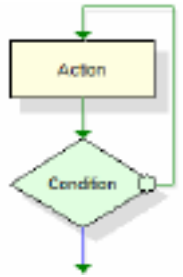


FORMATION AFPA
- DEVELOPPEUR LOGICIEL –

(NIVEAU III)

JJP

ALGORITHME ET PSEUDO-CODE

INTRODUCTION

Pourquoi apprendre l'algorithmique pour apprendre à programmer ? En quoi a-t-on besoin d'un langage spécial, distinct des langages de programmation compréhensibles par les ordinateurs ?

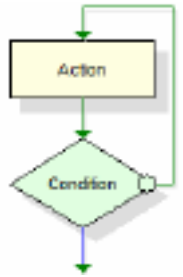
Parce que l'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage. Pour prendre une image, si un programme était une dissertation, l'algorithmique serait le plan, une fois mis de côté la rédaction et l'orthographe. Or, vous savez qu'il vaut mieux faire d'abord le plan et rédiger ensuite que l'inverse…

Apprendre l'algorithmique, c'est apprendre à manier la structure logique d'un programme informatique. Cette dimension est présente quelle que soit le langage de programmation ; mais lorsqu'on programme dans un langage (en C, en Visual Basic, etc.) on doit en plus se colleter les problèmes de syntaxe, ou de types d'instructions, propres à ce langage. Apprendre l'algorithmique de manière séparée, c'est donc sérier les difficultés pour mieux les vaincre.

A cela, il faut ajouter que des générations de programmeurs, souvent autodidactes ayant directement appris à programmer dans tel ou tel langage, ne font pas mentalement clairement la différence entre ce qui relève de la structure logique générale de toute programmation (les règles fondamentales de l'algorithmique) et ce qui relève du langage particulier qu'ils ont appris. Ces programmeurs, non seulement ont beaucoup plus de mal à passer ensuite à un langage différent, mais encore écrivent bien souvent des programmes qui même s'ils sont justes, restent laborieux. Car on n'ignore pas impunément les règles fondamentales de l'algorithmique… Alors, autant l'apprendre en tant que telle !

Avec quelles conventions écrit-on un algorithme ?

C'est pourquoi on utilise généralement une série de conventions appelée « pseudo-code », qui ressemble à un langage de programmation authentique dont on aurait évacué la plupart des problèmes de syntaxe. Ce pseudo-code est susceptible de varier légèrement d'un livre (ou d'un enseignant) à un autre. C'est bien normal : le pseudo-code, encore une fois, est purement conventionnel ; aucune machine n'est censée le reconnaître. Donc, chaque cuisinier peut faire sa sauce à sa guise, avec ses petites épices bien à lui, sans que cela prête à conséquence.



FORMATION AFPA
- DEVELOPPEUR LOGICIEL –

(NIVEAU III)

JJP

ALGORITHME ET PSEUDO-CODE

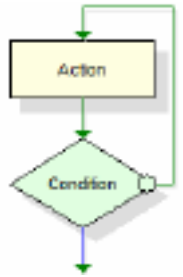
DEROULEMENT DU COURS

I.] ALGORIGRAMME ET ALGORITHME

1.) Définitions – Symboles - Structures
2.) Exercices

II.] ALGORITHME ET PSEUDO-CODE

1.) Structure générale d'un algorithme
2.) Les Variables
3.) Instruction d'affectation
4.) Expressions et opérateurs
5.) Lecture et Ecriture
6.) Les Tests
7.) Encore de la Logique
8.) Les Boucles
9.) Les Tableaux
10.) Techniques Rusées
11.) Tableaux Multidimensionnels
12.) Fonctions Prédéfinies
13.) Fichiers
14.) Procédures et Fonctions
15.) Notions Complémentaires



ALGORITHME ET PSEUDO-CODE

II.] ALGORITHME ET PSEUDO-CODE

1.) Structure générale d'un algorithme

Voici la structure générale d'un algorithme :

```
ALGORITHME  identifiant_algorithme

        <Partie Declarations>

DEBUT
        <Partie Instructions>
FIN
```

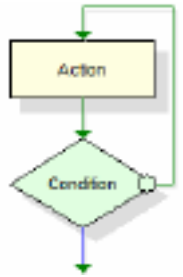
Exemple :

```
ALGORITHME  EXO_1

        Variable iCompteur en Numérique

DEBUT
        iCompteur ← 1
ou encore :
        iCompteur= 1
FIN
```

Comme de nombreux langages impératifs, il est composé de deux parties distinctes : les déclarations et les instructions.



FORMATION AFPA
- DEVELOPPEUR LOGICIEL –

(NIVEAU III)

JJP

ALGORITHME ET PSEUDO-CODE

2.) Les Variables

A quoi servent les variables ?

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de résultats obtenus par le programme, intermédiaires ou définitifs. Ces données peuvent être de plusieurs types (on en reparlera) : elles peuvent être des nombres, du texte, etc. Toujours est-il que **dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une variable.**

Pour employer une image, une variable est une boîte, que le programme (l'ordinateur) va repérer par une étiquette. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.

Déclaration des variables

La première chose à faire avant de pouvoir utiliser une variable est de créer la boîte et de lui coller une étiquette. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle la déclaration des variables.

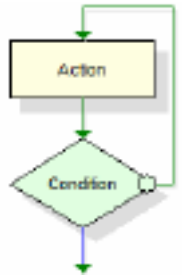
Le nom de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages. Toutefois, une règle absolue est qu'un nom de variable peut comporter des lettres et des chiffres, mais qu'il exclut la plupart des signes de ponctuation, en particulier les espaces. Un nom de variable correct commence également impérativement par une lettre.

Les noms de variables ne peuvent contenir que des caractères compris dans les intervalles suivants :

- 'a' .. 'z'
- 'A' .. 'Z'
- '0' .. '9'
- le caractère _ (souligné/underscore).

Pour construire un identifiant, il faudra respecter les règles suivantes :

- il ne peut en aucun cas commencer par un chiffre ;
- tout identifiant doit avoir été déclaré avant d'être utilisé ;
- un identifiant doit bien entendu être différent d'un mot clé, ceci pour éviter toute ambiguïté (par exemple : longueur);
- enfin, pour faciliter l'écriture et la lecture des algorithmes, il est très fortement conseillé d'utiliser des identifiants explicites (par exemple : largeur_image ou largeurImage et non xi). Idéalement, il est bien de pouvoir connaitre son type (et donc d'ajouter son type, par exemple : i_largeur_image ou iLargeImage)



FORMATION AFPA
- DEVELOPPEUR LOGICIEL –

(NIVEAU III)

JJP

ALGORITHME ET PSEUDO-CODE

Les types : types numériques classiques

Commençons par le cas très fréquent, celui d'une variable destinée à recevoir des nombres.
Le type de variable choisi pour un nombre va déterminer :

- les valeurs maximales et minimales des nombres pouvant être stockés dans la variable
- la précision de ces nombres (dans le cas de nombres décimaux).

