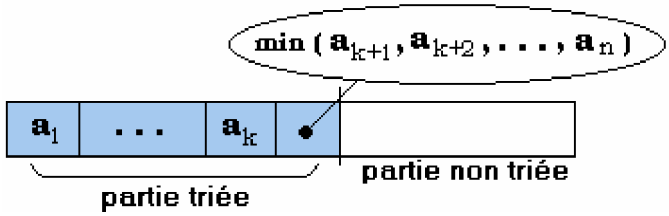
si $\mathbf{a}_{k+1} > \mathbf{a}_p$ **alors** $\mathbf{a}_{k+1} \leftarrow \mathbf{a}_p$ **Fsi**



et l'on obtient ainsi à la fin de l'examen de la sous-liste ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$) la valeur min ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$) stockée dans la cellule $\mathbf{a}_{k+1}$.



La sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$, $\mathbf{a}_{k+1}$) est maintenant triée :



Et l'on recommence la boucle de recherche du minimum sur la nouvelle sous-liste ($\mathbf{a}_{k+2}$, $\mathbf{a}_{k+3}$, ... , $\mathbf{a}_n$) etc...

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

C) Algorithme :

Une version maladroite de l'algorithme mais exacte a été fournie par un groupe d'étudiants elle est dénommée / **Version maladroite 1**/.

Elle échange physiquement et systématiquement l'élément frontière Tab[i] avec un élément Tab[j] dont la valeur est plus petite (la suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_i$) est triée) :

Maladroit

```
Algorithme Tri_Selection /Version maladroite 1/
local:   m, i , j , n, temp ∈  Entiers naturels
Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
        début
         pour i de 1 jusquà n-1faire // recommence une sous-suite
          m ← i ; // i est l'indice de l'élément frontière Tab[ i ]
          pour j de i+1 jusquà n faire// liste non-triée : (a_{i+1} , a_{i+2} , ... , a_n )
           si Tab[ j ] <  Tab[ m ] alors // a_j est le nouveau minimum partiel
            m ← j ;
            temp ←Tab[ m ] ;
            Tab[ m ] ← Tab[ i ] ;
            Tab[ i ]  ← temp //on échange les positions de a_i et de a_j
            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 a_i et a_j de la boucle interne "**pour** j **de** i+1 **jusquà n faire**" pour le déporter à la fin de cette boucle.

Amélioration

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.

Version maladroite	Version améliorée

pour j **de** i+1 **jusquà n faire**
  si Tab[j] < Tab[m] **alors**
  m ← j ;
  temp ←Tab[m] ;
  Tab[m] ← Tab[i] ;
  Tab[i] ← temp
  m ← i ;
  Fsi
fpour
	pour j **de** i+1 **jusquà n faire**
	  si Tab[j] < Tab[m] **alors**
	  m ← j ;
	  Fsi
	fpour;
	temp ← Tab[m] ;
	Tab[m] ←Tab[i] ;
	Tab[i] ← temp

A la fin de la boucle interne "**pour** j **de** i+1 **jusquà n faire**" la variable m contient l'indice de min(**a**i+1, **a**k+2, ... , **a**n) et l'on permute l'élément concerné (d'indice m) avec l'élément frontière ***ai*** :

```
Algorithme Tri_Selection  /Version 2 améliorée/
 local:   m, i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
          début
           pour i de 1 jusquà n-1 faire // recommence une sous-suite
            m ← i ; // i est l'indice de l'élément frontière ai = Tab[ i ]
            pour j de i+1 jusquà n faire// (a(i+1) , a(i+2) , ... , 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
```

D) Complexité :

Choix opération

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 du tri par sélection en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons 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.

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

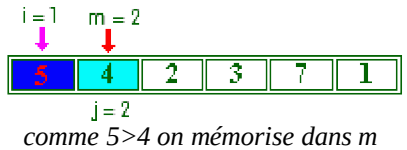
L'échange a lieu systématiquement dans la boucle principale "**pour** i **de** 1 **jusqà** n-1 **faire**" qui s'exécute n-1 fois :

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

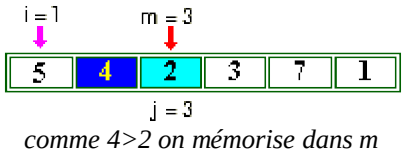
Un échange valant 3 transferts (affectation) la complexité en transfert est O(3n) = O(n)

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 **$O(n^2)$**.

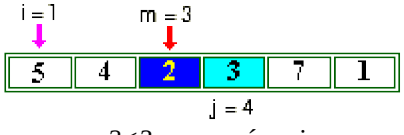
Exemple : soit la liste à 6 éléments (**5 , 4 , 2 , 3 , 7 ,1**), appliquons la version 2 du tri par sélection sur cette liste d'entiers. Visualisons les différents états de la liste pour la première itération externe contrôlée par i (i = 1) et pour les itérations internes contrôlées par l'indice j (de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



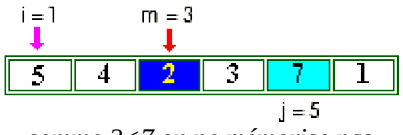
comme 5>4 on mémorise dans m



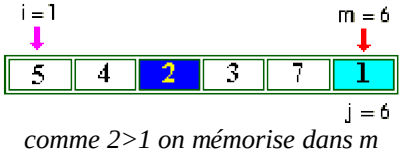
comme 4>2 on mémorise dans m



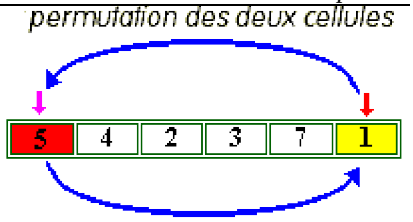
comme 2<3 on ne mémorise pas



comme 2<7 on ne mémorise pas

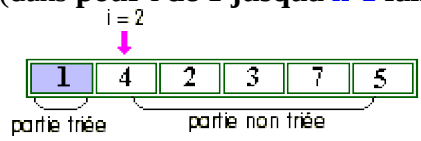


comme 2>1 on mémorise dans m



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 (dans **pour** i **de** 1 **jusqà n-1 faire** , il passe à i = 2) :



etc....

E) Programme C# (tableau d'entiers) :

```
class ApplicationTriSelection {
 static int[] table;  // le tableau à trier, par exemple 19 éléments de l'index 1 à l'index 19

   static void AfficherTable()
   {
       // Affichage du tableau
       int n = table.Length - 1;
       for (int i = 1; i <= n; i++)
           System.Console.Write(table[i] + " , ");
       System.Console.WriteLine();
   }

   static void InitTable()
   {
       int[] tableau = { 0 ,25, 7 , 14 , 26 , 25 , 53 , 74 , 99 ,
            24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52 };
       table = tableau;
   }
```

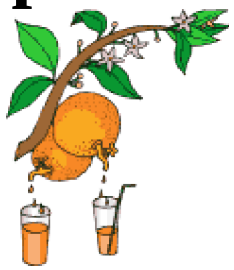
```
   static void TriParSelection()
   {
       int m, i, j, temp;
       int n = table.Length - 1;
       for ( i = 1; i <= n - 1; i++)
       {
           m = i;
           for ( j = i + 1; j <= n; j++)
               if (table[j] < table[m]) m = j;
           temp = table[m];
           table[m] = table[i];
           table[i] = temp;
       }
   }
```

