INFORMATIQUE
Programmation 2 **Cours 2**

F.DIDIER

Elaboration d'algorithmes itératifs

I) Introduction.

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 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 à mon avis, comme on va le voir, de mieux expliciter et de mieux contrôler la construction d'un algorithme.
Il faut ajouter que la manière de procéder présente une forte analogie avec le raisonnement par récurrence, raisonnement avec lequel l'étudiant est en principe familiarisé depuis le lycée.
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.

II) 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]
ftq
```

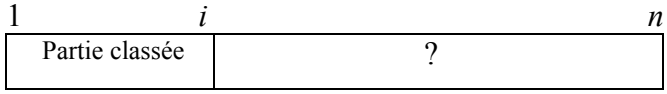
Cette façon de procéder montre comment on prouve la validité de l'algorithme au fur et à mesure de son élaboration.
En effet la situation générale choisie en [1] est en fait l'invariant qui caractérise la boucle *tantque*.
Cette situation est satisfaite au départ à cause de l'étape [4], elle reste vraie 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.

III) Exemples.

1) 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[1..i] est classé.
La notation T[1..i] désigne les i premières composantes du tableau T.

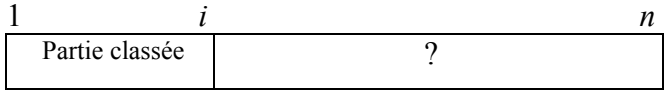
Pour illustrer cette situation on peut soit faire un dessin:



Soit procéder, plus formellement, en décrivant la situation par une formule logique :

$$\{\forall j, j \in [1, i[\Rightarrow T[j] \leq T[j+1]\}$$

[1]



[2] L'algorithme a terminé lorsque *i*=*n*.

[3] Pour se rapprocher de la fin, on incrémente *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 1 et *i* Pour cela «on doit amener T[*i*] à sa place dans T[1..*i*]». Ceci est un autre problème qui sera résolu en utilisant la même démarche.

[4] T[1..1] étant trié, l'initialisation *i*:=1 convient.

Cette première analyse nous conduit à écrire la séquence suivante:

```
i:=1
tant que non i=n faire
    i:=i+1
   «amener T[i] à sa place dans T[1..i]»
ftq
```

L'invariant:

$$\{\forall j, j \in [1, i[\Rightarrow T[j] \leq T[j+1]\}$$

joint à la condition d'arrêt i=n s'écrit à la sortie de la boucle:

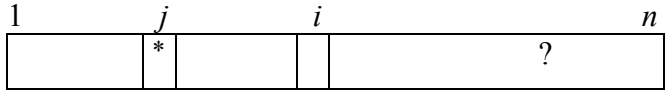
$$\{\forall j, j \in [1..n[\Rightarrow T[j] \leq T[j+1]\}$$

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 1 et chaque fois que l'étape [3] est exécutée *i* est incrémenté de 1, la condition d'arrêt *i*=*n* sera forcément vérifiée au bout de *n* étapes.

Toute réalisation de «amener T[*i*] à sa place dans T[1..*i*]» nous donne un algorithme de tri.

Par hypothèse de récurrence on sait que le tableau est classé entre 1 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[1..*j*-1] et T[*j*..*i*] classés.

[2] C'est terminé lorsque *j*=1 ou T[*j*-1]≤T[*j*].

[3] Echanger T[*j*-1] et T[*j*]
j:=*j*-1

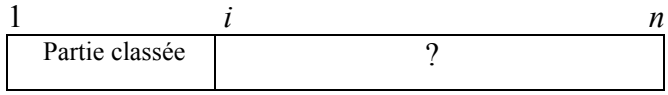
[4] L'initialisation *j*:=*i* satisfait la situation choisie en [1].

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

```
i:=1
tant que i≠n faire
    i:=i+1
    j:=i
    tant que j≠1 et T[j-1]>T[j] faire
        Echanger T[j-1] et T[j]
        J:=j-1
    ftq
ftq
```

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

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



La nuance réside dans le fait que la partie triée est T[1..*i*-1], et non T[1..*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[1..*i*-1], puis vient l'incrémentation de *i*, l'initialisation devient *i*:=2.
Comme on peut le constater la situation choisie guide totalement l'écriture de l'algorithme.

2) Tri d'un tableau par sélection.

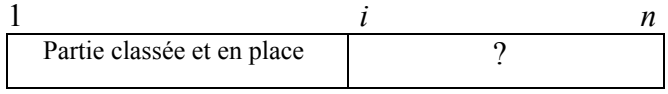
Comme il a déjà été dit et constaté la situation générale n'est pas unique. Par exemple, toujours pour le même problème, on peut faire une hypothèse plus forte sur la partie classée en ajoutant qu'elle est aussi définitivement en place.

Ce qui se traduit formellement par une conjonction de formules logiques:

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

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.

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

[4] L'initialisation i:=1 convient.

Ce qui nous conduit à écrire l'algorithme suivant:

```
i:=1
tant que i≠n faire
  «rechercher l'indice k du plus petit élmt de T[i..n]»
  échanger T[i] et T[k]
  i:=i+1
ftq
```

Toute réalisation de «rechercher l'indice k du plus petit élément de T[i..n]» 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 1 et est incrémentée chaque fois que l'étape 3 est exécutée, après *n* itérations la condition d'arrêt *i*=n est atteinte.
A la sortie de l'itération *i* vaut donc *n*. Ainsi, en remplaçant *i* par *n* 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:=1 satisfait-elle la situation choisie au départ?
On peut justifier ce choix soit intuitivement, soit formellement.
Intuitivement:
Lorsque i vaut 1, T[1..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 ces éléments sont classés et en place.
Formellement:
A ce stade il est nécessaire de faire quelques rappels:

a) Une implication P⇒Q est vraie si P est fausse.

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

Examinons maintenant la formule logique décrivant la situation générale.

$$\{\forall j, j \in [1..i-1[\Rightarrow T[j] \le T[j+1]\} \text{et } \{\forall k, k \in [i..n] \Rightarrow T[i-1] \le T[k]\}$$

Il faut qu'au départ (i vaut 1) cette formule soit vraie. Pour qu'une conjonction de deux formules soit vraie il faut que chacune des deux formules soit vraie.

Il est facile de constater que la première formule est vraie lorsque i vaut 1. En effet cette formule est de la forme P⇒Q, où P est de la forme $\forall x, x \in \phi$, car lorsque *i* vaut 1 l'intervalle semi-ouvert [1..*i*-1[possède zéro élément.
La seconde formule conduit à considérer T[0] qui n'existe pas, on ne peut rien démontrer!.
Le problème provient du fait que la condition T[1..*i*-1] en place est mal traduite par la seconde formule, car cette formule n'a pas de sens lorsque *i* est égal à 1.

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

Pour traduire le fait que T[1..*i*-1] sont en place, on doit écrire :
et là encore les rappels a) et b) nous permettent de conclure.

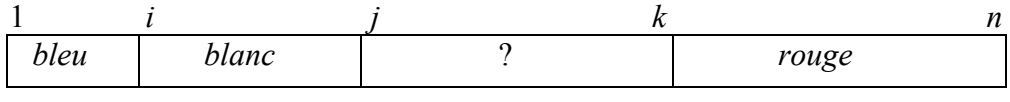
3) Le drapeau tricolore : Une illustration du danger de raisonner sur un dessin.

C'est le célèbre problème du à E.W.DIJKSTRA qui fut un des premiers informaticiens à s'intéresser aux preuves de programmes.
Dans sa version originale, on remplit aléatoirement un tableau T[1..n] avec les couleurs *bleu*, *blanc* et *rouge*. Le problème consiste à trouver un algorithme qui, par des échanges successifs de composantes, reconstitue le drapeau hollandais. L'algorithme ne doit pas utiliser de tableau intermédiaire et ne parcourir le tableau T qu'une seule fois.

Remarque:
Lorsqu'on pose cet exercice à des étudiants, il est préférable de modifier cet énoncé de manière à rendre discernables les composantes qui présentent le même critère, ceci pour éviter d'avoir à préciser constamment les règles du jeu. On peut, par exemple, remplacer les couleurs par des nombres réels aléatoires et prendre comme critère de regroupement : Négatif, nul, positif.
Cette modification évite d'avoir comme proposition, des solutions du style :
on prend trois compteurs, puis on parcourt le tableau une seule fois en examinant chacune des composantes pour incrémenter le bon compteur, puis on remplit à nouveau le tableau T avec autant de composantes de chaque couleur que l'indique son compteur…

Parmi toutes les situations générales envisageables (on peut disposer de différentes façons les zones ainsi que les frontières qui les délimitent), en voici une qui conduit à un algorithme simple à mettre en œuvre :

[1]



[2]

C'est terminé lorsque *k*=*j*-1.

[3]
Suivant la couleur de la composantes *j*, on effectue de manière exclusive l'un de ces trois groupes d'instructions :

- Si T[*j*] est blanc alors *j*:=*j*+1;
- Si T[*j*] est rouge alors échanger T[*j*] et T[*k*];
  k:=*k*-1;
- Si T[*j*] est bleu alors échanger T[*i*] et T[*j*] ;
  i:=*i*+1;
  j:=*j*+1;

Cette étape conserve l'invariant choisi en [1] et assure la terminaison de l'algorithme (soit *j* augmente, soit *k* diminue…).

[4]
Les initialisations *i*:=1; *j*:=1 et *k*:=*n* satisfont l'hypothèse de récurrence.

On peut, pour mieux s'en convaincre, décrire formellement la situation choisie et constater que la formule est vraie pour ces valeurs initiales.

$$\{\forall u, u \in [1, i[\Rightarrow T[u] \text{ bleu}\} \text{et } \{\forall u, u \in [i, j[\Rightarrow T[u] \text{ blanc}\} \text{et } \{\forall u, u \in]k, n] \Rightarrow T[u] \text{ rouge}\}$$

On obtient :