Tous les langages, quels qu'ils soient offrent un « bouquet » de types numériques, dont le détail est susceptible de varier légèrement d'un langage à l'autre. Grosso modo, on retrouve cependant les types suivants :

Type Numérique	Plage
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	$-3{,}40\text{x}10^{38}$ à $-1{,}40\text{x}10^{45}$ pour les valeurs négatives $1{,}40\text{x}10^{-45}$ à $3{,}40\text{x}10^{38}$ pour les valeurs positives
Réel double	$1{,}79\text{x}10^{308}$ à $-4{,}94\text{x}10^{-324}$ pour les valeurs négatives $4{,}94\text{x}10^{-324}$ à $1{,}79\text{x}10^{308}$ pour les valeurs positives

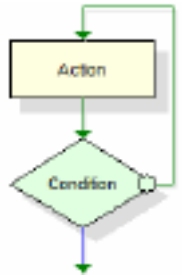
Pourquoi ne pas déclarer toutes les variables numériques en réel double, histoire de bétonner et d'être certain qu'il n'y aura pas de problème ? **En vertu du principe de l'économie de moyens !**

En algorithmique, on ne se tracassera pas trop avec les sous-types de variables numériques. On se contentera donc de préciser qu'il s'agit d'un nombre, en gardant en tête que dans un vrai langage, il faudra être plus précis.
En pseudo-code, une déclaration de variables aura ainsi cette tête :

Variable g **en Numérique**
ou encore
Variables PrixHT, TauxTVA, PrixTTC **en Numérique**

Même si nous n'emploierons pas ces types dans ce cours, certains langages autorisent d'autres types numériques :

- le type monétaire (avec strictement deux chiffres après la virgule)
- le type date (jour/mois/année).



ALGORITHME ET PSEUDO-CODE

Les types : type alphanumérique

Fort heureusement, les boîtes que sont les variables peuvent contenir bien d'autres informations que des nombres. Sans cela, on serait un peu embêté dès que l'on devrait stocker un nom de famille, par exemple. On dispose donc également du type alphanumérique.

Dans une variable de ce type, on stocke des caractères, qu'il s'agisse de lettres, de signes de ponctuation, d'espaces, ou même de chiffres. Quant au nombre maximal de caractères pouvant être stockés dans une seule variable string, cela peut dépendre du langage utilisé, mais surtout de la base de données (dans le cas où il y'en a une). Un groupe de caractères est souvent appelé chaîne de caractères.

En pseudo-code, une chaîne de caractères est toujours notée entre guillemets.

Les types : type booléen

Le dernier type de variables est le type booléen : on y stocke uniquement les valeurs logiques VRAI et FAUX (équivalent à TRUE et FALSE, 0 et 1, ou OUI et NON, ….)

Ce qui compte, c'est de comprendre que le type booléen est très économique en termes de place mémoire occupée, puisque pour stocker une telle information binaire, un seul bit suffit.

Note : le recours aux variables booléennes s'avère très souvent un puissant instrument de lisibilité des algorithmes. Il peut faciliter la vie de celui qui écrit l'algorithme, comme de celui qui le relit pour le corriger.

ALGORITHME ET PSEUDO-CODE

3.) Instruction d'affectation

Syntaxe et signification

La seule chose qu'on puisse faire avec une variable, c'est l'affecter, c'est-à-dire lui attribuer une valeur. Pour poursuivre la superbe métaphore filée déjà employée, on peut remplir la boîte.

En pseudo-code, l'instruction d'affectation se note avec le signe ← (tolérance pour « = ») :

```
Toto ← 24
```

Attribue la valeur 24 à la variable Toto.

Ceci, soit dit en passant, sous-entend impérativement que Toto soit une variable de type numérique. Si Toto a été défini dans un autre type, il faut bien comprendre que cette instruction provoquera une erreur.

On peut en revanche sans aucun problème attribuer à une variable la valeur d'une autre variable, telle quelle ou modifiée. Par exemple :

```
Tutu ← Toto
```

Signifie que la valeur de Tutu est maintenant celle de Toto.
Notez bien que cette instruction n'a en rien modifié la valeur de Toto : **une instruction d'affectation ne modifie que ce qui est situé à gauche de la flèche.**

```
Tutu ← Toto + 4
```

Si Toto contenait 12, Tutu vaut maintenant 16.

```
Tutu ← Tutu + 1
```

Si Tutu valait 6, il vaut maintenant 7. La valeur de Tutu est modifiée, puisque Tutu est la variable située à gauche de la flèche.

ALGORITHME ET PSEUDO-CODE

Pour revenir à présent sur le rôle des guillemets dans les chaînes de caractères et sur la confusion numéro 2 signalée plus haut, comparons maintenant deux algorithmes suivants :

Exemple n°1

Début

Riri ← "Loulou"

Fifi ← "Riri"

Fin

Exemple n°2

Début

Riri ← "Loulou"

Fifi ← Riri

Fin

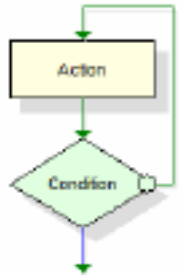
La seule différence entre les deux algorithmes consiste dans la présence ou dans l'absence des guillemets lors de la seconde affectation. Et l'on voit que cela change tout !

Dans l'exemple n°1, ce que l'on affecte à la variable Fifi, c'est la suite de caractères R – i – r - i. Et à la fin de l'algorithme, le contenu de la variable Fifi est donc « Riri ».

Dans l'exemple n°2, en revanche, Riri étant dépourvu de guillemets, n'est pas considéré comme une suite de caractères, mais comme un nom de variable. Le sens de la ligne devient donc : « affecte à la variable Fifi le contenu de la variable Riri ». A la fin de l'algorithme n°2, la valeur de la variable Fifi est donc « Loulou ». Ici, l'oubli des guillemets conduit certes à un résultat, mais à un résultat différent.

A noter, car c'est un cas très fréquent, que généralement, lorsqu'on oublie les guillemets lors d'une affectation de chaîne, ce qui se trouve à droite du signe d'affectation ne correspond à aucune variable précédemment déclarée et affectée. **Dans ce cas, l'oubli des guillemets se solde immédiatement par une erreur d'exécution.**

Ceci est une simple illustration. Mais elle résume l'ensemble des problèmes qui surviennent lorsqu'on oublie la règle des guillemets aux chaînes de caractères.



ALGORITHME ET PSEUDO-CODE

Ordre des instructions

Il va de soi que l'ordre dans lequel les instructions sont écrites va jouer un rôle essentiel dans le résultat final. Considérons les deux algorithmes suivants :

Exemple 1

Variable A **en Numérique**
Début
A ← 34
A ← 12
Fin

Exemple 2

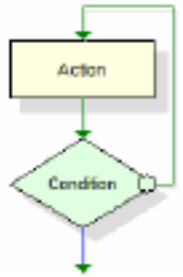
Variable A **en Numérique**
Début
A ← 12
A ← 34
Fin

Il est clair que dans le premier cas la valeur finale de A est 12, dans l'autre elle est 34 .

Il est tout aussi clair que ceci ne doit pas nous étonner : lorsqu'on indique le chemin à quelqu'un, dire « prenez tout droit sur 1km, puis à droite » n'envoie pas les gens au même endroit que si l'on dit « prenez à droite puis tout droit pendant 1 km ».

Enfin, il est également clair que si l'on met de côté leur vertu pédagogique, les deux algorithmes ci-dessus sont parfaitement idiots ; à tout le moins ils contiennent une incohérence. Il n'y a aucun intérêt à affecter une variable pour l'affecter différemment juste après. En l'occurrence, on aurait tout aussi bien atteint le même résultat en écrivant simplement :

FAIRE LES EXERCICES : 1.1 à 1.7



ALGORITHME ET PSEUDO-CODE

4.) Expressions et opérateurs

Préambule

Continuons sur l'affectation. Une condition nécessaire de validité d'une instruction d'affectation est que l'expression située à droite de la flèche soit du même type que la variable située à gauche.
C'est très logique : on ne peut pas ranger convenablement des outils dans un sac à provision, ni des légumes dans une trousse à outils… sauf à provoquer un résultat catastrophique.

Si tout cela n'est pas respecté, la machine sera incapable d'exécuter l'affectation, et déclenchera une erreur.

On va maintenant détailler ce que l'on entend par le terme d' opérateur.

Un opérateur est un signe qui relie deux valeurs, pour produire un résultat.