```
   static void Main(string[] args)
   {
       InitTable();
       System.Console.WriteLine("Tableau initial :");
       AfficherTable();
       TriParSelection();
       System.Console.WriteLine("Tableau une fois trié :");
       AfficherTable();
       System.Console.Read();
   }
}
```

Résultat de l'exécution du programme précédent :

Tableau initial (n°1 au n°19):
25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52
Tableau une fois trié (n°1 au n°19) :
7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

Le tri par insertion



- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme C#

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**).**

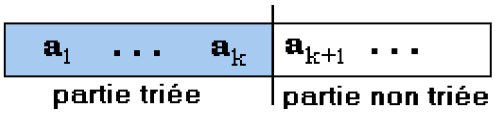
A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par insertion est de parcourir la liste non triée liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en la décomposant en deux parties : une partie 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.

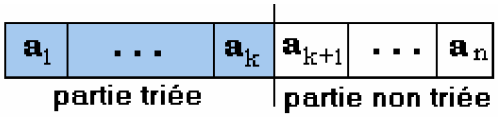
La liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est décomposée en deux parties : une partie triée ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$) et une partie non-triée ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$); l'élément $\mathbf{a}_{k+1}$ est appelé élément frontière (c'est le premier élément non trié).



B) Spécification concrète itérative

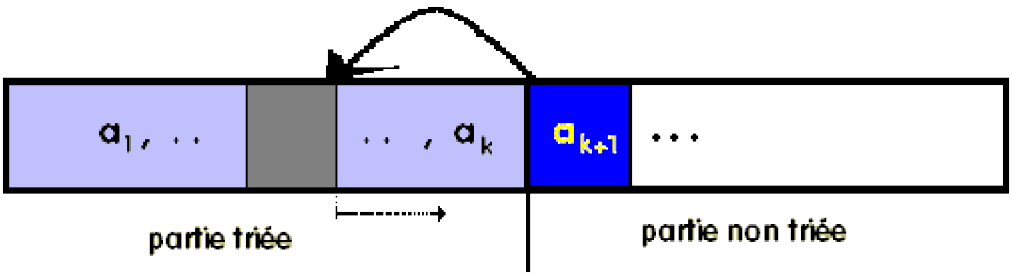
La suite (a_1, a_2, ... , a_n) est rangée dans un tableau à une dimension T[...] en mémoire centrale.

Le tableau contient une partie triée ((a_1, a_2, ... , a_k) en foncé à gauche) et une partie non triée ((a_{k+1}, a_{k+2}, ... , a_n); en blanc à droite) :



On insère l'élément frontière a_{k+1} en faisant varier j de k jusqu'à 2 , afin de balayer toute la partie (a1, a2, ... , ak) déjà rangée, on décale alors 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.

C) Algorithme :

```
Algorithme Tri_Insertion
 local:   i , j , n, v ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 0 à n éléments
 Sortie : Tab ∈  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 de2 jusqà 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 ;
```

Exercice

Un étudiant a proposé d'intégrer la comparaison dans le test de la boucle en écrivant ceci :

```
Tantque ( Tab[j-1] > v ) et ( j >0 ) faire
        Tab[ j ]  ← Tab[ j-1 ];
        j ← j-1;
FinTant ;
```

Il a eu des problèmes de dépassement d'indice de tableau lors de l'implémentation de son programme.

Essayez d'analyser l'origine du problème en notant que la présence d'une sentinelle élimine le problème.

D) Complexité :

Choix opération

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 l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

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 **jusquà 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 du tri par insertion au pire en nombre de transferts est de l'ordre de n^2, que l'on écrit $O(n^2)$.

E) Programme C# (tableau d'entiers) :

```
class ApplicationTriInsertion {
 static int[] table; // le tableau à trier, par exemple 19 éléments de l'index 1 à l'index 19

   static void AfficherTable()
   {
       // Affichage du tableau
       int n = table.Length - 1;
       for (int i = 1; i <= n; i++)
           System.Console.Write(table[i] + " , ");
       System.Console.WriteLine();
   }

   static void InitTable()
   {    // la sentinelle est l'entier le plus petit du type
       int[] tableau = { Int32.MinValue ,25, 7, 14, 26, 25, 53,
                         74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 ,
                         56 , 58 , 36 , 91 , 52 };
       table = tableau;
   }

   static void Main(string[] args)
   {
       InitTable();
       System.Console.WriteLine("Tableau initial :");
       AfficherTable();
       TriInsert();
       System.Console.WriteLine("Tableau une fois trié :");
       AfficherTable();
       System.Console.Read();
   }


   static void TriInsert()
   {
       // sous-programme de Tri par insertion :
       int n = table.Length - 1;
```

