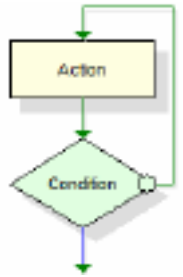


FORMATION AFPA
- DEVELOPPEUR LOGICIEL –

(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 ?

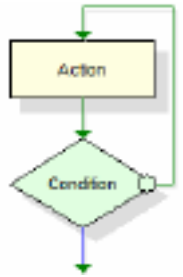
Parce que l'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage. Pour prendre une image, si un programme était une dissertation, l'algorithmique serait le plan, une fois mis de côté la rédaction et l'orthographe. Or, vous savez qu'il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

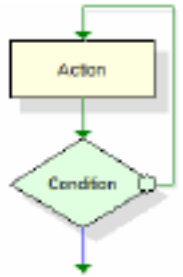
Exemple :

```
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.

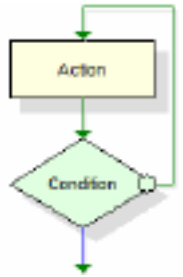
Le nom de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages. Toutefois, une règle absolue est qu'un nom de variable peut comporter des lettres et des chiffres, mais qu'il exclut la plupart des signes de ponctuation, en particulier les espaces. Un nom de variable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

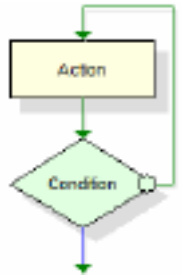
Pourquoi ne pas déclarer toutes les variables numériques en réel double, histoire de bétonner et d'être certain qu'il n'y aura pas de problème ? **En vertu du principe de l'économie de moyens !**

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

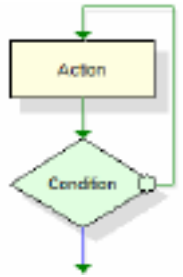
La seule différence entre les deux algorithmes consiste dans la présence ou dans l'absence des guillemets lors de la seconde affectation. Et l'on voit que cela change tout !

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

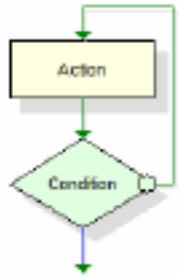
Variable A **en Numérique**
Début
A ← 12
A ← 34
Fin

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 :

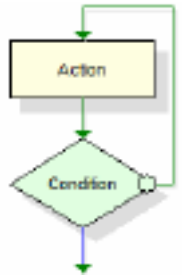
```
Variables A, B, C en Caractère
Début
A ← "Gloubi"
B ← "Boulga"
C ← A & B
Fin
```

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 :
$y = 3x + 2$

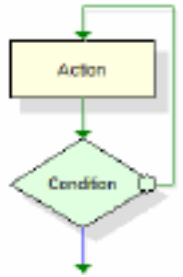
les « variables » x et y satisfaisant à l'équation existent en nombre infini (graphiquement, l'ensemble des solutions à cette équation dessine une droite). Lorsque j'écris :
$ax^2 + bx + c = 0$

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.

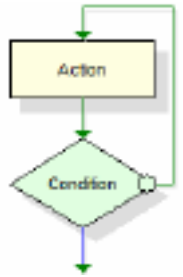
Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

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) :

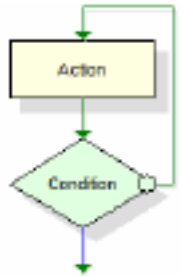
Si A ET B **Alors**
    Instructions 1
Sinon
    Instructions 2
Fin si

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
```

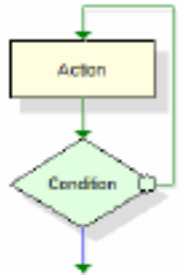
Dans cette dernière formulation, nous n'avons plus recours à une condition composée (mais au prix d'un test imbriqué supplémentaire)

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 ?

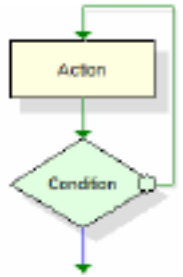
Prenons le cas d'une saisie au clavier (une lecture), où par exemple, le programme pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Mais tôt ou tard, l'utilisateur, maladroit, risque de taper autre chose que la réponse attendue. Dès lors, le programme peut planter soit 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).

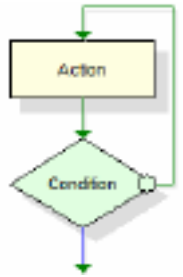
```
Tableau Note(11) en Entier
```

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.

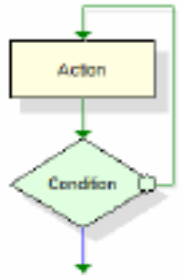
Je le re-re-répète, si l'on est dans un langage où les indices commencent à zéro, il faut en tenir compte à la déclaration :

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

ALGORITHME ET PSEUDO-CODE

9.) Techniques Rusées

Ce chapitre a pour but de détailler quelques techniques de programmation et leur connaissance en est indispensable.

Tri d'un tableau : le tri par sélection

Ranger des valeurs dans un ordre donné, c'est une technique à maitriser tellement elle est récurrente dans la programmation. Même si elles paraissent un peu ardues au départ, il vaut mieux avoir assimilé une ou deux techniques solidement éprouvée s.

Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons deux : le tri par sélection, et le tri à bulles.

Commençons par le tri par sélection.

Admettons que le but de la manœuvre soit de trier un tableau de 12 éléments dans l'ordre croissant. La technique du tri par sélection est la suivante : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis en met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier. Par exemple, si l'on part de :

45	122	12	3	21	78	64	53	89	28	84	46

On commence par rechercher, parmi les 12 valeurs, quel est le plus petit élément , et où il se trouve. On l'identifie en quatrième position (c'est le nombre 3), et on l'échange alors avec le premier élément (le nombre 45). Le tableau devient ainsi :

3	122	12	45	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément, mais cette fois, seulement à partir du deuxième (puisque le premier est maintenant correct, on n'y touche plus). On le trouve en troisième position (c'est le nombre 12). On échange donc le deuxième avec le troisième :

3	12	122	45	21	78	64	53	89	28	84	46

ALGORITHME ET PSEUDO-CODE

On recommence à chercher le plus petit élément à partir du troisième (puisque les deux premiers sont maintenant bien placés), et on le place correctement, en l'échangeant, ce qui donnera in fine :

3	12	21	45	122	78	64	53	89	28	84	46

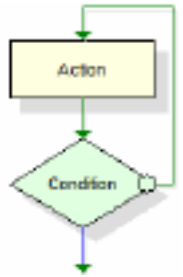
Et ainsi de suite jusqu'à l'avant dernier.

En bon français, nous pourrions décrire le processus de la manière suivante :

- Boucle principale : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.
- Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel et le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

Maintenant passons à l'algorithme. Celui-ci s'écrit :

```
// boucle principale : le point de départ se décale à chaque tour
Pour i ← 0 à 10
        // on considère provisoirement que t(i) est le plus petit élément
        posmini ← i
        // on examine tous les éléments suivants
        Pour j ← i + 1 à 11
                Si t(j) < t(posmini) Alors
                        posmini ← j
                Finsi
        j suivant
        // A cet endroit, on sait maintenant où est le plus petit élément.
        // Il ne reste plus qu'à effectuer la permutation.
        temp ← t(posmini)
        t(posmini) ← t(i)
        t(i) ← temp
        // On a placé correctement l'élément numéro i, on passe à présent au suivant.
i suivant
```



ALGORITHME ET PSEUDO-CODE

Un exemple de flag : la recherche dans un tableau

Nous allons maintenant nous intéresser au maniement d'une variable booléenne : la technique dite du « flag ».

Le flag, en anglais, est un petit drapeau, qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas. Et, aussitôt que cet événement a lieu, le petit drapeau se lève (la variable booléenne change de valeur). Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Un exemple extrêmement fréquent est la recherche de l'occurrence d'une valeur dans un tableau. On en profitera au passage pour corriger une erreur particulièrement fréquente chez le programmeur débutant.

Soit un tableau comportant, par exemple, 20 valeurs. Et on doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.

La première étape consiste à écrire les instructions de lecture / écriture, et la boucle qui parcourt le tableau :

```
        Tableau Tab(19) en Numérique
        Variable N en Numérique
Début
        Ecrire "Entrez la valeur à rechercher"
        Lire N
        Pour i ← 0 à 19
                ???
        i suivant
Fin
```

Il nous reste à combler les points d'interrogation de la boucle Pour. Évidemment, il va falloir comparer N à chaque élément du tableau : si les deux valeurs sont égales, alors N fait partie du tableau. Cela va se traduire, bien entendu, par un Si … Alors … Sinon. Et voilà le programmeur raisonnant hâtivement qui se vautre en écrivant :

```
        Tableau Tab(19) en Numérique
        Variable N en Numérique
Début
        Ecrire "Entrez la valeur à rechercher"
        Lire N
        Pour i ← 0 à 19
                Si N = Tab(i) Alors
                        Ecrire N "fait partie du tableau"
                Sinon
                        Ecrire N "ne fait pas partie du tableau"
                FinSi
        i suivant
Fin
```

ALGORITHME ET PSEUDO-CODE

Et cet algorithme est une véritable catastrophe.

Il suffit d'ailleurs de le faire tourner mentalement pour s'en rendre compte. De deux choses l'une : ou bien la valeur N figure dans le tableau, ou bien elle n'y figure pas. Mais dans tous les cas, l'algorithme ne doit produire qu'une seule réponse, quel que soit le nombre d'éléments que compte le tableau. Or, l'algorithme ci-dessus envoie à l'écran autant de messages qu'il y a de valeurs dans le tableau, en l'occurrence pas moins de 20 !

