

Algorithmique

Ce qu'il faut retenir

L'algorithmique est la science qui permet l'étude et l'analyse de traitements automatisés grâce à du **pseudo-code** (du code informatique écrit en langage naturel, donc pour nous ce sera en français). Il est aussi possible d'exprimer un **algorithme** avec un diagramme qu'on appelle **organigramme de programmation** ou **logigramme**.

L'objectif du pseudo-code est de **structurer sa pensée** pour pouvoir exprimer les actions à réaliser dans une grammaire proche de celle attendue par la machine. L'avantage du pseudo-code est d'être **indépendante du langage de programmation**. Un algorithme en pseudo-code peut donc être théoriquement compris par quelqu'un qui ne connaît pas votre langage de programmation : vos futurs clients ! D'où l'intérêt de le rédiger avec lisibilité, concision et un niveau de détail adapté au processus que vous souhaitez décrire.

Un algorithme est une suite **d'instructions** qui seront exécutées les unes après les autres, c'est à dire **de manière séquentielle / linéaire / les unes après les autres**.

Une **instruction** peut être :

- Une **déclaration** (de programme, de procédure, de fonction ou de variable)
- Une **assignation de variable** (aussi appelée "affectation")
- Une **structure de contrôle**
- Un **appel de fonction**

Un algorithme ou un programme **valide** est un algorithme qui contient **au moins 1 instruction**.

Exemple :

```
BEGIN PROGRAM // This is the program declaration
|
| // Usually we have to declare variables before we can use them.
| // If you have only one variable, feel free to write a one-line DECLARE instruction.
| DECLARE
| |
| |----->myArray : INTEGER[10] // array of size 10 containing integer values
| |----->number : INTEGER // a single integer value
| |
| END DECLARE
|
| FOR number FROM 0 TO 9 DO
| |
| |-----> // Indentation inside a block is MANDATORY (lisibility concerns)
| |----->myArray[i] = i
| |
| END FOR
|
END PROGRAM
```

1. Les bases d'un algorithme

Un algorithme est quelque chose qui est à la fois assez souple et informel mais dont la structuration est très codifiée. Voici les termes à connaître absolument :

- Grammaire
- Programme
- Variable
- Procédure
- Fonction
- Instruction
- Expression

1.1. La grammaire

Pour définir la syntaxe d'un langage, on utilise très souvent la BNF ([Backus-Naur Form](#)¹). Par la suite j'utiliserai quelques fois la BNF pour décrire une syntaxe si elle n'est pas trop compliquée.

NOTE :

- <Xyz> signifie "élément Xyz" et invite à aller voir la définition de Xyz plus bas
- | signifie "ou"
- ::= signifie "est composé de"
- {<xyz>} signifie que "l'élément xyz est attendu un nombre indéterminé de fois"
- [<xyz>] signifie que "l'élément xyz est optionnel"
- ... Tout le reste doit être pris comme argent comptant !

Exemple :

```
// Ici je souhaite décrire comment s'écrit un nom de famille.
// Un nom de famille est une succession d'un nombre indéfini de majuscules,
// suivies par un tiret et un autre nom de famille, ou bien suivies par un espace
// et un autre nom de famille.
//
NomDeFamille ::= {<MAJUSCULE>} {[-][NomDeFamille]} | {" "<NomDeFamille>}
MAJUSCULE ::= A|B|C|D...|Y|Z

// Avec ces deux lignes, je peux énoncer tous les cas possibles de noms de famille.
// Par exemple : "PICASSO", "BACKUS-NAUR" ou "VAN GOGH".
```

Ce formalisme est très puissant car il permet à la BNF de se décrire elle-même en BNF ! Cette particularité en fait donc un **méta-langage**.

1 https://fr.wikipedia.org/wiki/Forme_de_Backus-Naur

1.2. Le Programme

Dans la vraie vie, les programmes (ou applications, logiciels, progiciels...) comportent plusieurs algorithmes, qu'on regroupe dans des fonctions, des procédures ou des sous-programmes si le code devient trop long.

Le programme sert d'une part à indiquer un "**point d'entrée**" (comme la fonction `main()` en Java ou en C++) et d'autre part c'est lui qui va appeler et orchestrer les appels aux différentes fonctions et procédures permettant de réaliser notre traitement.