Dans le chapitre suivant, nous allons étudier les opérateurs possibles dépendent du type des valeurs.

Opérateurs numériques

Ce sont les quatre opérations arithmétiques tout ce qu'il y a de classique.

+ : addition
- : soustraction
* : multiplication
/ : division

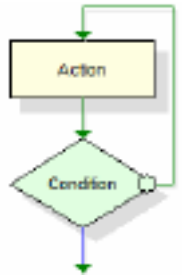
Mentionnons également le ^ qui signifie « puissance ». 45 au carré s'écrira donc 45 ^ 2 :

^ : puissance (ou racine)

Enfin, on a le droit d'utiliser les parenthèses, avec les mêmes règles qu'en mathématiques. La multiplication et la division ont « naturellement » priorité sur l'addition et la soustraction. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle.

Cela signifie qu'en informatique, 12 * 3 + 5 et (12 * 3) + 5 valent strictement la même chose, à savoir 41. Pourquoi dès lors se fatiguer à mettre des parenthèses inutiles ?

En revanche, 12 * (3 + 5) vaut 12 * 8 soit 96.



ALGORITHME ET PSEUDO-CODE

Opérateur alphanumérique : &

Cet opérateur permet de concaténer, autrement dit d'agglomérer, deux chaînes de caractères. Par exemple :

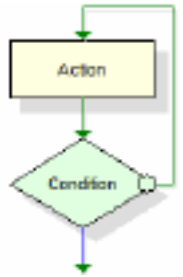
```
Variables A, B, C en Caractère
Début
A ← "Gloubi"
B ← "Boulga"
C ← A & B
Fin
```

La valeur de C à la fin de l'algorithme est "GloubiBoulga".
Suivant les langages, et suivant le type de variables, on peut écrire ceci :

```
Variables A, B, C en Caractère
Début
A ← "Gloubi"
B ← "Boulga"
C ← A + B (aussi équivalent « A . B »)
Fin
```

Opérateurs logiques (ou booléens)

Il s'agit du ET, du OU, du NON (du FALSE OU TRUE, de l'état 0 ou 1).



ALGORITHME ET PSEUDO-CODE

2 remarques pour conclure

Deux remarques pour terminer :

Maintenant que nous sommes familiers des variables et que nous les manipulons correctement, j'attire votre attention sur la trompeuse similitude de vocabulaire entre les mathématiques et l'informatique. En mathématiques, une « variable » est généralement une inconnue, qui recouvre un nombre non précisé de valeurs. Lorsque j'écris :
$y = 3x + 2$

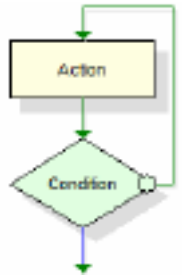
les « variables » x et y satisfaisant à l'équation existent en nombre infini (graphiquement, l'ensemble des solutions à cette équation dessine une droite). Lorsque j'écris :
$ax^2 + bx + c = 0$

la « variable » x désigne les solutions à cette équation, c'est-à-dire zéro, une ou deux valeurs à la fois…

En informatique, une variable possède à un moment donné une valeur et une seule. A la rigueur, elle peut ne pas avoir de valeur du tout (une fois qu'elle a été déclarée, et tant qu'on ne l'a pas affectée. A signaler que dans certains langages, les variables non encore affectées sont considérées comme valant automatiquement zéro). Mais ce qui est important, c'est que cette valeur justement, ne « varie » pas à proprement parler. Du moins ne varie-t-elle que lorsqu'elle est l'objet d'une instruction d'affectation.

La deuxième remarque concerne le signe de l'affectation. En algorithmique, comme on l'a vu, c'est le signe ←. Mais en pratique, la quasi totalité des langages emploient le signe égal. Et là, pour les débutants, la confusion avec les maths est également facile. En maths, A = B et B = A sont deux propositions strictement équivalentes. En informatique, absolument pas, puisque cela revient à écrire A ← B et B ← A, deux choses bien différentes. De même, A = A + 1, qui en mathématiques, constitue une équation sans solution, représente en programmation une action tout à fait licite (et de surcroît extrêmement courante). Donc, attention ! ! ! La meilleure des vaccinations contre cette confusion consiste à bien employer le signe ← en pseudo-code, signe qui a le mérite de ne pas laisser place à l'ambiguïté. Une fois acquis les bons réflexes avec ce signe, vous n'aurez plus aucune difficulté à passer au = des langages de programmation.

FAIRE LES EXERCICES : 1.8 à 1.9



ALGORITHME ET PSEUDO-CODE

5.) Lecture et Ecriture

Préambule

Imaginons que nous ayons fait un programme pour calculer le carré d'un nombre, mettons 12. Si on a fait au plus simple, on a écrit un truc du genre :

Variable A **en Numérique**
Début
A ← 12^2
Fin

D'une part, ce programme nous donne le carré de 12. Mais si l'on veut le carré d'un autre nombre que 12, il faut réécrire le programme. Bof.

D'autre part, le résultat est calculé par la machine. Mais elle le garde soigneusement pour elle, et le pauvre utilisateur qui fait exécuter ce programme, lui, ne saura jamais quel est le carré de 12. Re-bof.

C'est pourquoi, il existe des d'instructions pour permettre à la machine de dialoguer avec l'utilisateur.

Dans un sens, ces instructions permettent à l'utilisateur de rentrer des valeurs au clavier pour qu'elles soient utilisées par le programme. **Cette opération est la lecture.**

Dans l'autre sens, d'autres instructions permettent au programme de communiquer des valeurs à l'utilisateur en les affichant à l'écran. **Cette opération est l'écriture.**

Remarque essentielle : Un algorithme, c'est une suite d'instructions qui programme la machine, pas l'utilisateur ! Donc quand on dit à la machine de lire une valeur, cela implique que l'utilisateur va devoir écrire cette valeur. Et quand on demande à la machine d'écrire une valeur, c'est pour que l'utilisateur puisse la lire. Lecture et écriture sont donc des termes qui comme toujours en programmation, doivent être compris du point de vue de la machine qui sera chargée de les exécuter.

Contrairement à l'algorithme, où l'on utilisait l'instruction « AFFICHER » afin de simplifier la compréhension de la personne non informaticienne, nous allons dorénavant adapter notre vocabulaire à la machine et au développeur.

ALGORITHME ET PSEUDO-CODE

Les instructions de lecture et d'écriture

Tout bêtement, pour que l'utilisateur entre la (nouvelle) valeur de Titi, on mettra :

```
Lire Titi
```

Dès que le programme rencontre une instruction Lire, l'exécution s'interrompt, attendant la frappe d'une valeur au clavier

Dès lors, aussitôt que la touche Entrée (Enter) a été frappée, l'exécution reprend. Dans le sens inverse, pour écrire quelque chose à l'écran, c'est aussi simple que :

```
Ecrire Toto
```

Avant de Lire une variable, il est très fortement conseillé d'écrire des **libellés** à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper (sinon, le pauvre utilisateur passe son temps à se demander ce que l'ordinateur attend de lui) :

```
Ecrire "Entrez votre nom : "
Lire NomFamille
```

Lecture et Ecriture sont des instructions algorithmiques qui ne présentent pas de difficultés particulières, une fois qu'on a bien assimilé ce problème du sens du dialogue (homme → machine, ou machine ← homme).

FAIRE LES EXERCICES : 2.1 à 2.4

ALGORITHME ET PSEUDO-CODE

6.) Les tests

Cette structure logique répond au nom de **test**. Toutefois, ceux qui tiennent absolument à briller en société parleront également de **structure alternative**.

Structure d'un test

Il n'y a que **deux formes possibles** pour un test ; la première est la plus simple, la seconde la plus complexe.

