ALGORITHMIQUE ET LANGAGES DE PROGRAMMATION

Jacques ARSAC(*)

1. DE L'EMPIRISME À LA PROGRAMMATION.

Dans les débuts de l'informatique, un cours de programmation consistait essentiellement en une description des organigrammes, comme représentation graphique de la structure d'un programme (tellement embrouillée qu'il fallait bien en passer par là), et la liste des instructions d'un langage : les constantes entières s'écrivent avec des chiffres sans point décimal ni partie exposant, un identificateur... l'instruction LIRE, etc. Pour le reste, l'apprenti programmeur devait se débrouiller tout seul. On lui demandait de se jeter à l'eau. et de survivre par ses propres moyens.

On juge un arbre à ses fruits. Ceux de la programmation empirique furent dénoncés pour la première fois de façon sérieuse au colloque de Monterey en 1972 sur « le coût élevé du logiciel » : il est toujours délivré en retard, plein d'erreurs, intransportable d'une machine à l'autre... Les statistiques de l'US Air Force donnaient un coût d'écriture de $75 par instruction (vous écrivez i := 0, et vous recevez 75 dollars), ce qui est négligeable devant le coût de mise au point : 4000 dollars par instruction. Pour créer un programme, on rédigeait quelque chose ressemblant à ce que l'on cherche, puis on faisait des essais pour le mettre au point. Dans ses « notes de programmation structurée » présentées pour la première fois à l'Université du Maryland en 1969, Edsger Dijkstra rappelle ce principe fondamental des sciences : « **Essayer un programme peut servir à montrer qu'il contient des erreurs, jamais qu'il est juste.** »

(*) Professeur à l'Université P. et M. Curie. Correspondant de l'Académie des Sciences. Chargé d'une mission d'inspection générale.

Le problème de la programmation n'est pas de connaître les mots pour dire un programme, mais d'avoir un programme à dire. Il fallait donc recentrer l'enseignement : non pas enseigner un langage, mais faire que les étudiants aient quelque chose à dire On utilise un ordinateur pour lui faire exécuter une tâche fastidieuse ou réaliser des calculs complexes. Il faut lui donner soit la marche à suivre pour cette tâche, soit la méthode de résolution de ce problème, choses que l'on regroupe sous le nom « **Algorithme** ». Or, ce n'est pas parce que nous sommes capables de faire une certaine tâche que nous pouvons dire la marche à suivre pour y arriver : la plupart des candidats du jeu « des chiffres et des lettres » de la télévision sont capables de trouver une combinaison d'au plus 6 plaques imposées pour obtenir un résultat lui aussi imposé, ils ne savent pas eux-mêmes par quelle méthode ils trouvent le résultat. Le « pifomètre » est leur façon habituelle. Le problème de la programmation est donc d'abord celui de la création d'un algorithme. « Nous n'enseignons plus un langage de programmation, nous faisons de l'algorithmique. »

Reste que ce mot est bien vague : les anglais le qualifieraient de « buzzword », ou, selon Bernard de Morlaix, *nomina nuda tenemus*. Pour beaucoup, « un algorithme est un ensemble de directives. Cet ensemble est structuré car l'ordre dans lequel les actions doivent être exécutées est fondamental » (J.P. Laurent, [LAU82]). Certains expliquent que la notion d'algorithme est banale, car nous en pratiquons souvent dans la vie courante : pour faire cuire un oeuf à la coque, pour changer une roue de voiture... J'ai même vu dans un ouvrage américain les partitions de musique présentées comme exemple d'algorithmes : elles décrivent la suite des actions à faire pour obtenir l'exécution d'une oeuvre...

L'ordre des actions étant « fondamental », la question de sa description se pose immédiatement. Ainsi Dieudonné [DIE85] propose dès la deuxième page de son livre la représentation arborescente des actions, puis leur composition « par la séquence, l'alternative SI... ALORS... SINON, l'itération : TANT QUE pas fini, répéter une action. » On ne sait plus alors ce qui reste à la programmation : on est déjà dans les structures de contrôle de PASCAL. Le soit-disant cours d'algorithmique est un cours de PASCAL qui n'ose pas dire son nom...

