Cours Algorithme

I. Les Variables

1. 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 données issues du disque dur, fournies par l'utilisateur (frappées au clavier), ou que sais-je encore. Il peut aussi 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.
En réalité, dans la mémoire vive de l'ordinateur, il n'y a bien sûr pas une vraie boîte, et pas davantage de vraie étiquette collée dessus (j'avais bien prévenu que la boîte et l'étiquette, c'était une image). Dans l'ordinateur, physiquement, il y a un emplacement de mémoire, repéré par une adresse binaire. Si on programmait dans un langage directement compréhensible par la machine, on devrait se fader de désigner nos données par de superbes 10011001 et autres 01001001 (enchanté !).
Mauvaise nouvelle : de tels langages existent ! Ils portent le doux nom d'assembleur. Bonne nouvelle : ce ne sont pas les seuls langages disponibles.
Les langages informatiques plus évolués (ce sont ceux que presque tout le monde emploie) se chargent précisément, entre autres rôles, d'épargner au programmeur la gestion fastidieuse des emplacements mémoire et de leurs adresses. Et, comme vous commencez à le comprendre, il est beaucoup plus facile d'employer les étiquettes de son choix, que de devoir manier des adresses binaires.

2. 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. C'est un genre de déclaration certes moins romantique qu'une déclaration d'amour, mais d'un autre côté moins désagréable qu'une déclaration d'impôts.
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. Quant au nombre maximal de signes pour un nom de variable, il dépend du langage utilisé.

En pseudo-code algorithmique, on est bien sûr libre du nombre de signes pour un nom de variable, même si pour des raisons purement pratiques, et au grand désespoir de Stéphane Bern, on évite généralement les noms à rallonge.
Lorsqu'on déclare une variable, il ne suffit pas de créer une boîte (réserver un emplacement mémoire) ; encore doit-on préciser ce que l'on voudra mettre dedans, car de cela dépendent la taille de la boîte (de l'emplacement mémoire) et le type de codage utilisé.

2.1 Types numériques classiques
Commençons par le cas très fréquent, celui d'une variable destinée à recevoir des nombres.
Si l'on réserve un octet pour coder un nombre, je rappelle pour ceux qui dormaient en lisant le chapitre précédent qu'on ne pourra coder que $2^8 = 256$ valeurs différentes. Cela peut signifier par exemple les nombres entiers de 1 à 256, ou de 0 à 255, ou de –127 à +128… Si l'on réserve deux octets, on a droit à 65 536 valeurs ; avec trois octets, 16 777 216, etc. Et là se pose un autre problème : ce codage doit-il représenter des nombres décimaux ? des nombres négatifs ?
Bref, le type de codage (autrement dit, 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
Byte (octet)	0 à 255
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

En pseudo-code, une déclaration de variables aura ainsi cette tête :

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

2.2 Autres types numériques
Certains langages autorisent d'autres types numériques, notamment :

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

Nous n'emploierons pas ces types dans ce cours ; mais je les signale, car vous ne manquerez pas de les rencontrer en programmation proprement dite.

2.3 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 (également appelé type caractère, type chaîne ou en anglais, le type string – mais ne fantasmez pas trop vite, les string, c'est loin d'être aussi excitant que le nom le suggère. Une étudiante qui se reconnaîtra si elle lit ces lignes a d'ailleurs mis le doigt - si j'ose m'exprimer ainsi - sur le fait qu'il en va de même en ce qui concerne les bytes).
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. Le nombre maximal de caractères pouvant être stockés dans une seule variable string dépend du langage utilisé.
Un groupe de caractères (y compris un groupe de un, ou de zéro caractères), qu'il soit ou non stocké dans une variable, d'ailleurs, est donc souvent appelé chaîne de caractères.

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

Pour éviter deux sources principales de possibles confusions :

- la confusion entre des nombres et des suites de chiffres. Par exemple, 423 peut représenter le nombre 423 (quatre cent vingt-trois), ou la suite de caractères 4, 2, et 3. Et ce n'est pas du tout la même chose ! Avec le premier, on peut faire des calculs, avec le second, point du tout. Dès lors, les guillemets permettent d'éviter toute ambiguïté : s'il n'y en a pas, 423 est quatre cent vingt trois. S'il y en a, "423" représente la suite des chiffres 4, 2, 3.
- …Mais ce n'est pas le pire. L'autre confusion, bien plus grave - et bien plus fréquente – consiste à se mélanger les pinceaux entre le nom d'une variable et son contenu. Pour parler simplement, cela consiste à confondre l'étiquette d'une boîte et ce qu'il y a à l'intérieur… On reviendra sur ce point crucial dans quelques instants.

2.4 Type booléen

Le dernier type de variables est le type booléen : on y stocke uniquement les valeurs logiques VRAI et FAUX.
On peut représenter ces notions abstraites de VRAI et de FAUX par tout ce qu'on veut : de l'anglais (TRUE et FALSE) ou des nombres (0 et 1). Peu importe. 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.

3. L'instruction d'affectation

3.1 Syntaxe et signification

Ouf, après tout ce baratin préliminaire, on aborde enfin nos premières véritables manipulations d'algorithmique. Pas trop tôt, certes, mais pas moyen de faire autrement !
En fait, la variable (la boîte) n'est pas un outil bien sorcier à manipuler. A la différence du couteau suisse ou du superbe robot ménager vendu sur Télé Boutique Achat, on ne peut pas faire trente-six mille choses avec une variable, mais seulement une et une seule.
Cette 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 ←

Ainsi :

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. C'est un peu comme si, en donnant un ordre à quelqu'un, on accolait un verbe et un complément incompatibles, du genre « Epluchez la casserole ». Même dotée de la meilleure volonté du monde, la ménagère lisant cette phrase ne pourrait qu'interrompre dubitativement sa tâche. Alors, un ordinateur, vous pensez bien…
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. De même que précédemment, Toto vaut toujours 12.
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.
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.

3.2 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 :

Exemple 1
Variable A **en Numérique**
Début
A ← 12
Fin

Exemple 2
Variable A **en Numérique**
Début
A ← 34
Fin

Tous les éléments sont maintenant en votre possession pour que ce soit à vous de jouer !

4. Expressions et opérateurs

Si on fait le point, on s'aperçoit que dans une instruction d'affectation, on trouve :

- à gauche de la flèche, un nom de variable, et uniquement cela. En ce monde empli de doutes qu'est celui de l'algorithmique, c'est une des rares règles d'or qui marche à tous les coups : si

on voit à gauche d'une flèche d'affectation autre chose qu'un nom de variable, on peut être certain à 100% qu'il s'agit d'une erreur.

- à droite de la flèche, ce qu'on appelle une expression. Voilà encore un mot qui est trompeur ; en effet, ce mot existe dans le langage courant, où il revêt bien des significations. Mais en informatique, le terme d'expression ne désigne qu'une seule chose, et qui plus est une chose très précise :

Une expression est un ensemble de valeurs, reliées par des opérateurs, et équivalent à une seule valeur
Cette définition vous paraît peut-être obscure. Mais réfléchissez-y quelques minutes, et vous verrez qu'elle recouvre quelque chose d'assez simple sur le fond. Par exemple, voyons quelques expressions de type numérique. Ainsi :
7
5+4
123-45+844
Toto-12+5-Riri
…sont toutes des expressions valides, pour peu que Toto et Riri soient bien des nombres. Car dans le cas contraire, la quatrième expression n'a pas de sens. En l'occurrence, les opérateurs que j'ai employés sont l'addition (+) et la soustraction (-).
Revenons pour le moment sur l'affectation. Une condition supplémentaire (en plus des deux précédentes) 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 l'un des trois points énumérés ci-dessus n'est pas respecté, la machine sera incapable d'exécuter l'affectation, et déclenchera une erreur (est-il besoin de dire que si aucun de ces points n'est respecté, il y aura aussi 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.
Les opérateurs possibles dépendent du type des valeurs qui sont en jeu. Allons-y, faisons le tour, c'est un peu fastidieux, mais comme dit le sage au petit scarabée, quand c'est fait, c'est plus à faire.

4.1 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.
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. Rien de difficile là-dedans, que du normal.

4.2 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"

5. Deux remarques pour terminer

Maintenant que nous sommes familiers des variables et que nous les manipulons les yeux fermés (mais les neurones en éveil, toutefois), 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.

6. Exercices

Enonce des Exercices	Corrigés des Exercices
Exercice 1.1 Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ? **Variables** A, B **en Entier** **Début** A ← 1	Exercice 1.1 Après La valeur des variables est : A ← 1 A = 1 B = ? B ← A + 3 A = 1 B = 4 A ← 3 **A = 3** **B = 4**

B ← A + 3
A ← 3
Fin

Exercice 1.2
Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?
Variables A, B, C **en Entier**
Début
A ← 5
B ← 3
C ← A + B
A ← 2
C ← B – A
Fin

Exercice 1.2

Après	La valeur des variables est :		
A ← 5	A = 5	B = ?	C = ?
B ← 3	A = 5	B = 3	C = ?
C ← A + B	A = 5	B = 3	C = 8
A ← 2	A = 2	B = 3	C = 8
C ← B – A	**A = 2**	**B = 3**	**C = 1**

Exercice 1.3
Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?
Variables A, B **en Entier**
Début
A ← 5
B ← A + 4
A ← A + 1
B ← A – 4
Fin

Exercice 1.3

Après	La valeur des variables est :	
A ← 5	A = 5	B = ?
B ← A + 4	A = 5	B = 9
A ← A + 1	A = 6	B = 9
B ← A – 4	**A = 6**	**B = 2**

Exercice 1.4
Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?
Variables A, B, C **en Entier**
Début
A ← 3
B ← 10
C ← A + B
B ← A + B
A ← C
Fin

Exercice 1.4

Après	La valeur des variables est :		
A ← 3	A = 3	B = ?	C = ?
B ← 10	A = 3	B = 10	C = ?
C ← A + B	A = 3	B = 10	C = 13
B ← A + B	A = 3	B = 13	C = 13
A ← C	**A = 13**	**B = 13**	**C = 13**

Exercice 1.5
Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?
Variables A, B **en Entier**
Début
A ← 5
B ← 2
A ← B
B ← A
Fin

Exercice 1.5

Après	La valeur des variables est :	
A ← 5	A = 5	B = ?
B ← 2	A = 5	B = 2
A ← B	A = 2	B = 2
B ← A	**A = 2**	**B = 2**

Les deux dernières instructions ne permettent donc pas d'échanger les deux valeurs de B et A, puisque l'une des deux valeurs (celle de A) est

Moralité : les deux dernières instructions permettent-elles d'échanger les deux valeurs de B et A ? Si l'on inverse les deux dernières instructions, cela change-t-il quelque chose ?	ici écrasée. Si l'on inverse les deux dernières instructions, cela ne changera rien du tout, hormis le fait que cette fois c'est la valeur de B qui sera écrasée.
Exercice 1.6 Plus difficile, mais c'est un classique absolu, qu'il faut absolument maîtriser : écrire un algorithme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable.	Exercice 1.6 **Début** C ← A A ← B B ← C **Fin**
Exercice 1.7 Une variante du précédent : on dispose de trois variables A, B et C. Ecrivez un algorithme transférant à B la valeur de A, à C la valeur de B et à A la valeur de C (toujours quels que soient les contenus préalables de ces variables).	Exercice 1.7 **Début** D ← C C ← B B ← A A ← D **Fin**
Exercice 1.8 Que produit l'algorithme suivant ? **Variables** A, B, C **en Caractères** **Début** A ← "423" B ← "12" C ← A + B **Fin**	Exercice 1.8 Il ne peut produire qu'une erreur d'exécution, puisqu'on ne peut pas additionner des caractères.
Exercice 1.9 Que produit l'algorithme suivant ? **Variables** A, B, C **en Caractères** **Début** A ← "423" B ← "12" C ← A & B **Fin**	Exercice 1.9 …En revanche, on peut les concaténer. A la fin de l'algorithme, C vaudra donc "42312".

II. Lecture et Ecriture

Trifouiller des variables en mémoire vive par un chouette programme, c'est vrai que c'est très marrant, et d'ailleurs on a tous bien rigolé au chapitre précédent. Cela dit, à la fin de la foire, on peut tout de même se demander à quoi ça sert.
En effet. 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. C'est très gentil à lui. 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 indubitablement 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, heureusement, il existe des d'instructions pour permettre à la machine de dialoguer avec l'utilisateur (et Lycée de Versailles, eût ajouté l'estimé Pierre Dac, qui en précurseur méconnu de l'algorithmique, affirmait tout aussi profondément que « *rien ne sert de penser, il faut réfléchir avant* »).
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 : A première vue, on peut avoir l'impression que les informaticiens étaient beurrés comme des petits lus lorsqu'ils ont baptisé ces opérations ; puisque quand l'utilisateur doit écrire au clavier, on appelle ça la lecture, et quand il doit lire sur l'écran on appelle çà l'écriture. Mais avant d'agonir d'insultes une digne corporation, il faut réfléchir un peu plus loin. 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. Et là, tout devient parfaitement logique. Et toc.

1. 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… et c'est très désagréable !) :

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).

Enonce des Exercices	Corrigés des Exercices
Exercice 2.1 Quel résultat produit le programme suivant ? Variables val, double numériques Début Val ← 231 Double ← Val * 2 Ecrire Val Ecrire Double Fin	Exercice 2.1 On verra apparaître à l'écran 231, puis 462 (qui vaut 231 * 2)
Exercice 2.2 Ecrire un programme qui demande un nombre à l'utilisateur, puis qui calcule et affiche le carré de ce nombre.	Exercice 2.2 **Variables** nb, carr **en Entier** **Début** **Ecrire** "Entrez un nombre :" **Lire** nb carr ← nb * nb **Ecrire** "Son carré est : ", carr **Fin** En fait, on pourrait tout aussi bien économiser la variable carr en remplaçant les deux avant-dernières lignes par : **Ecrire** "Son carré est : ", nb*nb C'est une question de style ; dans un cas, on privilégie la lisibilité de l'algorithme, dans l'autre, on privilégie l'économie d'une variable.
Exercice 2.3 Ecrire un programme qui lit le prix HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant. Faire en sorte que des libellés apparaissent clairement.	Exercice 2.3 **Variables** nb, pht, ttva, pttc **en Numérique** **Début** **Ecrire** "Entrez le prix hors taxes :" **Lire** pht **Ecrire** "Entrez le nombre d'articles :" **Lire** nb **Ecrire** "Entrez le taux de TVA :" **Lire** ttva pttc ← nb * pht * (1 + ttva) **Ecrire** "Le prix toutes taxes est : ", pttc **Fin** Là aussi, on pourrait squeezer une variable et une ligne en écrivant directement. : **Ecrire** "Le prix toutes taxes est : ", nb * pht * (1 + ttva) C'est plus rapide, plus léger en mémoire, mais un peu plus difficile à relire (et à écrire !)
Exercice 2.4 Ecrire un algorithme utilisant des variables de type chaîne de	Exercice 2.4 **Variables** t1, t2, t3, t4 **en Caractère** **Début**

caractères, et affichant quatre variantes possibles de la célèbre « belle marquise, vos beaux yeux me font mourir d'amour ». On ne se soucie pas de la ponctuation, ni des majuscules.	t1 ← "belle Marquise" t2 ← "vos beaux yeux" t3 ← "me font mourir" t4 ← "d'amour" **Ecrire** t1 & " " & t2 & " " & t3 & " " & t4 **Ecrire** t3 & " " & t2 & " " & t4 & " " & t1 **Ecrire** t2 & " " & t3 & " " & t1 & " " & t4 **Ecrire** t4 & " " & t1 & " " & t2 & " " & t3 **Fin**

III. Les Tests

1. De quoi s'agit-il ?

Reprenons le cas de notre « programmation algorithmique du touriste égaré ». Normalement, l'algorithme ressemblera à quelque chose comme : « *Allez tout droit jusqu'au prochain carrefour, puis prenez à droite et ensuite la deuxième à gauche, et vous y êtes* ».
Mais en cas de doute légitime de votre part, cela pourrait devenir : « *Allez tout droit jusqu'au prochain carrefour et là regardez à droite. Si la rue est autorisée à la circulation, alors prenez la et ensuite c'est la deuxième à gauche. Mais si en revanche elle est en sens interdit, alors continuez jusqu'à la prochaine à droite, prenez celle-là, et ensuite la première à droite* ».
Ce deuxième algorithme a ceci de supérieur au premier qu'il prévoit, en fonction d'une situation pouvant se présenter de deux façons différentes, deux façons différentes d'agir. Cela suppose que l'interlocuteur (le touriste) sache analyser la condition que nous avons fixée à son comportement (« la rue est-elle en sens interdit ? ») pour effectuer la série d'actions correspondante.
Eh bien, croyez le ou non, mais les ordinateurs possèdent cette aptitude, sans laquelle d'ailleurs nous aurions bien du mal à les programmer. Nous allons donc pouvoir parler à notre ordinateur comme à notre touriste, et lui donner des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre. Cette structure logique répond au doux nom de test. Toutefois, ceux qui tiennent absolument à briller en société parleront également de structure alternative.