Il y a donc une erreur manifeste de conception : l'écriture du message ne peut se trouver à l'intérieur de la boucle : elle doit figurer à l'extérieur. On sait si la valeur était dans le tableau ou non uniquement lorsque le balayage du tableau est entièrement accompli.

Nous réécrivons donc cet algorithme en plaçant le test après la boucle. On se contentera de faire dépendre pour le moment la réponse d'une variable booléenne que nous appellerons bTrouve.

```
        Tableau Tab(19) en Numérique
        Variable N en Numérique
Début
        Ecrire "Entrez la valeur à rechercher"
        Lire N
        Pour i ← 0 à 19
                ???
        i suivant
        Si bTrouve Alors
                Ecrire "N fait partie du tableau"
        Sinon
                Ecrire "N ne fait pas partie du tableau"
        FinSi
Fin
```

Il ne nous reste plus qu'à gérer la variable bTrouve. Ceci se fait en deux étapes.

- un test figurant dans la boucle, indiquant lorsque la variable bTrouve doit devenir vraie (à savoir, lorsque la valeur N est rencontrée dans le tableau). Et attention : le test est asymétrique. Il ne comporte pas de "sinon". On reviendra là dessus dans un instant.
- l'affectation par défaut de la variable bTrouve, dont la valeur de départ doit être évidemment Faux.

ALGORITHME ET PSEUDO-CODE

Au total, l'algorithme complet donne :

```
        Tableau Tab(19) en Numérique
        Variable N en Numérique
Début
        Ecrire "Entrez la valeur à rechercher"
        Lire N
        bTrouve ← Faux
        Pour i ← 0 à 19
                Si N = Tab(i) Alors
                        bTrouve ← Vrai
                FinSi
        i suivant
        Si bTrouve Alors
                Ecrire "N fait partie du tableau"
        Sinon
                Ecrire "N ne fait pas partie du tableau"
        FinSi
Fin
```

La difficulté est de comprendre que dans une recherche, le problème ne se formule pas de la même manière selon qu'on le prend par un bout ou par un autre. On peut résumer l'affaire ainsi : il suffit que N soit égal à une seule valeur de Tab pour qu'elle fasse partie du tableau. En revanche, il faut qu'elle soit différente de toutes les valeurs de Tab pour qu'elle n'en fasse pas partie.

Voilà la raison qui nous oblige à passer par une variable booléenne , un « drapeau » qui peut se lever, mais jamais se rabaisser. Et cette technique de flag (que nous pourrions élégamment surnommer « gestion asymétrique de variable booléenne ») doit être mise en œuvre chaque fois que l'on se trouve devant pareille situation.

Autrement dit, connaître ce type de raisonnement est indispensable, et savoir le reproduire à bon escient ne l'est pas moins.

Dans ce cas précis, il est vrai, on pourrait à juste titre faire remarquer que l'utilisation de la technique du flag, si elle permet une subtile mais ferme progression pédagogique, ne donne néanmoins pas un résultat optimum. En effet, dans l'hypothèse où la machine trouve la valeur recherchée quelque part au milieu du tableau, notre algorithme continue – assez bêtement, il faut bien le dire – la recherche jusqu'au bout du tableau, alors qu'on pourrait s'arrêter net.

Le meilleur algorithme possible, même s'il n'utilise pas de flag, consiste donc à remplacer la boucle **Pour** par une boucle **TantQue** : en effet, là, on ne sait plus combien de tours de boucle il va falloir faire (puisqu'on risque de s'arrêter avant la fin du tableau). Pour savoir quelle condition suit le TantQue, raisonnons à l'envers : on s'arrêtera quand on aura trouvé la valeur cherchée... ou qu'on sera arrivés à la fin du tableau. Appliquons la transformation de Morgan : Il faut donc poursuivre la recherche tant qu'on n'a pas trouvé la valeur et qu'on n'est pas parvenu à la fin du tableau.

ALGORITHME ET PSEUDO-CODE

Démonstration :

```
        Tableau Tab(19) en Numérique
        Variable N en Numérique
Début
        Ecrire "Entrez la valeur à rechercher"
        Lire N
        i ← 0
        TantQue N <> T(i) et i < 19
                i ← i + 1
        FinTantQue
        Si N = Tab(i) Alors
                Ecrire "N fait partie du tableau"
        Sinon
                Ecrire "N ne fait pas partie du tableau"
        FinSi
Fin
```

Tri de tableau + flag = tri à bulles

Et maintenant, nous en arrivons à la formule : tri de tableau + flag = tri à bulles.

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel tout élément est plus petit que celui qui le suit.

En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands « remontent » ainsi peu à peu vers les dernières places, ce qui explique la dénomination de « tri à bulle ».

En quoi le tri à bulles implique-t-il l'utilisation d'un flag ? Eh bien, par ce qu'on ne sait jamais par avance combien de remontées de bulles on doit effectuer. En fait, tout ce qu'on peut dire, c'est qu'on devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés. Ceci est typiquement un cas de question « asymétrique » : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié.

Nous baptiserons le flag Yapermute, car cette variable booléenne va nous indiquer si nous venons ou non de procéder à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est signe que le tableau est trié, et donc qu'on peut arrêter la machine à bulles). La boucle principale sera alors :

ALGORITHME ET PSEUDO-CODE

```
        Variable Yapermute en Booléen
Début
        …
        TantQue Yapermute
                …
        FinTantQue
Fin
```

Que va-t-on faire à l'intérieur de la boucle ? Prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, et procéder à un échange si nécessaire. C'est parti :

```
        Variable Yapermute en Booléen
Début
        …
        TantQue Yapermute
                Pour i ← 0 à 10
                        Si t(i) > t(i+1) Alors
                                temp ← t(i)
                                t(i) ← t(i+1)
                                t(i+1) ← temp
                        Finsi
                i suivant
        FinTantQue
Fin
```

Mais il ne faut pas oublier un détail capital : la gestion de notre flag. L'idée, c'est que cette variable va nous signaler le fait qu'il y a eu au moins une permutation effectuée. Il faut donc :

- lui attribuer la valeur Vrai dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
- la remettre à Faux à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés),
- dernier point, il ne faut pas oublier de lancer la boucle principale, et pour cela de donner la valeur Vrai au flag au tout départ de l'algorithme.

ALGORITHME ET PSEUDO-CODE

La solution complète donne donc :

```
        Variable Yapermute en Booléen
Début
        …
        Yapermute ← Vrai
        TantQue Yapermute
                Yapermute ← Faux
                Pour i ← 0 à 10
                        Si t(i) > t(i+1) alors
                                temp ← t(i)
                                t(i) ← t(i+1)
                                t(i+1) ← temp
                                Yapermute ← Vrai
                        Finsi
                i suivant
        FinTantQue
Fin
```

La recherche dichotomique

Nous allons terminer ce chapitre par une technique célèbre de recherche, qui révèle toute son utilité lorsque le nombre d'éléments est très élevé. Par exemple, imaginons que nous ayons un programme qui doive vérifier si un mot existe dans le dictionnaire. Nous pouvons supposer que le dictionnaire a été préalablement entré dans un tableau (à raison d'un mot par emplacement). Ceci peut nous mener à, disons à la louche, 40 000 mots.

Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier. Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20 000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, un être humain qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots sont triés.

Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve pile poil au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans le première moitié du dico. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié.

ALGORITHME ET PSEUDO-CODE

A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

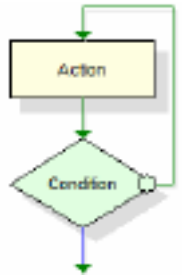
A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne en terme de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.

- Au départ, on cherche le mot parmi 40 000.
- Après le test n°1, on ne le cherche plus que parmi 20 000.
- Après le test n°2, on ne le cherche plus que parmi 10 000.
- Après le test n°3, on ne le cherche plus que parmi 5 000.
- etc.
- Après le test n°15, on ne le cherche plus que parmi 2.
- Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas. Moralité : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment ! Il n'y a pas photo sur l'écart de performances. Attention, toutefois, même si c'est évident, je le répète avec force : **la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.**

FAIRE LES EXERCICES 7.1 à 7.6



ALGORITHME ET PSEUDO-CODE

10.) Tableaux Multidimensionnels

Pourquoi plusieurs dimensions ?

Prenons le cas de la modélisation d'un jeu de dames, et du déplacement des pions sur le damier. Je rappelle qu'un pion qui est sur une case blanche peut se déplacer (pour simplifier) sur les quatre cases blanches adjacentes.

Avec les outils que nous avons abordés jusque là, le plus simple serait évidemment de modéliser le damier (même si ce dernier est supposé fait 100 cases, prenons un damier à 64 cases - 8x8) sous la forme d'un tableau. Chaque case est un emplacement du tableau, qui contient par exemple 0 si elle est vide, et 1 s'il y a un pion. On attribue comme indices aux cases les numéros 1 à 8 pour la première ligne, 9 à 16 pour la deuxième ligne, et ainsi de suite jusqu'à 64.

Un pion placé dans la case numéro i, autrement dit la valeur 1 de Cases(i), peut bouger vers les cases contiguës en diagonale. Cela va nous obliger à de petites acrobaties intellectuelles : la case située juste au-dessus de la case numéro i ayant comme indice i-8, les cases valables sont celles d'indice i-7 et i-9. De même, la case située juste en dessous ayant comme indice i+8, les cases valables sont celles d'indice i+7 et i+9.

Bien sûr, on peut fabriquer tout un programme comme cela, mais le moins qu'on puisse dire est que cela ne facilite pas la clarté de l'algorithme.

Il serait évidemment plus simple de modéliser un damier par… un damier !