```
I:=1;j:=1;k:=n;
Tant que j≤k faire
    Cas T[j] dans
    Blanc :j:=j+1;
    Rouge :début
              Echanger T[j] et T[k];
              k:=k-1
           fin
     bleu :début
              échanger T[i] et T[j] ;
              i:=i+1;
              j:=j+1
           fin
  fincas
ftq
```

Examinons une autre situation générale qui montre qu'il est possible d'écrire une étape [3] fausse lorsqu'on raisonne sur dessin qui représente une «situation trop générale» !

[1]

T	1	*i*	*j*	*k*	*n*
	bleu	*blanc*	*Rouge*	?	

[2]
C'est terminé lorsque *k* = *n*+1.

[3]
Suivant la couleur de la composantes *k*, on effectue de manière exclusive l'un de ces trois groupes d'instructions :

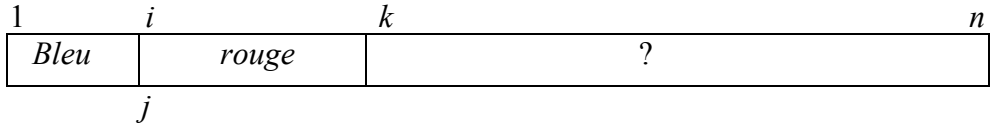
- Si T[*k*] est rouge alors *k*:=*k*+1;
- Si T[*k*] est blanc alors échanger T[*j*] et T[*k*];
  j:=*j*+1;
  k:=*k*+1;

- Si T[*k*] est bleu alors échanger T[*k*] et T[*i*];
  échanger T[*j*] et T[*k*];
  i:=*i*+1;
  j:=*j*+1;
  k:=*k*+1;

Cette étape semble conserver l'invariant et assure de manière évidente la terminaison de l'algorithme, car quel que soit la possibilité envisagée, *k* est toujours incrémenté.

[4]
Les initialisations i:=1; j:=1; k:=1 conviennent.

L'étape [3], telle qu'elle est écrite, ne conserve pas toujours l'invariant. En effet il est tout à fait possible qu'à un moment donné, on n'ait pas encore rencontré de boule blanche (il se peut aussi que le tableau n'en contienne aucune), on est dans la situation «particulière» suivante:



Dans cette situation i est égal à j. Examinons comment agit l'étape [3] dans le cas où la composante k est de couleur bleue.
Les deux échanges laissent la composante *k* (bleue) à sa place !. La situation générale choisie n'est plus conservée.
Cet exemple montre qu'il faut être prudent lorsqu'on raisonne avec un dessin. On peut malgré tout s'en tirer, sans changer la situation choisie, en modifiant l'étape [3] précédente de la façon suivante:

- Si T[k] est bleu alors échanger T[k] et T[j];
  échanger T[j] et T[i];
  i:=i+1;
  j:=j+1;
  k:=k+1;

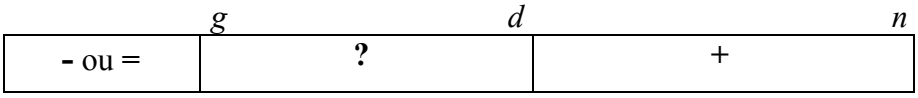
4) <u>La recherche dichotomique : Un exemple où le problème de l'arrêt peut s'avérer délicat</u>.

Soit T[1..n] 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 mettre en œuvre 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 suffisent !
Voilà un algorithme très important que très peu d'étudiants de licence ou de maîtrise sont capables d'écrire sans aucune erreur.

Parmi toutes les situations générales imaginables (il en existe un grand nombre), en voici une qui conduit à une version assez classique.

[1]

T



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:=(g + d) div 2
si T[k] ≤X alors g:=k+1 sinon d:=k-1

Cette étape conserve de façon évidente la situation choisie en [1]. D'autre part soit g augmente, soit d diminue, la condition d'arrêt sera toujours atteinte.

[4]

Les initialisations g:=1 et d:=n conviennent.

On obtient :

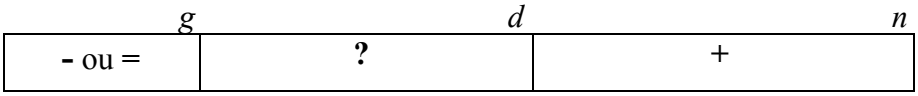
```
g:=1; d:=n;
tant que d≥g faire
     k:=(g+d) div 2 ;
     si T[k] ≤ X alors g:=k+1 sinon d:=k-1
ftq
Si d<1 ou T[d] ≠ X alors « pas trouvé »
                   sinon « trouvé en d »
```