2. 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
Finsi

Si booléen **Alors**
  Instructions 1
Sinon
  Instructions 2
Finsi

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 FinSi.
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 « Finsi ». 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 FinSi 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.
Exprimé sous forme de pseudo-code, la programmation de notre touriste de tout à l'heure donnerait donc quelque chose du genre :

Allez tout droit jusqu'au prochain carrefour
Si la rue à droite est autorisée à la circulation **Alors**
  Tournez à droite
  Avancez
  Prenez la deuxième à gauche
Sinon
  Continuez jusqu'à la prochaine rue à droite
  Prenez cette rue
  Prenez la première à droite
Finsi

3. 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 constituant la condition constitue donc, si l'on veut, une affirmation, qui a un moment donné est VRAIE ou FAUSSE.
A 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 (rappelez vous le code ASCII vu dans le préambule), les majuscules étant systématiquement placées avant les minuscules. Ainsi on a :
"t" < "w" VRAI
"Maman" > "Papa" FAUX
"maman" > "Papa" VRAI

4. 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.
Comme on l'a évoqué plus haut, 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.
- 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

Non C1	
C1 Vrai	Faux
C1 Faux	Vrai

5. Tests imbriqués

Graphiquement, on peut très facilement représenter un SI comme un aiguillage de chemin de fer (ou un aiguillage de train électrique, c'est moins lourd à porter). 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).
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"
FinSi
Si Temp > 0 Et Temp < 100 Alors
  Ecrire "C'est du liquide"
Finsi
Si Temp > 100 Alors
  Ecrire "C'est de la vapeur"
Finsi
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

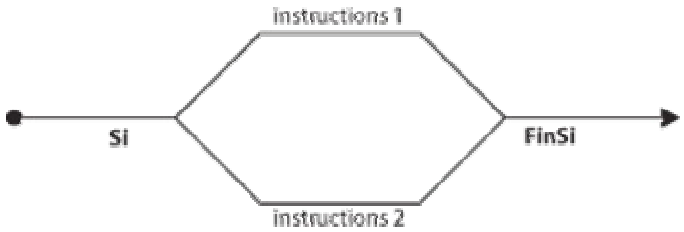
température de l'eau (la valeur de la variable Temp). Il serait ainsi bien plus rationnel d'imbriquer les tests de cette manière :

```
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"
  Finsi
Finsi
Fin
```

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.

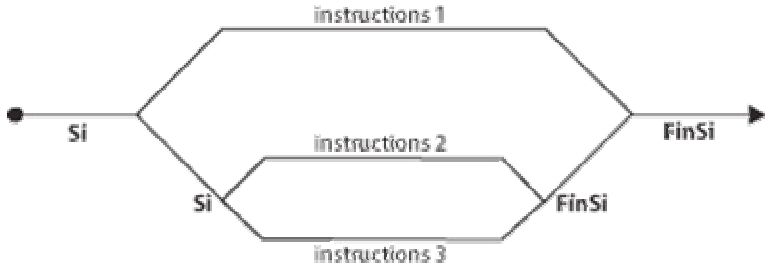
6. De l'aiguillage à la gare de tri

« J'ai l'âme ferroviaire : je regarde passer les vaches » (Léo Ferré)
Cette citation n'apporte peut-être pas grand chose à cet exposé, mais je l'aime bien, alors c'était le moment ou jamais.
En effet, dans un programme, une structure SI peut être facilement comparée à un aiguillage de train. La voie principale se sépare en deux, le train devant rouler ou sur l'une, ou sur l'autre, et les deux voies se rejoignant tôt ou tard pour ne plus en former qu'une seule, lors du FinSi. On peut schématiser cela ainsi :



Mais dans certains cas, ce ne sont pas deux voies qu'il nous faut, mais trois, ou même plus. Dans le cas de l'état de l'eau, il nous faut trois voies pour notre « train », puisque l'eau peut être solide, liquide ou gazeuse. Alors, nous n'avons pas eu le choix : pour deux voies, il nous fallait un aiguillage, pour trois voies il nous en faut deux, imbriqués l'un dans l'autre.

Cette structure (telle que nous l'avons programmée à la page précédente) devrait être schématisée comme suit :



Soyons bien clairs : cette structure est la seule possible du point de vue logique (même si on peut toujours mettre le bas en haut et le haut en bas). Mais du point de vue de l'écriture, le pseudo-code algorithmique admet une simplification supplémentaire. Ainsi, il est possible (mais non obligatoire, que l'algorithme initial :

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"
  Finsi
Finsi
Fin
devienne :
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"
Finsi
Fin

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
Le SinonSi permet en quelque sorte de créer (en réalité, de simuler) des aiguillages à plus de deux branches. On peut ainsi enchaîner les SinonSi les uns derrière les autres pour simuler un aiguillage à autant de branches que l'on souhaite.

7. Variables Booléennes

Jusqu'ici, pour écrire nos des 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"
Finsi
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 (rassurez-vous, rien à voir avec le flagrant délit des policiers).

Enonce des Exercices	Corrigés des Exercices

Exercice 3.1
 	
Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on laisse de côté le cas où le nombre vaut zéro).
	Exercice 3.1
	Variable n **en Entier**
	Début
	Ecrire "Entrez un nombre : "
	Lire n
	Si n > 0 **Alors**
	  Ecrire "Ce nombre est positif"
	Sinon
	  Ecrire "Ce nombre est négatif"
	Finsi
	Fin

Exercice 3.2
 	
Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul). Attention toutefois : on ne doit **pas** calculer le produit des deux nombres.
	Exercice 3.2
	Variables m, n **en Entier**
	Début
	Ecrire "Entrez deux nombres : "
	Lire m, n
	Si (m > 0 ET n > 0) OU (m < 0 ET n < 0) **Alors**
	  Ecrire "Leur produit est positif"
	Sinon
	  Ecrire "Leur produit est négatif"
	Finsi
	Fin

Exercice 3.3
 	
Ecrire un algorithme qui demande trois noms à l'utilisateur et l'informe ensuite s'ils sont rangés ou non dans l'ordre alphabétique.
	Exercice 3.3
	Variables a, b, c **en Caractèr**e
	Début
	Ecrire "Entrez successivement trois noms : "
	Lire a, b, c
	Si a < b ET b < c **Alors**
	  Ecrire "Ces noms sont classés alphabétiquement"
	Sinon
	  Ecrire "Ces noms ne sont pas classés"
	Finsi
	Fin

Exercice 3.4
 	
Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on inclut cette fois le traitement du cas où le nombre vaut zéro).
	Exercice 3.4
	Variable n **en Entier**
	Début
	Ecrire "Entrez un nombre : "
	Lire n
	Si n < 0 **Alors**
	  Ecrire "Ce nombre est négatif"
	SinonSi n = 0 **Alors**
	  Ecrire "Ce nombre est nul"
	Sinon
	  Ecrire "Ce nombre est positif"
	Finsi
	Fin

Exercice 3.5
 	
Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si le produit est négatif ou positif (on inclut cette fois le traitement du cas où le produit peut être nul). Attention toutefois, on ne doit pas calculer
	Exercice 3.5
	Variables m, n **en Entier**
	Début
	Ecrire "Entrez deux nombres : "
	Lire m, n
	Si m = 0 OU n = 0 **Alors**
	  Ecrire "Le produit est nul"
	SinonSi (m < 0 ET n < 0) OU (m > 0 ET n > 0) **Alors**
	  Ecrire "Le produit est positif"
	Sinon
	  Ecrire "Le produit est négatif"

le produit !	**Finsi** **Fin** Si on souhaite simplifier l'écriture de la condition lourde du SinonSi, on peut toujours passer par des variables booléennes intermédiaires. Une astuce de sioux consiste également à employer un Xor (c'est l'un des rares cas dans lesquels il est pertinent)
Exercice 3.6 Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie : • "Poussin" de 6 à 7 ans • "Pupille" de 8 à 9 ans • "Minime" de 10 à 11 ans • "Cadet" après 12 ans Peut-on concevoir plusieurs algorithmes équivalents menant à ce résultat ?	Exercice 3.6 **Variable** age **en Entier** **Début** **Ecrire** "Entrez l'âge de l'enfant : " **Lire** age **Si** age >= 12 **Alors** **Ecrire** "Catégorie Cadet" **SinonSi** age >= 10 Alors **Ecrire** "Catégorie Minime" **SinonSi** age >= 8 Alors **Ecrire** "Catégorie Pupille" **SinonSi** age >= 6 Alors **Ecrire** "Catégorie Poussin" **Finsi** **Fin** On peut évidemment écrire cet algorithme de différentes façons, ne serait-ce qu'en commençant par la catégorie la plus jeune.

8. 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, bien plus intéressante.
Spontanément, 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
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
Finsi

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 de vérité, et vous noterez la symétrie entre celle du ET et celle du OU. Dans les deux tables, il y a trois cas sur quatre qui mènent à un résultat, et un sur quatre qui mène au résultat inverse. Alors, rien d'étonnant à ce qu'une situation qui s'exprime avec une des tables (un des opérateurs logiques) puisse tout aussi 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
Finsi

équivaut à :

Si NON A OU NON B **Alors**
  Instructions 2
Sinon
  Instructions 1
Finsi

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

Enonce des Exercices	Corrigés des Exercices
Exercice 4.1 Formulez un algorithme équivalent à l'algorithme suivant : **Si** Tutu > Toto + 4 OU Tata = "OK" **Alors** Tutu ← Tutu + 1 **Sinon** Tutu ← Tutu – 1 **Finsi**	Exercice 4.1 Aucune difficulté, il suffit d'appliquer la règle de la transformation du OU en ET vue en cours (loi de Morgan). Attention toutefois à la rigueur dans la transformation des conditions en leur contraire... **Si** Tutu <= Toto + 4 ET Tata <> "OK" **Alors** Tutu ← Tutu - 1 **Sinon** Tutu ← Tutu + 1 **Finsi**
Exercice 4.2 Cet algorithme est destiné à prédire l'avenir, et il doit être infaillible ! Il lira au clavier l'heure et les minutes, et il affichera l'heure qu'il sera une minute plus tard. Par exemple, si l'utilisateur tape 21 puis 32, l'algorithme doit répondre : "Dans une minute, il sera 21 heure(s) 33". NB : on suppose que l'utilisateur entre une heure valide. Pas besoin donc de la vérifier.	Exercice 4.2 **Variables** h, m **en Numérique** **Début** **Ecrire** "Entrez les heures, puis les minutes : " **Lire** h, m m ← m + 1 **Si** m = 60 **Alors** m ← 0 h ← h + 1 **FinSi** **Si** h = 24 **Alors** h ← 0 **FinSi** **Ecrire** "Dans une minute il sera ", h, "heure(s) ", m, "minute(s)" **Fin**
Exercice 4.3 De même que le précédent, cet algorithme doit demander une heure et en afficher une autre. Mais cette fois, il doit gérer également les secondes, et afficher l'heure qu'il sera une seconde plus tard. Par exemple, si l'utilisateur tape 21, puis 32, puis 8, l'algorithme doit répondre : "Dans une seconde, il sera 21 heure(s), 32 minute(s) et 9 seconde(s)". NB : là encore, on suppose que l'utilisateur entre une date valide.	Exercice 4.3 **Variables** h, m, s **en Numérique** **Début** **Ecrire** "Entrez les heures, puis les minutes, puis les secondes : " **Lire** h, m, s s ← s + 1 **Si** s = 60 **Alors** s ← 0 m ← m + 1 **FinSi** **Si** m = 60 **Alors** m ← 0 h ← h + 1 **FinSi** **Si** h = 24 **Alors** h ← 0 **FinSi** **Ecrire** "Dans une seconde il sera ", h, "h", m, "m et ", s, "s"

	Fin
Exercice 4.4 Un magasin de reprographie facture 0,10 E les dix premières photocopies, 0,09 E les vingt suivantes et 0,08 E au-delà. Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.	Exercice 4.4 **Variables** n, p **en Numérique** **Début** **Ecrire** "Nombre de photocopies : " **Lire** n **Si** n <= 10 **Alors** p ← n * 0,1 **SinonSi** n <= 30 **Alors** p ← 10 * 0,1 + (n – 10) * 0,09 **Sinon** p ← 10 * 0,1 + 20 * 0,09 + (n – 30) * 0,08 **FinSi** **Ecrire** "Le prix total est: ", p **Fin**
Exercice 4.5 Les habitants de Zorglub paient l'impôt selon les règles suivantes : • les hommes de plus de 20 ans paient l'impôt • les femmes paient l'impôt si elles ont entre 18 et 35 ans • les autres ne paient pas d'impôt Le programme demandera donc l'âge et le sexe du Zorglubien, et se prononcera donc ensuite sur le fait que l'habitant est imposable.	Exercice 4.5 **Variable** sex **en Caractère** **Variable** age **en Numérique** **Variables** C1, C2 **en Booléen** **Début** **Ecrire** "Entrez le sexe (M/F) : " **Lire** sex **Ecrire** "Entrez l'âge: " **Lire** age C1 ← sex = "M" ET age > 20 C2 ← sex = "F" ET (age > 18 ET age < 35) **Si** C1 ou C2 **Alors** **Ecrire** "Imposable" **Sinon** **Ecrire** "Non Imposable" **FinSi** **Fin**
Exercice 4.6 Les élections législatives, en Guignolerie Septentrionale, obéissent à la règle suivante : • lorsque l'un des candidats obtient plus de 50% des suffrages, il est élu dès le premier tour. • en cas de deuxième tour, peuvent participer uniquement les candidats ayant obtenu	Exercice 4.6 Cet exercice, du pur point de vue algorithmique, n'est pas très méchant. En revanche, il représente dignement la catégorie des énoncés piégés. En effet, rien de plus facile que d'écrire : si le candidat a plus de 50%, il est élu, sinon s'il a plus de 12,5 %, il est au deuxième tour, sinon il est éliminé. Hé hé hé... mais il ne faut pas oublier que le candidat peut très bien avoir eu 20 % mais être tout de même éliminé, tout simplement parce que l'un des autres a fait plus de 50 % et donc qu'il n'y a pas de deuxième tour !... Moralité : ne jamais se jeter sur la programmation avant d'avoir soigneusement mené l'analyse du problème à traiter. **Variables** A, B, C, D **en Numérique**

au moins 12,5% des voix au premier tour.

Vous devez écrire un algorithme qui permette la saisie des scores de quatre candidats au premier tour. Cet algorithme traitera ensuite le candidat numéro 1 (et **uniquement** lui) : il dira s'il est élu, battu, s'il se trouve en ballottage favorable (il participe au second tour en étant arrivé en tête à l'issue du premier tour) ou défavorable (il participe au second tour sans avoir été en tête au premier tour).

```
Début
Ecrire "Entrez les scores des quatre prétendants :"
Lire A, B, C, D
C1 ← A > 50
C2 ← B > 50 ou C > 50 ou D > 50
C3 ← A >= B et A >= C et A >= D
C4 ← A >= 12,5
Si C1 Alors
  Ecrire “Elu au premier tour"
Sinonsi C2 ou Non(C4) Alors
  Ecrire “Battu, éliminé, sorti !!!”
SinonSi C3 Alors
  Ecrire "Ballotage favorable"
Sinon
  Ecrire "Ballotage défavorable"
FinSi
Fin
```

Exercice 4.7

Une compagnie d'assurance automobile propose à ses clients quatre familles de tarifs identifiables par une couleur, du moins au plus onéreux : tarifs bleu, vert, orange et rouge. Le tarif dépend de la situation du conducteur :

- un conducteur de moins de 25 ans et titulaire du permis depuis moins de deux ans, se voit attribuer le tarif rouge, si toutefois il n'a jamais été responsable d'accident. Sinon, la compagnie refuse de l'assurer.
- un conducteur de moins de 25 ans et titulaire du permis depuis plus de deux ans, ou de plus de 25 ans mais titulaire du permis depuis moins de deux ans a le droit au tarif orange s'il n'a jamais provoqué d'accident, au tarif rouge pour un accident, sinon il est refusé.

Exercice 4.7

Là encore, on illustre l'utilité d'une bonne analyse. Je propose deux corrigés différents. Le premier suit l'énoncé pas à pas. C'est juste, mais c'est vraiment lourd. La deuxième version s'appuie sur une vraie compréhension d'une situation pas si embrouillée qu'elle n'en a l'air.
Dans les deux cas, un recours aux variables booléennes aère sérieusement l'écriture.
Donc, premier corrigé, on suit le texte de l'énoncé pas à pas :

```
Variables age, perm, acc, assur en Numérique
Variables C1, C2, C3 en Booléen
Variable situ en Caractère
Début
Ecrire "Entrez l’âge: "
Lire age
Ecrire "Entrez le nombre d'années de permis: "
Lire perm
Ecrire "Entrez le nombre d'accidents: "
Lire acc
Ecrire "Entrez le nombre d'années d'assurance: "
Lire assur
C1 ← age >= 25
C2 ← perm >= 2
C3 ← assur > 1
Si Non(C1) et Non(C2) Alors
  Si acc = 0 Alors
    situ ← "Rouge"
  Sinon
    situ ← "Refusé"
  FinSi
Sinonsi ((Non(C1) et C2) ou (C1 et Non(C2)) Alors
```

- un conducteur de plus de 25 ans titulaire du permis depuis plus de deux ans bénéficie du tarif vert s'il n'est à l'origine d'aucun accident et du tarif orange pour un accident, du tarif rouge pour deux accidents, et refusé au-delà
- De plus, pour encourager la fidélité des clients acceptés, la compagnie propose un contrat de la couleur immédiatement la plus avantageuse s'il est entré dans la maison depuis plus d'un an.