Tableaux à deux dimensions

L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par une seule, mais par deux coordonnées.

Un tel tableau se déclare ainsi :

Tableau Cases(7, 7) **en Numérique**

Cela veut dire : réserve moi un espace de mémoire pour 8 x 8 entiers, et quand j'aurai besoin de l'une de ces valeurs, je les repèrerai par deux indices (comme à la bataille navale, ou Excel, la seule différence étant que pour les coordonnées, on n'utilise pas de lettres, juste des chiffres).

Pour notre problème de dames, les choses vont sérieusement s'éclaircir. La case qui contient le pion est dorénavant Cases(i, j). Et les quatre cases disponibles sont Cases(i-1, j-1), Cases(i-1, j+1), Cases(i+1, j-1) et Cases(i+1, j+1).

REMARQUE ESSENTIELLE :
Il n'y a aucune différence qualitative entre un tableau à deux dimensions (i, j) et un tableau à une dimension (i * j). De même que le jeu de dames qu'on vient d'évoquer, tout problème qui peut être modélisé d'une manière peut aussi être modélisé de l'autre. Simplement, l'une ou l'autre de ces techniques correspond plus spontanément à tel ou tel problème, et facilite donc (ou complique, si on a choisi la mauvaise option) l'écriture et la lisibilité de l'algorithme.

ALGORITHME ET PSEUDO-CODE

Une autre remarque : une question classique à propos des tableaux à deux dimensions est de savoir si le premier indice représente les lignes ou le deuxième les colonnes, ou l'inverse. En fait, cela n'a aucun sens. « Lignes » et « Colonnes » sont des concepts graphiques, visuels, qui s'appliquent à des objets du monde réel ; les indices des tableaux ne sont que des coordonnées logiques, pointant sur des adresses de mémoire vive. Si cela ne vous convainc pas, pensez à un jeu de bataille navale classique : les lettres doivent-elles désigner les lignes et les chiffres les colonnes ? Aucune importance ! Chaque joueur peut même choisir une convention différente, aucune importance ! L'essentiel est qu'une fois une convention choisie, un joueur conserve la même tout au long de la partie, bien entendu.

Tableaux à n dimensions

Si vous avez compris le principe des tableaux à deux dimensions, sur le fond, il n'y a aucun problème à passer au maniement de tableaux à trois, quatre, ou pourquoi pas neuf dimensions. C'est exactement la même chose. Si je déclare un tableau Titi(2, 4, 3, 3), il s'agit d'un espace mémoire contenant 3 x 5 x 4 x 4 = 240 valeurs. Chaque valeur y est repérée par quatre coordonnées.

FAIRE LES EXERCICES 8.1 à 8.7

ALGORITHME ET PSEUDO-CODE

11.) Les Fonctions Prédéfinies

Certains traitements ne peuvent être effectués par un algorithme, aussi savant soit-il. D'autres ne peuvent l'être qu'au prix de souffrances indicibles.

C'est par exemple le cas du calcul du sinus d'un angle : pour en obtenir une valeur approchée, il faudrait appliquer une formule d'une complexité à vous glacer le sang. Aussi, que se passe-t-il sur les petites calculatrices que vous connaissez tous ? On vous fournit quelques touches spéciales, dites touches de fonctions, qui vous permettent par exemple de connaître immédiatement ce résultat. Sur votre calculatrice, si vous voulez connaître le sinus de 35°, vous taperez 35, puis la touche SIN, et vous aurez le résultat.

Tout langage de programmation propose ainsi un certain nombre de fonctions ; certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs – et pénibles - algorithmes.

Structure générale des fonctions

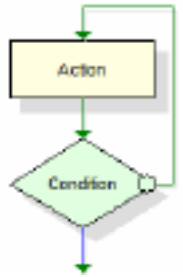
Reprenons l'exemple du sinus. Les langages informatiques, qui se doivent tout de même de savoir faire la même chose qu'une calculatrice à 2 €, proposent généralement une fonction SIN. Si nous voulons stocker le sinus de 35 dans la variable A, nous écrirons :

A ← Sin(35)

Une fonction est donc constituée de trois parties :

- le nom proprement dit de la fonction. Ce nom ne s'invente pas ! Il doit impérativement correspondre à une fonction proposée par le langage. Dans notre exemple, ce nom est SIN.
- deux parenthèses, une ouvrante, une fermante. **Ces parenthèses sont toujours obligatoires, même lorsqu'on n'écrit rien à l'intérieur.**
- une liste de valeurs, indispensables à la bonne exécution de la fonction. Ces valeurs s'appellent des arguments, ou des paramètres. Certaines fonctions exigent un seul argument, d'autres deux, etc. et d'autres encore aucun. A noter que même dans le cas de ces fonctions n'exigeant aucun argument, les parenthèses restent obligatoires. Le nombre d'arguments nécessaire pour une fonction donnée ne s'invente pas : il est fixé par le langage. Par exemple, la fonction sinus a besoin d'un argument (ce n'est pas surprenant, cet argument est la valeur de l'angle). Si vous essayez de l'exécuter en lui donnant deux arguments, ou aucun, cela déclenchera une erreur à l'exécution. Notez également que les arguments doivent être d'un certain type, et qu'il faut respecter ces types.

Les choses à ne pas faire :
Il consiste à affecter une fonction, quelle qu'elle soit.



ALGORITHME ET PSEUDO-CODE

Toute écriture plaçant une fonction à gauche d'une instruction d'affectation est aberrante, pour deux raisons symétriques :

- d'une part, on ne peut affecter qu'une variable, à l'exclusion de tout autre chose.
- ensuite, parce qu'une fonction a pour rôle de produire, de renvoyer, de valoir (tout cela est synonyme), un résultat. Pas d'en recevoir un.

FAIRE L'EXERCICE 9.1

Les fonctions de texte

Une catégorie privilégiée de fonctions est celle qui nous permet de manipuler des chaînes de caractères. Nous avons déjà vu qu'on pouvait facilement « coller » deux chaînes l'une à l'autre avec l'opérateur de concaténation &. Mais ce que nous ne pouvions pas faire, et qui va être maintenant possible, c'est pratiquer des extractions de chaînes.

Tous les langages proposent la plupart des fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

- Length(chaîne) : renvoie le nombre de caractères d'une chaîne
- Mid(chaîne,n1,n2) : renvoie un extrait de la chaîne, commençant au caractère n1 et faisant n2 caractères de long.

Ce sont les deux seules fonctions de chaînes réellement indispensables. Cependant, pour nous épargner des algorithmes fastidieux, les langages proposent également :

- Left(chaîne,n) : renvoie les n caractères les plus à gauche dans chaîne.
- Right(chaîne,n) : renvoie les n caractères les plus à droite dans chaîne
- Trouve(chaîne1,chaîne2) : renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie zéro.

Exemples :

Length("Bonjour, ça va ?") vaut 16
Length("") vaut 0
Mid("Zorro is back", 4, 7) vaut "ro is b"
Mid("Zorro is back", 12, 1) vaut "c"
Left("Et pourtant…", 8) vaut "Et pourt"
Right("Et pourtant…", 4) vaut "t…"
Trouve("Un pur bonheur", "pur") vaut 4
Trouve("Un pur bonheur", "techno") vaut 0

ALGORITHME ET PSEUDO-CODE

Il existe aussi dans tous les langages une fonction qui renvoie le caractère correspondant à un code Ascii donné (fonction Asc), et son contraire (fonction Chr) :

Asc("N") vaut 78
Chr(63) vaut "?"

Si les programmes informatiques ont fréquemment à traiter des nombres, ils doivent tout aussi fréquemment gérer des séries de caractères (des chaînes), et connaître les techniques de base sur les chaînes est plus qu'utile : c'est indispensable.

FAIRE LES EXERCICES 9.2 à 9.6

Trois fonctions numériques classiques

Partie Entière

Une fonction extrêmement répandue est celle qui permet de récupérer la partie entière d'un nombre :

Après : A ← Ent(3,228) A vaut 3

Cette fonction est notamment indispensable pour effectuer le test de parité.

Modulo

Cette fonction permet de récupérer le reste de la division d'un nombre par un deuxième nombre. Par exemple :

A ← Mod(10,3) A vaut 1 car 10 = 3*3 + 1
B ← Mod(12,2) B vaut 0 car 12 = 6*2
C ← Mod(44,8) C vaut 4 car 44 = 5*8 + 4

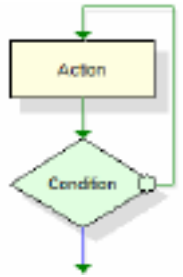
Génération de nombres aléatoires

Une autre fonction classique , et très utile, est celle qui génère un nombre choisi au hasard : **random**.

Dans tous les langages, cette fonction existe et produit le résultat suivant :

Après : Toto ← random(0, 100) On a : 0 =< Toto =< 100

FAIRE LES EXERCICES 9.7 à 9.11



ALGORITHME ET PSEUDO-CODE

Les fonctions de conversion

Dernière grande catégorie de fonctions, là aussi disponibles dans tous les langages, car leur rôle est parfois incontournable, les fonctions dites de conversion.

Rappelez-vous ce que nous avons vu dans les premières pages de ce cours : il existe différents types de variables, qui déterminent notamment le type de codage qui sera utilisé. Prenons le chiffre 3. Si je le stocke dans une variable de type alphanumérique, il sera codé en tant que caractère, sur un octet. Si en revanche je le stocke dans une variable de type entier, il sera codé sur deux octets. Et la configuration des bits sera complètement différente dans les deux cas.

Une conclusion évidente, et sur laquelle on a déjà eu l'occasion d'insister, c'est qu'on ne peut pas faire n'importe quoi avec n'importe quoi, et qu'on ne peut pas par exemple multiplier "3" et "5", si 3 et 5 sont stockés dans des variables de type caractère.

