Une méthode pour élaborer des algorithmes itératifs

F. Didier-IREM Campus Luminy
didier@irem.univ-mrs.fr

Atelier 5a
Séminaire des IREM et de la revue "Repère IREM"
CIRM (Marseille-Luminy)
15 au 19 mars 2010

1 Construction de programmes itératifs par la mise en évidence de l'invariant

La démarche présentée tente, sans introduire un formalisme excessif, d'apporter des éléments de réponses aux questions fondamentales qui se posent en algorithmique :

– Comment élaborer un algorithme ?
– L'algorithme s'arrête-t-il toujours ?
– Comment prouver qu'un algorithme résout effectivement le problème posé ?

Une des principales difficultés dans l'élaboration d'un algorithme est de contrôler son aspect dynamique (c'est à dire la façon dont il se comporte en fonction des données qu'on lui fournit en entrée), en ne considérant que son aspect statique (on n'a sous les yeux qu'une suite finie d'instructions). La méthode ci-dessous permet d'apporter une aide à la mise au point d'algorithmes itératifs et de contrôler cet aspect dynamique.

L'idée principale de la méthode consiste à raisonner en terme de **situation** alors que l'étudiant débutant, lui, raisonne en terme **d'action**. Par exemple, la description d'un algorithme de tri commence le plus souvent par des phrases du type : "je compare le premier élément avec le second, s'il est plus grand je les échange, puis je compare le second avec le troisième......". On finit par s'y perdre et ce n'est que très rarement que l'algorithme ainsi explicité arrive à être codé correctement. Le fait de raisonner en terme de situation permet, comme on va le voir, de mieux expliciter et de mieux contrôler la construction d'un algorithme. Pour terminer cette introduction, on peut ajouter que cette façon de procéder peut aussi s'avérer utile, dans le cas où un algorithme est connu, pour se convaincre ou convaincre un auditoire que l'algorithme résout bien le problème posé.

1.1 La démarche

On sait que ce qui caractérise une itération est son invariant. Trouver l'invariant d'une boucle n'est pas toujours chose aisée. L'idée maîtresse de la méthode est de construire cet invariant parallèlement à l'élaboration de l'algorithme et non pas de concevoir l'algorithme itératif pour ensuite en rechercher l'invariant. Etant donné un problème dont on soupçonne qu'il a une solution itérative, on va s'efforcer de mettre en évidence les étapes suivantes :

1 Proposer une situation générale décrivant le problème posé (hypothèse de récurrence). C'est cette étape qui est peut être la plus délicate car elle exige de faire preuve d'imagination. On peut toujours supposer que l'algorithme (on le cherche !) a commencé à "travailler" pour résoudre le problème posé et qu'on l'arrête avant qu'il ait fini : On essaye alors de décrire, de manière

très précise, une situation dans laquelle les données qu'il manipule puissent se trouver. Il est possible, comme on le verra sur des exemples, d'imaginer plusieurs situations générales.

2 Chercher la condition d'arrêt. A partir de la situation imaginée en [1], on doit formuler la condition qui permet d'affirmer que l'algorithme a terminé son travail. La situation dans laquelle il se trouve alors, est appelée situation finale.

3 Se "rapprocher" de la situation finale, tout en faisant le nécessaire pour conserver une situation générale analogue à celle choisie en [1].

4 Initialiser les variables introduites dans la description de la situation générale pour que celle-ci soit vraie au départ (c'est à dire avant que l'algorithme ait commencé à travailler).

Une fois cette étude conduite l'algorithme aura la structure suivante :

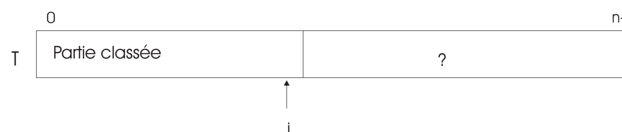
[4]
tant que non [2] faire [3]

Cette façon de procéder montre comment on prouve la validité de l'algorithme au fur et à mesure de son élaboration. En effet la situation générale choisie en [1] est en fait l'invariant qui caractérise la boucle $tantque$. Cette situation est satisfaite au départ à cause de l'étape [4], elle reste vraie à chaque itération (étape [3]). Ainsi lorsque la condition d'arrêt est atteinte cette situation nous permet d'affirmer que le problème est résolu. C'est également en analysant l'étape [3] qu'on peut prouver la terminaison de l'algorithme.