Nous nous proposons de revenir sur la notion d'algorithme, d'en indiquer des modes de représentation ne prenant en compte que les propriétés de ces objets. Ainsi se manifestera plus clairement peut-être la distance entre algorithme et programme. Mais il apparaîtra aussitôt que

les structures des langages habituellement utilisés dans l'enseignement (PASCAL, pour ne pas le nommer) ne sont pas les plus adaptées. D'autres langages sont possibles, et le cercle se referme. Est-il vraiment possible de parler d'algorithme sans entraîner avec soi un langage de programmation ? On ne fait pas de l'algorithmique pour élaborer une théorie mathématique, mais comme un pas vers la programmation. N'est-il pas raisonnable de prendre l'objectif en compte dès le début ?

2. LA NOTION D'ALGORITHME.

On sait que le mot vient du nom d'un mathématicien arabe du X° siècle, Al Kwarismi. Mais il ne fut pas le premier à dire comment réaliser un calcul. La méthode la plus connue pour calculer le plus grand commun diviseur de deux entiers est désignée sous le nom « Algorithme d'Euclide ». Il fallut attendre le XX° siècle pour que des mathématiciens étudient l'objet « Algorithme » afin d'en décrire les propriétés. Alan Turing imagine une machine fictive munie d'une tête de lecture devant laquelle défile un ruban portant des caractères 0 ou 1, comme une bande magnétique. Pour dire ce que fait la machine, on suppose qu'elle peut se trouver dans un certain nombre (fini) d'états, et peut agir sur le ruban : écrire un caractère, l'avancer ou le reculer d'une case. Les règles qui commandent son action ont toutes la même forme :

SI état i ET lu = 0 ALORS action i_0, état j_0 IS

SI état i ET lu = 1 ALORS action i_1, état j_1 IS

où lu désigne le caractère lu sur le ruban, action i_0 ou i_1 ont la forme décrite plus haut, j_0 ou j_1 sont le nouvel état pris par la machine. Un de ces états est l'état final : quand elle y arrive, la machine s'arrête. Turing a démontré que toute fonction calculable peut être calculée par une telle machine, autrement dit, ce **formalisme permet d'exprimer TOUS les algorithmes.** Un algorithme pour machine de Turing est un ensemble fini de régles de cette forme. **Cet ensemble n'est pas ordonné.** A chaque instant, la machine étant dans un certain état, seules les deux règles mentionnant cet état sont envisageables; la lecture d'un caractère sur le ruban dit celle qui sera appliquée.

Dans les années 50, le mathématicien russe A. Markof fit une théorie des algorithmes. Pour les décrire, il utilise des règles ayant une des deux formes que voici :

SI condition i ALORS action i IS

SI condition i ALORS action i; FINI IS

Ses notations étaient un peu différentes : par exemple, il utilisait un point après la mention de l'action à faire, au lieu du mot FINI. Ce sont des détails d'écriture non significatifs. Pour exécuter l'algorithme, on cherche une condition vraie, on exécute l'action correspondante. S'il n'y a pas le mot FINI, on recommence en cherchant une nouvelle condition vraie. S'il y a le mot FINI, l'algorithme est terminé.