Pourquoi ne pas en tirer les conséquences, et stocker convenablement les nombres dans des variables numériques, les caractères dans des variables alphanumériques, comme nous l'avons toujours fait ?

Parce qu'il y a des situations où on n'a pas le choix ! Nous allons voir dès le chapitre suivant un mode de stockage (les fichiers textes) où toutes les informations, quelles qu'elles soient, sont obligatoirement stockées sous forme de caractères. Dès lors, si l'on veut pouvoir récupérer des nombres et faire des opérations dessus, il va bien falloir être capable de convertir ces chaînes en numériques.

Aussi, tous les langages proposent-ils une palette de fonctions destinées à opérer de telles conversions. On trouvera au moins une fonction destinée à convertir une chaîne en numérique (appelons-la Cnum en pseudo-code), et une convertissant un nombre en caractère (Ccar).

ALGORITHME ET PSEUDO-CODE

12.) Les Fichiers

Jusqu'à présent, les informations utilisées dans nos programmes ne pouvaient provenir que de deux sources : soit elles étaient inclues dans l'algorithme lui-même, par le programmeur, soit elles étaient entrées en cours de route par l'utilisateur. Et cela ne suffit pas à combler les besoins réels des informaticiens.

Imaginons que l'on veuille écrire un programme gérant un carnet d'adresses. D'une exécution du programme à l'autre, l'utilisateur doit pouvoir retrouver son carnet à jour, avec les modifications qu'il y a apportées la dernière fois qu'il a exécuté le programme. Les données du carnet d'adresse ne peuvent donc être inclues dans l'algorithme, et encore moins être entrées au clavier à chaque nouvelle exécution !

Les fichiers sont là pour combler ce manque. Ils servent à stocker des informations de manière permanente, entre deux exécutions d'un programme. Car si les variables, qui sont des adresses de mémoire vive, disparaissent à chaque fin d'exécution, les fichiers, eux sont stockés sur des périphériques à mémoire de masse (disque dur, CD Rom…).

Organisation des fichiers

Un premier grand critère, qui différencie les deux grandes catégories de fichiers, est le suivant : le fichier est-il ou non organisé sous forme de lignes successives ? Si oui, cela signifie vraisemblablement que ce fichier contient le même genre d'information à chaque ligne. Ces lignes sont alors appelées des enregistrements.

Prenons le cas classique, celui d'un carnet d'adresses. Le fichier est destiné à mémoriser les coordonnées d'un certain nombre de personnes. Pour chacune, il faudra noter le nom, le prénom, le numéro de téléphone et l'email. Dans ce cas, il peut paraître plus simple de stocker une personne par ligne du fichier (par enregistrement). Dit autrement, quand on prendra une ligne, on sera sûr qu'elle contient les informations concernant une personne, et uniquement cela. Un fichier ainsi codé sous forme d'enregistrements est appelé un fichier texte.

En fait, entre chaque enregistrement, sont stockés les octets correspondants aux caractères CR (code Ascii 13) et LF (code Ascii 10), signifiant un retour au début de la ligne suivante. Le plus souvent, le langage de programmation, dès lors qu'il s'agit d'un fichier texte, gèrera lui-même la lecture et l'écriture de ces deux caractères à chaque fin de ligne : c'est autant de moins dont le programmeur aura à s'occuper. Le programmeur, lui, n'aura qu'à dire à la machine de lire une ligne, ou d'en écrire une.

Ce type de fichier est couramment utilisé dès lors que l'on doit stocker des informations pouvant être assimilées à une base de données.

Le second type de fichier, vous l'aurez deviné, se définit a contrario : il rassemble les fichiers qui ne possèdent pas de structure de lignes (d'enregistrement). Les octets, quels qu'il soient, sont écrits à la queue leu leu. Ces fichiers sont appelés des fichiers binaires. Naturellement, leur structure différente implique un traitement différent par le programmeur. Tous les fichiers qui ne codent pas une base de données sont obligatoirement des fichiers binaires : cela concerne par exemple un fichier son, une image, un programme exécutable, etc. .

Autre différence majeure entre fichiers texte et fichiers binaires : dans un fichier texte, toutes les données sont écrites sous forme de... texte. Cela veut dire que les nombres y sont représentés sous forme de suite de chiffres (des chaînes de caractères). Ces nombres doivent donc être convertis en chaînes lors de l'écriture dans le fichier. Inversement, lors de la lecture du fichier, on devra convertir ces chaînes en nombre si l'on veut pouvoir les utiliser dans des calculs.

ALGORITHME ET PSEUDO-CODE

Ceci a comme autre implication qu'un fichier texte est directement lisible, alors qu'un fichier binaire ne l'est pas(sauf bien sîr en écrivant soi-même un programme approprié). Si l'on ouvre un fichier texte via un éditeur de textes, comme le bloc-notes de Windows, on y reconnaîtra toutes les informations (ce sont des caractères, stockés comme tels). La même chose avec un fichier binaire ne nous produit à l'écran qu'une quantité de signes incompréhensibles.

Structure des enregistrements

Savoir que les fichiers peuvent être structurés en enregistrements, c'est bien. Mais savoir comment sont à leur tour structurés ces enregistrements, c'est mieux. Or, là aussi, il y a deux grandes possibilités. Ces deux grandes variantes pour structurer les données au sein d'un fichier texte sont la délimitation et les champs de largeur fixe.

Reprenons le cas du carnet d'adresses, avec dedans le nom, le prénom, le téléphone et l'email. Les données, sur le fichier texte, peuvent être organisées ainsi :

Structure n°1
"Fonfec";"Sophie";0142156487;"fonfec@yahoo.fr"
"Zétofrais";"Mélanie";0456912347;"zétofrais@free.fr"
"Herbien";"Jean-Philippe";0289765194;"vantard@free.fr"
"Hergébel";"Octave";0149875231;"rg@aol.fr"

ou ainsi :

Structure n°2
Fonfec Sophie 0142156487 fonfec@yahoo.fr
Zétofrais Mélanie 0456912347 zetofrais@free.fr
Herbien Jean-Philippe 0289765194 vantard@free.fr
Hergébel Octave 0149875231 rg@aol.fr

La structure n°1 est dite délimitée ; Elle utilise un caractère spécial, appelé caractère de délimitation, qui permet de repérer quand finit un champ et quand commence le suivant. Il va de soi que ce caractère de délimitation doit être strictement interdit à l'intérieur de chaque champ, faute de quoi la structure devient proprement illisible.

La structure n°2, elle, est dite à champs de largeur fixe. Il n'y a pas de caractère de délimitation, mais on sait que les x premiers caractères de chaque ligne stockent le nom, les y suivants le prénom, etc. Cela impose bien entendu de ne pas saisir un renseignement plus long que le champ prévu pour l'accueillir.

- L'avantage de la structure n°1 est son faible encombrement en place mémoire ; il n'y a aucun espace perdu, et un fichier texte codé de cette manière occupe le minimum de place possible. Mais elle possède en revanche un inconvénient majeur, qui est la lenteur de la lecture. En effet, chaque fois que l'on récupère une ligne dans le fichier, il faut alors parcourir un par un tous les caractères pour repérer chaque occurrence du caractère de séparation avant de pouvoir découper cette ligne en différents champs.

- La structure n°2, à l'inverse, gaspille de la place mémoire, puisque le fichier est un vrai gruyère plein de trous. Mais d'un autre côté, la récupération des différents champs est très rapide. Lorsqu'on récupère une ligne, il suffit de la découper en différentes chaînes de longueur prédéfinie, et le tour est joué.

ALGORITHME ET PSEUDO-CODE

Types d'accès

On vient de voir que l'organisation des données au sein des enregistrements du fichier pouvait s'effecteur selon deux grands choix stratégiques. Mais il existe une autre ligne de partage des fichiers : le type d'accès, autrement dit la manière dont la machine va pouvoir aller rechercher les informations contenues dans le fichier. On distingue :

- L'accès séquentiel : on ne peut accéder qu'à la donnée suivant celle qu'on vient de lire. On ne peut donc accéder à une information qu'en ayant au préalable examiné celle qui la précède. Dans le cas d'un fichier texte, cela signifie qu'on lit le fichier ligne par ligne (enregistrement par enregistrement).

Dans le cadre de ce cours, on se limitera volontairement au type de base : le fichier texte en accès séquentiel.

Note : en PHP, nous verrons que nous pouvons récupérer le contenu du fichier en une seule ligne de code.

Instructions (fichiers texte en accès séquentiel)

Si l'on veut travailler sur un fichier, la première chose à faire est de l'ouvrir. Cela se fait en attribuant au fichier un numéro de canal. On ne peut ouvrir qu'un seul fichier par canal, mais quel que soit le langage, on dispose toujours de plusieurs canaux, donc pas de soucis.

L'important est que lorsqu'on ouvre un fichier, on stipule ce qu'on va en faire : lire, écrire ou ajouter.

- Si on ouvre un fichier pour lecture, on pourra uniquement récupérer les informations qu'il contient, sans les modifier en aucune manière.
- Si on ouvre un fichier pour écriture, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront intégralement écrasées Et on ne pourra pas accéder aux informations qui existaient précédemment.
- Si on ouvre un fichier pour ajout, on ne peut ni lire, ni modifier les informations existantes. Mais on pourra, comme vous commencez à vous en douter, ajouter de nouvelles lignes (je rappelle qu'au terme de lignes, on préférera celui d'enregistrements.

Au premier abord, ces instructions peuvent sembler limitées : il n'y a même pas d'instructions qui permettent de supprimer un enregistrement d'un fichier.