```
        for (int i = 2; i <= n; i++)
        {
            int v = table[i];
            int j = i;
            while (table[j - 1] > v)
            {
                table[j] = table[j - 1];
                j--;
            }
            table[j] = v;
        }
    }
}
```

Résultat de l'exécution du programme précédent :

Tableau initial (n°1 au n°19):
25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52
Tableau une fois trié (n°1 au n°19) :
7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

Le tri rapide

méthode Sedgewick

- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme Delphi - Java

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

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par insertion est de parcourir la liste L = liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en la divisant systématiquement en deux sous-listes L1 et L2. L'une de ces deux sous-listes est telle que tous ses éléments sont inférieurs à tous ceux de l'autre liste , la division en sous-liste a lieu en travaillant séparément sur chacune des deux sous-listes en appliquant à nouveau la même division à chaque sous-liste 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**.

Soit sur un exemple de liste L :

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

Nous obtenons deux sous-listes L1 et L2 :

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 ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau de dimension unT[...] 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 appliquant la même action sur les deux sous-listes obtenues après partition, 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 tel que :

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

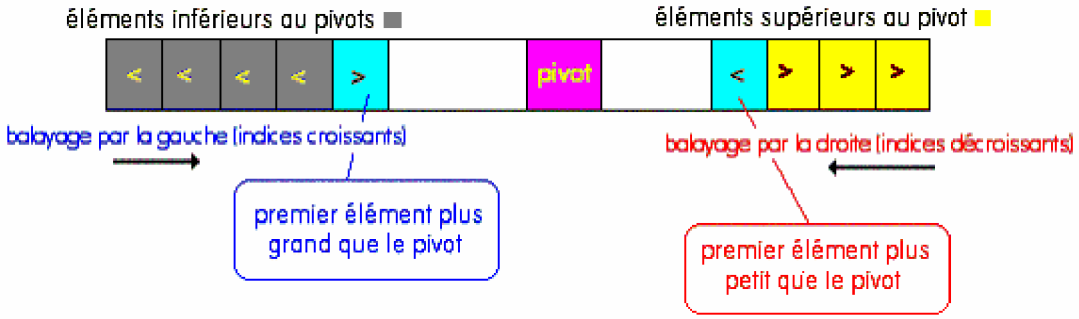
[x .. pivot'] = T[G] , .. , T[i-1]
(où : x = T[G] et pivot' = T[i-1]) tels que les T[G] , .. , T[i-1] sont tous inférieurs à T[i] ,

[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] ,

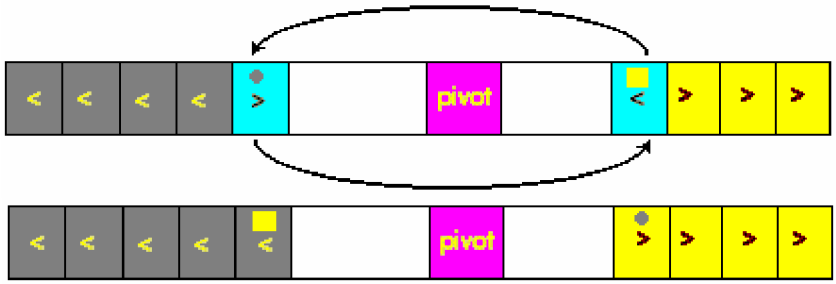
Il est proposé de **choisir arbitrairement le pivot** que l'on cherche à placer, puis ensuite de balayer la liste à réarranger dans les deux sens (par la gauche et par la droite) en construisant une sous-liste à gauche dont les éléments ont une valeur inférieure à celle du pivot et une sous-liste à droite 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 des deux éléments mal placés :
  [**4, 14**, **3**, **42**, 16, **2** , **23**, **45**, **18**, **38**] ----> [**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 jusqà 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).

Choix opération

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

Comme tous les algorithmes qui divisent et traitent le problème en deux sous-problèmes le nombre moyen de comparaisons est en **O(n.log(n))** que l'on nomme **complexité moyenne.** La notation **log** (x) est utilisée pour le logarithme à base 2, $\mathbf{log_2}$ (x).

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….

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

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriQsort {
 static int[] table;  // le tableau à trier, par exemple 19 éléments de l'index 1 à l'index 19

   static void AfficherTable()
   {
       // Affichage du tableau
       int n = table.Length - 1;
       for (int i = 1; i <= n; i++)
           System.Console.Write(table[i] + " , ");
       System.Console.WriteLine();
   }

   static void InitTable()
   {    // la sentinelle est l'entier le plus petit du type
       int[] tableau = { Int32.MinValue ,25, 7, 14, 26, 25, 53,
                         74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 ,
                         56 , 58 , 36 , 91 , 52, Int32.MaxValue };
       table = tableau;
   }

   static void Main(string[] args)
   {
       InitTable();
       System.Console.WriteLine("Tableau initial :");
       AfficherTable();
       int n = table.Length - 1;
       QSort(1, n);
       System.Console.WriteLine("Tableau une fois trié :");
       AfficherTable();
       System.Console.Read();
```

```
    }


    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);
        }
    }
}
```

Résultat de l'exécution du programme précédent :

Tableau initial (n°1 au n°19):
25 , 7 , 14 , 26 , 25 , 53 , 74 , 99 , 24 , 98 , 89 , 35 , 59 , 38 , 56 , 58 , 36 , 91 , 52
Tableau une fois trié (n°1 au n°19) :
7 , 14 , 24 , 25 , 25 , 26 , 35 , 36 , 38 , 52 , 53 , 56 , 58 , 59 , 74 , 89 , 91 , 98 , 99

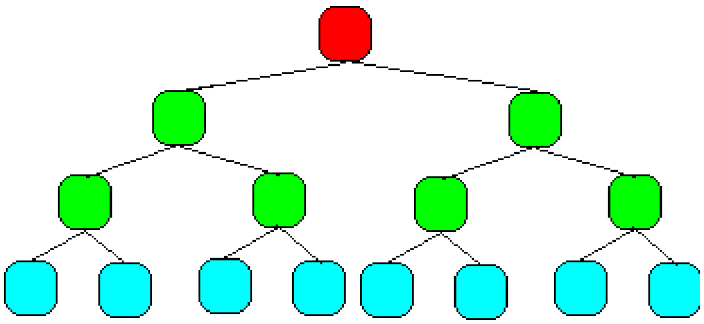
Le tri par arbre

C'est un tri également appelé tri par tas (*heapsort*, en anglais). Il utilise une structure de données temporaire dénommée "tas" comme mémoire de travail.

Définitions préliminaires

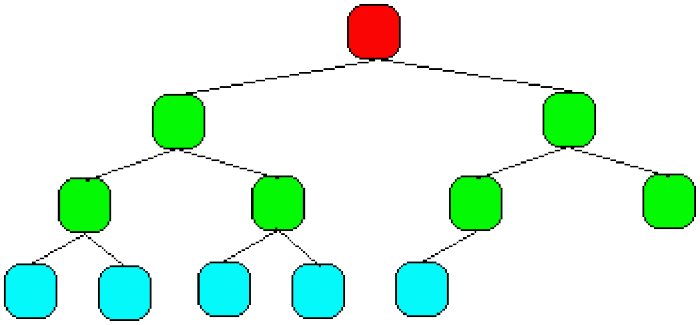
Définition - 1 / Arbre parfait :

c'est un arbre binaire dont tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles), dans ce cas l'arbre parfait est un arbre binaire incomplet et les feuilles du dernier niveau doivent être regroupées à partir de la gauche de l'arbre
.

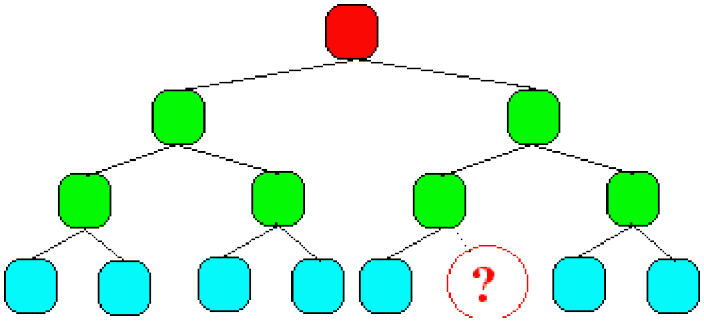


un arbre parfait complet

Amputons l'arbre parfait précédent de ses trois feuilles situées sur le bord droit, les cinq premières feuilles de gauche ne changeant pas, on obtient toujours un arbre parfait mais il est incomplet :



*un autre arbre **parfait** incomplet*

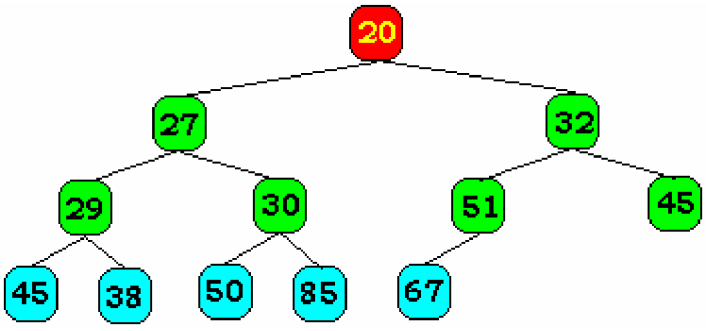


*exemple d'arbre **non parfait***

Définition - 2 / Arbre partiellement ordonné :

C'est un arbre étiqueté dont les noeuds appartiennent à un ensemble muni d'une relation d'ordre total (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses fils ont une valeur supérieure ou égale à celle de leur père.