1.2 Tri d'un tableau par insertion

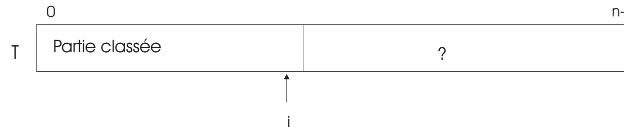
Le premier exemple consiste à établir un algorithme permettant de classer suivant l'ordre croissant un tableau de n nombres. Essayons de trouver une situation générale décrivant le problème posé. Pour cela supposons que l'on arrête un algorithme de tri avant que tout le tableau soit classé, on peut imaginer qu'il a commencé à mettre de l'ordre et que par exemple $T[0..i]$ est classé. La notation $T[0..i]$ désigne les $i+1$ premières composantes du tableau T.

Pour illustrer cette situation on peut soit faire un dessin :



Soit procéder, plus formellement, en décrivant la situation par une formule logique :
$\{\forall j, j \in [0, i[\Rightarrow T[j] \leq T[j+1]\}$

[1]



[2] L'algorithme a terminé lorsque $i = n - 1$.

[3] Pour se rapprocher de la fin, on incrémente d'abord i de 1, mais si on veut conserver une situation analogue à celle choisie, cela ne suffit pas, car il faut que le tableau soit classé entre 0 et i. Pour cela "on doit amener $T[i]$ à sa place dans $T[0..i]$". Ceci est un autre problème qui sera résolu en utilisant la même démarche.

[4] $T[0..0]$ étant trié, l'initialisation $i \leftarrow 0$ convient.

Cette première analyse nous conduit à écrire la séquence suivante :

$i \leftarrow 0$
tant que non $i = n - 1$ **faire**
| $i \leftarrow i + 1$
| "amener $T[i]$ à sa place dans $T[0..i]$"

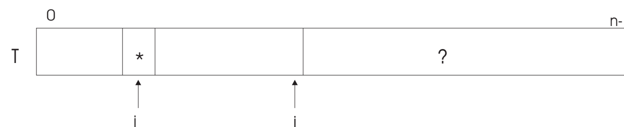
L'invariant :$\{\forall j,\ j \in [0, i[\Rightarrow T[j] \leq T[j+1]\}$ joint à la condition d'arrêt $i = n - 1$ s'écrit à la sortie de la boucle :
$\{\forall j, j \in [0, n-1[\Rightarrow T[j] \leq T[j+1]\}$

Ce qui veut précisément dire que tout le tableau est classé. La démonstration de la terminaison de cet algorithme est triviale car i est initialisé avec 0 et chaque fois que l'étape [3] est exécutée i est incrémenté de 1, la condition d'arrêt $i = n-1$ sera forcément vérifiée au bout de n étapes.

Toute réalisation de "amener $T[i]$ à sa place dans $T[0..i]$" nous donne un algorithme de tri.

Par hypothèse on sait que le tableau est classé entre 0 et $i-1$, on suppose qu'un algorithme a commencé à travailler et l'élément qui était initialement en i s'est "rapproché" de sa place par des échanges successifs et se trouve en j lorsqu'on a interrompu l'algorithme. Sur le schéma qui suit, l'élément qui était initialement en i est matérialisé par une étoile.

[1] Avec $T[0..j-1]$ et $T[j..i]$ classés.



[2] C'est terminé lorsque $j = 0$ ou $T[j-1] \leq T[j]$.

[3]
Echanger $T[j-1]$ et $T[j]$
$j \leftarrow j-1$

[4] L'initialisation $j \leftarrow i$ satisfait la situation choisie en [1].

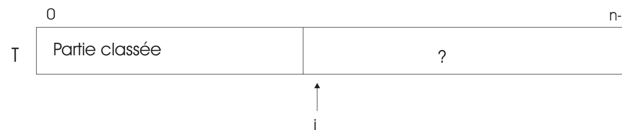
Ce qui nous conduit finalement à écrire l'algorithme de tri par insertion :

```
i ← 0
tant que i ≠ n − 1 faire
  | i ← i + 1
  | j ← i
  | tant que j ≠ 0 et T[j − 1] > T[j] faire
  |   | Echanger T[j − 1] et T[j]
  |   | j ← j − 1
```

On remarquera que dans cette ultime version les expressions booléennes qui suivent les $tantque$ sont les négations des conditions d'arrêt.

On aurait pu raisonner à partir d'une situation générale sensiblement différente de celle choisie :

[1]



La nuance réside dans le fait que la partie triée est $T[0..i-1]$, et non $T[0..i]$ comme dans la première version. Cette situation est tout aussi acceptable que la précédente. En raisonnant avec celle-ci on obtient un algorithme, certes voisin du précédent, mais comportant plusieurs modifications. La condition d'arrêt devient $i = n$, le corps de l'itération consiste à "amener l'élément qui se trouve en i à sa place dans $T[0..i-1]$, puis vient l'incrémentation de i, l'initialisation devient $i \leftarrow 1$. Comme on peut le constater la situation choisie guide totalement l'écriture de l'algorithme.