ALGORITHME ET PSEUDO-CODE

Pour ouvrir un fichier texte, on écrira par exemple :

```
Ouvrir "Exemple.txt" sur 4 en Lecture
```

Ici, "Exemple.txt" est le nom du fichier sur le disque dur, 4 est le numéro de canal, et ce fichier a donc été ouvert en lecture. Vous l'aviez sans doute pressenti. Allons plus loin :

```
        Variables Truc, Nom, Prénom, Tel, Mail en Caractères
Début
        Ouvrir "Exemple.txt" sur 4 en Lecture
        LireFichier 4, Truc
        Nom ← Mid(Truc, 1, 20)
        Prénom ← Mid(Truc, 21, 15)
        Tel ← Mid(Truc, 36, 10)
        Mail ← Mid(Truc, 46, 20)
```

L'instruction LireFichier récupère donc dans la variable spécifiée l'enregistrement suivant dans le fichier... "suivant", oui, mais par rapport à quoi ? Par rapport au dernier enregistrement lu. C'est en cela que le fichier est dit séquentiel. En l'occurrence, on récupère donc la première ligne, donc, le premier enregistrement du fichier, dans la variable Truc. Ensuite, le fichier étant organisé sous forme de champs de largeur fixe, il suffit de tronçonner cette variable Truc en autant de morceaux qu'il y a de champs dans l'enregistrement, et d'envoyer ces tronçons dans différentes variables.

La suite du raisonnement s'impose avec une logique impitoyable : lire un fichier séquentiel de bout en bout suppose de programmer une boucle. Comme on sait rarement à l'avance combien d'enregistrements comporte le fichier, la combine consiste neuf fois sur dix à utiliser la fonction EOF (acronyme pour End Of File). Cette fonction renvoie la valeur Vrai si on a atteint la fin du fichier (auquel cas une lecture supplémentaire déclencherait une erreur). L'algorithme, ultra classique, en pareil cas est donc :

```
        Variable Truc en Caractère
Début
        Ouvrir "Exemple.txt" sur 5 en Lecture
        Tantque Non EOF(5)
                LireFichier 5, Truc
                …
        FinTantQue
        Fermer 5
Fin
```

Et neuf fois sur dix également, si l'on veut stocker au fur et à mesure en mémoire vive les informations lues dans le fichier, on a recours à un ou plusieurs tableaux. Et comme on ne sait pas d'avance combien il y aurait d'enregistrements dans le fichier, on ne sait pas davantage combien il doit y avoir d'emplacements dans les tableaux.

ALGORITHME ET PSEUDO-CODE

En rassemblant l'ensemble des connaissances acquises, nous pouvons donc écrire le prototype du code qui effectue la lecture intégrale d'un fichier séquentiel, tout en recopiant l'ensemble des informations en mémoire vive :

```
        Tableaux Nom(), Prénom(), Tel(), Mail() en Caractère
Début
        Ouvrir "Exemple.txt" sur 5 en Lecture
        i ← -1
        Tantque Non EOF(5)
                LireFichier 5, Truc
                i ← i + 1
                Redim Nom(i)
                Redim Prénom(i)
                Redim Tel(i)
                Redim Mail(i)
                Nom(i) ← Mid(Truc, 1, 20)
                Prénom(i) ← Mid(Truc, 21, 15)
                Tel(i) ← Mid(Truc, 36, 10)
                Mail(i) ← Mid(Truc, 46, 20)
        FinTantQue
        Fermer 5
Fin
```

Ici, on a fait le choix de recopier le fichier dans quatre tableaux distincts. On aurait pu également tout recopier dans un seul tableau : chaque case du tableau aurait alors été occupée par une ligne complète (un enregistrement) du fichier. Cette solution nous aurait fait gagner du temps au départ, mais elle alourdit ensuite le code, puisque chaque fois que l'on a besoin d'une information au sein d'une case du tableau, il faudra aller procéder à une extraction via la fonction MID. Ce qu'on gagne par un bout, on le perd donc par l'autre.

Mais surtout, comme on va le voir bientôt, il y a autre possibilité, bien meilleure, qui cumule les avantages sans avoir aucun des inconvénients.

Pour une opération d'écriture, ou d'ajout, il faut d'abord impérativement, sous peine de semer la panique dans la structure du fichier, constituer une chaîne équivalente à la nouvelle ligne du fichier. Cette chaîne doit donc être « calibrée » de la bonne manière, avec les différents champs qui « tombent » aux emplacements corrects. Le moyen le plus simple pour s'épargner de longs traitements est de procéder avec des chaînes correctement dimensionnées dès leur déclaration (la plupart des langages offrent cette possibilité) :

```
Ouvrir "Exemple.txt" sur 3 en Ajout
Variable Truc en Caractère
Variables Nom*20, Prénom*15, Tel*10, Mail*20 en Caractère
```

ALGORITHME ET PSEUDO-CODE

Une telle déclaration assure que quel que soit le contenu de la variable Nom, par exemple, celle-ci comptera toujours 20 caractères. Si son contenu est plus petit, alors un nombre correct d'espaces sera automatiquement ajouté pour combler. Si on tente d'y entrer un contenu trop long, celui-ci sera automatiquement tronqué. Voyons la suite :

```
Nom ← "Jokers"
Prenom ← "Midnight"
Tel ← "0348946532"
Mail ← "allstars@rockandroll.com"
Truc ← Nom & Prénom & Tel & Mail
EcrireFichier 3, Truc
```

Et pour finir, une fois qu'on en a terminé avec un fichier, il ne faut pas oublier de fermer ce fichier. On libère ainsi le canal qu'il occupait (et accessoirement, on pourra utiliser ce canal dans la suite du programme pour un autre fichier… ou pour le même).

Les éléments dont nous avons besoin de connaitre en PHP

- Découper une chaine par un délimiteur "/", et ajouter les éléments découpés dans un tableau :

```
$aOfElements= explode("/",  $chaine);
```

- Rechercher une chaine "$sChaineAtrouver" et remplacer par une chaine "$sChaineRemplacante" dans un texte "$texte" :

```
$texte= str_replace($sChaineAtrouver, $sChaineRemplacante, $texte);
```

- Compléter par des espaces à droite d'une variable :

```
$sNom= str_pad($_POST["sNom"], 20, " ");
```

- Récupérer les 20 premiers caractères d'une chaine :

```
$sNom= substr($chaine, 0, 20);
```

- Récupérer les 10 caractères à partir du caractère 40 d'une chaine :

```
$sTel= substr($chaine, 40, 10);
```

- Ouvrir un fichier en lecture :

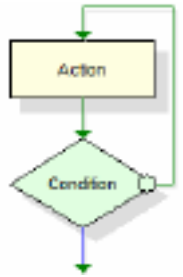
```
$sReferenceFichier = fopen('Carnet.txt', 'r');
if ($sReferenceFichier)  {
        while (!feof($sReferenceFichier))        {
                $buffer = fgets($sReferenceFichier);
        }
        fclose($sReferenceFichier);
}
```

ALGORITHME ET PSEUDO-CODE

- Ouvrir un fichier en écriture et écrire tout un contenu "$sTextFinal" dedans :

```
$sReferenceFichier = fopen('Carnet.txt', 'W');
if ($sReferenceFichier)
{
        fputs($sReferenceFichier, $sTextFinal);
        fclose($sReferenceFichier);
}
```

FAIRE LES EXERCICES 10.1 à 10.3



ALGORITHME ET PSEUDO-CODE

Stratégies de traitement

Il existe globalement deux manières de traiter les fichiers textes :

- l'une consiste à s'en tenir au fichier proprement dit, c'est-à-dire à modifier directement (ou presque) les informations sur le disque dur. C'est parfois un peu acrobatique, lorsqu'on veut supprimer un élément d'un fichier : on programme alors une boucle avec un test, qui recopie dans un deuxième fichier tous les éléments du premier fichier sauf un ; et il faut ensuite recopier intégralement le deuxième fichier à la place du premier fichier.
- l'autre stratégie consiste, comme on l'a vu, à passer par un ou plusieurs tableaux. En fait, le principe fondamental de cette approche est de commencer, avant toute autre chose, par recopier l'intégralité du fichier de départ en mémoire vive. Ensuite, on ne manipule que cette mémoire vive (concrètement, un ou plusieurs tableaux). Et lorsque le traitement est terminé, on recopie à nouveau dans l'autre sens, depuis la mémoire vive vers le fichier d'origine.

Les avantages de la seconde technique sont nombreux, et 99 fois sur 100, c'est ainsi qu'il faudra procéder :

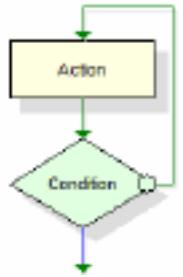
- la rapidité : les accès en mémoire vive sont des milliers de fois plus rapides (nanosecondes) que les accès aux mémoires de masse (millisecondes au mieux pour un disque dur). En basculant le fichier du départ dans un tableau, on minimise le nombre ultérieur d'accès disque, tous les traitements étant ensuite effectués en mémoire.
- la facilité de programmation : bien qu'il faille écrire les instructions de recopie du fichier dans le tableau, pour peu qu'on doive tripoter les informations dans tous les sens, c'est largement plus facile de faire cela avec un tableau qu'avec des fichiers.

Pourquoi, alors, ne fait-on pas cela à tous les coups ? Y a-t-il des cas où il vaut mieux en rester aux fichiers et ne pas passer par des tableaux ?

La recopie d'un très gros fichier en mémoire vive exige des ressources qui peuvent atteindre des dimensions considérables. Donc, dans le cas d'immenses fichiers (très rares, cependant), cette recopie en mémoire peut s'avérer problématique.