Le code d'un programme s'écrit entre des balises DEBUT et FIN (on accepte le français et l'anglais mais il vaut mieux s'adresser dans la langue de son interlocuteur) :

```
PROGRAMME ::= <DEBUT> <EOL> <instructions> <EOL> <FIN>
DEBUT     ::= DEBUT_PROGRAM[ME] | BEGIN_PROGRAM[ME] | START_PROGRAM[ME]
FIN       ::= FIN_PROGRAM[ME] | END_PROGRAM[ME] | STOP_PROGRAM[ME]
EOL       ::= "caractère de fin de ligne (CR-LF sous windows ou LF sous Mac et Linux)"

instructions ::= ...à définir ultérieurement ;)
```

Exemple :

```
BEGIN PROGRAM

    // This program does absolutely nothing, but does it very well ;)
    // BTW it's a valid program declaration even though it does nothing.

END PROGRAM
```

1.3. Les variables

Une **variable** c'est une **zone de la mémoire** dont la taille varie en fonction de son type (de 1 à 8 octets). On peut comparer une variable à une cellule d'un tableau Excel : elle a un nom (son libellé de cellule dans Excel) et une adresse (A5 par exemple).

La valeur d'une variable ne change QUE LORS D'UNE AFFECTATION. Lors d'une affectation de variable "`a = <une expression quelconque>`", l'ordinateur calcule la valeur de l'expression qui est à droite du signe égal puis la stocke dans la zone de mémoire de la variable. **L'expression qui sert à l'affectation n'est JAMAIS réévaluée** (à moins qu'on retrouve la même instruction plus loin dans le programme évidemment).

En algorithmique, on déclare toujours le type d'une variable "`<nomVariable> : <TYPE>`" où `<TYPE>` est parmi :

| | | |
|------------|---------|--|
| •ENTIER | INTEGER | // ex. : -5 49 45698520 ...etc. |
| •REEL | REAL | // ex. : 3.14 2.0112 -1125874.0 ...etc. |
| •BOOLEEN | BOOLEAN | // ex. : VRAI ou FAUX |
| •CARACTERE | CHAR | // ex. : 'a' 'B' '&' '\$' ...etc. |
| •CHAINE | STRING | // ex. : "Variable" "Pomme" "Panda" ..etc. |

1.4. Les procédures & fonctions

Une procédure (aussi appelée sous-programme ou sous-routine) est destinée à réaliser une tâche complexe à plusieurs endroits du programme.

À l'inverse des fonctions, les procédures ne retournent aucun résultat donc on ne peut pas les utiliser dans des expressions. Un appel de procédure est forcément une instruction terminale (on ne peut pas l'utiliser autrement que toute seule).

Exemple :

```
// This procedure displays the message passed in parameters
// in the console and repeats this operation 10 times.
//
BEGIN PROCEDURE Repeat10Times(message : STRING)

    FOR (count FROM 1 TO 10) DO
        PRINT(message)
    END FOR

END PROCEDURE

// -----
// Main program
// -----
BEGIN PROGRAM
    Repeat10Times("I will survive")

    //PRINT( Repeat10Times("I will survive") ) // NOT ALLOWED, this will generate an error
END PROGRAM
```

Une fonction est comme une procédure mais **elle retourne un résultat**. La distinction procédure/fonction se fait en algorithmique et dans les anciens langages de programmation mais on les distingue de moins en moins dans les nouveaux langages de programmation. En Java on utilise le terme de "**méthodes**" pour parler indifféremment des procédures et des fonctions.

Puisqu'une fonction retourne un résultat, l'appel à une fonction est une expression qui peut s'utiliser dans une opération mathématique, en paramètre d'une autre fonction...etc.

Exemple :

```
// Function computing the mean of 2 numbers n1 and n2.
// Returns a REAL number.
//
DECLARE FUNCTION Mean( n1 : INTEGER, n2 : INTEGER ) : REAL
    RETURN (n1 + n2) / 2
END FUNCTION
```

Plus de détails dans le chapitre "4. Les fonctions".

1.5. Les expressions

Une expression, ça peut être plein de choses. C'est un terme général qui désigne une sous-partie d'une instruction et qui reflète **une valeur et qui peut être utilisée seulement dans certains contextes**.