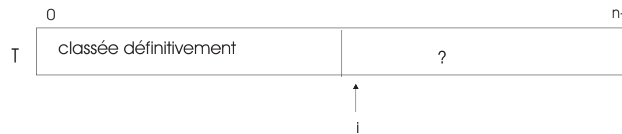
1.3 Tri d'un tableau par sélection

Comme il a déjà été dit et constaté la situation générale n'est pas unique. Par exemple, toujours pour le même problème, on peut faire une hypothèse plus forte sur la partie classée en ajoutant qu'elle est aussi définitivement en place. Ce qui se traduit formellement par une conjonction de formules logiques :

$$\{\forall j, j \in [0, i-1[\Rightarrow T[j] \leq T[j+1]\}$$
et
$$\{(\forall j, j \in [0, i-1] \text{ et } \forall k, k \in [i, n-1]) \Rightarrow T[j] \leq T[k]\}$$

La première formule indique que le tableau est classé jusqu'à $i-1$, la seconde formule que cette partie est définitivement en place.

[1]



[2] C'est terminé lorsque $i = n - 1$.

[3]
"Rechercher l'indice k du plus petit élément de $T[i..n-1]$"
Echanger $T[i]$ et $T[k]$.
Incrémenter i de 1.

[4] L'initialisation $i \leftarrow 0$ convient.

Ce qui nous conduit à écrire l'algorithme suivant :

$i \leftarrow 0$
tant que $i \neq n-1$ **faire**
  "Rechercher l'indice k du plus petit élément de $T[i..n-1]$"
  Echanger $T[i]$ et $T[k]$
  $i \leftarrow i+1$

Toute réalisation de "rechercher l'indice k du plus petit élément de $T[i..n-1]$" conduit à un algorithme de tri. L'analyse de ce sous problème est laissée au soin du lecteur. Cet algorithme se termine car la variable i est initialisée avec 0 et est incrémentée chaque fois que l'étape 3 est exécutée, après n itérations la condition d'arrêt $i = n - 1$ est atteinte. A la sortie de l'itération i vaut donc $n - 1$. Ainsi, en remplaçant i par $n - 1$ dans les formules logiques qui décrivent la situation générale choisie, on obtient précisément la définition du fait que le tableau T est classé, ce qui prouve la validité de notre algorithme.

Revenons un instant sur l'étape [4]. Pourquoi l'initialisation $i \leftarrow 0$ satisfait-elle la situation choisie au départ ? On peut justifier ce choix soit intuitivement, soit formellement.
Intuitivement :
Lorsque i vaut 0, $T[0..i-1]$ désigne le tableau vide. On peut dire du tableau vide

qu'il a toutes les propriétés que l'on veut, puisqu'il ne possède aucun élément. En particulier, on peut dire que ses éléments sont classés et en place.
Formellement :
A ce stade il est nécessaire de faire quelques rappels :

a) Si P est fausse, l'implication $P \Rightarrow Q$ est vraie.

b) La propriété $\forall x, x \in \emptyset$ est toujours fausse ($\emptyset$ désigne l'ensemble vide).

Examinons maintenant la formule logique décrivant la situation générale.

$$\{\forall j, j \in [0, i-1[\Rightarrow T[j] \leq T[j+1]\}$$
$$\text{et}$$
$$\{(\forall j, j \in [0, i-1] \text{ et } \forall k, k \in [i, n-1]) \Rightarrow T[j] \leq T[k]\}$$

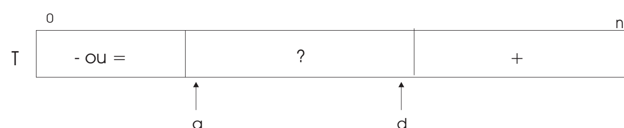
Il faut qu'au départ (i vaut 0) cette formule soit vraie. Pour qu'une conjonction de deux formules soit vraie il suffit que chacune des deux formules soit vraie. Il est facile de constater que la première formule est vraie lorsque i vaut 0. En effet cette formule est de la forme $P \Rightarrow Q$, où P est de la forme $\forall x, x \in \emptyset$, car lorsque i vaut 0 l'intervalle semi-ouvert $[0..i-1[$ possède zéro élément.
La seconde formule et les rappels a) et b) nous permettent encore de conclure.

