Chapitre 1

Notions d'algorithmique

La notion d'algorithme est très largement antérieure à l'invention de l'informatique. Un **algorithme** est un processus systématique de résolution d'un problème en un nombre fini d'étapes *.

Le lecteur connaît peut-être déjà certains algorithmes :

- l'algorithme de résolution des équations du second degré : calcul du discriminant, puis en fonction de son signe calcul des solutions ;
- la méthode du pivot due à Gauss, qui est un algorithme général de résolution des sytèmes linéaires ;
- l'algorithme de Dijkstra qui sert à résoudre le problème du plus court chemin en théorie des graphes ;
- l'algorithme d'Euclide qui permet de calculer le pgcd de deux entiers.

Un algorithme n'est d'ailleurs pas nécessairement mathématique. Ainsi lorsque vous exécutez une recette de cuisine ou lorsque vous montez un meuble vendu en pièces détachées vous exécutez sans le savoir des algorithmes !

On le voit, la suite finie d'opérations amenant à la résolution d'un problème peut être proposée à une machine mais aussi à une personne. Pour que notre algorithme puisse être effectué, il faut qu'il soit constitué uniquement d'opérations compréhensibles par notre interlocuteur. Dans notre cas, l'interlocuteur sera une machine. Or, nous ne connaissons pas le langage de la machine. C'est là qu'interviennent les **langages informatiques**.
Le processus est le même pour tous les langages informatiques : on écrit un **code source** dans un certain langage. Le code source est ensuite transformé en **exécutable** (c'est-à-dire un code écrit en langage machine). Le logiciel qui permet ce transfert est appelé **compilateur**.

---

*. Au neuvième siècle, à Bagdad, Mohammed Al Khwarizmi rédige un ouvrage dans lequel il expose des méthodes systématiques de résolution des équations du premier et du second degré. Dans la version latine de son ouvrage le traducteur déforme son nom en « Algoritmi ». C'est ce qui a donné notre mot algorithme.
Notons qu'Al Khwarizmi est sans doute l'un des plus grands mathématiciens de son temps. On lui doit aussi l'introduction du zéro et de la numération décimale de position (les chiffres "arabes"). C'est enfin à lui que l'on doit le mot algèbre.

Le langage employé dans cet ouvrage est le Pascal. Le compilateur est le Turbo-Pascal. Mais avant d'apprendre les bases de celui-ci, on se propose dans ce chapitre d'étudier l'élaboration générale des algorithmes informatiques, en dehors des contraintes spécifiques à ce langage.

1.1 Principes généraux de programmation

1.1.1 Structure d'un algorithme

Dans une bonne recette de cuisine, on commence par énumérer les ingrédients (pour la tarte aux pommes : Farine, 150g; Beurre, 75g; Pommes, 1kg). De même, dans la notice de montage d'un meuble sont énumérés les outils nécessaires (6 vis, 10 pointes, etc.).

De manière générale, un algorithme se décompose lui aussi en deux parties : d'abord la liste des données qui serviront dans l'algorithme, puis les instructions qui amènent à la résolution du problème posé.

Pour construire la partie relative aux instructions, nous étudions à présent deux outils théoriques d'aide à la création d'algorithmes.

1.1.2 L'analyse descendante

C'est une technique rigoureuse de résolution des problèmes. Elle est adaptée aussi bien à la programmation qu'à la vie professionnelle.
Elle consiste à décomposer tout problème non élémentaire en sous-problèmes plus simples. Plus précisément, pour résoudre un problème on suit la démarche suivante :

- si la solution du problème est élémentaire, alors l'écrire ;
- sinon, décomposer ce problème en sous-problèmes plus simples ;
- si les sous-problèmes ne sont pas élémentaires réitérer le procédé jusqu'à ce qu'on n'ait que des problèmes élémentaires.

EXEMPLE

Imaginons que l'on veuille programmer un robot domestique pour qu'il sache faire cuire les spaghettis.
On peut d'abord décomposer grossièrement son travail en :

- Faire chauffer de l'eau ;
- Quand celle-ci est bouillante jeter les spaghettis dedans ;
- Au bout de 10 mn, les égoutter.