Voici des exemples d'expressions :

- Un nombre
- Les valeurs VRAI et FAUX
- Une variable
- Une opération mathématique (entre deux expressions de mêmes types)
- Un appel de fonction

Voici les cas où on trouve des expressions :

```
maVariable = <expression>
maVariable = maFonction() // ici "maFonction()" est une expression
maVariable = maFonction(5) // ici "5" est une expression, "maFonction(5)" aussi
maFonction() // ici "maFonction()" est une instruction, pas une expression
fonction(autreFonction()) // ici
IF (maVariable == VRAI)... // ici "maVariable == VRAI" est une expression
WHILE (maVariable)... // ici "maVariable" est une expression
```

1.6. Les instructions

En première approximation, on peut dire qu'une instruction est une ligne de code. Cependant, ce n'est pas QUE ça. Un bloc de code IF...THEN...ELSE par exemple est aussi une instruction, elle-même composée d'autres instructions.

Quand une instruction contient d'autres instructions, on parle de **bloc**. Les structures de contrôle contiennent plusieurs parties, dont des blocs.

Comme mentionné au début du document, une instruction peut être :

- Une **déclaration** (de programme, de procédure, de fonction ou de variable)
- Une **assignation de variable** (aussi appelée "affectation")
- Une **structure de contrôle**
- Un **appel de fonction**

Notez que mettre une variable toute seule sur une ligne n'est pas une instruction valide. On ne peut pas se contenter d'écrire le nom d'une variable toute seule sans opérateur car en algorithmique ça n'a aucun sens.

2. Les structures de contrôle

Pour répéter des instructions sans avoir à écrire le même code des milliers de fois, ou bien pour modifier le flot d'exécution du programme en fonction de la valeur d'une variable à un instant T, on utilise **des structures de contrôle**. Par exemple : IF/THEN/ELSE ou WHILE.

Il existe 4 catégories de structures de contrôle :

- Les **branchements conditionnels** (ou tests)
- Les **boucles** (qui incluent une condition de sortie)
- Les **itérations** (des boucles avec une variable auto-incrémentée)
- Les **sauts**.

Une condition est une **expression booléenne**, c'est-à-dire qui ne peut être que **TRUE** ou **FALSE**. On peut « chaîner » plusieurs conditions entre elles avec des opérateurs AND et OR. On peut aussi faire la négation d'une condition avec l'opérateur NOT.

Un branchement conditionnel est une instruction qui permet d'exécuter soit un bloc de code si la condition est vraie ou un autre bloc si la condition est fausse.

Exemple :

```
IF (<condition>) THEN
|
|-----><instructions_if_true> // any block of 1 or more instructions
|
ELSE
|
|-----><instructions_if_false> // any block of 1 or more instructions
|
END_IF
```

IMPORTANT : Toute expression booléenne est utilisable comme condition d'une boucle.

Une boucle est un bloc d'instructions qui se répétera tant que la condition sera vraie au moment où le programme la testera (la condition peut devenir fausse pendant l'exécution du bloc mais cela n'a aucune incidence tant que le bloc complet n'a pas été exécuté).

Exemples :

```
WHILE (<condition>) DO
|
|-----><instructions> // any bloc of 1 or more instructions to repeat
|                       // while <condition> == true
|
END WHILE

// Variant: the DO...WHILE loop executes the block BEFORE the condition is tested.
DO
|
|-----><instructions> // any bloc of 1 or more instructions executed at least once
|                       // (even if <condition> == false)
|
WHILE (<condition>)
```

Une itération est une boucle dans laquelle on compte le nombre de répétitions. Par abus de langage vous entendrez souvent « **première itération** » ou « **chaque itération** » au lieu de « première répétition » ou « chaque répétition ».

Exemple :

```
FOR (I FROM 1 TO 10) DO
|
|       <instructions> // any bloc of 1 or more instructions repeated 10 times
|
END FOR
```

3. Les structures de données

Une variable n'est qu'une boîte qui peut contenir n'importe quoi. Quand une variable est un nombre, on ne se pose pas (trop) la question de savoir comment obtenir sa valeur : il suffit de la référencer par son nom.