1.4 La recherche dichotomique

Soit $T[0..n-1]$ un tableau de n éléments classés par ordre croissant. Le problème consiste à établir un algorithme qui permette de dire si un élément x appartient ou pas au tableau T. L'algorithme recherché ne doit pas effectuer une recherche séquentielle, mais doit utiliser le principe de la dichotomie :
à chaque étape de l'itération, l'intervalle de recherche doit être divisé par deux. Pour rechercher un élément dans un tableau ayant un million d'éléments, au plus vingt comparaisons suffiront !
Cet exemple de la recherche dichotomique est très riche en soi, car les variations de situation sont multiples et la méthode permet de parfaitement contrôler ce que l'on écrit. Il faut ajouter que les versions, la plupart du temps erronées, que les étudiants écrivent, peuvent être plus facilement corrigées si l'on cherche à mettre en évidence la situation générale à partir de laquelle leur algorithme semble construit.

1.4.1 Version a

[1]



où + indique la zone des éléments de T qui sont strictement plus grands que x et - ou = indique la zone des éléments inférieurs ou égaux à x.

[2] C'est terminé lorsque $d < g$

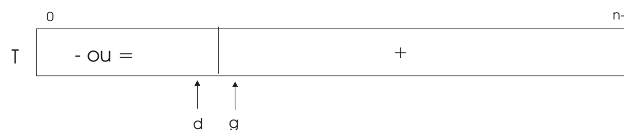
[3]
$k \leftarrow (g + d)$ **div** 2
si $T[k] \leq x$ **alors** $g \leftarrow k + 1$ **sinon** $d \leftarrow k - 1$

[4] Les initialisations $g \leftarrow 0$ et $d \leftarrow n - 1$ conviennent.

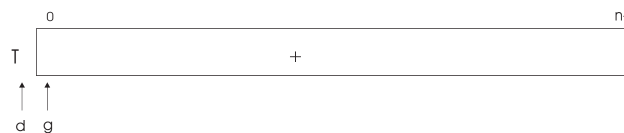
L'étape [3] conserve de façon évidente la situation choisie en [1]. D'autre part, à chaque étape, soit g augmente, soit d diminue, la condition d'arrêt sera toujours atteinte. On obtient :

$g \leftarrow 0$
$d \leftarrow n - 1$
tant que $g \leq d$ **faire**
  $k \leftarrow (g + d)$ **div** 2
  si $T[k] \leq x$ **alors** $g \leftarrow k + 1$ **sinon** $d \leftarrow k - 1$
si $d \geq 0$ et $T[d] = x$ **alors** " trouvé en d " **sinon** " pas trouvé "

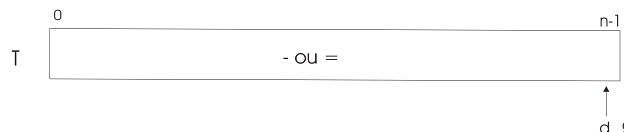
Lorsque la situation finale est atteinte, il suffit d'examiner les diverses possibités pour conclure.
Les deux indices d et g ont changé de valeur :



Seul d a changé de valeur :



Seul g a changé de valeur :

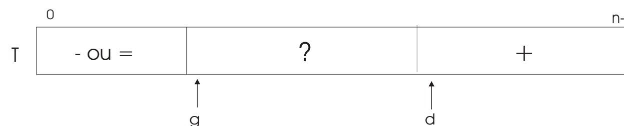


Attention, il ne faut pas oublier de tester si $d < 0$, car dans ce cas, accéder à $T[d]$ déclencherait une erreur.

1.4.2 Version b

Voici une autre situation, très voisine de la précédente, qui conduit à un algorithme sensiblement différent.

[1]



[2] C'est terminé lorsque $d = g$

[3]
$k \leftarrow (g + d)$ **div** 2
si $T[k] \leq x$ **alors** $g \leftarrow k + 1$ **sinon** $d \leftarrow k$
Cette étape conserve de façon évidente la situation choisie en [1]

[4] Les initialisations $g \leftarrow 0$ et $d \leftarrow n$ conviennent.

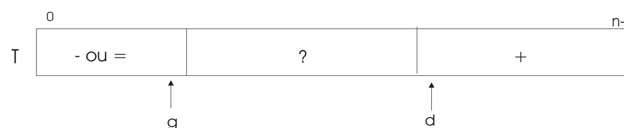
On obtient :

$g \leftarrow 0$
$d \leftarrow n$
tant que $g \neq d$ **faire**
  $k \leftarrow (g + d)$ **div** 2
  si $T[k] \leq x$ **alors** $g \leftarrow k + 1$ **sinon** $d \leftarrow k$
si $g > 0$ et $T[g - 1] = x$ **alors** " trouvé en g-1 "**sinon** " pas trouvé "

1.4.3 Version c

Voici une autre situation, très voisine des précédentes, qui conduit à un algorithme différent.