Ecrire l'algorithme permettant de saisir les données nécessaires (sans contrôle de saisie) et de traiter ce problème. Avant de se lancer à corps perdu dans cet exercice, on pourra réfléchir un peu et s'apercevoir qu'il est plus simple qu'il n'en a l'air (cela s'appelle faire une analyse !)

```
  Si acc = 0 Alors
    situ ← "Orange"
  SinonSi acc = 1 Alors
    situ ← "Rouge"
  Sinon
    situ ← "Refusé"
  FinSi
Sinon
  Si acc = 0 Alors
    situ ← "Vert"
  SinonSi acc = 1 Alors
    situ ← "Orange"
  SinonSi acc = 2 Alors
    situ ← "Rouge"
  Sinon
    situ ← "Refusé"
  FinSi
FinSi
Si C3 Alors
  Si situ = "Rouge" Alors
    situ ← "Orange"
  SinonSi situ = "Orange" Alors
    situ ← "Orange"
  SinonSi situ = "Vert" Alors
    situ ← "Bleu"
  FinSi
FinSi
Ecrire "Votre situation : ", situ
Fin
```

Vous trouvez cela compliqué ? Oh, certes oui, ça l'est ! Et d'autant plus qu'en lisant entre les lignes, on pouvait s'apercevoir que ce galimatias de tarifs recouvre en fait une logique très simple : un système à points. Et il suffit de comptabiliser les points pour que tout s'éclaire... Reprenons juste après l'affectation des trois variables booléennes C1, C2, et C3. On écrit :

```
P ← 0
Si Non(C1) Alors
  P ← P + 1
FinSi
Si Non(C2) Alors
  P ← P + 1
FinSi
P ← P + acc
Si P < 3 et C3 Alors
  P ← P - 1
FinSi
Si P = -1 Alors
  situ ← "Bleu"
```

	SinonSi P = 0 **Alors**
	  situ ← "Vert"
	SinonSi P = 1 **Alors**
	  situ ← "Orange"
	SinonSi P = 2 **Alors**
	  situ ← "Rouge"
	Sinon
	  situ ← "Refusé"
	FinSi
	Ecrire "Votre situation : ", situ
	Fin
	Cool, non ?

Exercice 4.8
Ecrivez un algorithme qui a près avoir demandé un numéro de jour, de mois et d'année à l'utilisateur, renvoie s'il s'agit ou non d'une date valide.
Cet exercice est certes d'un manque d'originalité affligeant, mais après tout, en algorithmique comme ailleurs, il faut connaître ses classiques ! Et quand on a fait cela une fois dans sa vie, on apprécie pleinement l'existence d'un type numérique « date » dans certains langages…).
Il n'est sans doute pas inutile de rappeler rapidement que le mois de février compte 28 jours, sauf si l'année est bissextile, auquel cas il en compte 29. L'année est bissextile si elle est divisible par quatre. Toutefois, les années divisibles par 100 ne sont pas bissextiles, mais les années divisibles par 400 le sont. Ouf !
Un dernier petit détail : vous ne savez pas, pour l'instant, exprimer correctement en pseudo-code l'idée qu'un nombre A est divisible par un nombre B. Aussi, vous vous contenterez d'écrire en bons télégraphistes que A divisible par B se dit « A dp B ».
	Exercice 4.8
	En ce qui concerne le début de cet algorithme, il n'y a aucune difficulté. C'est de la saisie bête et même pas méchante:
	Variables J, M, A, JMax **en Numérique**
	Variables VJ, VM, B **en Booleen**
	Début
	Ecrire "Entrez le numéro du jour"
	Lire J
	Ecrire "Entrez le numéro du mois"
	Lire M
	Ecrire "Entrez l'année"
	Lire A
	C'est évidemment ensuite que les ennuis commencent… La première manière d'aborder la chose consiste à se dire que fondamentalement, la structure logique de ce problème est très simple. Si nous créons deux variables booléennes VJ et VM, représentant respectivement la validité du jour et du mois entrés, la fin de l'algorithme sera d'une simplicité biblique (l'année est valide par définition, si on évacue le débat byzantin concernant l'existence de l'année zéro) :
	Si VJ et VM **alors**
	  Ecrire "La date est valide"
	Sinon
	  Ecrire "La date n'est pas valide"
	FinSi
	Toute la difficulté consiste à affecter correctement les variables VJ et VM, selon les valeurs des variables J, M et A. Dans l'absolu, VJ et VM pourraient être les objets d'une affectation monstrueuse, avec des conditions atrocement composées. Mais franchement, écrire ces conditions en une seule fois est un travail de bénédictin sans grand intérêt. Pour éviter d'en arriver à une telle extrémité, on peut sérier la difficulté en créant deux variables supplémentaires :
	
	B : variable booléenne qui indique s'il s'agit d'une année bissextile

JMax : variable numérique qui indiquera le dernier jour valable pour le mois entré.

Avec tout cela, on peut y aller et en ressortir vivant.
On commence par initialiser nos variables booléennes, puis on traite les années, puis les mois, puis les jours.
On note "dp" la condition "divisible par" :

```
B ← A dp 400 ou (non(A dp 100) et A dp 4)
Jmax ← 0
VM ← M >= 1 et M =< 12
Si VM Alors
  Si M = 2 et B Alors
    JMax ← 29
  SinonSi M = 2 Alors
    JMax ← 28
  SinonSi M = 4 ou M = 6 ou M = 9 ou M = 11 Alors
    JMax ← 30
  Sinon
    JMax ← 31
  FinSi
  VJ ← J >= 1 et J =< Jmax
FinSi
```

Cette solution a le mérite de ne pas trop compliquer la structure des tests, et notamment de ne pas répéter l'écriture finale à l'écran. Les variables booléennes intermédiaires nous épargnent des conditions composées trop lourdes, mais celles-ci restent néanmoins sérieuses.

Une approche différente consisterait à limiter les conditions composées, quitte à le payer par une structure beaucoup plus exigeante de tests imbriqués. Là encore, on évite de jouer les extrémistes et l'on s'autorise quelques conditions composées lorsque cela nous simplifie l'existence. On pourrait aussi dire que la solution précédente "part de la fin" du problème (la date est elle valide ou non ?), alors que celle qui suit "part du début" (quelles sont les données entrées au clavier ?) :

```
Si M < 1 ou M > 12 Alors
  Ecrire "Date Invalide"
SinonSi M = 2 Alors
  Si A dp 400 Alors
    Si J < 1 ou J > 29 Alors
      Ecrire "Date Invalide"
    Sinon
      Ecrire "Date Valide"
    FinSi
  SinonSi A dp 100 Alors
    Si J < 1 ou J > 28 Alors
      Ecrire "Date Invalide"
    Sinon
```

```
      Ecrire "Date Valide"
    FinSi
  SinonSi A dp 4 Alors
    Si J < 1 ou J > 28 Alors
      Ecrire "Date Invalide"
    Sinon
      Ecrire "Date Valide"
    FinSi
  Sinon
    Si J < 1 ou J > 28 Alors
      Ecrire "Date Invalide"
    Sinon
      Ecrire "Date Valide"
    FinSi
  FinSi
SinonSi M = 4 ou M = 6 ou M = 9 ou M = 11 Alors
  Si J < 1 ou J > 30 Alors
    Ecrire "Date Invalide"
  Sinon
    Ecrire "Date Valide"
  FinSi
Sinon
  Si J < 1 ou J > 31 Alors
    Ecrire "Date Invalide"
  Sinon
    Ecrire "Date Valide"
  FinSi
FinSi
```

On voit que dans ce cas, l'alternative finale (Date valide ou invalide) se trouve répétée un grand nombre de fois. Ce n'est en soi ni une bonne, ni une mauvaise chose. C'est simplement une question de choix stylistique.
Personnellement, j'avoue préférer assez nettement la première solution, qui fait ressortir beaucoup plus clairement la structure logique du problème (il n'y a qu'une seule alternative, autant que cette alternative ne soit écrite qu'une seule fois).

Il convient enfin de citer une solution très simple et élégante, un peu plus difficile peut-être à imaginer du premier coup, mais qui avec le recul apparaît comme très immédiate. Sur le fond, cela consiste à dire qu'il y a quatre cas pour qu'une date soit valide : celui d'un jour compris entre 1 et 31 dans un mois à 31 jours, celui d'un jour compris entre 1 et 30 dans un mois à 30 jours, celui d'un jour compris entre 1 et 29 en février d'une année bissextile, et celui d'un jour de février compris entre 1 et 28.
Ainsi :
B ← (A dp 4 et Non(A dp 100)) ou A dp 400
K1 ← (m=1 ou m=3 ou m=5 ou m=7 ou m=8 ou m=10 ou m=12) et (J>=1 et J=<31)

	K2 ← (m=4 ou m=6 ou m=9 ou m=11) et (J>=1 et J=<30) K3 ← m=2 et B et J>=1 et J=<29 K4 ← m=2 et J>=1 et J=<28 **Si** K1 ou K2 ou K3 ou K4 **Alors** **Ecrire** "Date valide" **Sinon** **Ecrire** "Date non valide" **FinSi** **Fin** Tout est alors réglé avec quelques variables booléennes et quelques conditions composées, en un minimum de lignes de code. La morale de ce long exercice - et non moins long corrigé, c'est qu'un problème de test un peu compliqué admet une pléiade de solutions justes... ...Mais que certaines sont plus astucieuses que d'autres !

9. Au-delà de la logique : le style

Ce titre un peu provocateur (mais néanmoins justifié) a pour but d'attirer maintenant votre attention sur un fait fondamental en algorithmique, fait que plusieurs remarques précédentes ont déjà dû vous faire soupçonner : 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
Finsi
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
  Finsi
Sinon
  Fermer la fenêtre
Finsi

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.

IV. Les Boucles

Et ça y est, on y est, on est arrivés, la voilà, c'est Broadway, la quatrième et dernière structure : ça est les boucles. Si vous voulez épater vos amis, vous pouvez également parler de structures répétitives, voire carrément de structures itératives. Ca calme, hein ? Bon, vous faites ce que vous voulez, ici on est entre nous, on parlera de boucles.
Les boucles, c'est généralement le point douloureux de l'apprenti programmeur. C'est là que ça coince, car autant il est assez facile de comprendre comment fonctionnent les boucles, autant il est souvent long d'acquérir les réflexes qui permettent de les élaborer judicieusement pour traiter un problème donné.
On peut dire en fait que les boucles constituent la seule vraie structure logique caractéristique de la programmation. Si vous avez utilisé un tableur comme Excel, par exemple, vous avez sans doute pu manier des choses équivalentes aux variables (les cellules, les formules) et aux tests (la fonction SI…). Mais les boucles, ça, ça n'a aucun équivalent. Cela n'existe que dans les langages de programmation proprement dits.
Le maniement des boucles, s'il ne différencie certes pas l'homme de la bête (il ne faut tout de même pas exagérer), est tout de même ce qui sépare en informatique le programmeur de l'utilisateur, même averti.
Alors, à vos futures – et inévitables - difficultés sur le sujet, il y a trois remèdes : de la rigueur, de la patience, et encore de la rigueur !

1. A quoi cela sert-il donc ?

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, facétieux ou 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.
A vue de nez, on pourrait essayer avec un SI. Voyons voir ce que ça donne :

```
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
FinSi
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 qu'une blague des Grosses Têtes, on n'en sortira pas, il y aura toujours moyen qu'un acharné flanque le programme par terre.
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, je le rappelle, peut être une variable booléenne ou, plus fréquemment, 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. Le manège enchanté 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 Caractère**
Début
Ecrire "Voulez vous un café ? (O/N)"
TantQue Rep <> "O" et Rep <> "N"
Lire Rep
FinTantQue
Fin

Là, on a le squelette de l'algorithme correct. Mais de même qu'un squelette ne suffit pas pour avoir un être vivant viable, il va nous falloir ajouter quelques muscles et organes sur cet algorithme pour qu'il fonctionne correctement.
Son principal défaut 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**
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

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
```

Le Gag De La Journée

C'est d'écrire une structure TantQue dans laquelle le booléen n'est jamais VRAI. Le programme ne rentre alors jamais dans la superbe boucle sur laquelle vous avez tant sué !
Mais la faute symétrique est au moins aussi désopilante.
Elle consiste à écrire une boucle dans laquelle le booléen ne devient jamais FAUX. L'ordinateur tourne alors dans la boucle comme un dératé et n'en sort plus. Seule solution, quitter le programme avec un démonte-pneu ou un bâton de dynamite. La « **boucle infinie** » est une des hantises les plus redoutées des programmeurs. C'est un peu comme le verre baveur, le poil à gratter ou le bleu de méthylène : c'est éculé, mais ça fait toujours rire.
Cette faute de programmation grossière – mais fréquente - ne manquera pas d'égayer l'ambiance collective de cette formation… et accessoirement d'étancher la soif proverbiale de vos enseignants.
Bon, eh bien vous allez pouvoir faire de chouettes algorithmes, déjà rien qu'avec ça…

2. 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
```

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, encore un mot qui fera forte impression sur votre entourage).
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(cf. Partie 9)

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 (parties 7 et 8) et chaînes de caractères (partie 9). 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.

3. Des boucles dans des boucles

(« *tout est dans tout... et réciproquement* »)
On rigole, on rigole !
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. Y a pas de raison.

```
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.
Des boucles peuvent donc être imbriquées (cas n°1) ou successives (cas n°2). Cependant, elles ne peuvent jamais, au grand jamais, être croisées. Cela n'aurait aucun sens logique, et de plus, bien peu de langages vous autoriseraient ne serait-ce qu'à écrire cette structure aberrante.
Variables Truc, Trac **en Entier**
Pour Truc ← …
  instructions
  Pour Trac ← …
    instructions
Truc **Suivant**
  instructions
  Trac **Suivant**
Pourquoi imbriquer des boucles ? Pour la même raison qu'on imbrique des tests. La traduction en bon français d'un test, c'est un « cas ». Eh bien un « cas » (par exemple, « est-ce un homme ou une femme ? ») peut très bien se subdiviser en d'autres cas (« a-t-il plus ou moins de 18 ans ? »).
De même, 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).
Dans la pratique de la programmation, la maîtrise des boucles imbriquées est nécessaire, même si elle n'est pas suffisante. Tout le contraire d'Alain Delon, en quelque sorte.

4. Et encore une bêtise à ne pas faire !

Examinons l'algorithme suivant :
Variable Truc **en Entier**
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. Il va sans dire que de telles manipulations perturbent complètement le déroulement normal de la boucle, et sont causes, sinon de plantages, tout au moins d'exécutions erratiques.

Enonce des Exercices	Corrigés des Exercices

Exercice 5.1
 	
Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.
	Exercice 5.1
	Variable N **en Entier**
	Debut
	N ← 0
	Ecrire "Entrez un nombre entre 1 et 3"
	TantQue N < 1 ou N > 3
	  Lire N
	  Si N < 1 ou N > 3 **Alors**
	  Ecrire "Saisie erronée. Recommencez"
	  FinSi
	  FinTantQue
	Fin

Exercice 5.2
 	
Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.
	Exercice 5.2
	Variable N **en Entier**
	Debut
	N ← 0
	Ecrire "Entrez un nombre entre 10 et 20"
	TantQue N < 10 ou N > 20
	  Lire N
	  Si N < 10 **Alors**
	  Ecrire "Plus grand !"
	  SinonSi N > 20 Alors
	  Ecrire "Plus petit !"
	  FinSi
	FinTantQue
	Fin

Exercice 5.3 Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.	Exercice 5.3 **Variables** N, i **en Entier** **Debut** **Ecrire** "Entrez un nombre : " **Lire** N **Ecrire** "Les 10 nombres suivants sont : " **Pour** i ← N + 1 à N + 10 **Ecrire** i i **Suivant** **Fin**

Exercice 5.4

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) :
Table de 7 :
7 x 1 = 7
7 x 2 = 14
…
7 x 10 = 70

Exercice 5.4

```
Variables N, i en Entier
Debut
Ecrire "Entrez un nombre : "
Lire N
Ecrire "La table de multiplication de ce nombre est : "
Pour i ← 1 à 10
  Ecrire N, " x ", i, " = ", n*i
i Suivant
Fin
```

Exercice 5.5

Ecrire un algorithme qui demande un nombre de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer :
1 + 2 + 3 + 4 + 5 = 15
NB : on souhaite afficher uniquement le résultat, pas la décomposition du calcul.

Exercice 5.5

```
Variables N, i, Som en Entier
Debut
Ecrire "Entrez un nombre : "
Lire N
Som ← 0
Pour i ← 1 à N
  Som ← Som + i
i Suivant
Ecrire "La somme est : ", Som
Fin
```

Exercice 5.6

Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle.
NB : la factorielle de 8, notée 8 !, vaut
1 x 2 x 3 x 4 x 5 x 6 x 7 x 8

Exercice 5.6

```
Variables N, i, F en Entier
Debut
Ecrire "Entrez un nombre : "
Lire N
F ← 1
Pour i ← 2 à N
  F ← F * i
i Suivant
Ecrire "La factorielle est : ", F
Fin
```

Exercice 5.7

Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dise ensuite quel était le plus grand parmi ces 20 nombres :
Entrez le nombre numéro 1 : 12
Entrez le nombre numéro 2 : 14
etc.

Exercice 5.7

```
Variables N, i, PG en Entier
Debut
PG ← 0
Pour i ← 1 à 20
  Ecrire "Entrez un nombre : "
  Lire N
  Si i = 1 ou N > PG Alors
    PG ← N
  FinSi
i Suivant
```

Entrez le nombre numéro 20 : 6
Le plus grand de ces nombres est : 14
Modifiez ensuite l'algorithme pour que le programme affiche de surcroît en quelle position avait été saisie ce nombre :
C'était le nombre numéro 2

Ecrire "Le nombre le plus grand était : ", PG
Fin

En ligne 3, on peut mettre n'importe quoi dans PG, il suffit que cette variable soit affectée pour que le premier passage en ligne 7 ne provoque pas d'erreur.

Pour la version améliorée, cela donne :

```
Variables N, i, PG, IPG en Entier
Debut
PG ← 0
Pour i ← 1 à 20
  Ecrire "Entrez un nombre : "
  Lire N
  Si i = 1 ou N > PG Alors
    PG ← N
    IPG ← i
  FinSi
i Suivant
Ecrire "Le nombre le plus grand était : ", PG
Ecrire "Il a été saisi en position numéro ", IPG
Fin
```

Exercice 5.8

Réécrire l'algorithme précédent, mais cette fois-ci on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un zéro.