Exemple :

```
BEGIN PROGRAM

  DECLARE myNumber : INTEGER

  myNumber = 5

  Print(myNumber) // This code will display the value of myNumber : '5'

END PROGRAM
```

Par contre, lorsqu'on souhaite stocker **plusieurs valeurs dans une même variable**, on utilise une **structures de données**. Il en existe beaucoup mais la principale structure avec laquelle il est possible de tout faire est : le tableau !

Pour des questions pratiques je vais cependant faire une introduction sur les 3 structures suivantes :

- Les tableaux
- Les listes
- Les tables de paires clé/valeur

Ce qui caractérise une structure de données, c'est d'abord **son organisation interne (c'est à dire la façon dont sont rangées les données dans la mémoire de l'ordinateur)** puis les moyens par lesquels on accède à ses éléments.

- Pour accéder à l'élément « n » d'un tableau ou d'une liste, on utilise la notation entre crochets avec un nombre entier : **array[n]** ou **list[n]**.
- Pour les paires clé/valeur, on utilise la clé entre crochets, sachant que cette clé n'est pas obligatoirement un nombre mais peut être aussi une chaîne de caractères : **map[«prenom »]**.


Exemple de tableau


Index: 012345

Values:

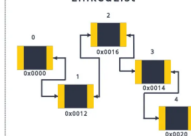
| | | | | | |
|----|----|---|---|----|---|
| 15 | 25 | 3 | 5 | 12 | 8 |
|----|----|---|---|----|---|

ArrayList





LinkedList



Exemple de paires clé/valeur

| KEY | VALUE |
|-----------|-------|
| Luke | true |
| Han | false |
| Chewbacca | false |
| Yoda | true |
| Leia | true |

3.1. Les tableaux (arrays)

Ce sont des structures **contiguës** à **1 dimension** et dont la taille est **invariable**. On peut imaginer un tableau comme une ligne de fichier Excel mais au lieu de référencer les cellules du tableau par le nom de la colonne, on les référence par le numéro de la colonne :

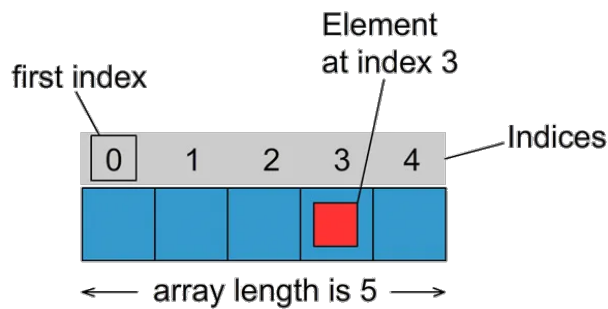


Figure 1: Illustration du concept de tableau

Exemple :

```
BEGIN PROGRAM

  DECLARE myArray : INTEGER[1] // Brackets « [1] » stands for "fixed size array of 1 elt"

  myArray[0] = 412           // Here we set the value of the 1st array element
  PRINT(myArray[0])         // This code will display '412'

  myArray = INTEGER[6]      // Here we create a new array with a size of '6'
  PRINT(myArray[0])         // This will display '0' !

  myArray[0] = 157          // Here we set the value of the 1st array element again
  PRINT(myArray[0])         // This will now display '157'

END PROGRAM
```

IMPORTANT : une fois qu'un tableau a été créé, on ne peut plus modifier sa taille. Pour retirer un élément et diminuer la taille d'un tableau, il faudra en créer un nouveau de taille inférieure et recopier un à un les éléments qu'on souhaite conserver. D'où l'intérêt des listes, qui sont en quelque sorte des tableaux dynamiques.

3.2. Les listes (*lists*)

En algorithmique, les listes n'existent pas ! Mais pour nos besoins et nos objectifs pédagogiques, on tolérera l'utilisation de listes. Ce sont des structures identiques aux tableaux, à ceci près qu'elles sont redimensionnables et donc plus simples à utiliser.

La différence principale entre une liste et un tableau, c'est que les éléments de la liste ne sont pas contigus en mémoire. C'est à dire qu'ils ne sont pas rangés dans des cases adjacentes. Deux éléments d'une liste qui se suivent ne sont pas nécessairement rangés l'un à côté de l'autre en mémoire. Nous verrons ceci plus en détails dans d'autre cours.

