Didactique de l'Informatique

Au ras des pâquerettes

Guy Chaty
IUT de Villetaneuse

1. Introduction

Il est des circonstances où l'on doit initier à l'algorithmique un public hétérogène composé en partie de personnes ayant une culture mathématique et informatique imprécise.

Pour faciliter leur compréhension, on est poussé parfois dans les derniers retranchements de tentatives d'explication simple. On finit par condenser des analyses en des formulations qui s'apparentent plus à des proverbes dictés par un certain bon sens qu'à des processus mathématiques. Et on s'aperçoit que cela ne manque pas d'intérêt d'autant plus qu'on peut établir un lien étroit entre ces formulations et la preuve mathématique des algorithmes concernés, et que rien ne nous empêche d'exposer celle-ci dans toute sa rigueur. Au bout du compte, on a aidé à la compréhension simultanée du processus algorithmique et du support mathématique.

De plus, ces formulations mettent en évidence ce qu'il peut y avoir de commun dans plusieurs algorithmes et préparent ainsi le regroupement en catégories de certains algorithmes d'où une économie de pensée et une aide à la mémorisation et à la recherche des algorithmes. Elles débouchent d'ailleurs parfois sur les schémas d'algorithmes [2] dont l'étude n'est pas envisagée ici.

Nous donnons ici quelques exemples de ces formulations. Les algorithmes étudiés sont des algorithmes élémentaires de base [1].

Nous utilisons un pseudo-langage d'expression des algorithmes possédant les primitives classiques : opérations arithmétiques, affectation, les primitives de choix : *si alors, si alors sinon*, les primitives d'itération : *tant que, répéter jusqu'à, pour*.

2. "Si c'est mieux, on le prend"

Considérons le problème de la recherche du plus petit élément, ou du plus grand, dans une suite (u_i) de n éléments d'un ensemble A. On suppose que A est totalement ordonné par une relation notée $\leqslant$.

Pour le résoudre avec les primitives classiques, l'idée est de définir une variable X initialisée à u_1 et de parcourir toute la suite du début à la fin, en ne modifiant à chaque étape la valeur de X que si l'élément courant est "meilleur" (plus petit ou plus grand) que cette valeur. L'essentiel de l'algorithme est donc (on a choisi l'option "plus petit") :

```
X := u1
tant que i<n faire
        i:=i+1
        si ui<X
                alors X:=ui
        fin si
fin tant que
```

L'idée à souligner, et qui n'est pas toujours vite et bien comprise, est que dans l'autre cas ($u_i \geqslant X$), on ne fait rien. D'où la formulation *"Si c'est mieux, on le prend"*.

La preuve mathématique de cet algorithme repose sur le raisonnement par récurrence suivant :

Si p(i) représente le minimum de l'ensemble des i premiers éléments de la suite, c'est-à-dire $p(i) = \min \{u_1, u_2, ..., u_i\}$,

alors $p(i + 1) = \min \{u_{i+1}, p(i)\}$.

Théorème facile à démontrer, mais demandant cependant quelque attention.

Le minimum d'une suite de 1 élément étant cet élément lui-même, on peut mettre en évidence une suite récurrente dont le dernier terme est la solution du problème.

```
p(1) = u1
p(2) = min(u2,p(1))
       ...
p(n - 1) = min(un-1,p(n - 2))
p(n) = min(un,p(n - 1))
```

Par conséquent, pour calculer le minimum p de la suite (u_i) de n éléments, il faut calculer les éléments successifs de la suite récurrente (p(i)). Ce que l'on peut exprimer par l'algorithme ci-dessus, la variable X recevant les valeurs p(i), i variant de 1 à n : la nouvelle valeur de X à l'étape i+1 est bien le minimum entre l'ancienne et u_i. L'algorithme ne fait que mettre en œuvre la suite (p(i)).

Notre formulation nous a quand même ouvert la voie !