Exercice 5.8

```
Variables N, i, PG, IPG en Entier
Debut
N ← 1
i ← 0
PG ← 0
TantQue N <> 0
  Ecrire "Entrez un nombre : "
  Lire N
  i ← i + 1
  Si i = 1 ou N > PG Alors
    PG ← N
    IPG ← i
  FinSi
FinTantQue
Ecrire "Le nombre le plus grand était : ", PG
Ecrire "Il a été saisi en position numéro ", IPG
Fin
```

Exercice 5.9

Lire la suite des prix (en euros entiers et terminée par zéro) des achats d'un client. Calculer la somme qu'il doit, lire la somme qu'il paye, et simuler la remise

Exercice 5.9

```
Variables E, somdue, M, Reste, Nb10E, Nb5E En Entier
Debut
E ← 1
somdue ← 0
TantQue E <> 0
  Ecrire "Entrez le montant : "
  Lire E
```

de la monnaie en affichant les textes "10 Euros", "5 Euros" et "1 Euro" autant de fois qu'il y a de coupures de chaque sorte à rendre.

```
  somdue ← somdue + E
FinTantQue
Ecrire "Vous devez :", somdue, " euros"
Ecrire "Montant versé :"
Lire M
Reste ← M - somdue
Nb10E ← 0
TantQue Reste >= 10
  Nb10E ← Nb10E + 1
  Reste ← Reste – 10
FinTantQue
Nb5E ← 0
Si Reste >= 5
  Nb5E ← 1
  Reste ← Reste – 5
FinSi
Ecrire "Rendu de la monnaie :"
Ecrire "Billets de 10 E : ", Nb10E
Ecrire "Billets de  5 E : ", Nb5E
Ecrire "Pièces de 1 E : ", reste
Fin
```

Exercice 5.10

Écrire un algorithme qui permette de connaître ses chances de gagner au tiercé, quarté, quinté et autres impôts volontaires.
On demande à l'utilisateur le nombre de chevaux partants, et le nombre de chevaux joués.
Les deux messages affichés devront être :
Dans l'ordre : une chance sur X de gagner
Dans le désordre : une chance sur Y de gagner
X et Y nous sont donnés par la formule suivante, si n est le nombre de chevaux partants et p le nombre de chevaux joués (on rappelle que le signe ! signifie "factorielle", comme dans l'exercice 5.6 ci-dessus) :
X = n ! / (n - p) !
Y = n ! / (p ! * (n – p) !)
NB : cet algorithme peut être écrit d'une manière simple,

Exercice 5.10
Spontanément, on est tenté d'écrire l'algorithme suivant :

```
Variables N, P, i, Numé, Déno1, Déno2 en Entier
Debut Ecrire "Entrez le nombre de chevaux partants : "
Lire N
Ecrire "Entrez le nombre de chevaux joués : "
Lire P
Numé ← 1
Pour i ← 2 à N
  Numé ← Numé * i
i Suivant
Déno1 ← 1
Pour i ← 2 à N-P
  Déno1 ← Déno1 * i
i Suivant
Déno2 ← 1
Pour i ← 2 à P
  Déno2 ← Déno2 * i
i Suivant
Ecrire "Dans l'ordre, une chance sur ", Numé / Déno1
Ecrire "Dans le désordre, une sur ", Numé / (Déno1 * Déno2)
Fin
```

Cette version, formellement juste, comporte tout de même deux faiblesses.

La première, et la plus grave, concerne la manière dont elle calcule le résultat final. Celui-ci est le quotient d'un nombre par

mais relativement peu performante. Ses performances peuvent être singulièrement augmentées par une petite astuce. Vous commencerez par écrire la manière la plus simple, puis vous identifierez le problème, et écrirez une deuxième version permettant de le résoudre.	un autre ; or, ces nombres auront rapidement tendance à être très grands. En calculant, comme on le fait ici, d'abord le numérateur, puis ensuite le dénominateur, on prend le risque de demander à la machine de stocker des nombres trop grands pour qu'elle soit capable de les coder (cf. le préambule). C'est d'autant plus bête que rien ne nous oblige à procéder ainsi : on n'est pas obligé de passer par la division de deux très grands nombres pour obtenir le résultat voulu.
	
	La deuxième remarque est qu'on a programmé ici trois boucles successives. Or, en y regardant bien, on peut voir qu'après simplification de la formule, ces trois boucles comportent le même nombre de tours ! (si vous ne me croyez pas, écrivez un exemple de calcul et biffez les nombres identiques au numérateur et au dénominateur). Ce triple calcul (ces trois boucles) peut donc être ramené(es) à un(e) seul(e). Et voilà le travail, qui est non seulement bien plus court, mais aussi plus performant :
	Variables N, P, i, O, F **en Entier**
	Debut
	Ecrire "Entrez le nombre de chevaux partants : "
	Lire N
	Ecrire "Entrez le nombre de chevaux joués : "
	Lire P
	A ← 1
	B ← 1
	Pour i ← 1 à P
	  A ← A * (i + N - P)
	  B ← B * i
	i **Suivant**
	Ecrire "Dans l'ordre, une chance sur ", A
	Ecrire "Dans le désordre, une chance sur ", A / B
	Fin

V. Les Tableaux

1. 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

Ouf ! C'est tout de même bigrement 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 le suicide direct.
Cerise sur le gâteau, 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 est carrément cuits.
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 – ô surprise – 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.
2. 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.
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). Au début, ça déroute, mais vous verrez, avec le temps, on se fait à tout, même au pire.

Tableau Note(11) **en Entier**
On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, bien sûr, 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 :

```
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.
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.

2. 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, pourquoi pas, au diable les varices) 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.
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.

3. Exercices

Enonce des Exercices	Corrigés des Exercices
Exercice 6.1 Ecrire un algorithme qui déclare et remplisse un tableau de 7 valeurs numériques en les mettant toutes à zéro.	Exercice 6.1 **Tableau** Truc(6) **en Numérique** **Variable** i **en Numérique** **Debut** **Pour** i ← 0 à 6 Truc(i) ← 0 i **Suivant** **Fin**
Exercice 6.2 Ecrire un algorithme qui déclare et remplisse un tableau contenant les six voyelles de l'alphabet latin.	Exercice 6.2 **Tableau** Truc(5) **en Caractère** **Debut** Truc(0) ← "a" Truc(1) ← "e" Truc(2) ← "i" Truc(3) ← "o" Truc(4) ← "u" Truc(5) ← "y" **Fin**

Exercice 6.3
 	
Ecrire un algorithme qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur.
	Exercice 6.3
	Tableau Notes(8) **en Numérique**
	Variable i **en Numérique**
	Pour i ← 0 à 8
	  Ecrire "Entrez la note numéro ", i + 1
	  Lire Notes(i)
	i **Suivant**
	Fin

Exercice 6.4
Que produit l'algorithme suivant ?
Tableau Nb(5) **en Entier**
Variable i **en Entier**
Début
Pour i ← 0 à 5
  Nb(i) ← i * i
i **suivant**
Pour i ← 0 à 5
  Ecrire Nb(i)
i **suivant**
Fin
Peut-on simplifier cet algorithme avec le même résultat ?
	Exercice 6.4
	Cet algorithme remplit un tableau avec six valeurs : 0, 1, 4, 9, 16, 25.
	Il les écrit ensuite à l'écran. Simplification :
	Tableau Nb(5) **en Numérique**
	Variable i **en Numérique**
	Début
	Pour i ← 0 à 5
	  Nb(i) ← i * i
	  Ecrire Nb(i)
	i **Suivant**
	Fin

Exercice 6.5

Que produit l'algorithme suivant ?

```
Tableau N(6) en Entier
Variables i, k en Entier
Début
N(0) ← 1
Pour k ← 1 à 6
  N(k) ← N(k-1) + 2
k Suivant
Pour i ← 0 à 6
  Ecrire N(i)
i suivant
Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?

Exercice 6.5

Cet algorithme remplit un tableau avec les sept valeurs : 1, 3, 5, 7, 9, 11, 13.
Il les écrit ensuite à l'écran. Simplification :

```
Tableau N(6) en Numérique
Variables i, k en Numérique
Début
N(0) ← 1
Ecrire N(0)
Pour k ← 1 à 6
  N(k) ← N(k-1) + 2
  Ecrire N(k)
k Suivant
Fin
```

Exercice 6.6

Que produit l'algorithme suivant ?

```
Tableau Suite(7) en Entier
Variable i en Entier
Début
Suite(0) ← 1
Suite(1) ← 1
Pour i ← 2 à 7
  Suite(i) ← Suite(i-1) + Suite(i-2)
i suivant
Pour i ← 0 à 7
  Ecrire Suite(i)
i suivant
Fin
```

Exercice 6.6

Cet algorithme remplit un tableau de 8 valeurs : 1, 1, 2, 3, 5, 8, 13, 21

Exercice 6.7

Ecrivez la fin de l'algorithme 6.3 afin que le calcul de la moyenne des notes soit effectué et affiché à l'écran.

Exercice 6.7

```
Variable S en Numérique
Tableau Notes(8) en Numérique
Debut
s ← 0
Pour i ← 0 à 8
  Ecrire "Entrez la note n° ", i + 1
  Lire Notes(i)
  s ← s + Notes(i)
i Suivant
Ecrire "Moyenne :", s/9
Fin
```

Exercice 6.8

Exercice 6.8

Ecrivez un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives.

```
Variables Nb, Nbpos, Nbneg en Numérique
Tableau T() en Numérique
Debut
Ecrire "Entrez le nombre de valeurs :"
Lire Nb
Redim T(Nb-1)
Nbpos ← 0
Nbneg ← 0
Pour i ← 0 à Nb - 1
  Ecrire "Entrez le nombre n° ", i + 1
  Lire T(i)
  Si T(i) > 0 alors
    Nbpos ← Nbpos + 1
  Sinon
    Nbneg ← Nbneg + 1
  Finsi
i Suivant
Ecrire "Nombre de valeurs positives : ", Nbpos
Ecrire "Nombre de valeurs négatives : ", Nbneg
Fin
```

Exercice 6.9

Ecrivez un algorithme calculant la somme des valeurs d'un tableau (on suppose que le tableau a été préalablement saisi).

Exercice 6.9

```
Variables i, Som, N en Numérique
Tableau T() en Numérique
Debut
… (on ne programme pas la saisie du tableau, dont on suppose qu'il compte N éléments)
Redim T(N-1)
…
Som ← 0
Pour i ← 0 à N - 1
  Som ← Som + T(i)
i Suivant
Ecrire "Somme des éléments du tableau : ", Som
Fin
```

Exercice 6.10

Ecrivez un algorithme constituant un tableau, à partir de deux tableaux de même longueur préalablement saisis. Le nouveau tableau sera la somme des éléments des deux tableaux de départ.

Tableau 1 :

4	8	7	9	1	5	4	6

Tableau 2 :

Exercice 6.10

```
Variables i, N en Numérique
Tableaux T1(), T2(), T3() en Numérique
Debut
… (on suppose que T1 et T2 comptent N éléments, et qu'ils sont déjà saisis)
Redim T3(N-1)
…
Pour i ← 0 à N - 1
  T3(i) ← T1(i) + T2(i)
i Suivant
Fin
```

7	6	5	2	1	3	7	4

Tableau à constituer :

11	14	12	11	2	8	11	10

Exercice 6.11

Toujours à partir de deux tableaux précédemment saisis, écrivez un algorithme qui calcule le schtroumpf des deux tableaux. Pour calculer le schtroumpf, il faut multiplier chaque élément du tableau 1 par chaque élément du tableau 2, et additionner le tout. Par exemple si l'on a :

Tableau 1 :

4	8	7	12

Tableau 2 :

3	6

Le Schtroumpf sera :

3 * 4 + 3 * 8 + 3 * 7 + 3 * 12 + 6 * 4 + 6 * 8 + 6 * 7 + 6 * 12 = 279

Exercice 6.11

```
Variables i, j, N1, N2, S en Numérique
Tableaux T1(), T2() en Numérique
Debut
… On ne programme pas la saisie des tableaux T1 et T2.
On suppose que T1 possède N1 éléments, et que T2 en possède T2)
…
S ← 0
Pour i ← 0 à N1 – 1
  Pour j ← 0 à N2 – 1
    S ← S + T1(i) * T2(j)
  j Suivant
i Suivant
Ecrire "Le schtroumpf est : ", S
Fin
```

Exercice 6.12

Ecrivez un algorithme qui permette la saisie d'un nombre quelconque de valeurs, sur le principe de l'ex 6.8. Toutes les valeurs doivent être ensuite augmentées de 1, et le nouveau tableau sera affiché à l'écran.

Exercice 6.12

```
Variables Nb, i en Numérique
Tableau T() en Numérique
Debut
Ecrire "Entrez le nombre de valeurs : "
Lire Nb
Redim T(Nb-1)
Pour i ← 0 à Nb - 1
  Ecrire "Entrez le nombre n° ", i + 1
  Lire T(i)
i Suivant
Ecrire "Nouveau tableau : "
Pour i ← 0 à Nb – 1
  T(i) ← T(i) + 1
  Ecrire T(i)
i Suivant
Fin
```

Exercice 6.13

Exercice 6.13

```
Variables Nb, Posmaxi en Numérique
```

Ecrivez un algorithme permettant, toujours sur le même principe, à l'utilisateur de saisir un nombre déterminé de valeurs. Le programme, une fois la saisie terminée, renvoie la plus grande valeur en précisant quelle position elle occupe dans le tableau. On prendra soin d'effectuer la saisie dans un premier temps, et la recherche de la plus grande valeur du tableau dans un second temps.

```
Tableau T() en Numérique
Ecrire "Entrez le nombre de valeurs :"
Lire Nb
Redim T(Nb-1)
Pour i ← 0 à Nb - 1
  Ecrire "Entrez le nombre n° ", i + 1
  Lire T(i)
i Suivant
Posmaxi ← 0
Pour i ← 0 à Nb - 1
  Si T(i) > T(Posmaxi) alors
    Posmaxi ← i
  Finsi
i Suivant
Ecrire "Element le plus grand : ", T(Posmaxi)
Ecrire "Position de cet élément : ", Posmaxi
Fin
```

Exercice 6.14

Toujours et encore sur le même principe, écrivez un algorithme permettant, à l'utilisateur de saisir les notes d'une classe. Le programme, une fois la saisie terminée, renvoie le nombre de ces notes supérieures à la moyenne **de la classe**.

Exercice 6.14

```
Variables Nb, i, Som, Moy, Nbsup en Numérique
Tableau T() en Numérique
Debut
Ecrire "Entrez le nombre de notes à saisir : "
Lire Nb
Redim T(Nb-1)
Pour i ← 0 à Nb - 1
  Ecrire "Entrez le nombre n° ", i + 1
  Lire T(i)
i Suivant
Som ← 0
Pour i ← 0 à Nb - 1
  Som ← Som + T(i)
i Suivant
Moy ← Som / Nb
NbSup ← 0
Pour i ← 0 à Nb - 1
  Si T(i) > Moy Alors
    NbSup ← NbSup + 1
  FinSi
i Suivant
Ecrire NbSup, " élèves dépassent la moyenne de la classe"
Fin
```

VI. Tri et Recherche dans un Tableau

1. Tri d'un tableau : le tri par Sélection

Première de ces ruses de sioux, et par ailleurs tarte à la crème absolue du programmeur, donc : le tri de tableau.
Combien de fois au cours d'une carrière (brillante) de développeur a-t-on besoin de ranger des valeurs dans un ordre donné ? C'est inimaginable. Aussi, plutôt qu'avoir à réinventer à chaque fois la roue, le fusil à tirer dans les coins, le fil à couper le roquefort et la poudre à maquiller, vaut-il mieux avoir assimilé une ou deux techniques solidement éprouvées, même si elles paraissent un peu ardues au départ.
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. Champagne !
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

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 cetera, et cetera, 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.

Cela 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**

2. Un exemple de flag : la recherche dans un tableau

Nous allons maintenant nous intéresser au maniement habile 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.
Tout ceci peut vous sembler un peu fumeux, mais cela devrait s'éclairer à l'aide d'un exemple extrêmement fréquent : 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, disons, 20 valeurs. L'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, évidente, consiste à écrire les instructions de lecture / écriture, et la boucle – car il y en a manifestement une – de parcours du 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 bingo, 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
```

Et patatras, 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. Faute de mieux, on se contentera de faire dépendre pour le moment la réponse d'une variable booléenne que nous appellerons Trouvé.

```
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 Trouvé 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 Trouvé. Ceci se fait en deux étapes.

- un test figurant dans la boucle, indiquant lorsque la variable Trouvé 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.
- last, but not least, l'affectation par défaut de la variable Trouvé, dont la valeur de départ doit être évidemment Faux.

Au total, l'algorithme complet – et juste ! – donne :

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

Méditons un peu sur cette affaire.
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.