Mais aucun de ces sous-problèmes n'est compréhensible par une machine. On peut alors raffiner. Le sous-problème « faire chauffer de l'eau » peut ainsi se décomposer en :

- Remplir une casserole d'eau ;
- La poser sur une plaque électrique ;
- Allumer celle-ci.

Certaines de ces opérations ne sont pas encore claires pour un robot. Par exemple on peut détailler « remplir la casserole d'eau » en :

- Placer la casserole sous le robinet ;
- Ouvrir le robinet ;
- Lorsque la casserole est pleine, fermer le robinet.

1.1.3 La programmation structurée

Il va sans dire qu'un langage informatique n'est capable de comprendre qu'un nombre limité d'instructions (faire un calcul - afficher un texte à l'écran - etc.) et de structures (conditionnelles - répétitives - etc.).
Cela semble être un inconvénient, c'est en fait un atout. En effet, cela nous oblige à clarifier au maximum nos algorithmes. Mieux, il existe une théorie, la programmation structurée, qui démontre que n'importe quel algorithme peut être écrit en utilisant les trois seules structures suivantes :

- La **séquence** : simple suite d'instructions élémentaires ;
- L'**alternative** : de la forme SI condition ALORS instruction ;
- L'**itération** : de la forme REPETE instruction JUSQU'A condition d'arrêt.

L'usage de l'analyse descendante et de la programmation structurée doit nous permettre d'écrire des algorithmes d'une grande clarté.

EXEMPLE

On veut écrire l'algorithme de résolution des équations du second degré, de la forme $aX^2 + bX + c = 0$.
Une première décomposition grossière du travail nous amène aux étapes suivantes :

- Demander a, b et c à l'utilisateur ;
- Calculer $\Delta = b^2 - 4ac$;
- En fonction du signe de Δ faire ce qui convient.

On considère que les deux premières instructions sont suffisamment élémentaires pour être comprises par la machine. Ce n'est pas le cas de la dernière, on la décompose donc en instructions plus simples :

- Si $\Delta < 0$ alors écrire « pas de solution réelle » ;
- Si $\Delta = 0$ alors
  - calculer $x = -b/(2a)$;
  - afficher x.
- Si $\Delta > 0$ alors
  - calculer $x_1 = (-b - \sqrt{\Delta})/(2a)$;
  - calculer $x_2 = (-b + \sqrt{\Delta})/(2a)$;
  - afficher x_1 et x_2.

Cette fois notre décomposition est assez détaillée. Grâce à l'analyse descendante, nous avons donc réussi à déterminer l'algorithme de résolution des équations du second degré en utilisant comme seules structures la séquence et l'alternative.

Par contre, notre algorithme ne contient pas d'itération. Pour compléter notre exemple nous pouvons en ajouter une en proposant que la résolution soit répétée

autant de fois que l'utilisateur le souhaite. Cela se fait en écrivant :
Répéter les instructions suivantes :

- résolution des équations du second degré ;
- Afficher « une autre équation (o/n) ? » ;
- Lire *réponse* ;

Jusqu'à ce que *réponse* soit égal à « n ».

On peut alors, finalement, écrire notre algorithme complet, sans oublier de citer d'abord la liste des données utilisées.

- **Données :**
  - a ,b, c, Δ, x, x_1, et x_2 : des nombres réels ;
  - *réponse* : un caractère.
- **Algorithme :**
  Répéter les instructions suivantes :
  - Demander a, b et c à l'utilisateur ;
  - Calculer $\Delta = b^2 - 4ac$;
  - Si $\Delta < 0$ alors écrire « pas de solution réelle » ;
  - Si $\Delta = 0$ alors
    - calculer $x = -b/(2a)$;
    - afficher x.
  - Si $\Delta > 0$ alors
    - calculer $x_1 = (-b - \sqrt{\Delta})/(2a)$;
    - calculer $x_2 = (-b + \sqrt{\Delta})/(2a)$;
    - afficher x_1 et x_2.
  - Ecrire « une autre équation (o/n) ? » ;
  - Lire *réponse* ;

  Jusqu'à ce que *réponse* soit égal à « n ».