Si booléen **Alors**
  Instructions
Fin si

Si booléen **Alors**
  Instructions 1
Sinon
  Instructions 2
Fin si

Ceci appelle quelques explications.

Un **booléen** est une **expression** dont la valeur est VRAI ou FAUX. Cela peut donc être (il n'y a que deux possibilités) :

- une **variable** (ou une expression) de type booléen
- une **condition**

Nous reviendrons dans quelques instants sur ce qu'est une **condition** en informatique.

Toujours est-il que la structure d'un test est relativement claire. Dans la forme la plus simple, arrivé à la première ligne (Si… Alors) la machine examine la valeur du booléen. Si ce booléen a pour valeur VRAI, elle exécute la série d'instructions. Cette série d'instructions peut être très brève comme très longue, cela n'a aucune importance. En revanche, dans le cas où le booléen est faux, l'ordinateur saute directement aux instructions situées après le **Fin Si**.

Dans le cas de la structure complète, c'est à peine plus compliqué. Dans le cas où le booléen est VRAI, et après avoir exécuté la série d'instructions 1, au moment où elle arrive au mot « Sinon », la machine saute directement à la première instruction située après le « **Fin si** ». De même, au cas où le booléen a comme valeur « Faux », la machine

saute directement à la première ligne située après le « Sinon » et exécute l'ensemble des « instructions 2 ». Dans tous les cas, les instructions situées juste après le **Fin Si** seront exécutées normalement.

En fait, la forme simplifiée correspond au cas où l'une des deux « branches » du Si est vide. Dès lors, plutôt qu'écrire « sinon ne rien faire du tout », il est plus simple de ne rien écrire. Et laisser un Si... complet, avec une des deux branches vides, est considéré comme une très grosse maladresse pour un programmeur, même si cela ne constitue pas à proprement parler une faute.

Qu'est ce qu'une condition ?

Une condition est une comparaison.

Cette définition est essentielle ! Elle signifie qu'une condition est composée de trois éléments :

- une valeur
- un **opérateur de comparaison**
- une autre valeur

Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères…). Mais si l'on veut que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type !

Les **opérateurs de comparaison** sont :

- égal à…
- différent de…
- strictement plus petit que…
- strictement plus grand que…
- plus petit ou égal à…
- plus grand ou égal à…

L'ensemble des trois éléments composant la condition constitue donc, si l'on veut, une affirmation, qui à un moment donné est VRAIE ou FAUSSE.

À noter que ces opérateurs de comparaison peuvent tout à fait s'employer avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique (voir le code ASCII pour information complémentaire), les majuscules étant systématiquement placées avant les minuscules.

ALGORITHME ET PSEUDO-CODE

Ainsi on a :

“t” < “w”	VRAI
“Maman” > “Papa“	FAUX
“maman” > “Papa”	VRAI

Remarque très importante

En formulant une condition dans un algorithme, il faut se méfier comme de la peste de certains raccourcis du langage courant, ou de certaines notations valides en mathématiques, mais qui mènent à des non-sens informatiques. Prenons par exemple la phrase « Toto est compris entre 5 et 8 ». On peut être tenté de la traduire par : **5 < Toto < 8**
Or, une telle expression, qui a du sens en français, comme en mathématiques, **ne veut rien dire en programmation**. En effet, elle comprend deux opérateurs de comparaison, soit un de trop, et trois valeurs, soit là aussi une de trop. On va voir dans un instant comment traduire convenablement une telle condition.

FAIRE L’EXERCICE : 3.1

Conditions composées

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus. Reprenons le cas « Toto est inclus entre 5 et 8 ». En fait cette phrase cache non une, mais **deux** conditions. Car elle revient à dire que « Toto est supérieur à 5 et Toto est inférieur à 8 ». Il y a donc bien là deux conditions, reliées par ce qu’on appelle un **opérateur logique**, le mot ET.
L’informatique met à notre disposition quatre opérateurs logiques : ET, OU, NON, et XOR.

- Le ET a le même sens en informatique que dans le langage courant. Pour que "Condition1 ET Condition2" soit VRAI, il faut impérativement que Condition1 soit VRAI et que Condition2 soit VRAI. Dans tous les autres cas, "Condition 1 et Condition2" sera faux.

ALGORITHME ET PSEUDO-CODE

- Il faut se méfier un peu plus du OU. Pour que "Condition1 OU Condition2" soit VRAI, il suffit que Condition1 soit VRAIE ou que Condition2 soit VRAIE. Le point important est que si Condition1 est VRAIE et que Condition2 est VRAIE aussi, Condition1 OU Condition2 reste VRAIE. Le OU informatique ne veut donc pas dire « ou bien »
- Le XOR (ou OU exclusif) fonctionne de la manière suivante. Pour que "Condition1 XOR Condition2" soit VRAI, il faut que soit Condition1 soit VRAI, soit que Condition2 soit VRAI. Si toutes les deux sont fausses, ou que toutes les deux sont VRAI, alors le résultat global est considéré comme FAUX. Le XOR est donc l'équivalent du "ou bien" du langage courant.

J'insiste toutefois sur le fait que le XOR est une rareté, dont il n'est pas strictement indispensable de s'encombrer en programmation.

- Enfin, le NON inverse une condition : NON(Condition1)est VRAI si Condition1 est FAUX, et il sera FAUX si Condition1 est VRAI. C'est l'équivalent pour les booléens du signe "moins" que l'on place devant les nombres.

Alors, vous vous demandez peut-être à quoi sert ce NON. Après tout, plutôt qu'écrire NON(Prix > 20), il serait plus simple d'écrire tout bonnement Prix=<20. Dans ce cas précis, c'est évident qu'on se complique inutilement la vie avec le NON. Mais si le NON n'est jamais indispensable, il y a tout de même des situations dans lesquelles il s'avère bien utile.

On représente fréquemment tout ceci dans des **tables de vérité** (C1 et C2 représentent deux conditions, et on envisage à chaque fois les quatre cas possibles)

C1 et C2	C2 Vrai	C2 Faux
C1 Vrai	Vrai	Faux
C1 Faux	Faux	Faux

C1 ou C2	C2 Vrai	C2 Faux
C1 Vrai	Vrai	Vrai
C1 Faux	Vrai	Faux

C1 xor C2	C2 Vrai	C2 Faux
C1 Vrai	Faux	Vrai
C1 Faux	Vrai	Faux

ALGORITHME ET PSEUDO-CODE

Non C1	
C1 Vrai	Faux
C1 Faux	Vrai

A EVITER :

C'est de formuler dans un test **une condition qui ne pourra jamais être vraie, ou jamais être fausse**. Si ce n'est pas fait exprès, c'est assez rigolo. Si c'est fait exprès, c'est encore plus drôle, car une condition dont on sait d'avance qu'elle sera toujours fausse n'est pas une condition. Dans tous les cas, cela veut dire qu'on a écrit un test qui n'en est pas un, et qui fonctionne comme s'il n'y en avait pas.
Cela peut être par exemple : Si Toto < 10 ET Toto > 15 Alors… (il est très difficile de trouver un nombre qui soit à la fois inférieur à 10 et supérieur à 15 !)

FAIRE L'EXERCICE : 3.2 à 3.3

Tests imbriqués

Graphiquement, on peut très facilement représenter un SI comme un aiguillage de chemin de fer. Un SI ouvre donc deux voies, correspondant à deux traitements différents. Mais il y a des tas de situations où deux voies ne suffisent pas. Par exemple, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles (solide, liquide ou gazeuse).