3. Tri de tableau + flag = tri à bulles

Et maintenant, nous en arrivons à la formule magique : 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. Cette constatation percutante semble digne de M. de Lapalisse, un ancien voisin à moi. Mais elle est plus profonde – et plus utile - qu'elle n'en a l'air.
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 charmante dénomination de « tri à bulle ». Comme quoi l'algorithmique n'exclut pas un minimum syndical de sens poétique.
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 :

```
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
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.

La solution complète donne donc :

```
Variable Yapermute en Booléen
Début
…
Yapermut ← Vrai
TantQue Yapermut
  Yapermut ← 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
      Yapermut ← Vrai
    Finsi
```

i **suivant**
FinTantQue
Fin
Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie de l'arsenal du programmeur bien armé.

4. La recherche dichotomique

Je ne sais pas si on progresse vraiment en algorithmique, mais en tout cas, qu'est-ce qu'on apprend comme vocabulaire !
Blague dans le coin, nous allons terminer ce chapitre migraineux 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. 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 entre la technique barbare et la

technique futée. 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.
Eh bien maintenant que je vous ai expliqué comment faire, vous n'avez plus qu'à traduire !

5. Exercices

Enonce des Exercices	Corrigés des Exercices

Exercice 7.1
 	
Ecrivez un algorithme qui permette de saisir un nombre quelconque de valeurs, et qui les range au fur et à mesure dans un tableau. Le programme, une fois la saisie terminée, doit dire si les éléments du tableau sont tous consécutifs ou non.
Par exemple, si le tableau est :
 	
12 13 14 15 16 17 18
 	
ses éléments sont tous consécutifs. En revanche, si le tableau est :
 	
9 10 11 15 16 17 18
 	
ses éléments ne sont pas tous consécutifs.
	Exercice 7.1
	Variables Nb, i **en Entier**
	Variable Flag **en Booleen**
	Tableau T() **en Entier**
	Debut
	Ecrire "Entrez le nombre de valeurs :"
	Lire Nb
	Redim T(Nb-1)
	Pour i ← 0 à Nb - 1
	  Ecrire "Entrez le nombre n° ", i + 1
	  Lire T(i)
	i **Suivant**
	Flag ← Vrai
	Pour i ← 1 à Nb - 1
	  Si T(i) <> T(i – 1) + 1 **Alors**
	  Flag ← Faux
	  FinSi
	i **Suivant**
	Si Flag **Alors**
	  Ecrire "Les nombres sont consécutifs"
	Sinon
	  Ecrire "Les nombres ne sont pas consécutifs"
	FinSi
	Fin
	Cette programmation est sans doute la plus spontanée, mais elle présente le défaut d'examiner la totalité du tableau, même lorsqu'on découvre dès le départ deux éléments non consécutifs. Aussi, dans le cas d'un grand tableau, est-elle dispendieuse en temps de traitement. Une autre manière de procéder serait de sortir de la boucle dès que deux éléments non consécutifs sont détectés. La deuxième partie de l'algorithme deviendrait donc :
	i ← 1
	TantQue T(i) = T(i – 1) + 1 et i < Nb - 1
	  i ← i + 1
	FinTantQue
	Si T(i) = T(i – 1) + 1 **Alors**
	  Ecrire "Les nombres sont consécutifs"
	Sinon
	  Ecrire "Les nombres ne sont pas consécutifs"
	FinSi

Exercice 7.2	Exercice 7.2 On suppose que N est le nombre d'éléments du tableau. Tri par

Ecrivez un algorithme qui trie un tableau dans l'ordre décroissant. Vous écrirez bien entendu deux versions de cet algorithme, l'une employant le tri par insertion, l'autre le tri à bulles.	insertion : … **Pour** i ← 0 à N - 2 posmaxi = i **Pour** j ← i + 1 à N - 1 **Si** t(j) > t(posmaxi) **alors** posmaxi ← j **Finsi** j **suivant** temp ← t(posmaxi) t(posmaxi) ← t(i) t(i) ← temp i **suivant** **Fin** Tri à bulles : … Yapermut ← Vrai **TantQue** Yapermut Yapermut ← Faux **Pour** i ← 0 à N - 2 **Si** t(i) < t(i + 1) **Alors** temp ← t(i) t(i) ← t(i + 1) t(i + 1) ← temp Yapermut ← Vrai **Finsi** i **suivant** **FinTantQue** **Fin**
Exercice 7.3 Ecrivez un algorithme qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement saisi (« les premiers seront les derniers… »)	Exercice 7.3 On suppose que n est le nombre d'éléments du tableau préalablement saisi … **Pour** i ← 0 à (N-1)/2 Temp ← T(i) T(i) ← T(N-1-i) T(N-1-i) ← Temp i **suivant** **Fin**
Exercice 7.4 Ecrivez un algorithme qui permette à l'utilisateur de supprimer une valeur d'un tableau préalablement saisi. L'utilisateur donnera l'indice de la valeur qu'il souhaite supprimer. Attention, il ne	Exercice 7.4 … **Ecrire** "Rang de la valeur à supprimer ?" **Lire** S **Pour** i ← S à N-2 T(i) ← T(i+1) i **suivant** **Redim** T(N–1)

s'agit pas de remettre une valeur à zéro, mais bel et bien de la supprimer du tableau lui-même ! Si le tableau de départ était :

12	8	4	45	64	9	2

Et que l'utilisateur souhaite supprimer la valeur d'indice 4, le nouveau tableau sera :

12	8	4	45	9	2

Exercice 7.5

Ecrivez l'algorithme qui recherche un mot saisi au clavier dans un dictionnaire. Le dictionnaire est supposé être codé dans un tableau préalablement rempli et trié.

Fin

Exercice 7.5

N est le nombre d'éléments du tableau Dico(), contenant les mots du dictionnaire, tableau préalablement rempli.

Variables Sup, Inf, Comp **en Entier**
Variables Fini **en Booléen**
Début
Ecrire "Entrez le mot à vérifier"
Lire Mot
On définit les bornes de la partie du tableau à considérer
Sup ← N - 1
Inf ← 0
Fini ← Faux
TantQue Non Fini
Comp désigne l'indice de l'élément à comparer. En bonne rigueur, il faudra veiller à ce que Comp soit bien un nombre entier, ce qui pourra s'effectuer de différentes manières selon les langages.
  Comp ← (Sup + Inf)/2
Si le mot se situe avant le point de comparaison, alors la borne supérieure change, la borne inférieure ne bouge pas.
  Si Mot < Dico(Comp) **Alors**
    Sup ← Comp - 1
Sinon, c'est l'inverse
  Sinon
    Inf ← Comp + 1
  FinSi
  Fini ← Mot = Dico(Comp) ou Sup < Inf
FinTantQue
Si Mot = Dico(Comp) **Alors**
  Ecrire "le mot existe"
Sinon
  Ecrire "Il n'existe pas"
Finsi
Fin

VII. Tableaux Multidimensionnels

1. Pourquoi plusieurs dimensions ?

Une seule ne suffisait-elle pas déjà amplement à notre bonheur, me demanderez-vous ? Certes, répondrai-je, mais vous allez voir qu'avec deux (et davantage encore) c'est carrément le nirvana.
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 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.
Arrivés à ce stade, les fines mouches du genre de Cyprien L. m'écriront pour faire remarquer qu'un damier, cela possède 100 cases et non 64, et qu'entre les damiers et les échiquiers, je me suis joyeusement emmêlé les pédales. A ces fines mouches, je ferai une double réponse de prof :

1. C'était pour voir si vous suiviez.

2. Si le prof décide contre toute évidence que les damiers font 64 cases, c'est le prof qui a raison et l'évidence qui a tort. Rompez.

Reprenons. 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 !

2. 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.

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. Je ne répondrai pas à cette question non parce que j'ai décidé de bouder, mais parce qu'elle 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.

3. 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.
Le principal obstacle au maniement systématique de ces tableaux à plus de trois dimensions est que le programmeur, quand il conçoit son algorithme, aime bien faire des petits gribouillis, des dessins immondes, imaginer les boucles dans sa tête, etc. Or, autant il est facile d'imaginer concrètement un tableau à une dimension, autant cela reste faisable pour deux dimensions, autant cela devient l'apanage d'une minorité privilégiée pour les tableaux à trois dimensions (je n'en fais malheureusement pas partie) et hors de portée de tout mortel au-delà. C'est comme ça, l'esprit humain a du mal à se représenter les choses dans l'espace, et crie grâce dès qu'il saute dans l'hyperespace (oui, c'est comme ça que ça s'appelle au delà de trois dimensions).
Donc, pour des raisons uniquement pratiques, les tableaux à plus de trois dimensions sont rarement utilisés par des programmeurs non matheux (car les matheux, de par leur formation, ont une fâcheuse propension à manier des espaces à n dimensions comme qui rigole, mais ce sont bien les seuls, et laissons les dans leur coin, c'est pas des gens comme nous).

4. Exercices

Enonce des Exercices	Corrigés des Exercices

Exercice 8.1
 	
Écrivez un algorithme remplissant un tableau de 6 sur 13, avec des zéros.
	Exercice 8.1
	Tableau Truc(5, 12) **en Entier**
	Debut
	Pour i ← 0 à 5
	  Pour j ← 0 à 12
	  Truc(i, j) ← 0
	  j **Suivant**
	i **Suivant**
	Fin

Exercice 8.2 Quel résultat produira cet algorithme ? **Tableau** X(1, 2) **en Entier** **Variables** i, j, val **en Entier** **Début** Val ← 1	Exercice 8.2 Cet algorithme remplit un tableau de la manière suivante: X(0, 0) = 1 X(0, 1) = 2 X(0, 2) = 3

```
Pour i ← 0 à 1
  Pour j ← 0 à 2
    X(i, j) ← Val
    Val ← Val + 1
  j Suivant
i Suivant
Pour i ← 0 à 1
  Pour j ← 0 à 2
    Ecrire X(i, j)
  j Suivant
i Suivant
Fin
```

X(1, 0) = 4
X(1, 1) = 5
X(1, 2) = 6
Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Exercice 8.3
Quel résultat produira cet algorithme ?

```
Tableau X(1, 2) en Entier
Variables i, j, val en Entier
Début
Val ← 1
Pour i ← 0 à 1
  Pour j ← 0 à 2
    X(i, j) ← Val
    Val ← Val + 1
  j Suivant
i Suivant
Pour j ← 0 à 2
  Pour i ← 0 à 1
    Ecrire X(i, j)
  i Suivant
j Suivant
Fin
```

Exercice 8.3

Cet algorithme remplit un tableau de la manière suivante:
X(0, 0) = 1
X(1, 0) = 4
X(0, 1) = 2
X(1, 1) = 5
X(0, 2) = 3
X(1, 2) = 6
Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Exercice 8.4
Quel résultat produira cet algorithme ?

```
Tableau T(3, 1) en Entier
Variables k, m, en Entier
Début
Pour k ← 0 à 3
  Pour m ← 0 à 1
    T(k, m) ← k + m
  m Suivant
k Suivant
Pour k ← 0 à 3
  Pour m ← 0 à 1
    Ecrire T(k, m)
  m Suivant
k Suivant
```

Exercice 8.4

Cet algorithme remplit un tableau de la manière suivante:
T(0, 0) = 0
T(0, 1) = 1
T(1, 0) = 1
T(1, 1) = 2
T(2, 0) = 2
T(2, 1) = 3
T(3, 0) = 3
T(3, 1) = 4
Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Fin

Exercice 8.5

Mêmes questions, en remplaçant la ligne :
T(k, m) ← k + m
par
T(k, m) ← 2 * k + (m + 1)
puis par :
T(k, m) ← (k + 1) + 4 * m

Exercice 8.5
Version a : cet algorithme remplit un tableau de la manière suivante:
T(0, 0) = 1
T(0, 1) = 2
T(1, 0) = 3
T(1, 1) = 4
T(2, 0) = 5
T(2, 1) = 6
T(3, 0) = 7
T(3, 1) = 8
Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Version b : cet algorithme remplit un tableau de la manière suivante:
T(0, 0) = 1
T(0, 1) = 5
T(1, 0) = 2
T(1, 1) = 6
T(2, 0) = 3
T(2, 1) = 7
T(3, 0) = 4
T(3, 1) = 8
Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Exercice 8.6

Soit un tableau T à deux dimensions (12, 8) préalablement rempli de valeurs numériques.
Écrire un algorithme qui recherche la plus grande valeur au sein de ce tableau.

Exercice 8.6
Variables i, j, iMax, jMax **en Numérique**
Tableau T(12, 8) **en Numérique**
Le principe de la recherche dans un tableau à deux dimensions est strictement le même que dans un tableau à une dimension, ce qui ne doit pas nous étonner. La seule chose qui change, c'est qu'ici le balayage requiert deux boucles imbriquées, au lieu d'une seule.

```
Debut
iMax ← 0
jMax ← 0
Pour i ← 0 à 12
  Pour j ← 0 à 8
    Si T(i,j) > T(iMax,jMax) Alors
      iMax ← i
      jMax ← j
    FinSi
  j Suivant
i Suivant
Ecrire "Le plus grand élément est ", T(iMax, jMax)
Ecrire "Il se trouve aux indices ", iMax, "; ", jMax
Fin
```

Exercice 8.7
Écrire un algorithme de jeu de dames très simplifié.
L'ordinateur demande à l'utilisateur dans quelle case se trouve son pion (quelle ligne, quelle colonne). On met en place un contrôle de saisie afin de vérifier la validité des valeurs entrées.
Ensuite, on demande à l'utilisateur quel mouvement il veut effectuer : 0 (en haut à gauche), 1 (en haut à droite), 2 (en bas à gauche), 3 (en bas à droite).
Si le mouvement est impossible (i.e. on sort du damier), on le signale à l'utilisateur et on s'arrête là . Sinon, on déplace le pion et on affiche le damier résultant, en affichant un « O » pour une case vide et un « X » pour la case où se trouve le pion.

Exercice 8.7
Variables i, j , posi, posj, i2, j2 **en Entier**
Variables Correct, MoveOK **en Booléen**
Tableau Damier(7, 7) **en Booléen**
Tableau Mouv(3, 1) **en Entier**
Le damier contenant un seul pion, on choisit de le coder à l'économie, en le représentant par un tableau de booléens à deux dimensions. Dans chacun des emplacements de ce damier, Faux signifie l'absence du pion, Vrai sa présence.

Par ailleurs, on emploie une méchante astuce, pas obligatoire, mais bien pratique dans beaucoup de situations. L'idée est de faire correspondre les choix possibles de l'utilisateur avec les mouvements du pion. On entre donc dans un tableau Mouv à deux dimensions, les déplacements du pion selon les quatre directions, en prenant soin que chaque ligne du tableau corresponde à une saisie de l'utilisateur. La première valeur étant le déplacement en i, la seconde le déplacement en j. Ceci nous épargnera par la suite de faire quatre fois les mêmes tests.
Debut
Choix 0 : pion en haut à droite
Mouv(0, 0) ← -1
Mouv(0, 1) ← -1
Choix 1 : pion en haut à droite
Mouv(1, 0) ← -1
Mouv(1, 1) ← 1
Choix 2 : pion en bas à gauche
Mouv(2, 0) ← 1
Mouv(2, 1) ← -1
Choix 3 : pion en bas à droite
Mouv(3, 0) ← 1
Mouv(3, 1) ← 1
Initialisation du damier; le pion n'est pour le moment nulle part
Pour i ← 0 à 7
  Pour j ← 0 à 7
    Damier(i, j) ← Faux
  j **suivant**
i **suivant**
Saisie de la coordonnée en i ("posi") avec contrôle de saisie
Correct ← Faux
TantQue Non Correct
  Ecrire "Entrez la ligne de votre pion: "
  Lire posi
  Si posi >= 0 et posi <= 7 **Alors**
    Correct ← vrai
  Finsi
Fintantque
Saisie de la coordonnée en j ("posj") avec contrôle de saisie