Pour illustrer cette forme d'algorithme, voyons comment on peut ordonner trois nombres **a**, **b**, **c**. Une règle dira que s'ils sont dans le bon ordre, c'est fini :

SI a ≤ b ET b ≤ c ALORS FINI IS

Si ce n'est pas fini, c'est qu'une des deux paires au moins n'est pas dans le bon ordre. On la retourne. D'où l'algorithme :

(1) SI a ≤ b ET b ≤ c ALORS FINI IS

(2) SI a > b ALORS échanger a et b IS

(3) SI b > c ALORS échanger b et c IS

Appliquons ceci à la suite a=3 b=2 c=1. Les règles (2) ou (3) s'appliquent, prenons en une, par exemple (3). Il vient a=3 b=1 c=2. Cette fois, seule la règle (2) s'applique : a=1 b=3 c=2. Seule la règle (3) s'applique : a=1 b=2 c=3. La règle (1) s'applique, et c'est fini.

Il n'y a pas une différence énorme avec les machines de Turing : le ruban a disparu, il n'y a plus que deux états (l'état courant. à la fin de toute règle sans FINI, l'état final, pour les autres). Les actions ont des formes moins contraintes. Mais dans les deux cas. Les règles sont formées d'une condition commandant une action. L'ensemble des règles n'est pas ordonné. L'algorithme est itératif : il y a une boucle implicite autour de l'ensemble des règles. dont on sort quand on a atteint l'état final.

Markof précise la notion d'algorithme : c'est un ensemble de **régles précises** telles qu'en les appliquant à des données dans un domaine suffisamment large, on obtienne obligatoirement le résultat cherché. C'est la définition donnée par C. et P. Richard [CRE87]. Elle diffère de ce qui est dit dans de nombreux ouvrages en deux points :

- les règles doivent être précises. *Mettre de l'eau dans une casserole* n'est pas une règle précise. Ce ne peut être un algorithme. Une

partition de musique n'est pas un algorithme, parce que deux exécutants différents n'en donnent pas la même interprétation.

- les règles ne sont pas ordonnées. Là est la différence fondamentale avec la programmation.

3. ALGORITHME ET LANGAGE MATHÉMATIQUE.

3.1. Algorithmes simples.

Turing et Markof ont utilisé des formes contraintes pour dire les algorithmes parce qu'ils voulaient démontrer qu'elles permettent de tout dire. Mais de tous temps les mathématiciens ont exprimé des algorithmes dans leur propre langage. Supposons que je veuille calculer le montant taxes comprises d'un achat de plusieurs objets identiques. Je définirai ce montant comme suit :

(1) montant-TTC = montant-HT + TVA

(2) montant-HT = prix-unitaire * quantité

(3) TVA = montant-HT * taux

Je définirai les constantes et données du problème :

(4) taux = 0.186

(5) quantité = 7

(6) prix-unitaire = 13

Ces relations ont été mises dans un ordre quelconque, et le résultat ne dépend pas de cet ordre. Mais le calcul ne peut se faire dans un ordre quelconque. Si par exemple je commence par la relation (1) :

(1) montant-TTC = montant-HT + TVA

je constate que je ne connais ni le montant-HT, ni la TVA. Les relations sont ordonnées entre elles par une **relation de précédence :** on ne peut calculer une quantité que si toutes celles dont elle dépend sont connues; les variables figurant à droite doivent être calculées **avant** celles de gauche. Dans l'exemple ci-dessus, il faut commencer par les relations (4), (5), (6), dans un ordre quelconque, parce qu'elles ne dépendent de rien, puis (2) (tout ce qui est à droite est connu), puis (3), puis (1).

On est parti des données en allant vers le résultat. On peut aussi considérer les définitions ci-dessus comme celles de fonctions. On obtient alors un programme au sens fort du terme :

```
taux _ 0.186 ; quantité _ 7 ; prix-unitaire _ 13
AFFICHER &montant-TTC( )
TERMINER

FONCTION &montant-TTC( )
        RESULTAT &montant-HT( ) + &TVA( )
$&montant-TTC

FONCTION &montant-HT( )
        RESULTAT prix-unitaire * quantité
$&montant-HT

FONCTION &TVA( )
        RESULTAT &montant-HT( ) * taux
$&TVA
```

Le programme proprement dit appelle la fonction &montant-TTC, dont l'exécution appelle &montant-HT, puis &TVA qui rappelle &montant-HT (cette fonction sera évaluée deux fois. mais donnera les deux fois le même résultat). L'ordre dans lequel sont écrites les définitions des fonctions n'a pas d'importance.