3. "Tant que ce n'est pas fini, on avance"

Considérons les circonstances où l'on veut détecter ou insérer, dans une suite de n éléments d'un ensemble A, un élément de A ou une sous-suite d'éléments de A, vérifiant une condition.

Elles se présentent dans les problèmes suivants et peut donner lieu à différentes formulations.

a) ***"Tant qu'il y a de l'espoir, on avance"***.

- **Problème**. Rechercher un élément X dans une suite quelconque.

On avance dans la suite tant que X est différent de l'élément visité et qu'on n'est pas arrivé à la fin.

```
i:=1
tant que X≠ ui et i<n faire
      i:=i+1
fin tant que
trouvé:=X=ui
```

trouvé est une variable booléenne de valeur vraie ou fausse.

- **Problème.** Rechercher un élément X dans une suite triée, dans "l'ordre croissant" par exemple. On avance dans la suite tant que X est supérieur à l'élément visité et qu'on n'est pas arrivé à la fin.

```
i:=1
tant que X> ui et i<n faire
      i:=i+1
fin tant que
trouvé:=X=ui
```

Lorsque X est inférieur ou égal à u_i, soit il n'y a plus d'espoir dans le premier cas, soit on a trouvé X : dans les deux cas, on sort de la boucle.

b) ***"Tant que c'est bon, on avance"***

- **Problème.** Vérifier si une chaîne de caractères de longueur n représente un entier. On avance dans la suite tant que le caractère est un chiffre et qu'on n'est pas arrivé à la fin.

```
i:=1
tant que u_i est un chiffre et i≤n faire
      i:=i+1
fin tant que
chaîne-entier:=i=n+1
```

- **Problème.** Insérer un élément "à sa place" dans une suite triée dans "l'ordre croissant" par exemple.

On avance dans la suite tant que l'élément à insérer est supérieur à l'élément visité et qu'on n'est pas arrivé à la fin.

```
i:=1
tant que E> u_i et i≤n faire
      i:=i+1
fin tant que
```

A la sortie de la boucle, i donne la place cherchée.

- **Problème.** Trouver une monotonie (sous-suite maximale triée) dans une suite quelconque. On avance dans la suite tant que l'élément visité est inférieur à son suivant et qu'on n'est pas arrivé à la fin.

```
i:=1
tant que u_i≤u_(i+1) et i<n-1 faire
      i:=i+1
fin tant que
```

A la sortie de la boucle, i donne le rang du dernier terme de la monotonie.

Ces deux formulations *"Tant qu'il y a de l'espoir, on avance"* et *"Tant que c'est bon, on avance"* peuvent être rassemblées en une seule : *"Tant que ce n'est pas fini, on avance"*, le *"ce n'est pas fini"* correspondant à une condition attachée au problème qui est encore vérifiée, ou ne l'est plus, à l'étape considérée.

On tombe sur un schéma d'algorithme mais il n'est pas question ici d'aller du général au particulier : au contraire, la démarche est de faire prendre conscience intuitivement sur des cas précis du sens de l'algorithme

et de regrouper ensuite ce qui peut être regroupé. Notre propos est "au ras des pâquerettes" pour mieux préparer "l'envol".

4. "Tant que ce n'est pas fini, on calcule et on avance"

Dans les exemples précédents, la suite était donnée. On peut être amené à la construire. Il en est ainsi dans le calcul approché de Va par la méthode de Newton.

On calcule les éléments de la suite (x_i) :

$x_i = 1/2\ (x_{i-1} + a/x_{i-1})$ si $i>0$

$x_0 = a/2$

tant que $|x_i - x_{i-1}| / x_{i-1} > e$.

Ici, le "on avance" s'agrémente de calculs qui permettent de calculer le terme suivant de la suite.

```
algorithme racine
  objets A, E, X, Y
  début
  lire (A, E)
  X:=A/2
  Z:=1
  tant que Z > E faire
      Y:=(X+A/X)/2
      Z:=|Y-X| / X
      X:=Y
      fin tant que
  écrire (X)
```

Montrer la *finitude* de l'algorithme est chose aisée avec tous les exemples précédents. Cela peut être plus délicat avec l'application du proverbe *"Tant que ce n'est pas fini, on avance ou on recule"* !