```
Correct ← Faux
TantQue Non Correct
  Ecrire "Entrez la colonne de votre pion: "
  Lire posj
    Si posj >= 0 et posj <= 7 Alors
      Correct ← Vrai
    Finsi
Fintantque
Positionnement du pion sur le damier virtuel.
Damier(posi, posj) ← Vrai
Saisie du déplacement, avec contrôle
Ecrire "Quel déplacement ?"
Ecrire " - 0: en haut à gauche"
Ecrire " - 1: en haut à droite"
Ecrire " - 2: en bas à gauche"
Ecrire " - 3: en bas à droite"
Correct ← Faux
TantQue Non Correct
  Lire Dep
  Si Dep >= 0 et Dep <= 3 Alors
    Correct ← Vrai
  FinSi
FinTantQue
i2 et j2 sont les futures coordonnées du pion. La variable
booléenne MoveOK vérifie la validité de ce futur emplacement
i2 ← posi + Mouv(Dep, 0)
j2 ← posj + Mouv(Dep, 1)
MoveOK ← i2 >= 0 et i2 <= 7 et j2 >= 0 et j2 <= 7
Cas où le déplacement est valide
Si MoveOK Alors
  Damier(posi, posj) ← Faux
  Damier(i2, j2) ← Vrai
Affichage du nouveau damier
  Pour i ← 0 à 7
    Pour j ← 0 à 7
      Si Damier(i, j) Alors
        Ecrire " O ";
      Sinon
        Ecrire " X ";
      FinSi
    j suivant
    Ecrire ""
  i suivant
Sinon
Cas où le déplacement n'est pas valide
  Ecrire "Mouvement impossible"
FinSi
Fin
```

VIII. Les Fonctions Prédéfinies

1. 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 19F90, 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.

Et d'entrée, nous trouvons :
LE GAG DE LA JOURNEE
Il consiste à affecter une fonction, quelle qu'elle soit.
Toute écriture plaçant une fonction à gauche d'une instruction d'affectation est aberrante, pour deux raisons symétriques.

- d'une part, parce que nous le savons depuis le premier chapitre de ce cours extraordinaire, 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, donc.

L'affectation d'une fonction sera donc considérée comme l'une des pires fautes algorithmiques, et punie comme telle.
Tavernier...

2. 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 (moins douloureuses, il faut le noter, que les extractions dentaires).
Tous les langages, je dis bien tous, proposent peu ou prou les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

- Len(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 :
Len("Bonjour, ça va ?") vaut 16
Len("") 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
Il existe aussi dans tous les langages une fonction qui renvoie le caractère correspondant à un code Ascii donné (fonction Asc), et Lycée de Versailles (fonction Chr) :
Asc("N") vaut 78
Chr(63) vaut "?"
J'insiste ; à moins de programmer avec un langage un peu particulier, comme le C, qui traite en réalité les chaînes de caractères comme des tableaux, on ne pourrait pas se passer des deux fonctions Len et Mid pour traiter les chaînes. Or, si les programmes informatiques ont fréquemment à traiter des nombres, ils doivent tout aussi fréquemment gérer des séries de caractères (des chaînes). Je sais bien que cela devient un refrain, mais connaître les techniques de base sur les chaînes est plus qu'utile : c'est indispensable.

3. 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 célébrissime test de parité (voir exercice dans pas longtemps).

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
Cette fonction peut paraître un peu bizarre, est réservée aux seuls matheux. Mais vous aurez là aussi l'occasion de voir dans les exercices à venir que ce n'est pas le cas.

Génération de nombres aléatoires
Une autre fonction classique , car très utile, est celle qui génère un nombre choisi au hasard.

Tous les programmes de jeu, ou presque, ont besoin de ce type d'outils, qu'il s'agisse de simuler un lancer de dés ou le déplacement chaotique du vaisseau spatial de l'enfer de la mort piloté par l'infâme Zorglub, qui veut faire main basse sur l'Univers (heureusement vous êtes là pour l'en empêcher, ouf).
Mais il n'y a pas que les jeux qui ont besoin de générer des nombres aléatoires. La modélisation (physique, géographique, économique, etc.) a parfois recours à des modèles dits stochastiques (chouette, encore un nouveau mot savant !). Ce sont des modèles dans lesquels les variables se déduisent les unes des autres par des relations déterministes (autrement dit des calculs), mais où l'on simule la part d'incertitude par une « fourchette » de hasard.
Par exemple, un modèle démographique supposera qu'une femme a en moyenne x enfants au cours de sa vie, mettons 1,5. Mais il supposera aussi que sur une population donnée, ce chiffre peut fluctuer entre 1,35 et 1,65 (si on laisse une part d'incertitude de 10%). Chaque année, c'est-à-dire chaque série de calcul des valeurs du modèle, on aura ainsi besoin de faire choisir à la machine un nombre au hasard compris entre 1,35 et 1,65.
Dans tous les langages, cette fonction existe et produit le résultat suivant :
Après : Toto ← Alea() On a : 0 =< Toto < 1
En fait, on se rend compte avec un tout petit peu de pratique que cette fonction Aléa peut nous servir pour générer n'importe quel nombre compris dans n'importe quelle fourchette. Je sais bien que mes lecteurs ne sont guère matheux, mais là, on reste franchement en deçà du niveau de feu le BEPC :

- si Alea génère un nombre compris entre 0 et 1, Alea multiplié par Z produit un nombre entre 0 et Z. Donc, il faut estimer la « largeur » de la fourchette voulue et multiplier Alea par cette « largeur » désirée.
- ensuite, si la fourchette ne commence pas à zéro, il va suffire d'ajouter ou de retrancher quelque chose pour « caler » la fourchette au bon endroit.

Par exemple, si je veux générer un nombre entre 1,35 et 1,65 ; la « fourchette » mesure 0,30 de large.
Donc : 0 =< Alea()*0,30 < 0,30
Il suffit dès lors d'ajouter 1,35 pour obtenir la fourchette voulue. Si j'écris que :
Toto ← Alea()*0,30 + 1,35
Toto aura bien une valeur comprise entre 1,35 et 1,65. Et le tour est joué !
Bon, il est grand temps que vous montriez ce que vous avez appris…

4. 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. Jusque là, pas de scoop me direz-vous, à juste titre vous répondrai-je, mais attendez donc la suite.
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).

5. Exercices

Enonce des Exercices	Corrigés des Exercices
Exercice 9.1 Parmi ces affectations (considérées indépendamment les unes des autres), lesquelles provoqueront des erreurs, et pourquoi ? Variables A, B, C en Numérique Variables D, E en Caractère A ← Sin(B) A ← Sin(A + B * C) B ← Sin(A) – Sin(D) D ← Sin(A / B) C ← Cos(Sin(A)	Exercice 9.1 A ← Sin(B) Aucun problème A ← Sin(A + B * C) Aucun problème B ← Sin(A) – Sin(D) Erreur ! D est en caractère D ← Sin(A / B) Aucun problème… si B est différent de zéro C ← Cos(Sin(A) Erreur ! Il manque une parenthèse fermante
Exercice 9.2 Ecrivez un algorithme qui demande un mot à l'utilisateur et qui affiche à l'écran le nombre de lettres de ce mot (c'est vraiment tout bête).	Exercice 9.2 Vous étiez prévenus, c'est bête comme chou ! Il suffit de se servir de la fonction Len, et c'est réglé : **Variable** Mot **en Caractère** **Variable** Nb **en Entier** **Debut** **Ecrire** "Entrez un mot : " **Lire** Mot Nb ← Len(Mot) **Ecrire** "Ce mot compte ", Nb, " lettres" **Fin**
Exercice 9.3 Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de mots de cette phrase. On suppose que les mots ne sont séparés que par des espaces (et c'est déjà un petit peu moins bête).	Exercice 9.3 Là, on est obligé de compter par une boucle le nombre d'espaces de la phrase, et on en déduit le nombre de mots. La boucle examine les caractères de la phrase un par un, du premier au dernier, et les compare à l'espace. **Variable** Bla **en Caractère** **Variables** Nb, i **en Entier** **Debut** **Ecrire** "Entrez une phrase : " **Lire** Bla Nb ← 0 **Pour** i ← 1 à Len(Bla) **Si** Mid(Bla, i , 1) = " " **Alors** Nb ← Nb + 1

Exercice 9.4

Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de voyelles contenues dans cette phrase.
On pourra écrire deux solutions. La première déploie une condition composée bien fastidieuse. La deuxième, en utilisant la fonction Trouve, allège considérablement l'algorithme.

```
  FinSi
i suivant
Ecrire "Cette phrase compte ", Nb + 1, " mots"
Fin
```

Exercice 9.4
Solution 1 : pour chaque caractère du mot, on pose une très douloureuse condition composée. Le moins que l'on puisse dire, c'est que ce choix ne se distingue pas par son élégance. Cela dit, il marche, donc après tout, pourquoi pas.

```
Variable Bla en Caractère
Variables Nb, i, j en Entier
Debut
Ecrire "Entrez une phrase : "
Lire Bla
Nb ← 0
Pour i ← 1 à Len(Bla)
  Si Mid(Bla, i, 1) = "a" ou Mid(Bla, i, 1) = "e" ou Mid(Bla, i, 1) = "i" ou Mid(Bla, i, 1) = "o" ou Mid(Bla, i, 1) = "u" ou Mid(Bla, i, 1) = "y" Alors
    Nb ← Nb + 1
  FinSi
i suivant
Ecrire "Cette phrase compte ", Nb, " voyelles"
Fin
```

Solution 2 : on stocke toutes les voyelles dans une chaîne. Grâce à la fonction Trouve, on détecte immédiatement si le caractère examiné est une voyelle ou non. C'est nettement plus sympathique...

```
Variables Bla, Voy en Caractère
Variables Nb, i, j en Entier
Debut
Ecrire "Entrez une phrase : "
Lire Bla
Nb ← 0
Voy ← "aeiouy"
Pour i ← 1 à Len(Bla)
  Si Trouve(Voy, Mid(Bla, i, 1)) <> 0 Alors
    Nb ← Nb + 1
  FinSi
i suivant
Ecrire "Cette phrase compte ", Nb, " voyelles"
Fin
```

Exercice 9.5

Exercice 9.5
Il n'existe aucun moyen de supprimer directement un caractère d'une chaîne… autrement qu'en procédant par collage. Il faut donc concaténer ce qui se trouve à

	gauche du caractère à supprimer, avec ce qui se trouve à sa droite. Attention aux paramètres des fonctions Mid, ils n'ont rien d'évident ! **Variable** Bla **en Caractère** **Variables** Nb, i, j **en Entier** **Début** **Ecrire** "Entrez une phrase : " **Lire** Bla **Ecrire** "Entrez le rang du caractère à supprimer : " **Lire** Nb L ← Len(Bla) Bla ← Mid(Bla, 1, Nb – 1) & Mid(Bla, Nb + 1, L – Nb) **Ecrire** "La nouvelle phrase est : ", Bla **Fin**
Exercice 9.6 - Cryptographie 1 Un des plus anciens systèmes de cryptographie (aisément déchiffrable) consiste à décaler les lettres d'un message pour le rendre illisible. Ainsi, les A deviennent des B, les B des C, etc. Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui la code selon ce principe. Comme dans le cas précédent, le codage doit s'effectuer au niveau de la variable stockant la phrase, et pas seulement à l'écran.	Exercice 9.6 Sur l'ensemble des exercices de cryptographie, il y a deux grandes stratégies possibles : - soit transformer les caractères en leurs codes ASCII. L'algorithme revient donc ensuite à traiter des nombres. Une fois ces nombres transformés, il faut les reconvertir en caractères. - soit en rester au niveau des caractères, et procéder directement aux transformations à ce niveau. C'est cette dernière option qui est choisie ici, et pour tous les exercices de cryptographie à venir. Pour cet exercice, il y a une règle générale : pour chaque lettre, on détecte sa position dans l'alphabet, et on la remplace par la lettre occupant la position suivante. Seul cas particulier, la vingt-sixième lettre (le Z) doit être codée par la première (le A), et non par la vingt-septième, qui n'existe pas ! **Variables** Bla, Cod, Alpha **en Caractère** **Variables** i, Pos **en Entier** **Début** **Ecrire** "Entrez la phrase à coder : " **Lire** Bla Alpha ← "ABCDEFGHIJKLMNOPQRSTUVWXYZ" Cod ← "" **Pour** i ← 1 à Len(Bla) Let ← Mid(Bla, i, 1) **Si** Let <> "Z" **Alors** Pos ← Trouve(Alpha, Let) Cod ← Cod & Mid(Alpha, Pos + 1, 1) **Sinon**

Cod ← Cod & "A"
FinSi
i **Suivant**
Bla ← Cod
Ecrire "La phrase codée est : ", Bla
Fin

Exercice 9.7 - Cryptographie 2 - *le chiffre de César*

Une amélioration (relative) du principe précédent consiste à opérer avec un décalage non de 1, mais d'un nombre quelconque de lettres. Ainsi, par exemple, si l'on choisit un décalage de 12, les A deviennent des M, les B des N, etc. Réalisez un algorithme sur le même principe que le précédent, mais qui demande en plus quel est le décalage à utiliser. Votre sens proverbial de l'élégance vous interdira bien sûr une série de vingt-six "Si...Alors"

Exercice 9.7
Cet algorithme est une généralisation du précédent. Mais là, comme on ne connaît pas d'avance le décalage à appliquer, on ne sait pas a priori combien de "cas particuliers", à savoir de dépassements au-delà du Z, il va y avoir.
Il faut donc trouver un moyen simple de dire que si on obtient 27, il faut en réalité prendre la lettre numéro 1 de l'alphabet, que si on obtient 28, il faut en réalité prendre la numéro 2, etc. Ce moyen simple existe : il faut considérer le reste de la division par 26, autrement dit le modulo.
Il y a une petite ruse supplémentaire à appliquer, puisque 26 doit rester 26 et ne pas devenir 0.

```
Variable Bla, Cod, Alpha en Caractère
Variables i, Pos, Décal en Entier
Début
Ecrire "Entrez le décalage à appliquer : "
Lire Décal
Ecrire "Entrez la phrase à coder : "
Lire Bla
Alpha ← "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Cod ← ""
Pour i ← 1 à Len(Bla)
  Let ← Mid(Bla, i, 1)
  Pos ← Trouve(Alpha, Let)
  NouvPos ← Mod(Pos + Décal, 26)
  Si NouvPos = 0 Alors
    NouvPos ← 26
  FinSi
  Cod ← Cod & Mid(Alpha, NouvPos, 1)
i Suivant
Bla ← Cod
Ecrire "La phrase codée est : ", Bla
Fin
```

Exercice 9.8 - Cryptographie 3

Une technique ultérieure de cryptographie consista à opérer non avec un décalage systématique, mais par une substitution

Exercice 9.8
Là, c'est assez direct.

```
Variable Bla, Cod, Alpha en Caractère
Variables i, Pos, Décal en Entier
Début
Ecrire "Entrez l'alphabet clé : "
```

aléatoire. Pour cela, on utilise un alphabet-clé, dans lequel les lettres se succèdent de manière désordonnée, par exemple :
HYLUJPVREAKBNDOFSQZCWMGITX
C'est cette clé qui va servir ensuite à coder le message. Selon notre exemple, les A deviendront des H, les B des Y, les C des L, etc.
Ecrire un algorithme qui effectue ce cryptage (l'alphabet-clé sera saisi par l'utilisateur, et on suppose qu'il effectue une saisie correcte).

```
Lire Clé
Ecrire "Entrez la phrase à coder : "
Lire Bla
Alpha ← "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Cod ← ""
Pour i ← 1 à Len(Bla)
  Let ← Mid(Bla, i, 1)
  Pos ← Trouve(Alpha, Let)
  Cod ← Cod & Mid(Clé, Pos, 1)
i Suivant
Bla ← Cod
Ecrire "La phrase codée est : ", Bla
Fin
```

Exercice 9.9 - Cryptographie 4 - *le chiffre de Vigenère*

Un système de cryptographie beaucoup plus difficile à briser que les précédents fut inventé au XVIe siècle par le français Vigenère. Il consistait en une combinaison de différents chiffres de César.
On peut en effet écrire 25 alphabets décalés par rapport à l'alphabet normal :

- l'alphabet qui commence par B et finit par …YZA
- l'alphabet qui commence par C et finit par …ZAB
- etc.

Le codage va s'effectuer sur le principe du chiffre de César : on remplace la lettre d'origine par la lettre occupant la même place dans l'alphabet décalé.
Mais à la différence du chiffre de César, un même message va utiliser non un, mais plusieurs alphabets décalés. Pour savoir quels alphabets doivent être utilisés, et dans quel ordre, on utilise une clé.
Si cette clé est "VIGENERE" et le message "Il faut coder cette phrase", on procèdera comme suit :
La première lettre du message, I, est la 9e lettre de l'alphabet normal. Elle doit être codée en utilisant l'alphabet commençant par la première lettre de la clé, V. Dans cet

Exercice 9.9
Le codage de Vigenère n'est pas seulement plus difficile à briser; il est également un peu plus raide à programmer. La difficulté essentielle est de comprendre qu'il faut deux boucles: l'une pour parcourir la phrase à coder, l'autre pour parcourir la clé. Mais quand on y réfléchit bien, ces deux boucles ne doivent surtout pas être imbriquées. Et en réalité, quelle que soit la manière dont on l'écrit, elle n'en forment qu'une seule.

```
Variables Alpha, Bla, Cod, Clé, Let en Caractère
Variables i, Pos, PosClé, Décal en Entier
Début
Ecrire "Entrez la clé : "
Lire Clé
Ecrire "Entrez la phrase à coder : "
Lire Bla
Alpha ← "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Cod ← ""
PosClé ← 0
Pour i ← 1 à Len(Bla)
```

On gère la progression dans la clé. J'ai effectué cela "à la main" par une boucle, mais un joli emploi de la fonction Modulo aurait permis une programmation en une seule ligne!

```
  Posclé ← Posclé + 1
  Si PosClé > Len(Clé) Alors
    PosClé ← 1
  FinSi
```

On détermine quelle est la lettre clé et sa position dans l'alphabet

```
  LetClé ← Mid(Clé, PosClé, 1)
  PosLetClé ← Trouve(Alpha, LetClé)
```

On détermine la position de la lettre à coder et le décalage à appliquer. Là encore, une solution

alphabet, la 9e lettre est le D. I devient donc D.
La deuxième lettre du message, L, est la 12e lettre de l'alphabet normal. Elle doit être codée en utilisant l'alphabet commençant par la deuxième lettre de la clé, I. Dans cet alphabet, la 12e lettre est le S. L devient donc S, etc.
Quand on arrive à la dernière lettre de la clé, on recommence à la première.
Ecrire l'algorithme qui effectue un cryptage de Vigenère, en demandant bien sûr au départ la clé à l'utilisateur.

alternative aurait été d'employer Mod : cela nous aurait épargné le Si…

```
  Let ← Mid(Bla, i, 1)
  Pos ← Trouve(Alpha, Let)
  NouvPos ← Pos + PosLetClé
  Si NouvPos > 26 Alors
    NouvPos ← NouvPos – 26
  FinSi
  Cod ← Cod & Mid(Alpha, NouvPos, 1)
i Suivant
Bla ← Cod
Ecrire "La phrase codée est : ", Bla
Fin
```

Exercice 9.10

Ecrivez un algorithme qui demande un nombre entier à l'utilisateur. L'ordinateur affiche ensuite le message "Ce nombre est pair" ou "Ce nombre est impair" selon le cas.

Exercice 9.10
On en revient à des choses plus simples...