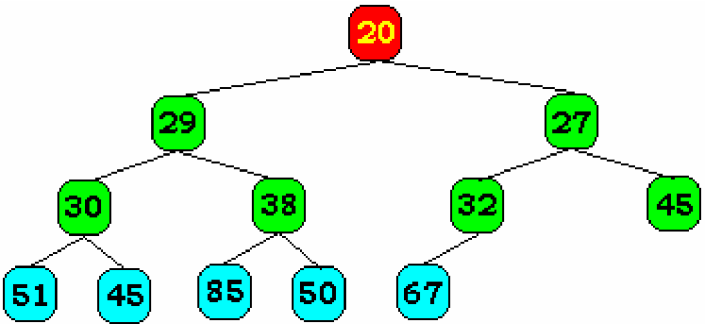
Exemple d'un arbre partiellement ordonné sur l'ensemble {20, 27,29, 30, 32, 38, 45, 45, 50, 51, 67 ,85 } d'entiers naturels :



Nous remarquons que **la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum** (le plus petit élément de l'ensemble), car la valeur de ce noeud par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses descendants c'est le minimum.

Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre.

Exemple d'un autre arbre partiellement ordonné sur le même ensemble {20, 27,29, 30, 32, 38, 45, 45, 50, 51, 67 ,85 } d'entiers naturels (il n'y a pas unicité) :



Définition - 3 / Le tas :

On appelle **tas** un tableau représentant un **arbre parfait partiellement ordonné**.

Principe du tri par tas

C'est une variante de méthode de tri par sélection où l'on parcourt le tableau des éléments en sélectionnant et conservant les minimas successifs (plus petits éléments partiels) dans un arbre parfait partiellement ordonné.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par tas est de parcourir la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en ajoutant chaque élément **a**k dans **un arbre parfait partiellement ordonné**.

- L'insertion d'un nouvel élément $\mathbf{a}_k$ dans l'arbre a lieu **dans la dernière feuille vide de l'arbre à partir de la gauche** (ou bien si le niveau est complet en recommençant un nouveau niveau par sa feuille la plus à gauche) et, en effectuant des échanges tant que la valeur de $\mathbf{a}_k$ est inférieur à celle de son pére.

- Lorsque tous les éléments de la liste seront placés dans l'arbre, l'élément minimum "$\mathbf{a}_i$" de la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) se retrouve à la racine de l'arbre qui est alors partiellement ordonné.

- On recommence le travail sur la nouvelle liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) -{$\mathbf{a}_i$}(c'est la liste précédente privée de son minimum),

  pour cela on supprime l'élément minimum **a**i de l'arbre pour le mettre dans la liste triée puis,

  on prend l'élément de la dernière feuille du dernier niveau et on le place à la racine.

  On effectue ensuite des échanges de contenu avec le fils dont le contenu est inférieur, en partant de la racine, et en descendant vers le fils avec lequel on a fait un échange, ceci tant qu'il n'a pas un contenu inférieur à ceux de ses deux fils (ou de son seul fils) ou tant qu'il n'est pas à une feuille.

- On recommence l'opération de suppression et d'échanges éventuels **jusqu'à ce que l'arbre ne contienne plus aucun élément**.

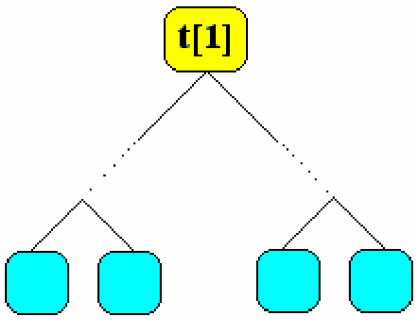
B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimension T[...] correspondant au tableau d'initialisation. Puis les éléments de ce tableau sont ajoutés et traités un par un dans un arbre avant d'être ajoutés dans un tableau trié en ordre décroissant ou croissant, selon le choix de l'utilisateur.

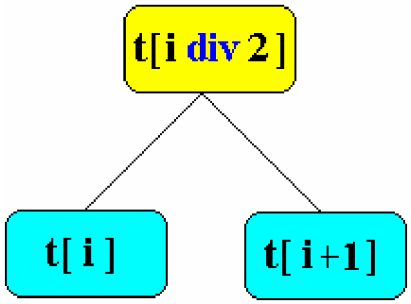
Signalons qu'un arbre binaire parfait se représente classiquement par un tableau :

Si t est ce tableau, on a les règles suivantes :

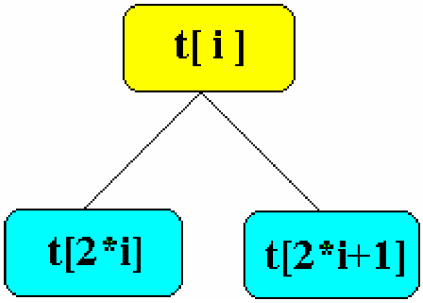
- t[1] est la racine :



- t[i **div** 2] est le père de t[i] pour i > 1 :



- t[2 * i] et t[2 * i + 1] sont les deux fils, s'ils existent, de t[i] :



- si **p** est le nombre de noeuds de l'arbre et si 2 * i = p, t[i] n'a qu'un fils, t[p].
  si i est supérieur à p **div** 2, t[i] est une feuille.

Figures illustrant le placement des éléments de la liste dans l'arbre

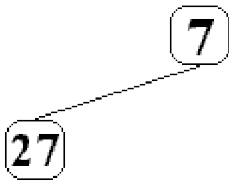
Exemple d'initialisation sur un tableau à 15 éléments :

7	27	41	30	10	31	22	33	23	17	3	25	44	7	25

Insertion du premier élément (le nombre **7**) à la racine de l'arbre :

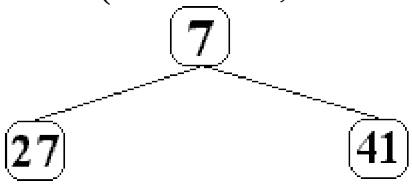
	27	41	30	10	31	22	33	23	17	3	25	44	7	25

Insertion du second élément (le nombre **27**, 27 > 7 donc c'est un fils du noeud 7) :

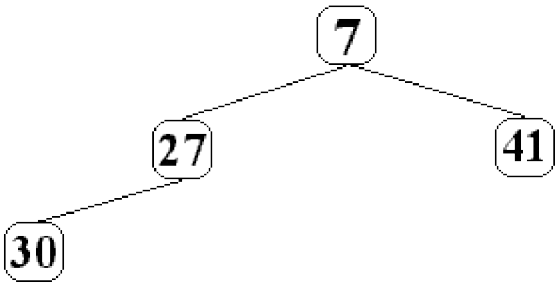


		41	30	10	31	22	33	23	17	3	25	44	7	25

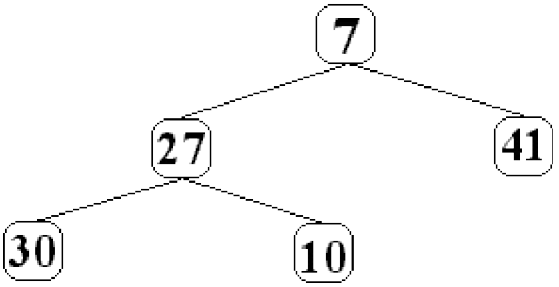
Insertion du troisième élément (le nombre **41**, 41 > 7 c'est un fils du noeud 7) :