On se limitera aux opérations suivantes :

- Ajouter un élément (opération la plus courante)
- Retirer un élément
- Vider la liste

Exemple :

```
BEGIN PROGRAM

  DECLARE myList : INTEGER{}

  // Add number '124' at the end of the list
  myList = myList + {124}

  // Take off the number '512' (assuming it is in the list and wherever it is)
  myList = myList - {512}

  // Empty the list
  myList = {}

END PROGRAM
```

NOTE : En Java, l'addition ou la soustraction d'objets dans une liste n'est pas possible avec l'opérateur « + » (il faut passer par un appel de fonction).

3.3. Les tables de paires clé-valeur (*maps*)

Cette structure de données n'existe pas non plus en algorithmique mais peut s'avérer utile pour modéliser certaines parties de vos algorithmes durant vos projets.

Une paire clé valeur est souvent représentée comme ça : « key=value ». Le but d'une *map* est de stocker des propriétés et la valeur associée à chacune de ces propriétés.

Voici une *map* pour illustrer l'exemple de code ci-après :

| Clé | Valeur |
|--------------------|--------|
| nom | PICASO |
| nationalité | ES |
| année_de_naissance | 1881 |

Exemple :

```
BEGIN PROGRAM

// Reminder: "{}" mean "resizable" whereas "[]" mean "fixed-size"
DECLARE myMap : MAP{STRING,STRING}

// Add a new key-value pair
myMap = myMap + {"name", "PICASO"}
myMap = myMap + {"nationality", "ES"}
myMap = myMap + {"year_of_birth", "1881"}

// Modify the value associated to the "name" key (orthograph) :
myMap["name"] = "PICASSO"

// Remove the key-value "nationality=xxxxx"
myMap = myMap - {"nationality"}

END PROGRAM
```

4. Les fonctions

Une fonction permet de **réutiliser** un ensemble de lignes de code, simplement en appelant cette fonction.

En algorithmique, on déclare les fonctions et les procédures en premier car il faut qu'elles aient été déclarées pour qu'on puisse l'utiliser.

Une fonction peut accepter des paramètres. Dans ce cas on les déclare entre parenthèses après le nom de la fonction, de la même façon qu'on déclare une variable.

Exemple :

```
BEGIN PROGRAM

// -----
// Sum(INTEGER, INTEGER)
//
// This function performs the sum of both parameters.
// p1 : The first number
// p2 : The second number
//
// Returns : the sum of p1 + p2
// -----
FUNCTION Sum(p1 : INTEGER
             |         p2 : INTEGER)
|
|----->DECLARE someUselessVariable : STRING
|
|----->// This variable is not visible outside the function
|----->someUselessVariable = "This variable will be destroyed when the function returns"
|
|----->RETURN p1 + p2
|
END FUNCTION

DECLARE myVariable : INTEGER

// Call 'Sum()' with parameters '5' and '2' and store the result into 'myVariable' :
myVariable = Sum(5, 2)
PRINT( myVariable ) // Will display '7'

// You can also directly call the function inside the PRINT statement :
PRINT( Sum(100, 28) ) // Will display '128'

END PROGRAM
```

5. Quelques exercices pour s'entraîner

Exercice 1 : inverser les éléments d'un tableaux

1. Créer un tableau de N éléments, le remplir avec des entiers de 1 à N.
2. Puis inverser les valeurs du tableau dans une boucle.

Exercice 2 : agrandir un tableau

1. Créer un tableau de N éléments et le remplir avec des valeurs de 1 à N.
2. Créer un second tableau de taille N+1 et recopier les valeurs du premier tableau.
3. Mettre manuellement la dernière valeur à la valeur N+1.

Exercice 3 : trier un tableau

1. Créer le tableau suivant : [12, 16, 21, 4, 87, 30, 17].
2. Parcourir le tableau à la recherche du plus petit élément.
3. Le placer au début puis incrémenter la variable de parcours du tableau.
4. Recommencer à l'étape 2.

Quelques liens sur le net :

<https://info.blaisepascal.fr/pseudo-code> (vous verrez que la notation utilisée est légèrement différente)
<https://http://www.france-ioi.org/> (très bon entraînement)



Licence : Creative Commons CC BY-SA
© Alexandre DERMONT, janvier 2023