```
Variable Nb en Entier
Ecrire "Entrez votre nombre : "
Lire Nb
Si Nb/2 = Ent(Nb/2) Alors
  Ecrire "Ce nombre est pair"
Sinon
  Ecrire "Ce nombre est impair"
FinSi
Fin
```

Exercice 9.11
Ecrivez les algorithmes qui génèrent un nombre Glup aléatoire tel que …

- 0 =< Glup < 2
- –1 =< Glup < 1
- 1,35 =< Glup < 1,65
- Glup émule un dé à six faces
- –10,5 =< Glup < +6,5
- Glup émule la somme du jet simultané de deux dés à six faces

Exercice 9.11

```
a) Glup ← Alea() * 2
b) Glup ← Alea() * 2 - 1
c) Glup ← Alea() * 0,30 + 1,35
d) Glup ← Ent(Alea() * 6) + 1
e) Glup ← Alea() * 17 – 10,5
f) Glup ← Ent(Alea()*6) + Ent(Alea()*6) + 2
```

IX. Procédures et Fonctions

1. Fonctions personnalisées

1.1 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, et roule Raoul. 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 lourdingue. Elle contient des répétitions, et pour peu que le programme soit joufflu, 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 aiguë. L'ignorer, c'est donc forcément grave.
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, patatras, 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.
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 le nom de la variable dans laquelle on range la réponse. Alors, il doit bien y avoir un truc.
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 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
```

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 toujours 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()
...
```

Et le tour est joué ! 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. Elle n'est pas belle, la vie ?
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...

1.2 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. Cela n'est certes pas une découverte pour vous : nous avons longuement utilisé les arguments à propos des fonctions prédéfinies. Eh bien, quitte à construire nos propres fonctions, nous pouvons donc construire nos propres arguments. Voilà comment l'affaire se présente...
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
```

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 ?")
...
```

Et voilà le travail.
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.

1.3 Deux mots sur l'analyse fonctionnelle

Comme souvent en algorithmique, si l'on s'en tient à la manière dont marche l'outil, tout cela n'est en réalité pas très compliqué. Les fonctions personnalisées se déduisent très logiquement de la manière nous nous avons déjà expérimenté les fonctions prédéfinies.
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.

Cette partie de la réflexion s'appelle d'ailleurs l'analyse fonctionnelle d'un problème, et c'est toujours par elle qu'il faut commencer : en gros, dans un premier temps, on découpe le traitement en modules (algorithmique fonctionnelle), et dans un deuxième temps, on écrit chaque module (algorithmique classique). Cependant, avant d'en venir là, il nous faut découvrir deux autres outils, qui prennent le relais là où les fonctions deviennent incapables de nous aider.

2. Sous Procédures

2.1 Généralités

Les fonctions, c'est bien, mais dans certains cas, ça ne nous rend guère service.
Il peut en effet arriver que dans un programme, on ait à réaliser des tâches répétitives, mais que ces tâches n'aient pas pour rôle de générer une valeur particulière, ou qu'elles aient pour rôle d'en générer plus d'une à la fois. Vous ne voyez pas de quoi je veux parler ? Prenons deux exemples.
Premier exemple. Imaginons qu'au cours de mon application, j'aie plusieurs fois besoin d'effacer l'écran et de réafficher un bidule comme un petit logo en haut à gauche. On pourrait se dire qu'il faut créer une fonction pour faire cela. Mais quelle serait la valeur renvoyée par la fonction ? Aucune ! Effacer l'écran, ce n'est pas produire un résultat stockable dans une variable, et afficher un logo non plus. Voilà donc une situation ou j'ai besoin de répéter du code, mais où ce code n'a pas comme rôle de produire une valeur.
Deuxième exemple. Au cours de mon application, je dois plusieurs fois faire saisir un tableau d'entiers (mais à chaque fois, un tableau différent). Là encore, on serait tenté d'effectuer toutes ces saisies de tableaux dans une seule fonction. Mais problème, une fonction ne peut renvoyer qu'une seule valeur à la fois. Elle ne peut donc renvoyer un tableau, qui est une série de valeurs distinctes.
Alors, dans ces deux cas, faute de pouvoir traiter l'affaire par une fonction, devra-t-on en rester au code répétitif dont nous venons de dénoncer si vigoureusement les faiblesses ? Mmmmmh ? Vous vous doutez bien que non. Heureusement, tout est prévu, il y a une solution. Et celle-ci consiste à utiliser des sous-procédures.
En fait, les fonctions - que nous avons vues - ne sont finalement qu'un cas particulier des sous-procédures - que nous allons voir : celui où doit être renvoyé vers la procédure appelante une valeur et une seule. Dans tous les autres cas (celui où on ne renvoie aucune valeur, comme celui ou en en renvoie plusieurs), il faut donc avoir recours non à la forme particulière et simplifiée (la fonction), mais à la forme générale (la sous-procédure).
Parlons donc de ce qui est commun aux sous-procédures et aux fonctions, mais aussi de ce qui les différencie. Voici comment se présente une sous-procédure :
Procédure Bidule(...)
...
Fin Procédure
Dans la procédure principale, l'appel à la sous-procédure Bidule devient quant à lui :
Appeler Bidule(...)
Établissons un premier état des lieux.

- Alors qu'une fonction se caractérisait par les mots-clés **Fonction** ... **Fin Fonction**, une sous-procédure est identifiée par les mots-clés **Procédure ... Fin Procédure**. Oui, je sais, c'est un peu trivial comme remarque, mais, bon, on ne sait jamais.
- Lorsqu'une fonction était appelée, sa valeur (retournée) était toujours affectée à une variable (ou intégrée dans le calcul d'une expression). L'appel à une procédure, lui, est au contraire toujours une **instruction autonome**. "Exécute la procédure Bidule" est un ordre qui se suffit à lui-même.
- Toute fonction devait, pour cette raison, comporter l'instruction "Renvoyer". Pour la même raison, l'instruction "Renvoyer" n'est jamais utilisée dans une sous-procédure. La fonction est

une valeur calculée, qui renvoie son résultat vers la procédure principale. La sous-procédure, elle, est un traitement ; elle ne "vaut" rien.

- Même une fois qu'on a bien compris les trois premiers points, on n'est pas complètement au bout de nos peines.

2.2 Le problème des arguments

En effet, il nous reste à examiner ce qui peut bien se trouver dans les parenthèses, à la place des points de suspension, aussi bien dans la déclaration de la sous-procédure que dans l'appel. Vous vous en doutez bien : c'est là que vont se trouver les outils qui vont permettre l'échange d'informations entre la procédure principale et la sous-procédure (en fait, cette dernière phrase est trop restrictive : mieux vaudrait dire : entre la procédure appelante et la procédure appelée. Car une sous-procédure peut très bien en appeler elle-même une autre afin de pouvoir accomplir sa tâche)
De même qu'avec les fonctions, les valeurs qui circulent depuis la procédure (ou la fonction) appelante vers la sous-procédure appelée se nomment des arguments, ou des paramètres en entrée de la sous-procédure. Comme on le voit, qu'il s'agisse des sous-procédure ou des fonctions, ces choses jouant exactement le même rôle (transmettre une information depuis le code donneur d'ordres jusqu'au code sous-traitant), elle portent également le même nom. Unique petite différence, on a précisé cette fois qu'il s'agissait d'arguments, ou de paramètres, en entrée. Pourquoi donc ?
Tout simplement parce que que dans une sous-procédure, on peut être amené à vouloir renvoyer des résultats vers le programme principal ; or, là, à la différence des fonctions, rien n'est prévu : la sous-procédure, en tant que telle, ne "renvoie" rien du tout (comme on vient de le voir, elle est d'ailleurs dépourvue de l'instruction "renvoyer"). Ces résultats que la sous-procédure doit transmettre à la procédure appelante devront donc eux aussi être véhiculés par des paramètres. Mais cette fois, il s'agira de paramètres fonctionnant dans l'autre sens (du sous-traitant vers le donneur d'ordres) : on les appellera donc des paramètres en sortie.
Ceci nous permet de reformuler en d'autres termes la vérité fondamentale apprise un peu plus haut : toute sous-procédure possédant un et un seul paramètre en sortie peut également être écrite sous forme d'une fonction (et entre nous, c'est une formulation préférable car un peu plus facile à comprendre et donc à retenir).
Jusque là, ça va ? Si oui, prenez un cachet d'aspirine et poursuivez la lecture. Si non, prenez un cachet d'aspirine et recommencez depuis le début. Et dans les deux cas, n'oubliez pas le grand verre d'eau pour faire passer l'aspirine.
Il nous reste un détail à examiner, détail qui comme vous vous en doutez bien, a une certaine importance : comment fait-on pour faire comprendre à un langage quels sont les paramètres qui doivent fonctionner en entrée et quels sont ceux qui doivent fonctionner en sortie...

2.3 Comment ça marche tout ça ?

En fait, si je dis qu'un paramètre est "en entrée" ou "en sortie", j'énonce quelque chose à propos de son rôle dans le programme. Je dis ce que je souhaite qu'il fasse, la manière dont je veux qu'il se comporte. Mais les programmes eux-mêmes n'ont cure de mes désirs, et ce n'est pas cette classification qu'ils adoptent. C'est toute la différence entre dire qu'une prise électrique sert à brancher un rasoir ou une cafetière (ce qui caractérise son rôle), et dire qu'elle est en 220 V ou en 110 V (ce qui caractérise son type technique, et qui est l'information qui intéresse l'électricien). A l'image des électriciens, les langages se contrefichent de savoir quel sera le rôle (entrée ou sortie) d'un paramètre. Ce qu'ils exigent, c'est de connaître leur voltage... pardon, le mode de passage de ces paramètres. Il n'en existe que deux :

- le passage par valeur

- le passage par référence

...Voyons de plus près de quoi il s'agit.
Reprenons l'exemple que nous avons déjà utilisé plus haut, celui de notre fonction RepOuiNon. Comme nous l'avons vu, rien ne nous empêche de réécrire cette fonction sous la forme d'une procédure (puisqu'une fonction n'est qu'un cas particulier de sous-procédure). Nous laisserons pour l''instant de côté la question de savoir comment renvoyer la réponse (contenue dans la variable Truc) vers le programme principal. En revanche, nous allons déclarer que Msg est un paramètre dont la transmission doit se faire par valeur. Cela donnera la chose suivante :

Procédure RepOuiNon(Msg **en Caractère par valeur**)
Ecrire Msg
Truc ← ""
TantQue Truc <> "Oui" et Truc <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Truc
FinTantQue
??? Comment transmettre Truc à la procédure appelante ???
Fin Procédure

Quant à l'appel à cette sous-procédure, il pourra prendre par exemple cette forme :

M ← "Etes-vous marié ?"
Appeler RepOuiNon(M)

Que va-t-il se passer ?
Lorsque le programme principal arrive sur la première ligne, il affecte la variable M avec le libellé "Êtes-vous marié". La ligne suivante déclenche l'exécution de la sous-procédure. Celle-ci crée aussitôt une variable Msg. Celle-ci ayant été déclarée comme un paramètre passé **par valeur**, Msg va être affecté avec le même contenu que M. Cela signifie que Msg est dorénavant une copie de M. Les informations qui étaient contenues dans M ont été intégralement recopiées (en double) dans Msg. Cette copie subsistera tout au long de l'exécution de la sous-procédure RepOuiNon et sera détruite à la fin de celle-ci.
Une conséquence essentielle de tout cela est que si d'aventure la sous-procédure RepOuiNon contenait une instruction qui modifiait le contenu de la variable Msg, cela n'aurait aucune espèce de répercussion sur la procédure principale en général, et sur la variable M en particulier. La sous-procédure ne travaillant que sur une copie de la variable qui a été fournie par le programme principal, elle est incapable, même si on le souhaitait, de modifier la valeur de celle-ci. Dit d'une autre manière, dans une procédure, un paramètre passé par valeur ne peut être qu'un paramètre en entrée.
C'est en même temps une limite (aggravée par le fait que les informations ainsi recopiées occupent dorénavant deux fois plus de place en mémoire) et une sécurité : quand on transmet un paramètre par valeur, on est sûr et certain que même en cas de bug dans la sous-procédure, la valeur de la variable transmise ne sera jamais modifiée par erreur (c'est-à-dire écrasée) dans le programme principal.
Admettons à présent que nous déclarions un second paramètre, Truc, en précisant cette fois qu'il sera transmis par référence. Et adoptons pour la procédure l'écriture suivante :

Procédure RepOuiNon(Msg **en Caractère par valeur,** Truc **en Caractère par référence**)
Ecrire Msg
Truc ← ""
TantQue Truc <> "Oui" et Truc <> "Non"
  Ecrire "Tapez Oui ou Non"
  Lire Truc

FinTantQue
Fin Fonction
L'appel à la sous-procédure deviendrait par exemple :
M ← "Etes-vous marié ?"
Appeler RepOuiNon(M, T)
Ecrire "Votre réponse est ", T
Dépiautons le mécanisme de cette nouvelle écriture. En ce qui concerne la première ligne, celle qui affecte la variable M, rien de nouveau sous le soleil. Toutefois, l'appel à la sous-procédure provoque deux effets très différents. Comme on l'a déjà dit, la variable Msg est créée et immédiatement affectée avec une copie du contenu de M, puisqu'on a exigé un passage par valeur. Mais en ce qui concerne Truc, il en va tout autrement. Le fait qu'il s'agisse cette fois d'un passage par référence fait que la variable Truc ne contiendra pas la valeur de T, mais son adresse, c'est-à-dire sa référence. Dès lors, toute modification de Truc sera immédiatement redirigée, par ricochet en quelque sorte, sur T. Truc n'est pas une variable ordinaire : elle ne contient pas de valeur, mais seulement la référence à une valeur, qui elle, se trouve ailleurs (dans la variable T). Il s'agit donc d'un genre de variable complètement nouveau, et différent de ce que nous avons vu jusque là. Ce type de variable porte un nom : on l'appelle un pointeur. Tous les paramètres passés par référence sont des pointeurs, mais les pointeurs ne se limitent pas aux paramètres passés par référence (même si ce sont les seuls que nous verrons dans le cadre de ce cours). Il faut bien comprendre que ce type de variable étrange est géré directement par les langages : à partir du moment où une variable est considérée comme un pointeur, toute affectation de cette variable se traduit automatiquement par la modification de la variable sur laquelle elle pointe.
Passer un paramètre par référence, cela présente donc deux avantages. Et d'une, on gagne en occupation de place mémoire, puisque le paramètre en question ne recopie pas les informations envoyées par la procédure appelante, mais qu'il se contente d'en noter l'adresse. Et de deux, cela permet d'utiliser ce paramètre tant en lecture (en entrée) qu'en écriture (en sortie), puisque toute modification de la valeur du paramètre aura pour effet de modifier la variable correspondante dans la procédure appelante.
Nous pouvons résumer tout cela par un petit tableau :

	passage par valeur	**passage par référence**
utilisation en entrée	oui	oui
utilisation en sortie	non	oui

Mais alors, demanderez-vous dans un élan de touchante naïveté, si le passage par référence présente les deux avantages présentés il y a un instant, pourquoi ne pas s'en servir systématiquement ? Pourquoi s'embêter avec les passages par valeur, qui non seulement utilisent de la place en mémoire, mais qui de surcroît nous interdisent d'utiliser la variable comme un paramètre en sortie ?
Eh bien, justement, parce qu'on ne pourra pas utiliser comme paramètre en sortie, et que cet inconvénient se révèle être aussi, éventuellement, un avantage. Disons la chose autrement : c'est une sécurité. C'est la garantie que quel que soit le bug qui pourra affecter la sous-procédure, ce bug ne viendra jamais mettre le foutoir dans les variables du programme principal qu'elle ne doit pas toucher. Voilà pourquoi, lorsqu'on souhaite définir un paramètre dont on sait qu'il fonctionnera exclusivement en entrée, il est sage de le verrouiller, en quelque sorte, en le définissant comme passé par valeur. Et Lycée de Versailles, ne seront définis comme passés par référence que les paramètres dont on a absolument besoin qu'ils soient utilisés en sortie.

3. Variables publiques et privées

Résumons la situation. Nous venons de voir que nous pouvions 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. 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 privées, ou locales.
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 publique, ou globale.
La manière dont la déclaration d'une variable publique doit être faites est évidemment fonction de chaque langage de programmation. En pseudo-code algorithmique, on pourra utiliser le mot-clé Publique :
Variable Publique Toto **en Numérique**
Alors, pourquoi ne pas rendre toutes les variables publiques, et s'épargner ainsi de fastidieux efforts pour passer des paramètres ? C'est très simple, et c'est toujours la même chose : les variables globales consomment énormément de ressources en mémoire. En conséquence, le principe qui doit présider au choix entre variables publiques et privées doit être celui de l'économie de moyens : on ne déclare comme publiques 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 publiques.

4. 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.
N'importe quoi, c'est quoi ? C'est par exemple, comme on vient de le voir, mettre des variables globales partout, sous prétexte que c'est autant de paramètres qu'on n'aura pas à passer.
Mais on peut imaginer d'autres atrocités.
Par exemple, une fonction, dont un des paramètres d'entrée serait passé par référence, et modifié par la fonction. Ce qui signifierait que cette fonction produirait non pas un, mais deux résultats. Autrement dit, que sous des dehors de fonctions, elle se comporterait en réalité comme une sous-procédure.

Ou inversement, on peut concevoir une procédure qui modifierait la valeur d'un paramètre (et d'un seul) passé par référence. Il s'agirait là d'une procédure qui en réalité, serait une fonction. Quoique ce dernier exemple ne soit pas d'une gravité dramatique, il participe de la même logique consistant à embrouiller le code en faisant passer un outil pour un autre, au lieu d'adopter la structure la plus claire et la plus lisible possible.
Enfin, il ne faut pas écarter la possibilité de programmeurs particulièrement vicieux, qui par un savant 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.
Trèfle de plaisanteries : 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. Il convient donc :

1. de limiter au minimum l'utilisation des variables globales. Celles-ci doivent être employées avec nos célèbres amis italo-arméniens, c'est-à-dire 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, et des sous-procédures dans tous les autres cas (ce qui implique de ne jamais passer un paramètre par référence à une fonction : soit on n'en a pas besoin, soit on en a besoin, et ce n'est alors plus une fonction).

Respecter ces règles d'hygiène est indispensable si l'on veut qu'une application ressemble à autre chose qu'au palais du facteur Cheval. Car une architecture à laquelle on ne comprend rien, c'est sans doute très poétique, mais il y a des circonstances où l'efficacité est préférable à la poésie. Et, pour ceux qui en douteraient encore, la programmation informatique fait (hélas ?) partie de ces circonstances.
5. Algorithmes fonctionnels
Pour clore ce chapitre, voici quelques mots supplémentaires à propos de la structure générale d'une application. Comme on l'a dit à plusieurs reprises, celle-ci va couramment être formée d'une procédure principale, et de fonctions et de sous-procédures (qui vont au besoin elles-mêmes en appeler d'autres, etc.). L'exemple typique est celui d'un menu, ou d'un sommaire, qui « branche » sur différents traitements, donc différentes sous-procédures.
L'algorithme fonctionnel de l'application est le découpage et/ou la représentation graphique de cette structure générale, ayant comme objectif de faire comprendre d'un seul coup d'œil quelle procédure fait quoi, et quelle procédure appelle quelle autre. L'algorithme fonctionnel est donc en quelque sorte la construction du squelette de l'application. Il se situe à un niveau plus général, plus abstrait, que l'algorithme normal, qui lui, détaille pas à pas les traitements effectués au sein de chaque procédure.
Dans la construction – et la compréhension – d'une application, les deux documents sont indispensables, et constituent deux étapes successives de l'élaboration d'un projet. La troisième – et dernière – étape, consiste à écrire, pour chaque procédure et fonction, l'algorithme détaillé.
Exemple de réalisation d'un algorithme fonctionnel : Le Jeu du Pendu
Vous connaissez tous ce jeu : l'utilisateur doit deviner un mot choisi au hasard par l'ordinateur, en un minimum d'essais. Pour cela, il propose des lettres de l'alphabet. Si la lettre figure dans le mot à trouver, elle s'affiche. Si elle n'y figure pas, le nombre des mauvaises réponses augmente de 1. Au bout de dix mauvaises réponses, la partie est perdue.

Ce petit jeu va nous permettre de mettre en relief les trois étapes de la réalisation d'un algorithme un peu complexe ; bien entendu, on pourrait toujours ignorer ces trois étapes, et se lancer comme un dératé directement dans la gueule du loup, à savoir l'écriture de l'algorithme définitif. Mais, sauf à être particulièrement doué, mieux vaut respecter le canevas qui suit, car les difficultés se résolvent mieux quand on les saucissonne…

Etape 1 : le dictionnaire des données

Le but de cette étape est d'identifier les informations qui seront nécessaires au traitement du problème, et de choisir le type de codage qui sera le plus satisfaisant pour traiter ces informations. C'est un moment essentiel de la réflexion, qu'il ne faut surtout pas prendre à la légère… Or, neuf programmeurs débutants sur dix bâclent cette réflexion, quand ils ne la zappent pas purement et simplement. La punition ne se fait généralement pas attendre longtemps ; l'algorithme étant bâti sur de mauvaises fondations, le programmeur se rend compte tout en l'écrivant que le choix de codage des informations, par exemple, mène à des impasses. La précipitation est donc punie par le fait qu'on est obligé de tout reprendre depuis le début, et qu'on a au total perdu bien davantage de temps qu'on en a cru en gagner…

Donc, avant même d'écrire quoi que ce soit, les questions qu'il faut se poser sont les suivantes :

- de quelles informations le programme va-t-il avoir besoin pour venir à bout de sa tâche ?
- pour chacune de ces informations, quel est le meilleur codage ? Autrement dit, celui qui sans gaspiller de la place mémoire, permettra d'écrire l'algorithme le plus simple ?

Encore une fois, il ne faut pas hésiter à passer du temps sur ces questions, car certaines erreurs, ou certains oublis, se payent cher par la suite. Et inversement, le temps investi à ce niveau est largement rattrapé au moment du développement proprement dit.

Pour le jeu du pendu, voici la liste des informations dont on va avoir besoin :

- une liste de mots (si l'on veut éviter que le programme ne propose toujours le même mot à trouver, ce qui risquerait de devenir assez rapidement lassant…)
- le mot à deviner
- la lettre proposée par le joueur à chaque tour
- le nombre actuel de mauvaises réponses
- et enfin, last but not least, l'ensemble des lettres déjà trouvées par le joueur. Cette information est capitale ; le programme en aura besoin au moins pour deux choses : d'une part, pour savoir si le mot entier a été trouvé. D'autre part, pour afficher à chaque tour l'état actuel du mot (je rappelle qu'à chaque tour, les lettres trouvées sont affichées en clair par la machine, les lettres restant à deviner étant remplacées par des tirets).
- à cela, on pourrait ajouter une liste comprenant l'ensemble des lettres déjà proposées par le joueur, qu'elles soient correctes ou non ; ceci permettra d'interdire au joueur de proposer à nouveau une lettre précédemment jouée.

Cette liste d'informations n'est peut-être pas exhaustive ; nous aurons vraisemblablement besoin au cours de l'algorithme de quelques variables supplémentaires (des compteurs de boucles, des variables temporaires, etc.). Mais les informations essentielles sont bel et bien là. Se pose maintenant le problème de choisir le mode de codage le plus futé. Si, pour certaines informations, la question va être vite réglée, pour d'autres, il va falloir faire des choix (et si possible, des choix intelligents !).

C'est parti, mon kiki :

- Pour la liste des mots à trouver, il s'agit d'un ensemble d'informations de type alphanumérique. Ces informations pourraient faire partie du corps de la procédure principale,

et être ainsi stockées en mémoire vive, sous la forme d'un tableau de chaînes. Mais ce n'est certainement pas le plus judicieux. Toute cette place occupée risque de peser lourd inutilement, car il n'y a aucun intérêt à stocker l'ensemble des mots en mémoire vive. Et si l'on souhaite enrichir la liste des mots à trouver, on sera obligé de réécrire des lignes de programme… Conclusion, la liste des mots sera bien plus à sa place dans un fichier texte, dans lequel le programme ira piocher un seul mot, celui qu'il faudra trouver. Nous constituerons donc un fichier texte, appelé dico.txt, dans lequel figurera un mot par ligne (par enregistrement).

- Le mot à trouver, lui, ne pose aucun problème : il s'agit d'une information simple de type chaîne, qui pourra être stocké dans une variable appelée mot, de type caractère.
- De même, la lettre proposée par le joueur est une information simple de type chaîne, qui sera stockée dans une variable appelée lettre, de type caractère.
- Le nombre actuel de mauvaises réponses est une information qui pourra être stockée dans une variable numérique de type entier simple appelée MovRep.
- L'ensemble des lettres trouvées par le joueur est typiquement une information qui peut faire l'objet de plusieurs choix de codage ; rappelons qu'au moment de l'affichage, nous aurons besoin de savoir pour chaque lettre du mot à deviner si elle a été trouvée ou non. Une première possibilité, immédiate, serait de disposer d'une chaîne de caractères comprenant l'ensemble des lettres précédemment trouvées. Cette solution est loin d'être mauvaise, et on pourrait tout à fait l'adopter. Mais ici, on fera une autre choix, ne serait-ce que pour varier les plaisirs : on va se doter d'un tableau de booléens, comptant autant d'emplacements qu'il y a de lettres dans le mot à deviner. Chaque emplacement du tableau correspondra à une lettre du mot à trouver, et indiquera par sa valeur si la lettre a été découverte ou non (faux, la lettre n'a pas été devinée, vrai, elle l'a été). La correspondance entre les éléments du tableau et le mot à deviner étant immédiate, la programmation de nos boucles en sera facilitée. Nous baptiserons notre tableau de booléens du joli nom de « verif ».
- Enfin, l'ensemble des lettres proposées sera stockée sans soucis dans une chaîne de caractères nommée Propos.

Nous avons maintenant suffisamment gambergé pour dresser le tableau final de cette étape, à savoir le dictionnaire des données proprement dit :

.

Nom	**Type**	**Description**
Dico.txt	Fichier texte	Liste des mots à deviner
Mot	Caractère	Mot à deviner
Lettre	Caractère	Lettre proposée
MovRep	Entier	Nombre de mauvaises réponses
Verif()	Tableau de Booléens	Lettres précédemment devinées, en correspondance avec Mot
Propos	Caractère	Liste des lettres proposées

Etape 2 : l'algorithme fonctionnel

On peut à présent passer à la réalisation de l'algorithme fonctionnel, c'est-à-dire au découpage de notre problème en blocs logiques. Le but de la manœuvre est multiple :

- faciliter la réalisation de l'algorithme définitif en le tronçonnant en plus petits morceaux.

- Gagner du temps et de la légèreté en isolant au mieux les sous-procédures et fonctions qui méritent de l'être. Eviter ainsi éventuellement des répétitions multiples de code au cours du programme, répétitions qui ne diffèrent les unes des autres qu'à quelques variantes près.
- Permettre une division du travail entre programmeurs, chacun se voyant assigner la programmation de sous-procédures ou de fonctions spécifiques (cet aspect est essentiel dès qu'on quitte le bricolage personnel pour entrer dans le monde de la programmation professionnelle, donc collective).

Dans notre cas précis, un premier bloc se détache : il s'agit de ce qu'on pourrait appeler les préparatifs du jeu (choix du mot à deviner). Puisque le but est de renvoyer une valeur et une seule (le mot choisi par la machine), nous pouvons confier cette tâche à une fonction spécialisée ChoixDuMot (à noter que ce découpage est un choix de lisibilité, et pas une nécessité absolue ; on pourrait tout aussi bien faire cela dans la procédure principale).
Cette procédure principale, justement, va ensuite avoir nécessairement la forme d'une boucle Tantque : en effet , tant que la partie n'est pas finie, on recommence la série des traitements qui représentent un tour de jeu. Mais comment, justement, savoir si la partie est finie ? Elle peut se terminer soit parce que le nombre de mauvaises réponses a atteint 10, soit parce que toutes les lettres du mot ont été trouvées. Le mieux sera donc de confier l'examen de tout cela à une fonction spécialisée, PartieFinie, qui renverra une valeur numérique (0 pour signifier que la partie est en cours, 1 en cas de victoire, 2 en cas de défaite).
Passons maintenant au tour de jeu.
La première chose à faire, c'est d'afficher à l'écran l'état actuel du mot à deviner : un mélange de lettres en clair (celles qui ont été trouvées) et de tirets (correspondant aux lettres non encore trouvées). Tout ceci pourra être pris en charge par une sous-procédure spécialisée, appelée AffichageMot. Quant à l'initialisation des différentes variables, elle pourra être placée, de manière classique, dans la procédure principale elle-même.
Ensuite, on doit procéder à la saisie de la lettre proposée, en veillant à effectuer les contrôles de saisie adéquats. Là encore, une fonction spécialisée, SaisieLettre, sera toute indiquée.
Une fois la proposition faite, il convient de vérifier si elle correspond ou non à une lettre à deviner, et à en tirer les conséquences. Ceci sera fait par une sous-procédure appelée VérifLettre.
Enfin, une fois la partie terminée, on doit afficher les conclusions à l'écran ; on déclare à cet effet une dernière procédure, FinDePartie.
Nous pouvons, dans un algorithme fonctionnel complet, dresser un tableau des différentes procédures et fonctions, exactement comme nous l'avons fait juste avant pour les données (on s'épargnera cette peine dans le cas présent, ce que nous avons écrit ci-dessus suffisant amplement. Mais dans le cas d'une grosse application, un tel travail serait nécessaire et nous épargnerait bien des soucis).
On peut aussi schématiser le fonctionnement de notre application sous forme de blocs, chacun des blocs représentant une fonction ou une sous-procédure :
A ce stade, l'analyse dite fonctionnelle est terminée. Les fondations (solides, espérons-le) sont posées pour finaliser l'application.

Etape 3 : Algorithmes détaillés
Normalement, il ne nous reste plus qu'à traiter chaque procédure isolément. On commencera par les sous-procédures et fonctions, pour terminer par la rédaction de la procédure principale.

4. Exercices

Enonce des Exercices	Corrigés des Exercices
Exercice 11.1 Écrivez une fonction qui renvoie la somme de cinq nombres fournis en argument.	Exercice 11.1 Voilà un début en douceur... **Fonction** Sum(a, b, c, d, e) **Renvoyer** a + b + c + d + e **FinFonction**

Exercice 11.2
 	
Écrivez une fonction qui renvoie le nombre de voyelles contenues dans une chaîne de caractères passée en argument. Au passage, notez qu'une fonction a tout à fait le droit d'appeler une autre fonction.
	Exercice 11.2
	Fonction NbVoyelles(Mot en Caractère)
	Variables i, nb **en Numérique**
	Pour i ← 1 à Len(Mot)
	  Si Trouve("aeiouy", Mid(Mot, i, 1)) <> 0 **Alors**
	  nb ← nb + 1
	  FinSi
	i **suivant**
	Renvoyer nb
	FinFonction

Exercice 11.3
 	
Réécrivez la fonction Trouve, vue précédemment, à l'aide des fonctions Mid et Len (comme quoi, Trouve, à la différence de Mid et Len, n'est pas une fonction indispensable dans un langage).
	Exercice 11.3
	Fonction Trouve(a, b)
	Variable i **en Numérique**
	Début
	i ← 1
	TantQue i < Len(a) - Len(b) et b <> Mid(a, i, Len(b))
	  i ← i + 1
	FinTantQue
	Si b <> Mid(a, i, Len(b)) **Alors**
	  Renvoyer 0
	Sinon
	  Renvoyer i
	FinFonction

Exemple des Fonctions

Fonction ChoixDuMot
Quelques explications : on lit intégralement le fichier contenant la liste des mots. Au fur et à mesure, on range ces mots dans le tableau Liste, qui est redimensionné à chaque tour de boucle. Un tirage aléatoire intervient alors, qui permet de renvoyer un des mots au hasard.