Attention, il ne faut pas oublier de tester si d<1, car dans ce cas, parler de T[d] déclencherait une erreur.

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

[1]

T



Le seul changement est la position de g, qui s'est décalé à gauche du trait vertical.

[2]

C'est terminé lorsque $d=g$.

[3]

k:=(g + d) div 2
si T[k] ≤X alors g:=k sinon d:=k-1

Cette étape conserve de façon évidente la situation choisie en [1].

[4]
Les initialisations g:=0 et d:=n conviennent.

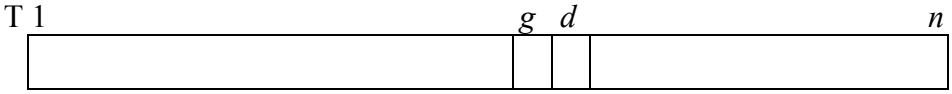
Tout semble parfaitement correct, et pourtant il y a une erreur. On n'est pas assuré que la condition d'arrêt soit atteinte dans tout les cas.

En effet, lorsque g≠d, on a les inégalités :

$$g \leq (g+d) \text{ div } 2 < d$$

Le point important est l'inégalité au sens large, l'égalité a lieu lorsque g est égal à d-1.

De la façon dont l'étape [3] est écrite, on risque de boucler indéfiniment. Il suffit de prendre un exemple pour s'en convaincre.



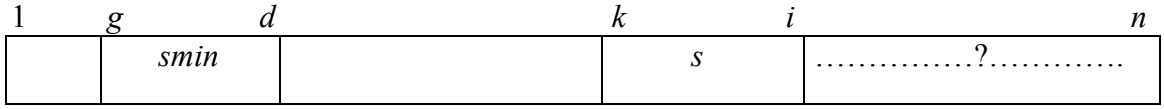
Ici on a k=(g + d) div 2=g, si T[g] est inférieur ou égal à l'élément X que l'on recherche, cet algorithme bouclera indéfiniment car g et d ne changerons plus de valeurs.

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.

5) Tranche de somme minimale: un exemple où la méthode permet d'expliquer un algorithme.

Une tranche d'un tableau T[1..n] est une suite de composantes consécutives de ce tableau T.
Lorsque le tableau T est remplit de nombres quelconques (il peut y avoir des nombres négatifs) on peut associer à chaque tranche la somme algébrique des éléments qui la composent.
Le problème est de déterminer une (il peut y en est plusieurs) tranche de somme minimale.
Toute la difficulté de ce problème provient du fait suivant:
On recherche un algorithme qui répond au problème en n'examinant les éléments du tableau T qu'une seule fois. On cherche en fait un algorithme en O(n).
Quand on y réfléchit pour la première fois cela semble impossible. Ce problème n'est pas du tout évident, je l'ai posé à de nombreux étudiants et enseignants dans des contextes très différents (stages, oraux de concours, TD..) à ce jour un seul a su le traiter entièrement sans aucune aide. Voici la solution :

[1]



La situation décrite par le schéma est la suivante : l'algorithme a commencé à examiner toute les composantes entre 1 et i, il a trouvé que la tranche de somme minimale commence en g, se

termine en *d* et a pour somme *smin*. Par ailleurs, parmi toutes les tranches qui s'achèvent en *i*, il en existe une de somme minimale, elle commence en *k* et a pour somme *s*.

[2]
C'est terminé lorsque *i*=*n*.

[3]
i:=*i*+1
{ *** on met à jour la tranche de somme minimale qui se termine en *i* *** }
si *s* + T[*i*] < T[*i*] alors *s*:= *s* + T[*i*]
sinon *s*:= T[*i*]
k:= *i*
{ *** on met éventuellement à jour la tranche de somme minimale *** }
si *s* < *smin* alors *smin*:= *s*
g:= *k*
d:= *i*

[4]
Les initialisations suivantes satisfont la situation générale choisie.
i:=1
g:=1
d:=1
k:=1
smin:=T[1]
s:= T[1]

On obtient l'algorithme :

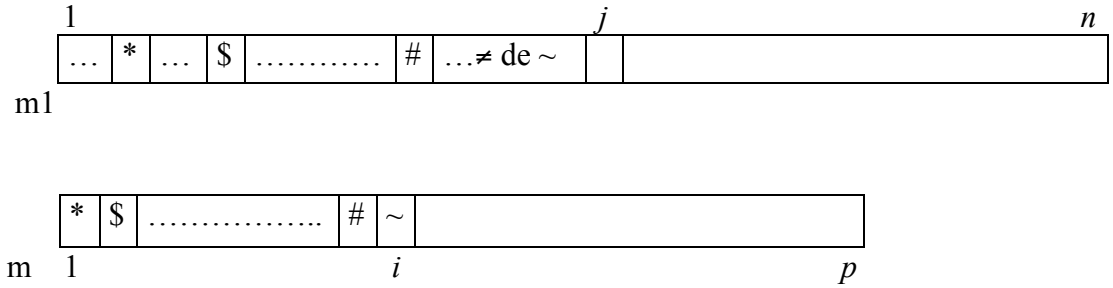
```
i:=1;
g:=1;
d:=1;
k:=1;
smin:=T[1];
s:=T[1];
tant que i≠n faire
  i:=i+1;
  si s+T[i]<T[i] alors s:=s+T[i]
                 sinon s:=T[i];
                       k:=i;