Toutefois, lorsque le fichier contient des données de type non homogènes (chaînes, numériques, etc.) cela risque d'être coton pour le stocker dans un tableau unique : il va falloir déclarer plusieurs tableaux, dont le maniement au final peut être aussi lourd que celui des fichiers de départ.

A moins... d'utiliser une ruse : créer des types de variables personnalisés, composés d'un « collage » de plusieurs types existants (10 caractères, puis un numérique, puis 15 caractères, etc.). Ce type de variable s'appelle un type structuré. Cette technique, bien qu'elle ne soit pas vraiment difficile, exige tout de même une certaine aisance... Voilà pourquoi on va maintenant en dire quelques mots.



ALGORITHME ET PSEUDO-CODE

Données structurées simples

Jusqu'à présent, voilà comment se présentaient nos possibilités en matière de mémoire vive : nous pouvions réserver un emplacement pour une information d'un certain type. Un tel emplacement s'appelle une variable. Nous pouvions aussi réserver une série d'emplacement numérotés pour une série d'informations **de même type**. Un tel emplacement s'appelle un tableau.

Voici maintenant que nous pouvons réserver **une série d'emplacements pour des données de type différents**. Un tel emplacement s'appelle une variable structurée. Son utilité, lorsqu'on traite des fichiers texte (et même, des fichiers en général), saute aux yeux : car on va pouvoir calquer chacune des lignes du fichier en mémoire vive, et considérer que chaque enregistrement sera recopié dans une variable et une seule, qui lui sera adaptée. Ainsi, le problème du "découpage" de chaque enregistrement en différentes variables (le nom, le prénom, le numéro de téléphone, etc.) sera résolu d'avance, puisqu'on aura une structure, un gabarit, en quelque sorte, tout prêt d'avance pour accueillir et prédécouper nos enregistrements.

Attention toutefois ; lorsque nous utilisions des variables de type prédéfini, comme des entiers, des booléens, etc. nous n'avions qu'une seule opération à effectuer : déclarer la variable en utilisant un des types existants. A présent que nous voulons créer un nouveau type de variable (par assemblage de types existants), il va falloir faire deux choses : d'abord, créer le type. Ensuite seulement, déclarer la (les) variable(s) d'après ce type.

Reprenons une fois de plus l'exemple du carnet d'adresses.

Nous allons donc, avant même la déclaration des variables, créer un type, une structure, calquée sur celle de nos enregistrements, et donc prête à les accueillir :

```
Structure Bottin
        Nom en Caractère * 20
        Prenom en Caractère * 15
        Tel en Caractère * 10
        Mail en Caractère * 20
Fin Structure
```

Ici, Bottin est le nom de ma structure. Ce mot jouera par la suite dans mon programme exactement le même rôle que les types prédéfinis comme Numérique, Caractère ou Booléen. Maintenant que la structure est définie, je vais pouvoir, dans la section du programme où s'effectuent les déclarations, créer une ou des variables correspondant à cette structure :

```
Variable Individu en Bottin
```

Et il est possible de remplir les différentes informations contenues au sein de la variable Individu de la manière suivante :

```
Individu ← "Joker", "Midnight", "0348946532", "allstars@rock.com"
```

ALGORITHME ET PSEUDO-CODE

On peut aussi avoir besoin d'accéder à un seul des champs de la variable structurée. Dans ce cas, on emploie le point :

```
Individu.Nom ← "Joker"
Individu.Prénom ← "Midnight"
Individu.Tel ← "0348946532"
Individu.Mail ← "allstars@rockandroll.com"
```

Ainsi, écrire correctement une information dans le fichier est un jeu d'enfant, puisqu'on dispose d'une variable Individu au bon gabarit. Une fois remplis les différents champs de cette variable - ce qu'on vient de faire -, il n'y a plus qu'à envoyer celle-ci directement dans le fichier.

EcrireFichier 3, Individu

De la même manière, dans l'autre sens, lorsque j'effectue une opération de lecture dans le fichier Adresses, cela en sera considérablement simplifiée : la structure étant faite pour cela, je peux dorénavant me contenter de recopier une ligne du fichier dans une variable de type Bottin, et le tour sera joué. Pour charger l'individu suivant du fichier en mémoire vive, il me suffira donc d'écrire :

LireFichier 5, Individu

Et là, direct, j'ai bien mes quatre renseignements accessibles dans les quatre champs de la variable individu. Tout cela, évidemment, parce que la structure de ma variable Individu correspond parfaitement à la structure des enregistrements de mon fichier.

Tableaux de données structurées

Si à partir des types simples, on peut créer des variables et des tableaux de variables, il en découle qu'à partir des types structurés, on peut créer des variables structurées... et des tableaux de variables structurées.

Cela veut dire que nous disposons d'une manière de gérer la mémoire vive qui va correspondre exactement à la structure d'un fichier texte (d'une base de données). Comme les structures se correspondent parfaitement, le nombre de manipulations à effectuer, autrement dit de lignes de programme à écrire, va être réduit au minimum. En fait, dans notre tableau structuré, les champs des emplacements du tableau correspondront aux champs du fichier texte, et les indices des emplacements du tableaux correspondront aux différentes lignes du fichier.

ALGORITHME ET PSEUDO-CODE

Voici, à titre d'illustration, l'algorithme complet de lecture du fichier Adresses et de sa recopie intégrale en mémoire vive, en employant un tableau structuré.

```
        Structure Bottin
                Nom en Caractère * 20
                Prenom en Caractère * 15
                Tel en Caractère * 10
                Mail en Caractère * 20
        Fin Structure
        Tableau Mespotes() en Bottin
Début
        Ouvrir "Exemple.txt" sur 3 en Lecture
        i ← -1
        Tantque Non EOF(3)
                i ← i + 1
                Redim Mespotes(i)
                LireFichier 3, Mespotes(i)
        FinTantQue
        Fermer 3
Fin
```

Une fois que ceci est réglé, on a tout ce qu'il faut ! Si je voulais écrire, à un moment, le mail de l'individu n°13 du fichier (donc le n°12 du tableau) à l'écran, il me suffirait de passer l'ordre :

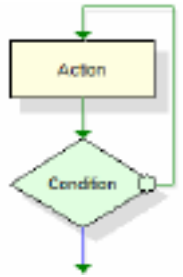
```
Ecrire Mespotes(12).Mail
```

REMARQUE FINALE SUR LES DONNÉES STRUCTURÉES

Même si le domaine de prédilection des données structurées est la gestion de fichiers, on peut tout à fait y avoir recours dans d'autres contextes, et organiser plus systématiquement les variables d'un programme sous la forme de telles structures.

En programmation dite **procédurale**, celle que nous étudions ici, ce type de stratégie reste relativement rare. Mais rare ne veut pas dire interdit, ou même inutile.

Et nous aurons l'occasion de voir qu'en programmation **objet**, ce type d'organisation des données devient fondamental.



ALGORITHME ET PSEUDO-CODE

Récapitulatif général

Lorsqu'on est amené à travailler avec des données situées dans un fichier, plusieurs choix, en partie indépendants les uns des autres, doivent être faits :

- sur l'organisation en enregistrements du fichier (choix entre fichier texte ou fichier binaire)
- sur le mode d'accès aux enregistrements du fichier (direct ou séquentiel)
- sur l'organisation des champs au sein des enregistrements (présence de séparateurs ou champs de largeur fixe)
- sur la méthode de traitement des informations (recopie intégrale préalable du fichier en mémoire vive ou non)
- sur le type de variables utilisées pour cette recopie en mémoire vive (plusieurs tableaux de type simple, ou un seul tableau de type structuré).

Chacune de ces options présente avantages et inconvénients, et il est impossible de donner une règle de conduite valable en toute circonstance. Il faut connaître ces techniques, et savoir choisir la bonne option selon le problème à traiter.

Voici une série d'exercices sur les fichiers texte, que l'on pourra traiter en employant les types structurés (c'est en tout cas le cas dans les corrigés).

FAIRE LES EXERCICES 10.4 à 10.9

ALGORITHME ET PSEUDO-CODE

14.) Procédures et Fonctions

1. Fonctions personnalisées

De quoi s'agit-il ?

Une application, surtout si elle est longue, a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement. Par exemple, la saisie d'une réponse par oui ou par non (et le contrôle qu'elle implique), peuvent être répétés dix fois à des moments différents de la même application, pour dix questions différentes.

La manière la plus évidente, mais aussi la moins habile, de programmer ce genre de choses, c'est bien entendu de répéter le code correspondant autant de fois que nécessaire. Apparemment, on ne se casse pas la tête : quand il faut que la machine interroge l'utilisateur, on recopie les lignes de codes voulues en ne changeant que le nécessaire. Mais en procédant de cette manière, la pire qui soit, on se prépare des lendemains qui déchantent...

D'abord, parce que si la structure d'un programme écrit de cette manière peut paraître simple, elle est en réalité inutilement lourde. Elle contient des répétitions, et pour peu que le programme soit conséquent, il peut devenir parfaitement illisible. Or, le fait d'être facilement modifiable donc lisible, y compris - et surtout - par ceux qui ne l'ont pas écrit est un critère essentiel pour un programme informatique ! Dès que l'on programme non pour soi-même, mais dans le cadre d'une organisation (entreprise ou autre), cette nécessité se fait sentir de manière importante. On ne peut pas l'ignorer.

En plus, à un autre niveau, une telle structure pose des problèmes considérables de maintenance : car en cas de modification du code, il va falloir traquer toutes les apparitions plus ou moins identiques de ce code pour faire convenablement la modification ! Et si l'on en oublie une, on a laissé un bug.

Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à regrouper les instructions qui le composent en un module séparé. Il ne restera alors plus qu'à appeler ce groupe d'instructions (qui n'existe donc désormais qu'en un exemplaire unique) à chaque fois qu'on en a besoin. Ainsi, la lisibilité est assurée ; le programme devient **modulaire**, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l'application.

ALGORITHME ET PSEUDO-CODE

Le corps du programme s'appelle alors la **procédure principale**, et ces groupes d'instructions auxquels on a recours s'appellent des **fonctions** et des **sous-procédures** (nous verrons un peu plus loin la différence entre ces deux termes).

Reprenons un exemple de question à laquelle l'utilisateur doit répondre par oui ou par non.

Mauvaise Structure :

```
...
Ecrire "Etes-vous marié ?"
Rep1 ← ""
TantQue Rep1 <> "Oui" et Rep1 <> "Non"
        Ecrire "Tapez Oui ou Non"
        Lire Rep1
FinTantQue
...
Ecrire "Avez-vous des enfants ?"
Rep2 ← ""
TantQue Rep2 <> "Oui" et Rep2 <> "Non"
        Ecrire "Tapez Oui ou Non"
        Lire Rep2
FinTantQue
...
```

On le voit bien, il y a là une répétition quasi identique du traitement à accomplir. A chaque fois, on demande une réponse par Oui ou Non, avec contrôle de saisie. La seule chose qui change, c'est l'intitulé de la question, et le nom de la variable dans laquelle on range la réponse.

La solution, on vient de le voir, consiste à **isoler les instructions** demandant une réponse par Oui ou Non, et à appeler ces instructions à chaque fois que nécessaire. Ainsi, on évite les répétitions inutiles, et on a découpé notre problème en petits morceaux autonomes.

Nous allons donc créer une **fonction** dont le rôle sera de **renvoyer** la réponse (oui ou non) de l'utilisateur. Ce mot de "fonction", en l'occurrence, ne doit pas nous surprendre : nous avons étudié précédemment des fonctions fournies

ALGORITHME ET PSEUDO-CODE

avec le langage, et nous avons vu que le but d'une fonction était de renvoyer une valeur. Eh bien, c'est exactement la même chose ici, sauf que c'est nous qui allons créer notre propre fonction, que nous appellerons RepOuiNon :

```
Fonction RepOuiNon() en caractère
        Truc ← ""
        TantQue Truc <> "Oui" et Truc <> "Non"
                Ecrire "Tapez Oui ou Non"
                Lire Truc
        FinTantQue
        Renvoyer Truc
Fin Fonction
```

On remarque au passage l'apparition d'un nouveau mot-clé : **Renvoyer**, qui indique quelle valeur doit prendre la fonction lorsqu'elle est utilisée par le programme. Cette valeur renvoyée par la fonction (ici, la valeur de la variable Truc) est en quelque sorte contenue dans le nom de la fonction lui-même, exactement comme c'était le cas dans les fonctions prédéfinies.

Une fonction s'écrit gébéralement **en-dehors de la procédure principale**. Selon les langages, cela peut prendre différentes formes. Mais ce qu'il faut comprendre, c'est que ces quelques lignes de codes sont en quelque sorte des satellites, qui existent en dehors du traitement lui-même. Simplement, elles sont à sa disposition, et il pourra y faire appel chaque fois que nécessaire. Si l'on reprend notre exemple, une fois notre fonction RepOuiNon écrite, le programme principal comprendra les lignes :

Bonne structure :

```
...
Ecrire "Etes-vous marié ?"
Rep1 ← RepOuiNon()
...
Ecrire "Avez-vous des enfants ?"
Rep2 ← RepOuiNon()
...
```

On a ainsi évité les répétitions inutiles, et si d'aventure, il y avait un bug dans notre contrôle de saisie, il suffirait de faire une seule correction dans la fonction RepOuiNon pour que ce bug soit éliminé de toute l'application.

ALGORITHME ET PSEUDO-CODE

Toutefois, les plus sagaces d'entre vous auront remarqué, tant dans le titre de la fonction que dans chacun des appels, la présence de **parenthèses**. Celles-ci, dès qu'on déclare ou qu'on appelle une fonction, sont obligatoires. Et si vous avez bien compris tout ce qui précède, vous devez avoir une petite idée de ce qu'on va pouvoir mettre dedans...

Passage d'arguments

Reprenons l'exemple qui précède et analysons-le. On écrit un message à l'écran, puis on appelle la fonction RepOuiNon pour poser une question ; puis, un peu plus loin, on écrit un autre message à l'écran, et on appelle de nouveau la fonction pour poser la même question, etc. C'est une démarche acceptable, mais qui peut encore être améliorée : puisque avant chaque question, on doit écrire un message, autant que cette écriture du message figure directement dans la fonction appelée. Cela implique deux choses :

- lorsqu'on appelle la fonction, on doit lui préciser quel message elle doit afficher avant de lire la réponse
- la fonction doit être « prévenue » qu'elle recevra un message, et être capable de le récupérer pour l'afficher.

En langage algorithmique, on dira que **le message devient un argument (ou un paramètre) de la fonction**. Nous avons longuement utilisé les arguments à propos des fonctions prédéfinies. La fonction sera dorénavant déclarée comme suit :

```
Fonction RepOuiNon(Msg en Caractère) en Caractère
        Ecrire Msg
        Truc ← ""
        TantQue Truc <> "Oui" et Truc <> "Non"
                Ecrire "Tapez Oui ou Non"
                Lire Truc
        FinTantQue
        Renvoyer Truc
Fin Fonction
```

ALGORITHME ET PSEUDO-CODE

Il y a donc maintenant entre les parenthèses une variable, Msg, dont on précise le type, et qui signale à la fonction qu'un argument doit lui être envoyé à chaque appel. Quant à ces appels, justement, ils se simplifieront encore dans la procédure principale, pour devenir :

```
...
Rep1 ← RepOuiNon("Etes-vous marié ?")
...
Rep2 ← RepOuiNon("Avez-vous des enfants ?")
...
```

Une remarque importante : là, on n'a passé qu'un seul argument en entrée. Mais bien entendu, on peut en passer autant qu'on veut, et créer des fonctions avec deux, trois, quatre, etc. arguments ; Simplement, il faut éviter d'être gourmands, et il suffit de passer ce dont on en a besoin, ni plus, ni moins !

Dans le cas que l'on vient de voir, le passage d'un argument à la fonction était élégant, mais pas indispensable. La preuve, cela marchait déjà très bien avec la première version. Mais on peut imaginer des situations où il faut absolument concevoir la fonction de sorte qu'on doive lui transmettre un certain nombre d'arguments si l'on veut qu'elle puisse remplir sa tâche. Prenons, par exemple, toutes les fonctions qui vont effectuer des calculs. Que ceux-ci soient simples ou compliqués, il va bien falloir envoyer à la fonction les valeurs grâce auxquelles elle sera censé produire son résultat (pensez tout bêtement à une fonction sur le modèle d'Excel, telle que celle qui doit calculer une somme ou une moyenne). C'est également vrai des fonctions qui traiteront des chaînes de caractères. Bref, dans 99% des cas, lorsqu'on créera une fonction, celle-ci devra comporter des arguments.

Deux mots sur l'analyse fonctionnelle

Le plus difficile, mais aussi le plus important, c'est d'acquérir le réflexe de **constituer systématiquement les fonctions adéquates quand on doit traiter un problème donné**, et de flairer la bonne manière de découper son algorithme en différentes fonctions pour le rendre léger, lisible et performant.

Le terme consacré parle d'ailleurs à ce sujet de **factorisation du code** : c'est une manière de parler reprise des matheux, qui « factorisent » un calcul, c'est-à-dire qui en regroupent les éléments communs pour éviter qu'ils ne se répètent. Cette factorisation doit, tant qu'à faire, être réalisée avant de rédiger le programme : il est toujours mieux de concevoir les choses directement dans leur meilleur état final possible. Mais même quand on s'est fait avoir, et qu'on a laissé passer des éléments de code répétitifs, il faut toujours les factoriser, c'est-à-dire les regrouper en fonctions, et ne pas laisser des redondances.

ALGORITHME ET PSEUDO-CODE

La phase de conception d'une application qui suit l'analyse et qui précède l'algorithmique proprement dite, et qui se préoccupe donc du découpage en modules du code s'appelle l'**analyse fonctionnelle** d'un problème. C'est une phase qu'il ne faut surtout pas omettre ! Donc, pour concevoir une application :

1. On identifie le problème à traiter, en inventoriant les fonctionnalités nécessaires, les tenants et les aboutissants, les règles explicites ou implicites, les cas compliqués, etc. C'est l'**analyse**.
2. On procède à un découpage de l'application entre une procédure qui jouera le rôle de chef d'orchestre, ou de donneur d'ordre, et des modules périphériques (fonctions) qui joueront le rôle de sous-traitants spécialisés. **C'est l'analyse fonctionnelle.**
3. On détaille l'enchaînement logique des traitements de chaque (sous-)procédure ou fonction : c'est l'algorithmique.
4. On procède sur machine à l'écriture (et au test) de chaque module dans le langage voulu : c'est le codage, ou la programmation proprement dite.