ALGORITHME ET PSEUDO-CODE

Une première solution serait la suivante :

```
        Variable Temp en Entier
Début
        Ecrire "Entrez la température de l'eau :"
        Lire Temp
        Si Temp =< 0 Alors
                Ecrire "C'est de la glace"
        Fin Si
        Si Temp > 0 Et Temp < 100 Alors
                Ecrire "C'est du liquide"
        Fin si
        Si Temp > 100 Alors
                Ecrire "C'est de la vapeur"
        Fin si
Fin
```

Vous constaterez que c'est un peu laborieux. Les conditions se ressemblent plus ou moins, et surtout on oblige la machine à examiner trois tests successifs alors que tous portent sur une même chose, la température de l'eau (la valeur de la variable Temp). Il serait ainsi bien plus rationnel d'**imbriquer** les tests de cette manière :

```
        Variable Temp en Entier
Début
        Ecrire "Entrez la température de l'eau :"
        Lire Temp
        Si Temp =< 0 Alors
                Ecrire "C'est de la glace"
        Sinon
                Si Temp < 100 Alors
                        Ecrire "C'est du liquide"
                Sinon
                        Ecrire "C'est de la vapeur"
                Fin si
        Fin si
Fin
```

ALGORITHME ET PSEUDO-CODE

Nous avons fait des économies : au lieu de devoir taper trois conditions, dont une composée, nous n'avons plus que deux conditions simples. Mais aussi, et surtout, nous avons fait des économies sur le temps d'exécution de l'ordinateur. Si la température est inférieure à zéro, celui-ci écrit dorénavant « C'est de la glace » et passe **directement** à la fin, sans être ralenti par l'examen d'autres possibilités (qui sont forcément fausses).

Cette deuxième version n'est donc pas seulement plus simple à écrire et plus lisible, elle est également plus performante à l'exécution.

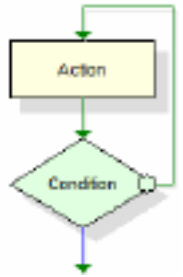
Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

A noter que l'on peut procéder ainsi :

Dans le cas de tests imbriqués, le Sinon et le Si peuvent être fusionnés en un SinonSi. On considère alors qu'il s'agit d'un seul bloc de test, conclu par un seul FinSi

Cela donnera ceci pour notre exemple précédent :

```
        Variable Temp en Entier
Début
        Ecrire "Entrez la température de l'eau :"
        Lire Temp
        Si Temp =< 0 Alors
                Ecrire "C'est de la glace"
        SinonSi Temp < 100 Alors
                Ecrire "C'est du liquide"
        Sinon
                Ecrire "C'est de la vapeur"
        Fin si
Fin
```



ALGORITHME ET PSEUDO-CODE

Variables Booléennes

Jusqu'ici, pour écrire nos tests, nous avons utilisé uniquement des **conditions**. Mais vous vous rappelez qu'il existe un type de variables (les booléennes) susceptibles de stocker les valeurs VRAI ou FAUX. En fait, on peut donc entrer des conditions dans ces variables, et tester ensuite la valeur de ces variables.

Reprenons l'exemple de l'eau. On pourrait le réécrire ainsi :

```
        Variable Temp en Entier
        Variables A, B en Booléen
Début
        Ecrire "Entrez la température de l'eau :"
        Lire Temp
        A ← Temp =< 0
        B ← Temp < 100
        Si A Alors
                Ecrire "C'est de la glace"
        SinonSi B Alors
                Ecrire "C'est du liquide"
        Sinon
                Ecrire "C'est de la vapeur"
        Fin si
Fin
```

A priori, cette technique ne présente guère d'intérêt : on a alourdi plutôt qu'allégé l'algorithme de départ, en ayant recours à deux variables supplémentaires.

- Mais souvenons-nous : une variable booléenne n'a besoin que d'un seul bit pour être stockée. De ce point de vue, l'alourdissement n'est donc pas considérable.
- dans certains cas, notamment celui de conditions composées très lourdes (avec plein de ET et de OU tout partout) cette technique peut faciliter le travail du programmeur, en améliorant nettement la lisibilité de l'algorithme. Les variables booléennes peuvent également s'avérer très utiles pour servir de **flag**, technique dont on reparlera plus loin.

FAIRE L'EXERCICE : 3.4 à 3.6

ALGORITHME ET PSEUDO-CODE

7.) Encore de la logique

Faut-il mettre un ET ? Faut-il mettre un OU ?

Une remarque pour commencer : dans le cas de conditions composées, les parenthèses jouent un rôle fondamental.

```
        Variables A, B, C, D, E en Booléen
        Variable X en Entier
Début
        Lire X
        A ← X > 12
        B ← X > 2
        C ← X < 6
        D ← (A ET B) OU C
        E ← A ET (B OU C)
Ecrire D, E
Fin
```

Si X = 3, alors on remarque que D sera VRAI alors que E sera FAUX.

S'il n'y a dans une condition que des ET, ou que des OU, en revanche, les parenthèses ne changent strictement rien.

Dans une condition composée employant à la fois des opérateurs ET et des opérateurs OU, la présence de parenthèses possède une influence sur le résultat, tout comme dans le cas d'une expression numérique comportant des multiplications et des additions.

On en arrive à une autre propriété des ET et des OU : on pense souvent que ET et OU s'excluent mutuellement, au sens où un problème donné s'exprime soit avec un ET, soit avec un OU. Pourtant, ce n'est pas si évident.

Quand faut-il ouvrir la fenêtre de la salle ? Uniquement si les conditions l'imposent, à savoir :

```
Si il fait trop chaud ET il ne pleut pas Alors
        Ouvrir la fenêtre
Sinon
        Fermer la fenêtre
Finsi
```

ALGORITHME ET PSEUDO-CODE

Cette petite règle pourrait tout aussi bien être formulée comme suit :

Si il ne fait pas trop chaud OU il pleut **Alors**
    Fermer la fenêtre
Sinon
    Ouvrir la fenêtre
Fin si

Ces deux formulations sont strictement équivalentes. Ce qui nous amène à la conclusion suivante :

Toute structure de test requérant une condition composée faisant intervenir l'opérateur ET peut être exprimée de manière équivalente avec un opérateur OU, et réciproquement.

Ceci est moins surprenant qu'il n'y paraît au premier abord. Jetez pour vous en convaincre un œil sur les tables (CA ET C2, C1 OU C2, C1 XOR C2, N C1) si bien être exprimée avec l'autre table (l'autre opérateur logique). Toute l'astuce consiste à savoir effectuer correctement ce passage.

Bien sûr, on ne peut pas se contenter de remplacer purement et simplement les ET par des OU ; ce serait un peu facile. La règle d'équivalence est la suivante (on peut la vérifier sur l'exemple de la fenêtre) :

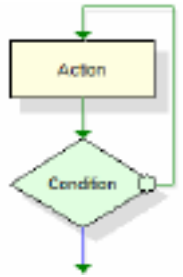
Si A ET B **Alors**
    Instructions 1
Sinon
    Instructions 2
Fin si

équivaut à :

Si NON A OU NON B **Alors**
    Instructions 2
Sinon
    Instructions 1
Fin si

Cette règle porte le nom de **transformation de Morgan**, du nom du mathématicien anglais qui l'a formulée.

FAIRE L'EXERCICE : 4.1 à 4.5



ALGORITHME ET PSEUDO-CODE

Au delà de la logique : le style

Il n'y a jamais une seule manière juste de traiter les structures alternatives. Et plus généralement, il n'y a jamais une seule manière juste de traiter un problème. Entre les différentes possibilités, qui ne sont parfois pas meilleures les unes que les autres, le choix est une affaire de **style**.

C'est pour cela qu'avec l'habitude, on reconnaît le style d'un programmeur aussi sûrement que s'il s'agissait de style littéraire.