  si s<smin alors smin:=s;
                  g:=d;
                  d:=i;
ftq
```

6) Un dernier exemple pour convaincre.

Ce dernier exemple consiste à trouver un algorithme permettant de déterminer si les lettres d'un mot *m* se trouvent dans le mot *m1* dans le même ordre. C'est le cas, par exemple, si *m* est le mot *arme* et *m1* le mot *algorithme*.

[1]



Commentons brièvement le schéma ci-dessus qui décrit la situation choisie.
Les *i*-1 premières lettres du mot *m* ont été trouvées dans le mot *m*1, les correspondances ont été matérialisées par des caractères particuliers. On a commencé à rechercher la ième lettre (~) de *m* à partir de la composante de *m1* qui suit le # et on ne l'a pas trouvée parmi toutes les composantes jusqu'à *j*-1 (la comparaison avec la lettre *m*1[*j*] n'a pas encore eu lieu).

[2]
C'est terminé lorsque *j*>*n* ou *i*>*p*.

Si on termine avec *j*>*n et i*≤*p* on a alors un échec: la ième lettre n'a pas été trouvée.
Si on termine avec *i*>*p*, on a alors un succès: toutes les lettres de *m* ont été trouvées.

[3]
Si *m*[i]=*m*1[*j*] alors *i*:=*i*+1
j:=*j*+1

[4] Les initialisations *i*:=1 et *j*:=1 conviennent.

On obtient l'algorithme:

```
i:=1;j:=1;
Tant que i≤p et j≤n faire
  si m[i]=m1[j] alors i:=i+1;
  j:=j+1
ftq;
si i>p alors «succés» sinon «echec» ;
```