[1]



[2] C'est terminé lorsque $d = g + 1$

[3]
$k \leftarrow (g + d)$ **div** 2

si $T[k] \leq x$ **alors** $g \leftarrow k$ **sinon** $d \leftarrow k$
Cette étape conserve de façon évidente la situation choisie en [1] [4]Les initialisations $g \leftarrow -1$ et $d \leftarrow n$ conviennent.

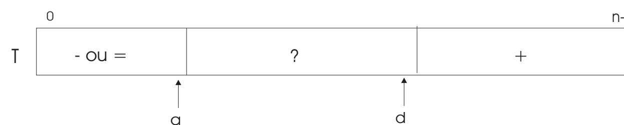
On obtient :

$g \leftarrow -1$
$d \leftarrow n$
tant que $g + 1 \neq d$ **faire**
  $k \leftarrow (g + d)$ **div** 2
  si $T[k] \leq x$ **alors** $g \leftarrow k$ **sinon** $d \leftarrow k$
si $g \geq 0$ et $T[g] = x$ **alors** " trouvé en g "**sinon** " pas trouvé "

1.4.4 Version d

Voici une autre situation, très voisine des précédentes, qui montre combien il est important de vérifier tous les points avant de conclure à la validité d'un algorithme.

[1]



[2] C'est terminé lorsque $g = d$

[3]
$k \leftarrow (g + d)$ **div** 2
si $T[k] \leq x$ **alors** $g \leftarrow k$ **sinon** $d \leftarrow k - 1$
Cette étape conserve de façon évidente la situation choisie en [1]

[4] Les initialisations $g \leftarrow -1$ et $d \leftarrow n - 1$ conviennent.

Tout semble parfaitement correct, et pourtant il y a une erreur. On n'est pas assuré que la condition d'arrêt soit atteinte dans tout les cas. En effet, dans le cas où d devient égal à $g+1$, $k \leftarrow (g+d)$ **div** 2 est égal à g et si le test $T[k] \leq x$ est satisfait l'instruction $g \leftarrow k$ ne permet pas à g d'augmenter et le programme boucle !

Pour être sûr de la terminaison, il faut écrire $g \geq d - 1$ pour condition d'arrêt. Lorsque cette condition d'arrêt est atteinte il faut examiner attentivement toutes les situations finales possibles avant de conclure. La conclusion est beaucoup plus

délicate à écrire.
Voici l'algorithme modifié :

$g \leftarrow -1$
$d \leftarrow n-1$
tant que $g < d-1$ **faire**
  $k \leftarrow (g+d)$ **div** 2
  si $T[k] \leq x$ **alors** $g \leftarrow k$ **sinon** $d \leftarrow k-1$
si $g \geq 0$ et $T[g]$ = x **alors** " trouvé en g"
sinon si $d \geq 0$ et $T[d]$ = x **alors** " trouvé en d" **sinon** "pas trouvé"

On peut aussi modifier le calcul de k pour s'assurer de l'arrêt de l'algorithme. En posant $k \leftarrow (g+d+1)$ **div** 2 et en supposant $g \neq d$, on a les inégalités :

$$g < k \leq d$$

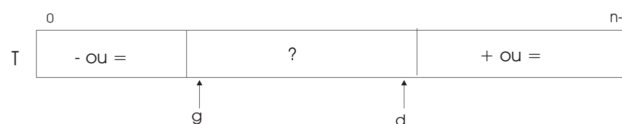
Lorsque l'étape 3 est exécutée soit $g \leftarrow k$, soit $d \leftarrow k-1$, dans tous les cas soit g augmente strictement, soit k diminue strictement. La condition d'arrêt $g = d$ sera donc toujours atteinte. On obtient l'algorithme :

$g \leftarrow -1$
$d \leftarrow n-1$
tant que $g \neq d$ **faire**
  $k \leftarrow (g+d+1)$ **div** 2
  si $T[k] \leq x$ **alors** $g \leftarrow k$ **sinon** $d \leftarrow k-1$
si $g \geq 0$ et $T[g]$ = x **alors** " trouvé en g" **sinon** "pas trouvé"

1.4.5 Version e

Voici encore une autre situation, qui conduit à l'écriture d'un algorithme que l'on trouve parfois dans les livres d'informatique.

[1]



[2] C'est terminé lorsque $g > d$

[3]
$k \leftarrow (g+d)$ **div** 2

si $T[k] \leq x$ **alors** $g \leftarrow k+1$
si $T[k] \geq x$ **alors** $d \leftarrow k-1$
Cette étape conserve de façon évidente la situation choisie en [1]

[4] Les initialisations $g \leftarrow 0$ et $d \leftarrow n-1$ conviennent.