Reprenons nos opérateurs de comparaison maintenant familiers, le ET et le OU. En fait, on s'aperçoit que l'on pourrait tout à fait s'en passer ! Par exemple, pour reprendre l'exemple de la fenêtre de la salle :

```
Si il fait trop chaud ET il ne pleut pas Alors
        Ouvrir la fenêtre
Sinon
        Fermer la fenêtre
Fin si
```

Possède un parfait équivalent algorithmique sous la forme de :

```
Si il fait trop chaud Alors
        Si il ne pleut pas Alors
                Ouvrir la fenêtre
        Sinon
                Fermer la fenêtre
        Fin si
Sinon
        Fermer la fenêtre
Fin si
```

Dans cette dernière formulation, nous n'avons plus recours à une condition composée (mais au prix d'un test imbriqué supplémentaire)

Et comme tout ce qui s'exprime par un ET peut aussi être exprimé par un OU, nous en concluons que le OU peut également être remplacé par un test imbriqué supplémentaire. On peut ainsi poser cette règle stylistique générale :

Dans une structure alternative complexe, les conditions composées, l'imbrication des structures de tests et l'emploi des variables booléennes ouvrent la possibilité de choix stylistiques différents. L'alourdissement des conditions allège les structures de tests et le nombre des booléens nécessaires ; l'emploi de booléens supplémentaires permet d'alléger les conditions et les structures de tests, et ainsi de suite.

FAIRE L'EXERCICE : 4.6 à 4.8

Si vous avez compris ce qui précède, et que l'exercice de la date ne vous pose plus aucun problème, alors vous savez tout ce qu'il y a à savoir sur les tests pour affronter n'importe quelle situation.

ALGORITHME ET PSEUDO-CODE

8.) Les boucles

Préambule

On peut retrouver plusieurs appellations : les **boucles**, ou les **structures répétitives**, ou les **structures itératives**.

A quoi les bouclent servent ?

Prenons le cas d'une saisie au clavier (une lecture), où par exemple, le programme pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Mais tôt ou tard, l'utilisateur, maladroit, risque de taper autre chose que la réponse attendue. Dès lors, le programme peut planter soit par une erreur d'exécution (parce que le type de réponse ne correspond pas au type de la variable attendu) soit par une erreur fonctionnelle (il se déroule normalement jusqu'au bout, mais en produisant des résultats fantaisistes).

Alors, dans tout programme un tant soit peu sérieux, on met en place ce qu'on appelle un **contrôle de saisie**, afin de vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme.

Voyons voir ce que ça donne avec un SI :

```
        Variable Rep en Caractère
Début
        Ecrire "Voulez vous un café ? (O/N)"
        Lire Rep
        Si Rep <> "O" et Rep <> "N" Alors
                Ecrire "Saisie erronnée. Recommencez"
                Lire Rep
        Fin Si
Fin
```

C'est impeccable. Du moins tant que l'utilisateur a le bon goût de ne se tromper qu'une seule fois, et d'entrer une valeur correcte à la deuxième demande. Si l'on veut également bétonner en cas de deuxième erreur, il faudrait rajouter un SI. Et ainsi de suite, on peut rajouter des centaines de SI, et écrire un algorithme aussi lourd, on n'en sortira pas, il y aura toujours moyen qu'un acharné vous mette le programme en erreur.

ALGORITHME ET PSEUDO-CODE

La solution consistant à aligner des SI… en pagaille est donc une impasse. La seule issue est donc de flanquer une **structure de boucle**, qui se présente ainsi :

```
TantQue booléen
  …
  Instructions
  …
FinTantQue
```

Le principe est simple : le programme arrive sur la ligne du TantQue. Il examine alors la valeur du booléen (qui, peut être une condition). Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne FinTantQue. Il retourne ensuite sur la ligne du TantQue, procède au même examen, et ainsi de suite. La boucle ne s'arrête que lorsque le booléen prend la valeur FAUX.

Illustration avec notre problème de contrôle de saisie. Une première approximation de la solution consiste à écrire :

```
        Variable Rep en Alphanumerique
Début
        Ecrire "Voulez vous un café ? (O/N)"
        TantQue Rep <> "O" et Rep <> "N"
                Lire Rep
        FinTantQue
Fin
```

Le principal défaut de ce squelette d'algorithme est de provoquer une erreur à chaque exécution. En effet, l'expression booléenne qui figure après le TantQue interroge la valeur de la variable Rep. Malheureusement, cette variable, si elle a été déclarée, n'a pas été affectée avant l'entrée dans la boucle. On teste donc une variable qui n'a pas de valeur, ce qui provoque une erreur et l'arrêt immédiat de l'exécution. Pour éviter ceci, on n'a pas le choix : il faut que la variable Rep ait déjà été affectée avant qu'on en arrive au premier tour de boucle. Pour cela, on peut faire une première lecture de Rep avant la boucle. Dans ce cas, celle-ci ne servira qu'en cas de mauvaise saisie lors de cette première lecture. L'algorithme devient alors :

```
        Variable Rep en Caractère (ou string ou alphanumerique)
Début
        Ecrire "Voulez vous un café ? (O/N)"
        Lire Rep
        TantQue Rep <> "O" et Rep <> "N"
                Lire Rep
        FinTantQue
Fin
```

Une autre possibilité, fréquemment employée, consiste à ne pas lire, mais à affecter arbitrairement la variable avant la boucle. Arbitrairement ? Pas tout à fait, puisque cette affectation doit avoir pour résultat de provoquer l'entrée obligatoire dans la boucle. L'affectation doit donc faire en sorte que le booléen soit mis à VRAI pour déclencher le

premier tour de la boucle. Dans notre exemple, on peut donc affecter Rep avec n'importe quelle valeur, hormis « O » et « N » : car dans ce cas, l'exécution sauterait la boucle, et Rep ne serait pas du tout lue au clavier. Cela donnera par exemple :

```
        Variable Rep en Caractère
Début
        Rep ← "X"
        Ecrire "Voulez vous un café ? (O/N)"
        TantQue Rep <> "O" et Rep <> "N"
                Lire Rep
        FinTantQue
Fin
```

Cette manière de procéder est à connaître, car elle est employée très fréquemment.

Il faut remarquer que les deux solutions (lecture initiale de Rep en dehors de la boucle ou affectation de Rep) rendent toutes deux l'algorithme satisfaisant, mais présentent une différence assez importante dans leur structure logique.

En effet, si l'on choisit d'effectuer une lecture préalable de Rep, la boucle ultérieure sera exécutée uniquement dans l'hypothèse d'une mauvaise saisie initiale. Si l'utilisateur saisit une valeur correcte à la première demande de Rep, l'algorithme passera sur la boucle sans entrer dedans.

En revanche, avec la deuxième solution (celle d'une affectation préalable de Rep), l'entrée de la boucle est forcée, et l'exécution de celle-ci, au moins une fois, est rendue obligatoire à chaque exécution du programme. Du point de vue de l'utilisateur, cette différence est tout à fait mineure ; et à la limite, il ne la remarquera même pas. Mais du point de vue du programmeur, il importe de bien comprendre que les cheminements des instructions ne seront pas les mêmes dans un cas et dans l'autre.