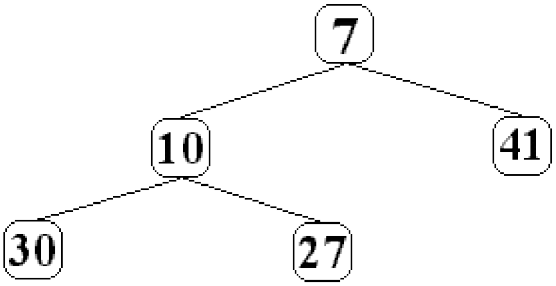
Insertion du quatrième élément (le nombre **30**, comme le niveau des fils du noeud 7 est complet, 30 est placé automatiquement sur une nouvelle feuille d'un nouveau niveau, puis il est comparé à son père 27, 30 > 27 c'est donc un fils du noeud 27 il n'y a pas d'échange) :



Insertion du cinquième élément (le nombre 10) : L'insertion du nouvel élément 10 dans l'arbre a lieu automatiquement dans la dernière feuille vide de l'arbre à partir de la gauche, ici le fils droit de 27 :

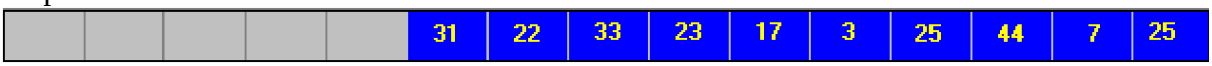


Puis 10 est comparé à son père 27, cette fois 10 est plus petit que 27, il y a donc échange des places entre 27 et 10, ce qui donne un nouvel arbre :

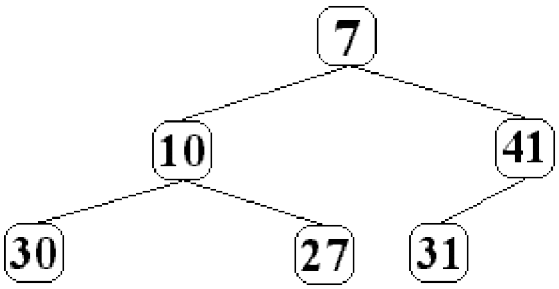


Puis 10 est comparé à son nouveau père 7, cette fois il n'y a pas d'échange car 10 est plus grand que 7.

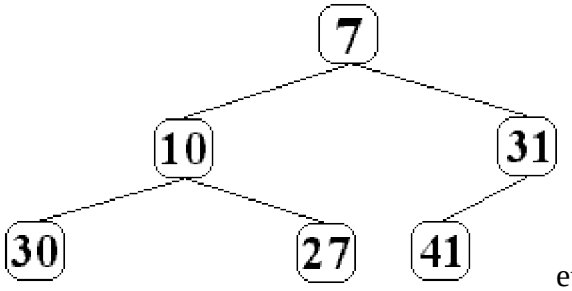
Le processus continue avec l'élément suivant de la liste le nombre 31:



31 est automatiquement rangé sur la première feuille disponible à gauche soit le fils gauche de 41:



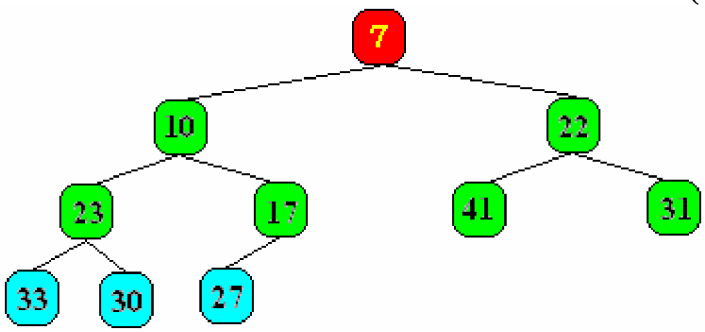
Puis 31 est comparé à son père, comme 31 < 41 il y a échange des valeurs, puis 31 est comparé à son nouveau père 7 comme 31 > 7 il n'y a plus d'échange :



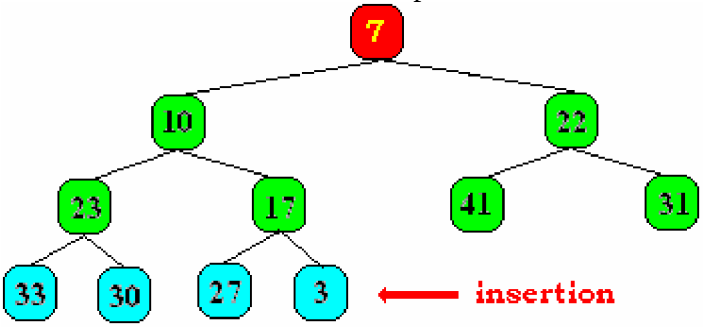
etc....

Supposons que l'arbre ait été construit sur les dix premiers éléments du tableau et observons maintenant comment l'élément minimum de la liste qui est le onzième élément, soit le nombre 3, est rangé dans l'arbre.

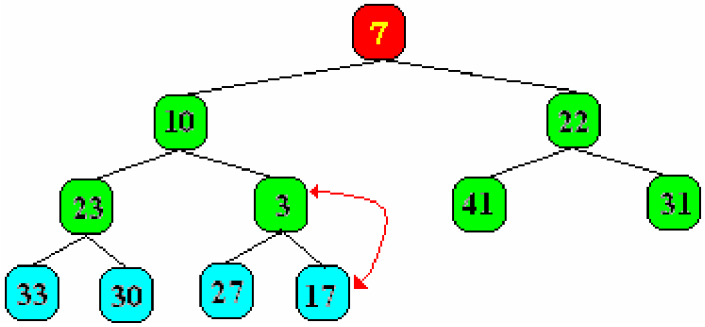
Voici l'état de l'arbre avant introduction du nombre 3 (quatre niveaux de nœuds) :



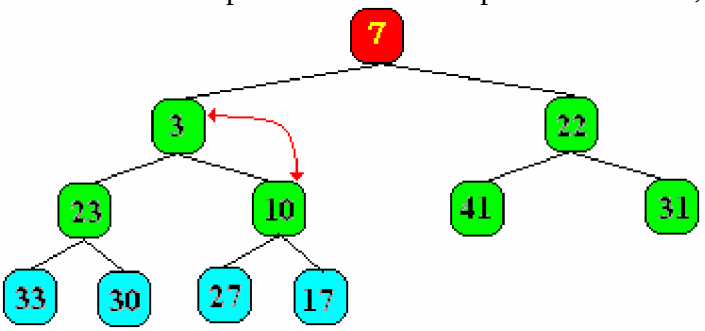
Le nombre 3 est introduit sur la première feuille libre du niveau quatre :