Pour l'algorithme suivant, où n est un entier naturel, on démontre facilement que toute exécution a une fin :

```
début
lire (n)
tant que n ≠ 1 faire
            si n est pair
                  alors n := n/2
                  sinon n := n + 1
             fin si
      fin tant que
  fin
```

Par contre, pour le suivant, on ne sait pas encore le démontrer !

```
début
lire (n)
tant que n ≠ 1 faire
            si n est pair
                  alors n := n/2
                  sinon n := 3n + 1
            fin si
      fin tant que
fin
```

4. "Tant que ce n'est pas fini, on avance...dans une arborescence"

On se place ici dans le cas du traitement d'une arborescence. La formulation "Tant que ce n'est pas fini, on avance" apparaît par exemple dans le problème suivant.

Problème. Tri arborescent d'une suite

Construction de l'arborescence binaire A

Analyse. On construit la racine et on y associe le premier élément. D'une manière générale, tant que l'élément considéré U n'est pas placé, on avance dans l'arborescence déjà construite, à gauche ou à droite du sommet S suivant que U est plus petit ou plus grand que p(S), élément associé à S. Si la place est libre, on construit le nouveau sommet et on lui associe U.

FINI, variable booléenne initialisée à faux, prendra la valeur vrai quand U sera placé.

```
répéter
     si U < p(S) alors
          si libre à gauche
               alors
                    placer U à gauche
                    FINI:=vrai
               sinon
                    aller à gauche
          fin si
     sinon
          si libre à droite...
          fin si
fin si
jusquà FINI
```

5."A chaque étape, on agit s'il le faut"

Le proverbe *"Si c'est mieux, on le prend"* du §1 est déjà un exemple de *" A chaque étape, on agit s'il le faut"*.

Donnons d'autres exemples.

- **Problème.** Chercher les diviseurs d'un entier en est un :

```
début
lire(n)
répéter
     si n mod d = 0
     alors écrire "d est un diviseur de n"
     fin si
d:=d+1
jusqu'à d>n
```

- Dans le "tri à bulles", on utilise :

```
pour k de 1 à i-1 faire
     si u_k > u_k+1
          alors échanger (u_k , u_k+1)
     fin si
fin pour
```

6. "Tant qu'il y a eu modification, recommencer"

La formulation "Tant qu'il y a eu modification, recommencer" apparaît dans les cas où la fin d'itération a lieu lorsqu'au précédent passage dans la boucle, les instructions de traitement n'ont pas été exécutées.

Ces cas se présentent dans des problèmes très différents à première vue.

Problème : Tri à bulles

Analyse (extrait)

Pour sortir de la boucle "parcours et échanges éventuels", on teste s'il y a eu effectivement échange au dernier passage : on peut utiliser une variable booléenne E initialisée à faux dans la boucle, et qui prend la valeur vraie en cas d'échange.

```
E:=vrai
Tant que E faire
     E:=faux
     {parcours et échanges éventuels}
     {E prend la valeur vrai en cas d'échange}
fin tant que
```

Problème : Recherche des composantes fortement connexes (cfc) d'un graphe.
Méthode : marquer + les descendants et – les ascendants d'un sommet x. Les sommets marqués + et – constituent la cfc de x.

Pour le marquage, il faut faire un balayage des sommets non encore "pris".

Il est nécessaire, pour terminer, de faire un balayage "blanc" où aucun sommet n'est marqué.

On peut utiliser une variable booléenne M initialisée à faux dans la boucle et qui prend la valeur vraie en cas de marquage.

```
M:=vrai
Tant que M faire
     M:=faux
     {balayage}
     {M prend la valeur vrai en cas de marqua-
     ge}
fin tant que
```

7. "Pour comparer, sauvegarder".

a) *Les doublons*

Considérons le problème suivant.