On obtient :

$g \leftarrow 0$
$d \leftarrow n-1$
tant que $g \leq d$ **faire**
  $k \leftarrow (g+d)$ **div** 2
  si $T[k] \leq x$ **alors** $g \leftarrow k+1$
  si $T[k] \geq x$ **alors** $d \leftarrow k-1$
si $g = d+2$ **alors** "trouvé en k" **sinon** "pas trouvé"

1.4.6 Version f

Voici une dernière situation qui, comme dans la version précédente, conduit à interrompre l'itération dans le cas où la dichotomie "tombe pile" sur l'élément recherché.

[1]

Trouve
ou

T	-	?	+

0 … n-1 ; g, d

[2] C'est terminé lorsque $g > d$ ou $trouve$

[3]
$k \leftarrow (g+d)$ **div** 2
si $T[k] = x$ **alors** $trouve \leftarrow vrai$
sinon si $T[k] < x$ **alors** $g \leftarrow k+1$ **sinon** $d \leftarrow k-1$
Cette étape conserve de façon évidente la situation choisie en [1]

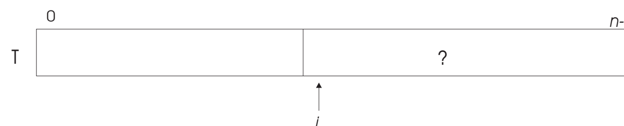
[4] Les initialisations $g \leftarrow 0$, $d \leftarrow n-1$ et $trouve \leftarrow faux$ conviennent.
On obtient :

$g \leftarrow 0$
$d \leftarrow n - 1$
$trouve \leftarrow faux$
tant que $g \leq d$ **et non** $trouve$ **faire**
  $k \leftarrow (g + d)$ **div** 2
  si $T[k] = x$ **alors** $trouve \leftarrow vrai$
  sinon si $T[k] < x$ **alors** $g \leftarrow k + 1$ **sinon** $d \leftarrow k - 1$
si $trouve$ **alors** "trouvé en k" **sinon** " pas trouvé"

1.5 Recherche d'un plus petit élément dans un tableau

Soit $T[0..n-1]$ un tableau de n éléments. On cherche un algorithme qui après avoir parcouru le tableau T une seule fois permet de connaître la valeur du plus petit élément de ce tableau. On peut imaginer comme situation générale celle où l'algorithme a examiné les i premières "cases" et repéré parmi celles-ci la valeur min du plus petit élément. Ce qui conduit à l'algorithme suivant.

[1]



et $\{\forall j, j \in [0, i[\Rightarrow min \leq T[j]\}$

[2] C'est terminé lorsque $i = n$

[3]
si $T[i] \leq min$ **alors** $min \leftarrow T[i]$
$i \leftarrow i + 1$

[4] Les initialisations $min \leftarrow T[0]$ et $i \leftarrow 1$ conviennent.

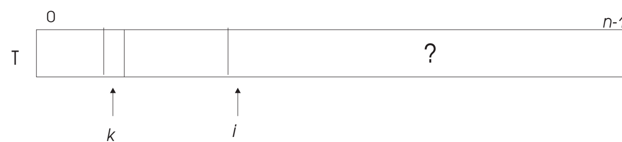
On obtient :

$min \leftarrow T[0]$
$i \leftarrow 1$
tant que $i \neq n$ **faire**
  si $T[i] \leq min$ **alors**
    $min \leftarrow T[i]$
  $i \leftarrow i + 1$

1.6 Recherche de l'emplacement du plus petit élément dans un tableau

Soit $T[0..n-1]$ un tableau de n éléments. On cherche un algorithme qui après avoir parcouru le tableau T une seule fois permet de connaître l'indice de la "case" où se trouve un (il peut y en avoir plusieurs) plus petit élément de ce tableau. On peut imaginer comme situation générale celle où l'algorithme a examiné les i premières "cases" et repéré parmi celles-ci l'indice k de l'endroit où se trouve le plus petit élément. Ce qui conduit à l'algorithme suivant.

[1]



et $\{\forall j, j \in [0, i[\Rightarrow T[k] \leq T[j]\}$

[2] C'est terminé lorsque $i = n$

[3]

si $T[i] \leq T[k]$ **alors** $k \leftarrow i$
$i \leftarrow i + 1$

[4] Les initialisations $k \leftarrow 0$ et $i \leftarrow 1$ conviennent.