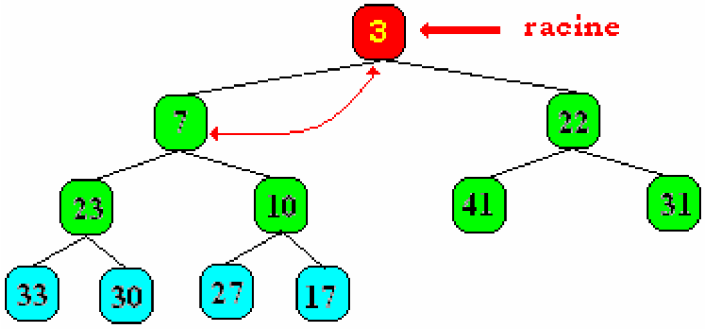
Il est comparé à son père le noeud 17, comme 3 < 17, il y a alors échange :



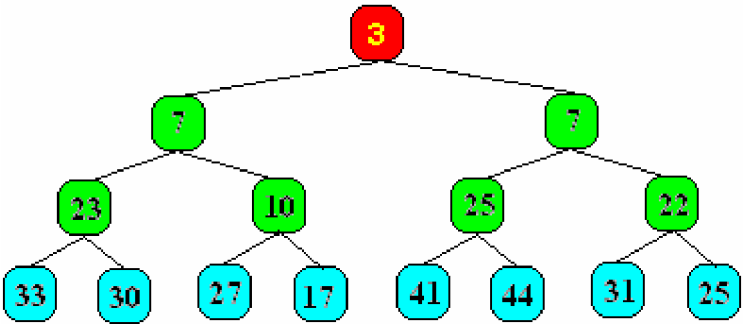
Il est ensuite comparé à son nouveau père le noeud 10, comme 3 < 10, il y a alors échange :



Il est enfin comparé à son dernier père (la racine de l'arbre), comme 3 < 7, il y a alors échange :



C'est l'élément **3** qui est maintenant **la racine** de l'arbre, comme les 4 éléments suivants sont plus grand que 3, ils seront rangés plus bas dans l'arbre (cf. figure ci-dessous) :

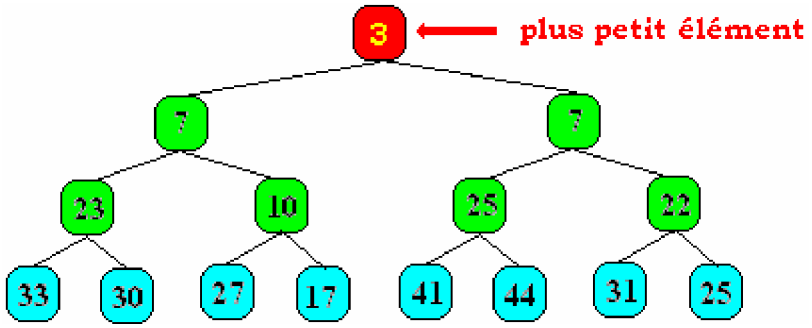


Conclusion sur le premier passage :

La liste initiale :

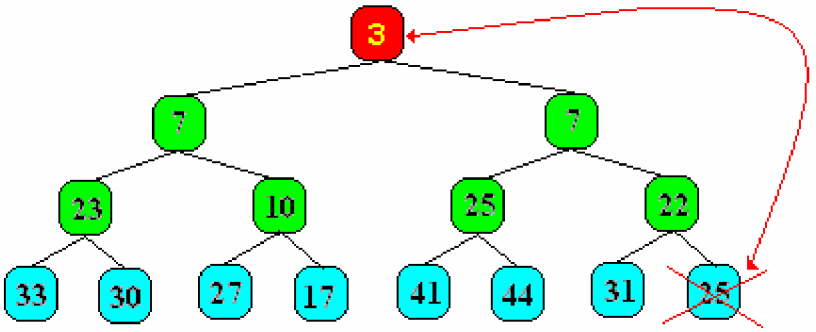
7	27	41	30	10	31	22	33	23	17	3	25	44	7	25

est finalement stockée ainsi :

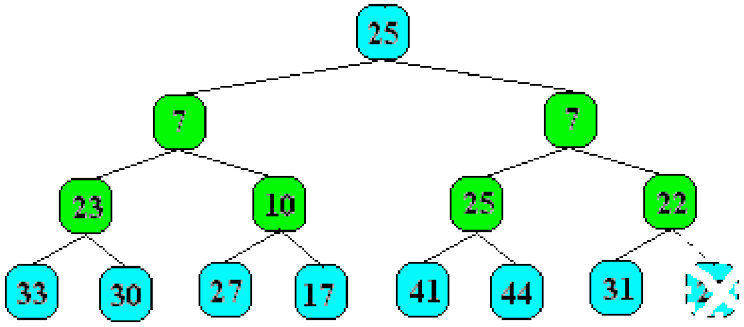


Figures illustrant la suppression de la racine

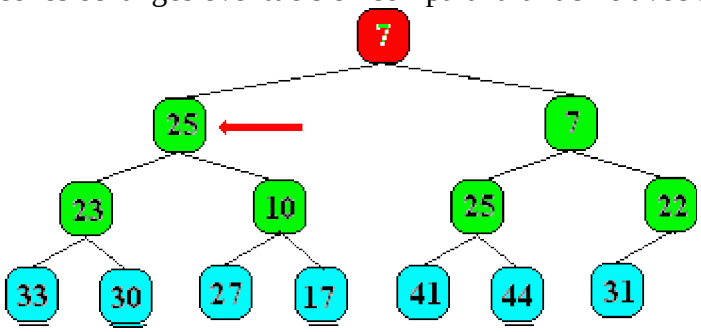
Le nombre **3** est le plus petit élément, on le supprime de l'arbre et l'on prend l'élément de la dernière feuille du dernier niveau (ici le nombre 25) et on le place à la racine (à la place qu'occupait le nombre 3)



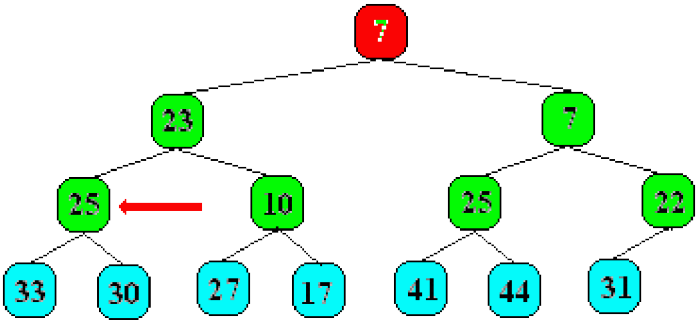
ce qui donne comme nouvelle disposition :



et l'on recommence les échanges éventuels en comparant la racine avec ses descendants :



le fils gauche 23 est inférieur à 25 => échange



Arrêt du processus (33 > 25 et 30 > 25)

On obtient le deuxième plus petit élément à la racine de l'arbre, ici le nombre 7.
Puis l' on continue à "vider" ainsi l'arbre et déplaçant les feuilles vers la racine et en échangeant les noeuds mal placés.

A la fin, lorsque l'arbre a été entièrement vidé, nous avons extrait à chaque étape le plus petit élément de chaque sous liste restante et nous obtenons les éléments de la liste classés par ordre croissant ou décroissant selon notre choix (dans notre exemple si nous stockons les minima successifs de gauche à droite dans une liste nous obtiendrons une liste classée par ordre croissant de gauche à droite).