Problème Lire un texte caractère par caractère (la fin est indiquée par *), compter le nombre de couples de lettres consécutives identiques (on suppose que le texte contient au moins un caractère différent de * et qu'une lettre ne se répète pas à la suite plus d'une fois), écrire ce nombre et la longueur du texte.

Analyse.
On lit le texte caractère par caractère (C) jusqu'à rencontrer le caractère *. En même temps, pour chaque caractère à partir du deuxième, on regarde s'il est identique au précédent (P). Celui-ci doit donc être conservé ! Si C=P , on incrémente un compteur (NC). Pour calculer la longueur, on utilise un autre compteur (I).

Algorithme.

```
algorithme LIRTEXT
début
NC := 0
lire (C)
I := 1
```

```
répéter
 P := C { la valeur de C est sauvegardée dans
         P}
 lire (C) { le nouveau caractère est placé
          dans C}
 si  C = P alors  NC := NC + 1
 fin si
 I := I + 1
jusque C = '*'
écrire ( NC, I)
fin
```

b) ***Tri d'un fichier en organisation séquentielle***

Un autre exemple de ce proverbe *"Pour comparer, sauvegarder"* est donné dans le tri d'un fichier séquentiel F, lorsqu'on répartit les monotonies de F sur deux fichiers FA et FB puis qu'on fusionne 2 à 2 les monotonies de FA et de FB sur le fichier F (qui ne sont pas nécessairement les monotonies obtenues dans la répartition). Pour détecter la fin de ces monotonies, on est amené à utiliser deux fenêtres pour chacun des fichiers FA et FB et à comparer leurs contenus.

8. "Là où l'on prend, là on avance"

Considérons le problème de la fusion de deux suites triées (a_i), $1 \leqslant i \leqslant n$, et (b_j), $1 \leqslant j \leqslant m$, en une suite triée (c_k) dont l'algorithme peut s'écrire (on a complété les suites par deux termes plus grands que les termes des deux suites pour éviter de vider une suite quand l'autre est terminée) :

```
début
i := 1; j:= 1; k := 1;
répéter
            Si  a_i <= b_j
                  alors      c_k:= u_i
                             i := i + 1
                  sinon      c_k := b_j
                             j:=j + 1
            Fin Si
            k := k + 1
jusqu'à FIN_SUITES
fin
```

FIN_SUITES est une variable booléenne égale à la conjonction : i = n + 1

et $j = m + 1$.

Pour étendre l'algorithme à la fusion de deux fichiers, on peut remplacer les incrémentations du type i:=i+1 par "avancer dans (u_i) " et exprimer la variable FIN_SUITES" en termes de fins de fichiers.

On a la formulation qui résume ces algorithmes : *"là où l'on prend, là on avance"*.

9. Conclusion

Nous avons montré sur quelques exemples qu'on pouvait aider à l'initiation aux algorithmes élémentaires de base en les exprimant sous forme de sentences proverbiales qui les résument.

Ceci permet un **regroupement des algorithmes** par familles, facilite la **compréhension** ainsi que la **mémorisation** de ces algorithmes, et n'empêche pas leur approfondissement mathématique.

De plus, pour certains étudiants, c'est une préparation à l'étude des schémas d'algorithmes. Enfin, les étudiants eux-mêmes peuvent se prendre au jeu et inventer à leur tour des proverbes, ce qui va dans le sens de l'objectif poursuivi et ne peut que réjouir le professeur.

Pour les élèves de l'enseignement secondaire et les personnes étrangères au monde informatique, ces formulations peuvent donner une idée des processus algorithmiques et constituer des bases d'une culture générale informatique.

Références.

[1] G.Chaty et J.Vicard, *Programmation*, Ellipses, Paris 1992.

[2] P-C. Scholl et J.-P. Peyrin, *Schémas algorithmiques fondamentaux*, Masson, Paris 1989.