```
Fonction ChoixDuMot()
Tableau Liste() en Caractère
Variables Nbmots, Choisi en Numérique
Ouvrir "Dico.txt" sur 1 en Lecture
Nbmots ← -1
Tantque Non EOF(1)
  Nbmots ← Nbmots + 1
  Redim Liste(Nbmots)
  LireFichier 1, Liste(Nbmots)
FinTantQue
Fermer 1
```

```
Choisi ← Ent(Alea() * Nbmots)
Renvoyer Liste(Choisi)
FinFonction
```

Fonction PartieFinie

On commence par vérifier le nombre de mauvaises réponses, motif de défaite. Ensuite, on regarde si la partie est gagnée, traitement qui s'apparente à une gestion de Flag : il suffit que l'une des lettres du mot à deviner n'ait pas été trouvée pour que la partie ne soit pas gagnée. La fonction aura besoin, comme arguments, du tableau Verif, de son nombre d'éléments et du nombre actuel de mauvaises réponses.

```
Fonction PartieFinie(t() en Booleen, n, x en Numérique)
Variables i, issue en Numerique
Si x = 10 Alors
  Renvoyer 2
Sinon
  Issue ← 1
  Pour i ← 0 à n
    Si Non t(i) Alors
      Issue ← 0
    FinSi
  i suivant
  Renvoyer Issue
FinSi
FinFonction
```

Procédure AffichageMot

Une même boucle nous permet de considérer une par une les lettres du mot à trouver (variable m), et de savoir si ces lettres ont été identifiées ou non.

```
Procédure AffichageMot(m en Caractère par Valeur, t() en Booléen par Valeur)
Variable Aff en Caractere
Variable i en Numerique
Aff ← ""
Pour i ← 0 à len(m) - 1
  Si Non t(i) Alors
    Aff ← Aff & "-"
  Sinon
    Aff ← Aff & Mid(mot, i + 1, 1)
  FinSi
i suivant
Ecrire Aff
FinProcédure
```

Remarque : cette procédure aurait également pu être écrite sous la forme d'une fonction, qui aurait renvoyé vers la procédure principale la chaîne de caractères Aff. L'écriture à l'écran de cette chaîne Aff aurait alors été faite par la procédure principale.

Voilà donc une situation où on peut assez indifféremment opter pour une sous-procédure ou pour une fonction.

Procédure SaisieLettre

On vérifie que le signe entré (paramètre b) est bien une seule lettre, qui ne figure pas dans les

propositions précédemment effectuées (paramètre a)
Procédure SaisieLettre(a, b **en Caractère par Référence**)
Variable Correct **en Booleen**
Variable Alpha **en Caractere**
Début
Correct ← Faux
Alpha ← "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
TantQue Non Correct
  Ecrire "Entrez la lettre proposée : "
  Lire b
  Si Trouve(alpha, b) = 0 Ou len(b) <> 1 **Alors**
    Ecrire "Ce n'est pas une lettre !"
  SinonSi Trouve(a, b) <> 0 **Alors**
    Ecrire "Lettre déjà proposée !"
  Sinon
    Correct ← Vrai
    a ← a & b
  FinSi
FinTantQue
Fin Procédure

Procédure VerifLettre
Les paramètres se multiplient… L est la lettre proposée, t() le tableau de booléens, M le mot à trouver et N le nombre de mauvaises propositions. Il n'y a pas de difficulté majeure dans cette procédure : on examine les lettres de M une à une, et on en tire les conséquences. Le flag sert à savoir si la lettre proposée faisait ou non partie du mot à deviner.
Procédure VerifLettre(L, M **en Caractère par Valeur**, t() **en Booléen par Référence**, N **en Numérique par Référence**)
Variable Correct **en Booleen**
Début
Correct ← Faux
Pour i ← 1 à Len(M)
  Si Mid(M, i, 1) = L **Alors**
    Correct ← Vrai
    T(i - 1) ← Vrai
  FinSi
FinTantQue
Si Non Correct **Alors**
  N ← N + 1
FinSi
Fin Procédure

Procédure Epilogue
Procédure Epilogue(M **en Caractère par Valeur**, N **en Numérique par Valeur**)
Début
Si N = 2 **Alors**
  Ecrire "Une mauvaise proposition de trop… Partie terminée !"
  Ecrire "Le mot à deviner était : ", M
Sinon

```
  Ecrire "Bravo ! Vous avez trouvé !"
FinSi
Fin Procédure
```

Procédure Principale

```
Procédure Principale
Variables Lettre, Mot, Propos en Caractere
Variables g i, MovRep en Numérique
Tableau Verif() en Booleen
Début
Mot ← ChoixDuMot()
Propos ← ""
Lettre ← ""
Redim Verif(Len(Mot)-1)
Pour i ← 0 à Len(Mot)-1
  Verif(i) ← Faux
i suivant
k ← 0
Tantque k = 0
  AffichageMot(Mot, Verif())
  SaisieLettre(Propos, Lettre)
  VerifLettre(Lettre, Mot, Verif(), MovRep)
  k ← PartieFinie(Verif(), len(mot), MovRep)
FinTantQue
Epilogue(Mot, k)
Fin
```