Les deux façons de calculer le résultat sont bien connues en informatique [ARS77, ARS83] : on peut partir des données. et chercher à chaque pas quelle valeur peut être calculée, parce que les quantités dont elle dépend sont connues : c'est ce qu'en anglais on appelle « data–flow » ; on peut partir du résultat, et remonter à partir de là jusqu'à une quantité connue : c'est , on attend d'en avoir besoin pour calculer une quantité.

3.2. Algorithmes récurrents.

La description d'un algorithme par une suite de définitions ne permet pas d'atteindre les mécanismes répétitifs. Il faut passer par les suites récurrentes. Pour ne pas faire de théorie inutile, prenons un exemple simple : on veut calculer le quotient **q** et le reste **r** de la division d'un entier **a** par un entier **b**>0. **q** est le nombre de fois que l'on peut retrancher **b** de **a**. On fait donc apparaître les valeurs successives

engendrées par la soustraction de **b**. La première, x(0), est **a**. On passe d'une valeur à la suivante en en retranchant **b**. Le processus s'arrête quand on ne peut plus retrancher **b**, donc pour le plus petit **i** tel que x(i)<0 (ce que dit l'opérateur m).

(1) x(i) = x(i-1) - b

(2) x(0) = a

(3) q = m i : x(i) < b

(4) r = x(q)

Là encore, ces relations ne sont pas ordonnées. L'ordre des calculs est donné par la nécessité : je ne peux calculer x(i) que si je connais x(i-1) : je dois les calculer dans l'ordre des i croissant, donc à partir de x(0) (relation (2)). Je m'arrête **dès que** je trouve un x(i) < b, ou, ce qui revient au même, je continue **tant que** x(i) ≥ b.

4. ALGORITHMES ET PROGRAMMES.

Cet exemple fait apparaître clairement la différence entre l'expression mathématique d'un algorithme et le programme qui commande l'exécution de cet algorithme par un ordinateur. D'une part, les calculs doivent être ordonnés. D'autre part. on s'efforce de ne pas multiplier le nombre de variables distinctes, pour économiser la place en mémoire. Pour l'algorithme de division, on utilisera une seule valeur de x, la nouvelle x(i) remplaçant l'ancienne x(i) :

```
x _ a ; i _ 0
TQ x ≥ b FAIRE
        x _ x - b ; i _ i + 1
BOUCLER
q _ i ; r _ x
```

Il n'est pas très difficile de *passer à la main* d'un algorithme exprimé par des relations ou des suites récurrentes à un programme. C'est la base de la *méthode déductive de Nancy* [DUC82] : on écrit les relations produisant le résultat. puis on les ordonne. Mais ceci peut échouer :

(1) résultat = si x>0 alors u sinon v is

(2) u = √x

(3) v = √(-x)

La relation (1) utilisant u et v, ceux-ci doivent être calculés en premier. Si je l'ordonne en un programme :

```
u _ RAC(x) ; v _ RAC(-x);
AFFICHER si x> ALORS u SINON v IS
```

celui-ci n'affiche effectivement le résultat que pour x = 0 ; dans tous les autres cas. il demande le calcul de la racine carrée d'un nombre négatif. Pourtant l'algorithme s'exécute correctement en évaluation retardée. La transformation automatique de tels algorithmes en programme s'est révélée impossible.

On a tenté de faire exécuter directement ces algorithmes par un ordinateur. Un système expert opère en cherchant quelle règle peut être appliquée, puis en exécutant l'action qu'elle commande. Cela ne vaut pas pour les algorithmes non récurrents.