On obtient :

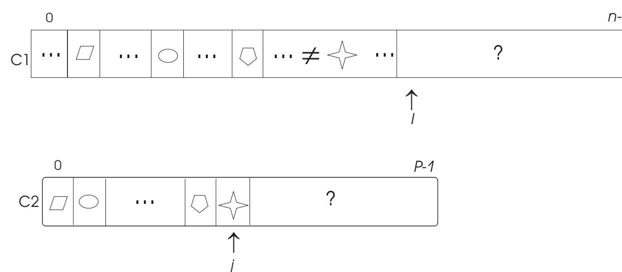
$k \leftarrow 0$
$i \leftarrow 1$
tant que $i \neq n$ **faire**
  si $T[i] \leq T[k]$ **alors**
    $k \leftarrow i$
  $i \leftarrow i + 1$

1.7 Recherche dans un tableau $C1$ d'une sous suite extraite égale à un tableau $C2$ donné

Soit $C1[0..n-1]$ un tableau de n éléments et $C2[0..p-1]$ un tableau de p éléments. On cherche un algorithme qui répond $vrai$ si tous les éléments du tableau $C2$ apparaissent dans le tableau $C1$ et ce dans le même ordre d'apparition, $faux$ sinon. Ainsi, deux éléments contigus dans le tableau $C2$ ne sont pas forcément contigus dans le tableau $C1$. Les tableaux $C1$ et $C2$ sont quelconques et peuvent

contenir des nombres, des caractères, ou tout autre chose. L'important est que l'on puisse tester l'égalité de deux objets entre eux.
On peut imaginer la situation suivante : l'algorithme a commencé sa recherche, a déjà trouvé les j premiers objets du tableau $C2$ dans le tableau $C1$, a commencé à rechercher $C2[j]$ (matérialisé par une étoile) dans le tableau $C1$ et ne l'a pas trouvé dans les cases qui se trouvent entre l'emplacement du dernier objet du tableau $C1$ égal à $C2[j-1]$ et $i-1$.
Ce qui nous conduit à établir la situation générale suivante :

[1]



[2] C'est terminé lorsque $i = n$ ou $j = p$

[3]
si $C2[j] = C1[i]$ **alors** $j \leftarrow j + 1$
$i \leftarrow i + 1$

[4] Les initialisations $i \leftarrow 0$ et $j \leftarrow 0$ conviennent.

On obtient :

$i \leftarrow 0$
$j \leftarrow 0$
tant que $i \neq n$ **et** $j \neq p$ **faire**
  si $C2[j] = C1[i]$ **alors**
    $j \leftarrow j + 1$
  $i \leftarrow i + 1$
si $j = p$ **alors Résultat Vrai sinon Résultat Faux**

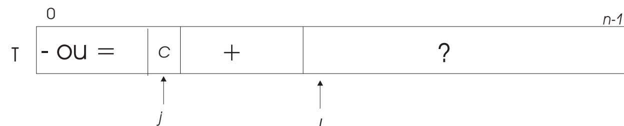
1.8 Partition dans un tableau

Soit $T[0..n-1]$ un tableau de n éléments et soit c l'élément $T[0]$. On cherche un algorithme qui par des échanges successifs, permette de placer cet élément de manière telle que tous les éléments qui lui sont inférieurs ou égaux soient placés

à sa gauche et les autres à sa droite. Cet élément de valeur c serait donc à sa place si l'on classait le tableau par ordre croissant. On impose de n'examiner qu'**une seule fois** les éléments et de ne pas utiliser de tableau auxiliaire. Le tableau T doit rester globalement invariant (i.e. aucun élément ne disparaît et aucun nouveau ne s'introduit), mais on n'exige pas que les deux parties de part et d'autre soient classées. On peut imaginer la situation ci-dessous, où l'étoile matérialise l'élément c qui sert à réaliser la partition. La zone des éléments qui lui sont inférieurs ou égaux est indiquée par **- ou =** et celle des éléments strictement supérieurs par **+**.

1.8.1 Version a

[1]



[2] C'est terminé lorsque $i = n$

[3]
si $T[i] > c$ **alors** $i \leftarrow i + 1$
sinon
échanger $T[j + 1]$ et $T[i]$
échanger $T[j + 1]$ et $T[j]$
$j \leftarrow j + 1$
$i \leftarrow i + 1$

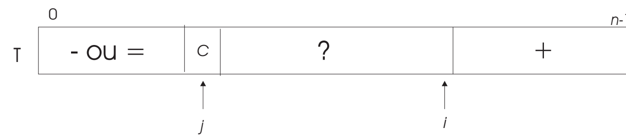
[4] Les initialisations $c \leftarrow T[0]$, $j \leftarrow 0$ et $i \leftarrow 1$ conviennent.