Pour terminer, remarquons que nous pourrions peaufiner nos solutions en ajoutant des affichages de libellés qui font encore un peu défaut. Ainsi, si l'on est un programmeur zélé, la première solution (celle qui inclut deux lectures de Rep, une en dehors de la boucle, l'autre à l'intérieur) pourrait devenir :

```
        Variable Rep en Caractère
Début
        Ecrire "Voulez vous un café ? (O/N)"
        Lire Rep
        TantQue Rep <> "O" et Rep <> "N"
                Ecrire "Vous devez répondre par O ou N. Recommencez"
                Lire Rep
        FinTantQue
        Ecrire "Saisie acceptée"
Fin
```

ALGORITHME ET PSEUDO-CODE

Quant à la deuxième solution, elle pourra devenir :

```
        Variable Rep en Caractère
Début
        Rep ← "X"
        Ecrire "Voulez vous un café ? (O/N)"
        TantQue Rep <> "O" et Rep <> "N"
                Lire Rep
                Si Rep <> "O" et Rep <> "N" Alors
                        Ecrire "Saisie Erronée, Recommencez"
                FinSi
        FinTantQue
Fin
```

ATTENTION :
- A ne pas écrire une structure TantQue dans laquelle le booléen n'est jamais VRAI. Le programme ne rentre alors jamais dans la boucle,
- A ne pas écrire une boucle dans laquelle le booléen ne devient jamais FAUX. L'ordinateur tourne alors dans la boucle et n'en sort plus.

FAIRE L'EXERCICE : 5.1 à 5.3

Boucler en comptant, ou compter en bouclant

Dans le dernier exercice, vous avez remarqué qu'une boucle pouvait être utilisée pour augmenter la valeur d'une variable. Cette utilisation des boucles est très fréquente, et dans ce cas, il arrive très souvent qu'on ait besoin d'effectuer un nombre **déterminé** de passages. Or, a priori, notre structure TantQue ne sait pas à l'avance combien de tours de boucle elle va effectuer (puisque le nombre de tours dépend de la valeur d'un booléen).

C'est pourquoi une autre structure de boucle est à notre disposition :

```
        Variable Truc en Entier
Début
        Truc ← 0
        TantQue Truc < 15
                Truc ← Truc + 1
                Ecrire "Passage numéro : ", Truc
        FinTantQue
Fin
```

ALGORITHME ET PSEUDO-CODE

Equivaut à :

```
        Variable Truc en Entier
Début
        Pour Truc ← 1 à 15
                Ecrire "Passage numéro : ", Truc
        Truc Suivant
Fin
```

Insistons : **la structure « Pour … Suivant » n'est pas du tout indispensable** ; on pourrait fort bien programmer toutes les situations de boucle uniquement avec un « Tant Que ». Le seul intérêt du « Pour » est d'épargner un peu de fatigue au programmeur, en lui évitant de gérer lui-même la progression de la variable qui lui sert de compteur (on parle d'**incrémentation**).

Dit d'une autre manière, la structure « Pour … Suivant » est un cas particulier de TantQue : celui où le programmeur peut dénombrer à l'avance le nombre de tours de boucles nécessaires.

Il faut noter que dans une structure « Pour … Suivant », la progression du compteur est laissée à votre libre disposition. Dans la plupart des cas, on a besoin d'une variable qui augmente de 1 à chaque tour de boucle. On ne précise alors rien à l'instruction « Pour » ; celle-ci, par défaut, comprend qu'il va falloir procéder à cette incrémentation de 1 à chaque passage, en commençant par la première valeur et en terminant par la deuxième.

Mais si vous souhaitez une progression plus spéciale, de 2 en 2, ou de 3 en 3, ou en arrière, de –1 en –1, ou de –10 en –10, ce n'est pas un problème : il suffira de le préciser à votre instruction « Pour » en lui rajoutant le mot « Pas » et la valeur de ce pas (Le « pas » dont nous parlons, c'est le « pas » du marcheur, « step » en anglais).

Naturellement, quand on stipule un pas négatif dans une boucle, la valeur initiale du compteur doit être **supérieure** à sa valeur finale si l'on veut que la boucle tourne ! Dans le cas contraire, on aura simplement écrit une boucle dans laquelle le programme ne rentrera jamais.

Nous pouvons donc maintenant donner la formulation générale d'une structure « Pour ». Sa syntaxe générale est :

```
Pour Compteur ← Initial à Final Pas ValeurDuPas
…
Instructions
…
Compteur suivant
```

Les structures **TantQue** sont employées dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble dont on ne connaît pas d'avance la quantité, comme par exemple :

- le contrôle d'une saisie
- la gestion des tours d'un jeu (tant que la partie n'est pas finie, on recommence)
- la lecture des enregistrements d'un fichier de taille inconnue(nous le verrons plus tard)

ALGORITHME ET PSEUDO-CODE

Les structures **Pour** sont employées dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble dont le programmeur connaît d'avance la quantité.

Nous verrons dans les chapitres suivants des séries d'éléments appelés tableaux et chaînes de caractères. Selon les cas, le balayage systématique des éléments de ces séries pourra être effectué par un Pour ou par un TantQue : tout dépend si la quantité d'éléments à balayer (donc le nombre de tours de boucles nécessaires) peut être dénombrée à l'avance par le programmeur ou non.

Des boucles dans des boucles

De même que les poupées russes contiennent d'autres poupées russes, de même qu'une structure SI … ALORS peut contenir d'autres structures SI … ALORS, une boucle peut tout à fait contenir d'autres boucles :

```
        Variables Truc, Trac en Entier
Début
        Pour Truc ← 1 à 15
                Ecrire "Il est passé par ici"
                Pour Trac ← 1 à 6
                        Ecrire "Il repassera par là"
                Trac Suivant
        Truc Suivant
Fin
```

Dans cet exemple, le programme écrira une fois "il est passé par ici" puis six fois de suite "il repassera par là", et ceci quinze fois en tout. A la fin, il y aura donc eu 15 x 6 = 90 passages dans la deuxième boucle (celle du milieu), donc 90 écritures à l'écran du message « il repassera par là ». Notez la différence marquante avec cette structure :

```
        Variables Truc, Trac en Entier
Début
        Pour Truc ← 1 à 15
                Ecrire "Il est passé par ici"
        Truc Suivant
        Pour Trac ← 1 à 6
                Ecrire "Il repassera par là"
        Trac Suivant
Fin
```

Ici, il y aura quinze écritures consécutives de "il est passé par ici", puis six écritures consécutives de "il repassera par là", et ce sera tout.

ALGORITHME ET PSEUDO-CODE

Des boucles peuvent donc être **imbriquées** (cas n°1) ou **successives** (cas n°2). Cependant, elles ne peuvent jamais, au grand jamais, être croisées. Dans la pratique de la programmation, la maîtrise des boucles imbriquées est nécessaire.

```
        Variables Truc, Trac en Entier
DEBUT
        Pour Truc ← …
                instructions
                Pour Trac ← …
                        instructions
        Truc Suivant
                instructions
        Trac Suivant
FIN
```

Pourquoi imbriquer des boucles ?

Une boucle, c'est un traitement systématique, un examen d'une série d'éléments un par un (par exemple, « prenons tous les employés de l'entreprise un par un »). Eh bien, on peut imaginer que pour chaque élément ainsi considéré (pour chaque employé), on doive procéder à un examen systématique d'autre chose (« prenons chacune des commandes que cet employé a traitées »). Voilà un exemple typique de boucles imbriquées : on devra programmer une boucle principale (celle qui prend les employés un par un) et à l'intérieur, une boucle secondaire (celle qui prend les commandes de cet employé une par une).

Dernière remarque à ne pas faire, en examinant l'algorithme suivant :

```
Début
        Pour Truc ← 1 à 15
                Truc ← Truc * 2
                Ecrire "Passage numéro : ", Truc
        Truc Suivant
Fin
```

Vous remarquerez que nous faisons ici gérer « en double » la variable Truc, ces deux gestions étant contradictoires. D'une part, la ligne

```
Pour…
```

augmente la valeur de Truc de 1 à chaque passage. D'autre part la ligne

```
Truc ← Truc * 2
```

double la valeur de Truc à chaque passage.

ALGORITHME ET PSEUDO-CODE

Et qu'on se le répète

La structure RÉPÉTER-JUSQU'À sous forme pseudo-code, présentée dans le tableau ci-dessus, est composée des mots réservés RÉPÉTER et JUSQU'À, d'une condition et d'une séquence d'instructions à exécuter jusqu'à ce que la condition devienne vraie (en d'autres mots, tant qu'elle est fausse).