Il est donc de la responsabilité du programmeur de rédiger le programme d'ordinateur à partir d'un algorithme. L'expérience a montré que l'écriture intermédiaire d'un algorithme sous une des formes présentées plus haut n'apporte pas grand chose. Il n'est pas beaucoup plus difficile de produire le programme directement que de composer l'algorithme intermédiaire. En outre, les notations algorithmiques sont le plus souvent beaucoup plus lourdes. Ainsi par exemple, lorsqu'on travaille sur un tableau, on change des valeurs dans le tableau par des affectations : a[i] _ a[i] + 1. Pour en rendre compte dans un algorithme. il faut fabriquer une suite récurrente de tableaux, le changement d'un élément donnant un nouveau tableau dont tous les éléments ont même valeur que dans l'ancien, un seul ayant changé. Un algorithme de tri ainsi rédigé est au moins aussi illisible que le programme correspondant, sinon plus.

Ce dont rêvent les professeurs, en parlant d'algorithmique, c'est de trouver une façon d'écrire **les programmes** indépendante d'un langage particulier, en jouant sur le fait que tous les langages sont fondés sur les mêmes constructions : l'affectation, la composition séquentielle, alternative, itérative. Mais c'est de la programmation, pas de l'algorithmique : celle-ci ne connaît pas la composition séquentielle...

Reste qu'une chose devrait être retenue des travaux de Turing ou Markof : la forme fondamentale d'une règle est :

SI condition ALORS action, état IS

Cette forme est la plus simple possible, et elle permet de rédiger des programmes clairs, proches de l'énoncé en langue naturelle. L'exemple du distributeur de billets à carte bancaire le montre bien. Un tel distributeur rend la carte s'il ne peut la lire, la refuse si elle est périmée, demande le code et garde la carte si on ne peut le donner en au plus 3 essais...

SI carte illisible ALORS message; rendre carte; FINI IS

SI carte périmée ALORS message; rendre carte; FINI IS

demander code

SI échec ALORS message; garder carte; FINI IS

...

Nous n'avons pas détaillé les messages, cela n'ajoute rien à ce que nous voulons montrer. Le seul état référencé est l'état final, et l'expérience (confirmant la théorie de Markof) montre que c'est suffisant. On aboutit ainsi à un style de programmation qui permet de **sérier les questions,** n'en traitant qu'une à la fois, et reprenant la recommandation de Descartes, dans son discours de la méthode : commencez par les cas les plus simples. C'est une façon d'opérer qui simplifierait grandement la tâche des apprentis programmeurs : ni Turing, ni Markof n'utilisent le SINON. On devrait le réserver au choix entre deux expressions (expression conditionnelle).

Le système EXEL [ANR74], dont sont sortis les langages SIOUX [VAS89] et LSE89 était fondé sur ces idées-là. Il se rapprochait de l'algorithmique. Les enseignants auraient certainement intérêt à les considérer...

Jacques ARSAC

RÉFÉRENCES

[ANR74] J. ARSAC, L. NOLIN, G. RUGGIU. J.P. VASSEUR
Le système de programmation structurée EXEL
Revue Tech. Thomson CSF, 6, 3, 1974, 715-736

[ARS77] J. ARSAC *La construction de programmes structurés*
Dunod, Paris, 1977

[ARS83] J. ARSAC *Les bases de la programmation*
Dunod, Paris, 1983

[CRE87] A. CREDI (nom collectif)
Approche de la programmation Belin, Paris, 1987

[DIE85] D. DIEUDONNE, J.F. BERTHON, A. NEVIANS,
Apprendre à programmer Cedic-Nathan 1985

[DUC82] A. DUCRIN (nom collectif), *Programmation*,
Dunod, Paris, 1982

[LAU82] J.P. LAURENT *Initiation à l'analyse et à la programmation*
Dunod, Paris, 1982

[VAS89] J.P. VASSEUR *La programmation naturelle*
Technea, Toulouse, 1989

[EPI91] *Répertoire informatisé des articles de 1971 à 1991.*