On obtient :

$c \leftarrow T[0]$
$j \leftarrow 0$
$i \leftarrow 1$
tant que $i \neq n$ **faire**
  si $T[i] \leq c$ **alors**
    échanger $T[j + 1]$ et $T[i]$
    échanger $T[j + 1]$ et $T[j]$
    $j \leftarrow j + 1$
  $i \leftarrow i + 1$

1.8.2 Version b

[1]



[2] C'est terminé lorsque $i = j$

[3]
si $T[j+1] > c$ **alors**
échanger $T[j+1]$ et $T[i]$
$i \leftarrow i - 1$
sinon
échanger $T[j+1]$ et $T[j]$
$j \leftarrow j + 1$

[4] Les initialisations $c \leftarrow T[0]$, $j \leftarrow 0$ et $i \leftarrow n - 1$ conviennent.

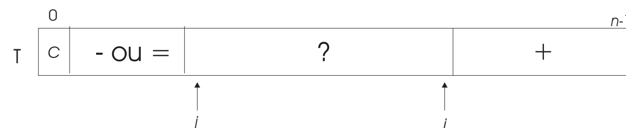
On obtient :

$c \leftarrow T[0]$
$j \leftarrow 0$
$i \leftarrow n - 1$
tant que $i \neq j$ **faire**
  si $T[j+1] > c$ **alors**
    échanger $T[j+1]$ et $T[i]$
    $i \leftarrow i - 1$
  sinon
    échanger $T[j+1]$ et $T[j]$
    $j \leftarrow j + 1$

1.8.3 Version c

On choisit de ne pas propager l'élément $T[0]$, on ne le mettra à sa place qu'une fois la partition réalisée.

[1]



[2] C'est terminé lorsque $i < j$

[3]
si $T[j] > c$ **alors**
échanger $T[j]$ et $T[i]$
$i \leftarrow i - 1$
sinon
$j \leftarrow j + 1$

[4] Les initialisations $c \leftarrow T[0]$, $j \leftarrow 1$ et $i \leftarrow n - 1$ conviennent.

On obtient :

$c \leftarrow T[0]$
$j \leftarrow 1$
$i \leftarrow n - 1$
tant que $i \geq j$ **faire**
  si $T[j] > c$ **alors**
    échanger $T[j]$ et $T[i]$
    $i \leftarrow i - 1$
  sinon
    $j \leftarrow j + 1$
$T[0] \leftarrow T[i]$
$T[i] \leftarrow c$

1.9 Recherche du $k^{\text{ème}}$ plus petit élément dans un tableau

On peut réordonner partiellement le tableau T en modifiant le tri par sélection pour mettre à leur place les k plus petits éléments. Le résultat sera l'élément $T[k-1]$ (le tableau étant indexé à partir de 0). On obtient un algorithme en $O(kn)$. Remarquons que cet algorithme nous fournit l'élément médian d'une suite de nombre en $O(n^2)$.

On peut répondre de manière plus performante au problème posé en utilisant l'algorithme de partition dans un tableau.

Pour cela, on modifiera l'algorithme qui effectue la partition du tableau par rapport au premier élément en le généralisant. Plus précisément au lieu de partitionner le tableau T entre les indices 0 et $n-1$ par rapport à $T[0]$, on écrira un algorithme qui partitionne une partie du tableau T entre les indices g(gauche) et d(droite) par rapport à $T[g]$. Pour rendre plus lisible l'algorithme on suppose que l'appel de la fonction $partition(T, g, d)$ renverra pour résultat l'emplacement p

de l'élément $T[g]$ qui a servi à effectuer la partition. Cet élément est donc "à sa place" si l'on ne considère que la portion du tableau T entre les indices g et d.
L'idée est la suivante :
On effectue une première partition du tableau T entre les indices 0 et $n-1$, si la position p renvoyée est égale à $k-1$, c'est terminé, sinon on demande à nouveau une partition, soit sur la partie gauche entre les indices 0 et $p-1$, soit sur la partie droite entre les indices $p+1$ et $n-1$ suivant que $p > k-1$ ou que $p < k-1$.
On réitère ce processus tant que le résultat de la partition est différent de $k-1$.
Ce qui nous conduit à écrire l'algorithme suivant :

$g \leftarrow 0$
$d \leftarrow n-1$
$p \leftarrow partition(T, g, d)$
tant que $p \neq k-1$ **faire**
  si $p > k-1$ **alors**
    $p \leftarrow partition(T, g, p-1)$
  sinon
    $p \leftarrow partition(T, p+1, d)$
résultat $T[k-1]$

En quoi cet algorithme est-il plus performant que le précédent qui utilisait le tri par sélection ?
Pour avoir une idée de son comportement lorsque n est très grand, supposons n égal à 2^q et que chaque appel à la fonction partition divise par 2 l'espace de recherche.
On effectuera ainsi n comparaisons lors du premier appel puis $2^{q-1} + 2^{q-2} + ... + 2 + 1$ comparaisons (dans le pire des cas) par la suite, soit $2^q - 1$. Ainsi l'efficacité de ce second algorithme est en $O(2n)$.