En résumé notre version de tri par tas comporte les étapes suivantes :

- initialisation : ajouter successivement chacun des n éléments dans le tas t[1..p]; p augmente de 1 après chaque adjonction . A la fin on a un tas de taille p = n.
- tant que p est plus grand que 1, supprimer le minimum du tas (p décroît de 1), réorganiser le tas, ranger le minimum obtenue à la $(p + 1)^{ième}$ place.

On en déduit l'algorithme ci-dessous composé de 2 sous algorithmes Ajouter pour la première étape, et Supprimer pour la seconde.

C) Algorithme :

```
Algorithme Ajouter
Entrée  P : entier ; x : entier // P nombre d'éléments dans le tas, x élément à ajouter
          Tas[1..max]  : tableau d'entiers // le tas
Sortie  P : entier
          Tas[1..max] // le tas
Local  j, temp : entiers
début
  P ← P + 1 ; // incrémentation du nombre d'éléments du tas
  j ← P ; // initialisation de j à la longueur du tas (position de la dernière feuille)
  Tas[P] ← x ; // ajout l'élément x à la dernière feuille dans le tas
  Tantque (j > 1) et (Tas[j] < Tas[j div 2]) faire ; // tant que l'on est pas arrivé à la racine et
que le "fils" est inférieur à son "père", on permute les 2 valeurs
        temp ← Tas[j] ;
        Tas[j] ← Tas[j div 2] ;
        Tas[j div 2] ← temp ;
        j ← j div 2 ;
  finTant
FinAjouter

Algorithme Supprimer
Entrée : P : entier // P nombre d'éléments contenu dans l'arbre
          Tas[1..max]  : tableau d'entiers // le tas
Sortie : P : entier // P nombre d'éléments contenu dans l'arbre
          Tas[1..max]  : tableau d'entiers // le tas
          Lemini : entier  // le plus petit de tous les éléments du tas
Local i, j, temp : entiers ;
début
  Lemini ← Tas[1] ; // retourne  la racine (minimum actuel) pour stockage éventuel
  Tas[1] ←Tas[P] ; // la racine prend la valeur de la dernière feuille de l'arbre
  P ← P - 1 ;
  j ←1 ;
  Tantque j <= (P div 2) faire
     // recherche de l'indice i du plus petit des descendants de Tas[j]
     si (2 * j = P ) ou (Tas[2 * j] < Tas[2 * j + 1])
           alors i ← 2 * j ;
     sinon i ←2 * j +1 ;
     Fsi;
    // Echange éventuel entre Tas[j] et son fils Tas[i]
    si Tlab[j] > Tlab[i] alors
          temp ← Tlab[j] ;
          Tlab[j] ← Tlab[i] ;
          Tlab[i] ← temp ;
          j ← i ;
    sinon Sortir ;
    Fsi;
  finTant
FinSupprimer
```

```
Algorithme Tri_par_arbre
Entrée  Tas[1..max] // le tas
          TabInit[1..max] // les données initiales
Sortie    TabTrie[1..max]: tableaux d'entiers //  tableau une fois trié
Local     P  : Entier //  P le nombre d'éléments à trier
          Lemin : entier  // le plus petit de tous les éléments du tas
début
   P ← 0;
  Tantque P < max faire
     Ajouter(P,Tas,TabInit[P+1]); // appel de l'algorithme Ajouter
  finTant;
  Tantque P >= 1 faire
     Supprimer(P,Tas,Lemin) ; // appel de l'algorithme Supprimer
     TabTrie[max-P] ← Lemin ; // stockage du minimum dans le nouveau tableau
  finTant;
Fin Tri_par_arbre
```

D) Complexité :

Cette version de l'algorithme construit le tas par n appels de la procédure **Ajouter** et effectue les sélections par n - 1 appels de la procédure supprimer.

Le coût et de l'ordre de $\sum_{i=1}^{n} \log_2 i$ comparaisons, au pire.

La complexité au pire en nombre de comparaisons est en O(n log n).

Le nombre d'échanges dans le tas est majoré par le nombre de comparaisons et il est du même ordre de grandeur.

La complexité au pire en nombre de transferts du tri par tas est donc en O(n log n).

Exercice : écrire un programme C# du tri par arbre.

3. Rechercher dans un tableau

Les tableaux sont des structures statiques contiguës, il est facile de rechercher un élément fixé dans cette structure. Nous exposons ci-après des algorithmes élémentaires de recherche utilisables dans des applications pratiques par le lecteur.

Essentiellement nous envisagerons la **recherche séquentielle** qui convient lorsqu'il y a peu d'éléments à consulter (quelques centaines), et la **recherche dichotomique** dans le cas où la liste est triée.

3.1 Recherche dans un tableau non trié

Algorithme de recherche séquentielle :

- Soit **t** un tableau d'entiers de **1..n** éléments non rangés.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)
- Complexité en O(n)

Nous proposons au lecteur 4 versions d'un même algorithme de recherche séquentielle. Le lecteur adoptera une de ces versions en fonction des possibilités du langage avec lequel il compte programmer sa recherche.

Version **Tantque** avec "**et alors**" (si le langage dispose d'un opérateur **et** optimisé)	Version **Tantque** avec "**et**" (opérateur **et** non optimisé)
i ← 1 ; **Tantque** (i ≤ n) **et alors** (t[i] ≠ Elt) **faire** i ← i+1 **finTant;** **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**	i ← 1 ; **Tantque** (i < n) **et** (t[i] ≠ Elt) **faire** i ← i+1 **finTant;** **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi**

Version **Tantque** avec sentinelle en fin de tableau (rajout d'une cellule supplémentaire en fin de tableau contenant systématiquement l'élément à rechercher)

Version **Tantque** avec sentinelle avec "**et alors**"	Version **Pour** avec instruction de sortie (conseillée si le langage dispose d'un opérateur **Sortirsi**)
t[n+1] ← Elt ; *// sentinelle rajoutée* i ← 1 ; **Tantque** (i ≤ n) **et alors** (t[i] ≠ Elt) **faire** i ← i+1 **finTant;** **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**	**pour** i ← 1 **jusquà** n **faire** **Sortirsi** t[i] = Elt **fpour**; **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**

3.2 Recherche dans un tableau trié

Spécifications d'un algorithme séquentiel

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**) ***On peut reprendre sans changement les versions de l'algorithme de recherche séquentielle précédent travaillant sur un tableau non trié.***
- Complexité moyenne en O(n)

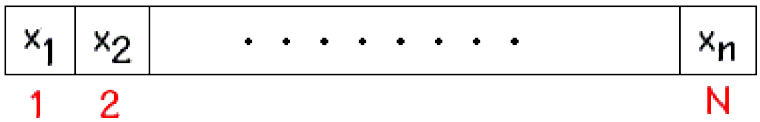
On peut aussi utiliser le fait que le dernier élément du tableau est **le plus grand élément** et s'en servir comme une sorte de **sentinelle**. Ci-dessous deux versions utilisant cette dernière remarque.