Variables locales et globales

Nous venons de voir que nous pouvions (et devions) découper un long traitement comportant éventuellement des redondances (notre application) en différents modules. Et nous avons vu que les informations pouvaient être transmises entre ces modules selon deux modes :

- si le module appelé est une fonction, par le **retour du résultat**
- dans tous les cas, par la **transmission de paramètres** (que ces paramètres soient passés par valeur ou par référence)

En fait, il existe un troisième et dernier moyen d'échanger des informations entre différentes procédures et fonctions : c'est de ne pas avoir besoin de les échanger, en faisant en sorte que ces procédures et fonctions partagent littéralement les mêmes variables, sous les mêmes noms. Cela suppose d'avoir recours à des variables particulières, lisibles et utilisables par n'importe quelle procédure ou fonction de l'application.
Par défaut, une variable est déclarée au sein d'une procédure ou d'une fonction. Elle est donc créée avec cette procédure, et disparaît avec elle. Durant tout le temps de son existence, une telle variable n'est visible que par la procédure qui l'a vu naître. Si je crée une variable Toto dans une procédure Bidule, et qu'en cours de route, ma procédure Bidule appelle une sous-procédure Machin, il est hors de question que Machin puisse accéder à Toto, ne serait-ce que pour connaître sa valeur (et ne parlons pas de la modifier). Voilà pourquoi ces variables par défaut sont dites **locales**.

ALGORITHME ET PSEUDO-CODE

Mais à côté de cela, il est possible de créer des variables qui certes, seront déclarées dans une procédure, mais qui du moment où elles existeront, seront des variables communes à toutes les procédures et fonctions de l'application. Avec de telles variables, le problème de la transmission des valeurs d'une procédure (ou d'une fonction) à l'autre ne se pose même plus : la variable Truc, existant pour toute l'application, est accessible et modifiable depuis n'importe quelle ligne de code de cette application. Plus besoin donc de la transmettre ou de la renvoyer. Une telle variable est alors dite **globale**.

ALGORITHME ET PSEUDO-CODE

Par défaut, toute variable est locale. La manière dont la déclaration d'une variable globale doit être faites est évidemment fonction de chaque langage de programmation. En pseudo-code algorithmique, on pourra utiliser le mot-clé **Globale** :

Variable Globale Toto **en Numérique**

Alors, pourquoi ne pas rendre toutes les variables globales, et s'épargner ainsi de fastidieux efforts pour passer des paramètres ? C'est très simple, et c'est toujours la même chose : d'une part, les variables globales consomment énormément de ressources en mémoire. En conséquence, le principe qui doit présider au choix entre variables globales et locales doit être celui de l'économie de moyens. Mais d'autre part, et surtout, multiplier les variables globales est une manière peu sécurisée de programmer. C'est exactement comme quand dans un navire, on fait sauter les compartiments internes : s'il y a une voie d'eau quelque part, c'est toute la coque qui se remplit, et le bateau coule. En programmant sous forme de modules ne s'échangeant des informations que via des arguments, on adopte une architecture compartimentée, et on réduit ainsi considérablement les risques qu'un problème quelque part contamine l'ensemble de la construction.

Moralité, on ne déclare comme globales que les variables qui doivent absolument l'être. Et chaque fois que possible, lorsqu'on crée une sous-procédure, on utilise le passage de paramètres plutôt que des variables globales.

Peut-on tout faire ?

A cette question, la réponse est bien évidemment : oui, on peut tout faire. Mais c'est précisément la raison pour laquelle on peut vite en arriver à faire aussi absolument n'importe quoi.

Par exemple, un mélange de paramètres passés par référence, de variables globales, de procédures et de fonctions mal choisies, finiraient par accoucher d'un code absolument illogique, illisible, et dans lequel la chasse à l'erreur relèverait de l'exploit.

Le principe qui doit guider tout programmeur est celui de la solidité et de la clarté du code. **Une application bien programmée est une application à l'architecture claire, dont les différents modules font ce qu'ils disent, disent ce qu'il font, et peuvent être testés (ou modifiés) un par un sans perturber le reste de la construction**.

ALGORITHME ET PSEUDO-CODE

Il convient donc :

1. de **limiter au minimum l'utilisation des variables globales**. Celles-ci doivent être employées avec parcimonie et à bon escient.
2. de **regrouper sous forme de modules distincts** tous les morceaux de code qui possèdent une certaine unité fonctionnelle (programmation par "blocs"). C'est-à-dire de faire la chasse aux lignes de codes redondantes, ou quasi-redondantes.
3. de faire de ces modules **des fonctions lorsqu'ils renvoient un résultat unique**

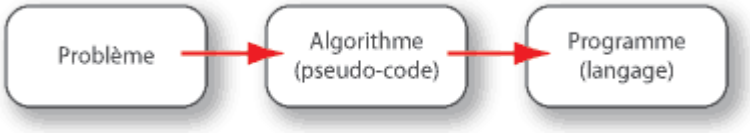
ALGORITHME ET PSEUDO-CODE

15.) Notions Complémentaires

Interprétation et compilation

Jusqu'ici, nous avons travaillé sur la première étape de la réalisation d'un programme : la rédaction de l'algorithme.

En fait, si l'algorithme est bien écrit, sans faute logique, l'étape suivante ne doit normalement poser aucun problème conceptuel. Il n'y a plus qu'à effectuer une simple traduction.



A partir de là, le travail du programmeur est virtuellement terminé (en réalité, il reste tout de même une inévitable phase de tests, de corrections, etc., qui s'avère souvent très longue). Mais en tout cas, pour l'ordinateur, c'est là que les ennuis commencent. En effet, aucun ordinateur n'est en soi apte à exécuter les instructions telles qu'elles sont rédigées dans tel ou tel langage ; l'ordinateur, lui, ne comprend qu'un seul langage, qui est un langage codé en binaire (à la rigueur en hexadécimal) et qui s'appelle le langage machine (ou assembleur).

C'est à cela que sert un langage à vous épargner la programmation en binaire (quasi impossible, vous vous en doutez) et vous permettre de vous faire comprendre de l'ordinateur d'une manière lisible.

C'est pourquoi tout langage, à partir d'un programme écrit, doit obligatoirement procéder à une **traduction** en langage machine pour que ce programme soit exécutable.

ALGORITHME ET PSEUDO-CODE

Il existe deux stratégies de traduction, ces deux stratégies étant parfois disponibles au sein du même langage.

- le langage traduit les instructions au fur et à mesure qu'elles se présentent. Cela s'appelle la **compilation à la volée**, ou **l'interprétation**.
- le langage commence par traduire l'ensemble du programme en langage machine, constituant ainsi un deuxième programme (un deuxième fichier) distinct physiquement et logiquement du premier. Ensuite, et ensuite seulement, il exécute ce second programme. Cela s'appelle la **compilation**

Il va de soi qu'un langage interprété est plus maniable : on peut exécuter directement son code - et donc le tester - au fur et à mesure qu'on le tape, sans passer à chaque fois par l'étape supplémentaire de la compilation. Mais il va aussi de soi qu'un programme compilé s'exécute beaucoup plus rapidement qu'un programme interprété : le gain est couramment d'un facteur 10, voire 20 ou plus.

Une logique vicelarde : la programmation récursive

La programmation des fonctions personnalisées a donné lieu à l'essor d'une logique un peu particulière, adaptée en particulier au traitement de certains problèmes mathématiques (ou de jeux) : la programmation récursive. Pour vous expliquer de quoi il retourne, nous allons reprendre un exemple cher à vos cœurs : le calcul d'une factorielle .

Rappelez-vous : la formule de calcul de la factorielle d'un nombre n s'écrit :

N ! = 1 x 2 x 3 x … x n

Nous avions programmé cela aussi sec avec une boucle Pour. Mais une autre manière de voir les choses, ni plus juste, ni moins juste, serait de dire que quel que soit le nombre n :

n ! = n x (n-1) !

En bon français : la factorielle d'un nombre, c'est ce nombre multiplié par la factorielle du nombre précédent. Encore une fois, c'est une manière ni plus juste ni moins juste de présenter les choses ; c'est simplement une manière différente.

Si l'on doit programmer cela, on peut alors imaginer une fonction Fact, chargée de calculer la factorielle. Cette fonction effectue la multiplication du nombre passé en argument par la factorielle du nombre précédent. Et cette factorielle du nombre précédent va bien entendu être elle-même calculée par la fonction Fact.

ALGORITHME ET PSEUDO-CODE

Autrement dit, on va créer une fonction qui pour fournir son résultat, **va s'appeler elle-même un certain nombre de fois**. C'est cela, la récursivité.

Toutefois, il nous manque une chose pour finir : quand ces auto-appels de la fonction Fact vont-ils s'arrêter ? Cela n'aura-t-il donc jamais de fin ? Si, bien sûr, on s'arrête quand on arrive au nombre 1, pour lequel la factorielle est par définition 1.

Cela produit l'écriture suivante, un peu déconcertante certes, mais parfois très pratique :

```
Fonction Fact (N en Numérique)
        Si N = 0 alors
                Renvoyer 1
        Sinon
                Renvoyer Fact(N-1) * N
        Finsi
Fin Fonction
```

Vous remarquerez que le processus récursif remplace en quelque sorte la boucle, c'est-à-dire un processus itératif. Vous remarquerez aussi qu'on traite le problème à l'envers : on part du nombre, et on remonte à rebours jusqu'à 1 pour pouvoir calculer la factorielle. Cet effet de rebours est caractéristique de la programmation récursive.

Pour conclure sur la récursivité, trois remarques fondamentales.

- la programmation récursive, pour traiter certains problèmes, est **très économique pour le programmeur** ; elle permet de faire les choses correctement, en très peu d'instructions.
- en revanche, elle est **très dispensieuse de ressources machine**. Car à l'exécution, la machine va être obligée de créer autant de variables temporaires que de « tours » de fonction en attente.
- et **tout problème formulé en termes récursifs peut également être formulé en termes itératifs !**

Donc, si la programmation récursive peut faciliter la vie du programmeur, elle n'est jamais indispensable. Mais cette technique est à voir, on peut toujours tomber un jour ou l'autre sur un programme ayant de telles fonctions écrites par un développeur.