1.2 Efficacité des algorithmes

1.2.1 Critères d'évaluation des bons algorithmes

On verra dans les prochains chapitres qu'on a souvent plusieurs façons de résoudre un même problème. Comment décider, entre deux algorithmes, lequel est le meilleur ?

Il existe beaucoup de critères d'évaluation des « bons » algorithmes : la clarté, la simplicité, etc. Mais lorsqu'un client commande un logiciel, il n'a que deux critères d'efficacité : la rapidité d'exécution et la précision du résultat.

Nous verrons au chapitre 2 que le manque de précision est souvent dû aux erreurs d'arrondis que peut faire l'ordinateur lors de chaque opération entre réels.

Dans la pratique, il est difficile d'évaluer le temps d'exécution d'un algorithme, celui-ci dépend en effet du choix du langage dans lequel on traduit l'algorithme, il dépend également de la machine sur laquelle on exécute le programme. Pour

évaluer l'efficacité d'un algorithme, on choisit donc de compter le nombre d'opérations élémentaires (additions - multiplications - divisions - comparaisons - appels à une fonction - etc.) effectuées par celui-ci, cette donnée étant moins dépendante du contexte de programmation que le temps d'exécution.

Ce nombre d'opérations mesure bien à la fois nos deux critères : d'une part, moins on fait d'opérations et moins on fait d'erreurs d'arrondis ; d'autre part, si le nombre d'opérations diminue, le programme gagne aussi en rapidité.

1.2.2 Complexité

Naturellement, le nombre d'opérations à effectuer dans un algorithme augmente avec le taille des données à traiter. Ainsi, dans un algorithme qui calcule les termes d'une suite u de u_0 à u_n, le nombre d'opérations effectuées augmente avec l'entier n. De même, dans un algorithme manipulant des polynômes, le nombre d'opérations augmente avec leur degré.

Pour savoir si un algorithme est efficace, il est donc intéressant de connaître le rapport entre la taille des données et le nombre d'opérations. Si l'augmentation du nombre d'opérations est proportionnelle à celle de la taille des données, on a un bon algorithme. Si par contre cette augmentation est exponentielle, mieux vaut trouver autre chose.

Bien sûr, l'étude de ce nombre d'opérations n'est intéressante que lorsque la taille des données devient importante. La complexité d'un algorithme se mesure donc pour n assez grand.

Définition. Soit n la taille des données d'un algorithme et f une fonction de n. On dit que la **complexité** de l'algorithme est en $\Theta(f(n))$ (lire « en theta de $f(n)$ » ou « de l'ordre de $f(n)$ ») s'il existe deux constantes strictement positives A et B telles que pour n assez grand, le nombre d'opérations élémentaires effectuées par l'algorithme soit compris entre $A.f(n)$ et $B.f(n)$.

REMARQUES

(1) $f(n)$ est en général choisie dans l'ensemble $\{1, \ln(n), n, n\ln(n), n^2, n^\alpha, a^n\}$.

(2) Notons que si $C(n)$ désigne le nombre exact d'opérations élémentaires effectuées par l'algorithme et s'il existe un réel α strictement positif tel que :

$$C(n) \underset{n\to+\infty}{\sim} \alpha.f(n)$$

alors la complexité de l'algorithme est en $\Theta(f(n))$.

EXEMPLES

(1) Soit P un polynôme de degré n. Nous verrons (cf. chapitre 7 § 7.2.3) que l'algorithme de Hörner, qui permet d'évaluer P en un nombre x nécessite $4n+2$ opérations élémentaires.
Ce nombre est équivalent à $4n$ au voisinage de l'infini, la complexité de l'algorithme de Hörner est donc en $\Theta(n)$.

Cela signifie que son exécution est à peu près proportionnelle au degré du polynôme.

(2) Soient n données numériques. On appelle algorithme de tri tout algorithme qui permet de replacer ces données dans l'ordre croissant.
Nous verrons (cf. chapitre 17) que l'algorithme de tri bulle a une complexité en $\Theta(n^2)$. Cela signifie que lorsque l'on double le nombre de données le coût de l'exécution de l'algorithme est environ multiplié par 4.
Il existe un meilleur algorithme de tri, c'est celui de tri rapide dont la complexité est – en moyenne – en $\Theta(n\ln(n))$ (même si elle peut être en $\Theta(n^2)$ dans certains cas particuliers).