Version **Tantque**	Version **pour**
si t[n] < Elt **alors** rang ← -1 **sinon** i ← 1 ; **Tantque** t[i] < Elt **faire** i ← i+1; **finTant;** **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi** **Fsi**	**si** t[n] < Elt **alors** rang ← -1 **sinon** **pour** i ← 1 **jusquà** n-1 **faire** **Sortirsi** t[i] ≥ Elt *// sortie de la boucle* **fpour**; **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi** **Fsi**

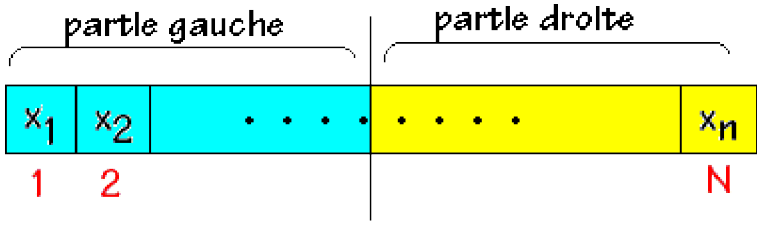
Spécifications d'un algorithme dichotomique

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**). **Au lieu de rechercher séquentiellement du premier jusqu'au dernier, on compare l'élément Elt à chercher au contenu du milieu du tableau. Si c'est le même, on retourne le rang du milieu, sinon l'on recommence sur la première moitié (ou la deuxième) si l'élément recherché est plus petit (ou plus grand) que le contenu du milieu du tableau.**
- Complexité moyenne en O(log(n))

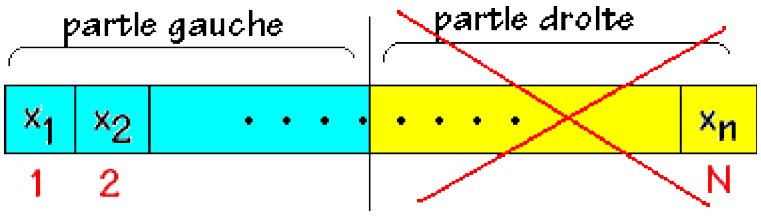
Soit un tableau au départ contenant les éléments (x_1, x_2, ... , x_n)



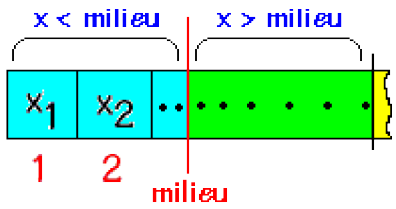
On recherche la présence de l'élément x dans ce tableau. On divise le tableau en 2 parties égales :



Soit x est inférieur à l'élément milieu et s'il est présent il ne peut être que dans la partie gauche, soit x est supérieur à l' élément milieu et s'il est présent il ne peut être que dans la partie droite. Supposons que x < milieu nous abandonnons la partie droite et nous recommençons la même division sur la partie gauche :



On divise à nouveau la partie gauche en deux parties égales avec un nouveau milieu :



Si x < milieu c'est la partie gauche qui est conservée sinon c'est la partie droite etc ...
Le principe de la division dichotomique aboutit à la fin à une dernière cellule dans laquelle soit x = milieu et x est présent dans la table, soit x ≠ milieu et x n'est pas présent dans la table.

Version itérative du corps de cet algorithme :

```
bas, milieu, haut, rang : entiers

    bas ← 1;
    haut ← N;
    Rang ← -1;
    repéter
      milieu ← (bas + haut) div 2;
      si x = t[milieu] alors
         Rang ← milieu
      sinon si t[milieu] < x alors
                bas ← milieu + 1
           sinon
                haut ← milieu-1
           fsi
      fsi
    jusquà ( x = t[milieu] ) ou ( bas haut )
```

Voici en C# , un programme complet traduisant la version itérative de cet algorithme :

```
class ApplicationRechDicho
{
    static int[] table; // le tableau à examiner cellules de 1 à 19
    static void AfficherTable()
    {
        // Affichage du tableau
        int n = table.Length - 1;
        for (int i = 1; i <= n; i++)
            System.Console.Write(table[i] + " , ");
        System.Console.WriteLine();
    }
    static void InitTable()
    {
        // La cellule de rang zéro est inutilisée (examen sur 19 éléments)
        int[] tableau = { 0, 53, 77, 11, 72, 28, 43, 65, 83, 39, 73, 82,
                          69, 65, 4, 95, 46, 12, 87, 75 };
        table = tableau;
    }
    static void TriInsert()
    {
        // sous-programme de Tri par insertion :
        int n = table.Length - 1;
        for (int i = 2; i <= n; i++)
        {
            int v = table[i];
            int j = i;
            while (table[j - 1] > v)
            {
```

```
                table[j] = table[j - 1];
                j = j - 1;
            }
            table[j] = v;
        }
    }
    static int RechDichoIter(int[] t, int Elt)
    {
        int n = t.Length - 1;
        int bas = 1, haut = n, milieu;
        int Rang = -1;
        do
        {
            milieu = (bas + haut) / 2;
            if (Elt == t[milieu]) Rang = milieu;
            else if (t[milieu] < Elt) bas = milieu + 1;
            else haut = milieu - 1;
        }
        while ((Elt != t[milieu]) & (bas <= haut));
        return Rang;
    }
    static void Main(string[] args)
    {
        InitTable();
        System.Console.WriteLine("Tableau initial :");
        AfficherTable();
        TriInsert();
        System.Console.WriteLine("Tableau trié :");
        AfficherTable();
        int x = Int32.Parse(System.Console.ReadLine()), rang;
        rang = RechDichoIter(table, x);
        if (rang > 0)
         System.Console.WriteLine("Elément " + x + " trouvé en : " + rang);
        else System.Console.WriteLine("Elément " + x + " non trouvé !");
         System.Console.Read();
    }
}
```

Dans le cas de langage de programmation acceptant la récursivité (comme C#), il est possible d'écrire une version récursive de cet algorithme dichotomique. Voici en C# un méthode traduisant la version récursive de cet algorithme :

```
static void RechDichoRecur(int Elt, int[] table,
                                   int bas, int haut, ref int rang)
{
  if (bas <= haut)
  {
    int milieu = (bas + haut) / 2;
    if (Elt == table[milieu]) rang = milieu;
    else
      if (Elt < table[milieu])
        RechDichoRecur(Elt, table, bas, milieu - 1, ref rang);
      else
        RechDichoRecur(Elt, table, milieu + 1, haut, ref rang);
  }
  else
    rang = -1;
}
```

Ci-après un code pour la méthode Main appelant la méthode récursive précédente :

```
static void Main(string[] args)
{
  InitTable();
  System.Console.WriteLine("Tableau initial :");
  AfficherTable();
  TriInsert();
  System.Console.WriteLine("Tableau trié :");
  AfficherTable();
  int rang = -1;
  int x = Int32.Parse(System.Console.ReadLine());
  RechDichoRecur(x, table, 1, table.Length - 1, ref rang);
  if (rang > 0)
    System.Console.WriteLine("Elément " + x + " trouvé en : " + rang);
  else System.Console.WriteLine("Elément " + x + " non trouvé !");
       System.Console.Read();
}
```