L'exemple ci-dessous exploite une structure RÉPÉTER-JUSQU'À afin de lire un entier positif et le valider :

```
        Variables iNombre en Entier
Début
        RÉPÉTER
                Ecrire "Saisissez un nombre Positif :"
                Lire iNombre
        JUSQU'A iNombre > 0

Fin
```

Puisque au moins une lecture doit être effectuée, la structure RÉPÉTER-JUSQU'À est préférable car elle fait obligatoirement une itération (i.e. une lecture) avant de vérifier la condition. La structure RÉPÉTER-JUSQU'À est généralement préférée à la structure TANTQUE lorsque les variables dont dépend la condition reçoivent leur valeur dans la séquence d'instructions, ce qui exige obligatoirement une première itération.

FAIRE L'EXERCICE : 5.4 à 5.11

ALGORITHME ET PSEUDO-CODE

9.) Les tableaux

Utilité des tableaux

Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions « Lire » distinctes, cela donnera obligatoirement une atrocité du genre :

```
Moy ← (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12)/12
```

C'est tout de même laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, alors là c'est tout simplement impensable ; et si en plus on est dans une situation on l'on ne peut pas savoir d'avance combien il y aura de valeurs à traiter, là on ne peut pas traiter le problème.

C'est pourquoi la programmation nous permet **de rassembler toutes ces variables en une seule**, au sein de laquelle chaque valeur sera désignée par un numéro. En bon français, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ». C'est largement plus pratique, vous vous en doutez.

Un ensemble de valeurs portant le même nom de variable et repérées par un nombre, s'appelle un tableau, ou encore une variable indicée.
Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle l'indice.
Chaque fois que l'on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément, entre parenthèses (ou entre crochets).

Notation et utilisation algorithmique

Dans notre exemple, nous créerons donc un tableau appelé Note. Chaque note individuelle (chaque élément du tableau Note) sera donc désignée Note(0), Note(1), etc. Eh oui, attention, les indices des tableaux commencent généralement à 0, et non à 1.

ALGORITHME ET PSEUDO-CODE

Un tableau doit être déclaré comme tel, en précisant le nombre et le type de valeurs qu'il contiendra (la déclaration des tableaux est susceptible de varier d'un langage à l'autre. Certains langages réclament le nombre d'éléments, d'autre le plus grand indice... C'est donc une affaire de conventions).

En nous calquant sur les choix les plus fréquents dans les langages de programmations, nous déciderons ici arbitrairement et une bonne fois pour toutes que :

- les "cases" sont numérotées à partir de zéro, autrement dit que le plus petit indice est zéro.
- lors de la déclaration d'un tableau, on précise la plus grande valeur de l'indice (différente, donc, du nombre de cases du tableau, puisque si on veut 12 emplacements, le plus grand indice sera 11).

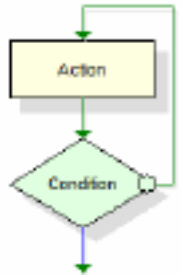
```
Tableau Note(11) en Entier
```

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, mais aussi tableaux de caractères, tableaux de booléens, tableaux de tout ce qui existe dans un langage donné comme type de variables. Par contre, hormis dans quelques rares langages, on ne peut pas faire un mixage de types différents de valeurs au sein d'un même tableau.

L'énorme avantage des tableaux, c'est qu'on va pouvoir les traiter en faisant des boucles. Par exemple, pour effectuer notre calcul de moyenne, cela donnera par exemple :

```
    (Variables) Tableau Note(11) en Numérique
    Variables Moy, Som en Numérique
Début
    Pour i ← 0 à 11
        Ecrire "Entrez la note n°", i
        Lire Note(i)
    i Suivant
    Som ← 0
    Pour i ← 0 à 11
        Som ← Som + Note(i)
    i Suivant
    Moy ← Som / 12
Fin
```

NB : On a fait deux boucles successives pour plus de lisibilité, mais on aurait tout aussi bien pu n'en écrire qu'une seule dans laquelle on aurait tout fait d'un seul coup.



ALGORITHME ET PSEUDO-CODE

Remarque générale : l'indice qui sert à désigner les éléments d'un tableau peut être exprimé directement comme un nombre en clair, mais il peut être aussi une variable, ou une expression calculée.

Dans un tableau, la valeur d'un indice doit toujours :

- **être égale au moins à 0** (dans quelques rares langages, le premier élément d'un tableau porte l'indice 1). Mais comme je l'ai déjà écrit plus haut, nous avons choisi ici de commencer la numérotation des indices à zéro, comme c'est le cas en langage C et en Visual Basic. Donc attention, Truc(6) est le septième élément du tableau Truc !
- **être un nombre entier** Quel que soit le langage, l'élément Truc(3,1416) n'existe jamais.
- **être inférieure ou égale au nombre d'éléments du tableau** (moins 1, si l'on commence la numérotation à zéro). Si le tableau Bidule a été déclaré comme ayant 25 éléments, la présence dans une ligne, sous une forme ou sous une autre, de Bidule(32) déclenchera automatiquement une erreur.

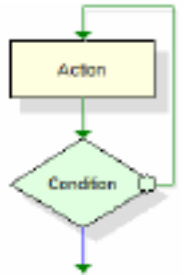
Je le re-re-répète, si l'on est dans un langage où les indices commencent à zéro, il faut en tenir compte à la déclaration :

Tableau Note(13) **en Numérique**

...créera un tableau de 14 éléments, le plus petit indice étant 0 et le plus grand 13.

ATTENTION :
Il consiste à confondre, dans sa tête et / ou dans un algorithme, l'**indice** d'un élément d'un tableau avec le **contenu** de cet élément. La troisième maison de la rue n'a pas forcément trois habitants, et la vingtième vingt habitants. En notation algorithmique, il n'y a aucun rapport entre i et truc(i).

FAIRE LES EXERCICES : 6.1 à 6.7



ALGORITHME ET PSEUDO-CODE

Tableaux dynamiques

Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter un tableau. Bien sûr, une solution consisterait à déclarer un tableau gigantesque (10 000 éléments par exemple) pour être sûr que « ça rentre ». Mais d'une part, on n'en sera jamais parfaitement sûr, d'autre part, en raison de l'immensité de la place mémoire réservée – et la plupart du temps non utilisée, c'est un gâchis préjudiciable à la rapidité, voire à la viabilité, de notre algorithme.

Aussi, pour parer à ce genre de situation, a-t-on la possibilité de déclarer le tableau sans préciser au départ son nombre d'éléments. Ce n'est que dans un second temps, au cours du programme, que l'on va fixer ce nombre via une instruction de redimensionnement : **Redim**.

Nous voyons cet aspect là, car certains langages le réclament. Et nous devons donc l'apprendre. Mais sinon, dans beaucoup de langages modernes, nous n'avons pas besoin de faire appel à un redimensionnement.

Notez que **tant qu'on n'a pas précisé le nombre d'éléments d'un tableau, d'une manière ou d'une autre, ce tableau est inutilisable**.

Exemple : on veut faire saisir des notes pour un calcul de moyenne, mais on ne sait pas combien il y aura de notes à saisir. Le début de l'algorithme sera quelque chose du genre :

```
        Tableau Notes() en Numérique
        Variable nb en Numérique
Début
        Ecrire "Combien y a-t-il de notes à saisir ?"
        Lire nb
        Redim Notes(nb-1)
        …
```

Cette technique n'a rien de sorcier, mais elle fait partie de l'arsenal de base de la programmation en gestion.

FAIRE LES EXERCICES : 6.8 à 6.14