REMARQUE

Notons que l'on peut parfois évaluer définitivement la difficulté d'un problème en évaluant la complexité minimale d'un algorithme le résolvant. Ainsi, il est démontré qu'il n'existe pas d'algorithme de tri dont la complexité moyenne soit inférieure à $\Theta(n\ln(n))$.

De même, l'algorithme de Hörner est l'algorithme optimal d'évaluation d'un polynôme en un point.

Chapitre 2

Structures de base du langage

2.1 Structure générale d'un programme

Un programme est une succession d'ordres donnés au compilateur. Ces ordres sont de deux types :

- les **instructions exécutables** qui lors de l'exécution feront effectuer une tâche à la machine (un calcul, un affichage à l'écran, etc.) ;
- les **déclarations** qui sont des instructions non exécutables, elles sont constituées d'informations destinées à aider le compilateur dans sa traduction.

Dans le langage Pascal l'agencement de ces instructions est imposé, celles-ci sont regroupées en trois parties :

- l'en-tête ;
- la partie déclarative ;
- la partie exécutive appelée **corps du programme**.

■ ***L'en-tête***

Celle-ci est composée d'une seule ligne commencant par le mot `PROGRAM`, elle permet de nommer le programme :

```
PROGRAM Test ;
```

Le nom donné au programme doit respecter un certain nombre de règles syntaxiques qui seront énoncées au paragraphe 2.2.1.
Notons que la présence de l'en-tête est facultative.

■ ***La partie déclarative***

La première règle fondamentale imposée par le langage Pascal peut s'énoncer ainsi : « **définir avant d'utiliser** ».
Dans la partie déclarative on doit donc déclarer toutes les données qui seront utilisées dans le programme. Ces données sont soit **constantes** soit **variables** et peuvent être de différents **types** : d'un type prédéfini (caractères, nombres entiers, nombres réels, etc.) ou d'un type défini par le concepteur du programme.

Nous étudierons en détail dans le paragraphe 2.2 la forme que doivent prendre ces déclarations.

REMARQUE

Outre les déclarations de constantes, variables et types, la partie déclarative est aussi l'endroit où l'on déclare les « sous-programmes » (procédures et fonctions) que nous étudierons ultérieurement.

■ ***Le corps du programme***

Il est constitué de la succession des instructions exécutables dans leur ordre d'exécution et séparées les unes des autres par un point virgule. Il est compris entre les mots `BEGIN` et `END.` (le point caractérisant la fin du programme).

En résumé, tout programme s'articule de la façon suivante :

```
PROGRAM nom_du_programme;          } En-tête

déclaration des constantes
déclaration des types              } Partie déclarative
déclaration des variables
déclaration des sous-programmes

BEGIN
    instructions exécutables       } Corps du programme
END.
```

2.2 La partie déclarative

2.2.1 Identificateurs

Avant de voir comment déclarer constantes, variables et types, il nous faut énoncer un certain nombre de règles nécessaires à la formation des mots dans le langage Pascal.

Les mots du langage sont de deux sortes :

- les **mots « réservés »** : ce sont soit des mots invariables qui jouent un rôle déterminé dans le langage (par exemple `PROGRAM`, `BEGIN`, `END`, etc.) ; soit des fonctions mathématiques prédéfinies (cf. § 2.3.1) comme `sin`, `cos` ou `exp` ; soit des procédures d'entrée-sortie (cf. § 2.3.3) comme `Write` et `Read`.
- les **identificateurs** : ce sont des mots que le programmeur choisit librement (dans le respect des règles qui vont suivre), et qui lui permettent de nommer son programme, ses sous-programmes (procédures et fonctions) et ses données (constantes, variables et types).

Les caractères autorisés pour construire un identificateur sont :

- les lettres majuscules et minuscules non accentuées,
- les chiffres,
- le caractère de soulignement : « _ ».

Par contre, les caractères spéciaux ($, #, %, “, etc.) et l'espace sont à proscrire. Enfin un mot ne peut commencer que par une lettre.