Informatique scientifique

O. de Viron

2006-2007

Table des matières

Chapitre 1

Préambule

Un ordinateur fait au bas mot 1 million d'opérations à la seconde, mais il a que ça à penser, aussi.

Jean-Marie Gourio

Les ordinateurs sont comme les dieux de l'Ancien Testament : avec beaucoup de règles, et sans pitié.

Joseph Campbell

Il y a 10 sortes de gens au monde : ceux qui connaissent le binaire et les autres.

Anonyme

1.1 Programmation et algorithme

Dans ce qui sera, pour quelques-uns d'entre vous, une toute collision avec la programmation, nous allons, essentiellement, nous confronter avec la notion d'algorithme. Imaginez que vous vous rendez à la SNCF pour demander comment aller à Pau, vous allez recevoir des instructions simples et non équivoques, par exemple prendre le train à la Gare de l'Est vers Strasbourg à 13h40, voie 10, puis prendre la correspondance pour Pau à 18h10, voie 3.

Essentiellement, ce faisant, vous exécutez un programme. Au contraire, si l'on vous avait dit *prenez un train à la gare, et descendez quand il s'arrête*, il y a fort à parier que vous ne seriez jamais arrivé à Pau, ou alors vous êtes allé consulter un horaire..

La programmation sur ordinateur, c'est presque pareil. Il faut donner à l'ordinateur des instructions simples et précise pour lui dire ce qu'il doit faire.

A ce propos, j'ai une bonne et une mauvaise nouvelle. Commençons par la bonne : L'ordinateur est GENTIL. Il fait exactement ce qu'on lui dit, sans râler, sans rouspéter, sans se poser de question. La mauvaise, c'est que l'ordinateur est gentil, il fait EXACTEMENT ce qu'on lui dit, sans râler, sans rouspéter, et SANS SE POSER DE QUESTION. Et comme il est trop bête pour se tromper, en général, ça plante. Supposons qu'on lui ait donner les instructions que la SNCF vous aurait données : prendre le train à la Gare de l'Est vers Strasbourg à 13h40, voie 10, puis prendre la correspondance pour Pau

à 18h10, voie 3. Il ne serait jamais arrivé à Pau, puisqu'on ne lui a pas dit de descendre du train à Strasbourg.

En un mot comme en cent, l'ordinateur à l'intelligence d'une amibe (et encore). Lui parler est donc un défi permanent. C'est la joie de la programmation.

Le programme s'appelle souvent algorithme, parce que ça fait plus savant, et la programmation algorithmique.

Je terminerai cette introduction en avouant que j'ai pompé honteusement l'excellent cours *L'algorithmique pour les non-matheux* de Christophe Darmangeat. Toutefois, les erreurs qui restent sont miennes. L'humour déplorable est parfois le sien, mais j'assume, si je l'ai gardé, c'est que cela m'a fait rire...

1.2 En plus, l'ordinateur ne parle même pas français

C'est bien connu, les ordinateurs sont comme le gros rock qui tâche : ils sont binaires. Mais ce qui est moins connu, c'est ce que ce qualificatif de « binaire »recouvre exactement, et ce qu'il implique. Aussi, avant de nous plonger dans les arcanes de l'algorithmique proprement dite, ferons-nous un détour par la notion de codage binaire. Contrairement aux apparences, nous ne sommes pas éloignés de notre sujet principal. Tout au contraire, ce que nous allons voir à présent constitue un ensemble de notions indispensables à l'écriture de programmes. Car pour parler à une machine, mieux vaut connaître son vocabulaire...

1.2.1 Pourquoi les ordinateurs sont-ils « binaires » ?

De nos jours, les ordinateurs sont ces machines merveilleuses capables de traiter du texte, d'afficher des tableaux de maître, de jouer de la musique ou de projeter des vidéos. On n'en est pas encore tout à fait à HAL, l'ordinateur de 2001 Odyssée de l'Espace, à « l'intelligence »si développée qu'il a peur de mourir... pardon, d'être débranché. Mais l'ordinateur paraît être une machine capable de tout faire. Pourtant, les ordinateurs ont beau sembler repousser toujours plus loin les limites de leur champ d'action, il ne faut pas oublier qu'en réalité, ces fiers-à-bras ne sont toujours capables que d'une seule chose : faire des calculs, et uniquement cela. Et encore, il ne faut pas croire qu'il comprend ce qu'il fait.

Lorsqu'un ordinateur traite du texte, du son, de l'image, de la vidéo, il traite en réalité des nombres. En fait, dire cela, c'est déjà lui faire trop d'honneur. D'une part, parce que l'ordinateur n'a pas vraiment d'idée de ce que c'est que traiter, ni même de nombre. En outre, même le simple nombre « 3 »reste hors de portée de l'intelligence d'un ordinateur, ce qui le situe largement en dessous de l'attachant chimpanzé Bonobo, qui sait, entre autres choses, faire des blagues à ses congénères et jouer au Pac-Man. Un ordinateur manipule exclusivement des informations binaires, dont on ne peut même pas dire sans être tendancieux qu'il s'agit de nombres.

Mais qu'est-ce qu'une information binaire ? C'est une information qui ne peut avoir que deux états : par exemple, ouvert/fermé, libre/occupé, militaire/civil, assis/couché, blanc/noir, vrai/faux, etc. Si l'on pense à des dispositifs physiques permettant de stocker ce genre d'information, on pourrait citer : chargé/non chargé, haut/bas, troué/non troué.

Je ne donne pas derniers exemples au hasard : ce sont précisément ceux dont se sert un ordinateur pour stocker l'ensemble des informations qu'il va devoir manipuler. En deux mots, la mémoire vive (la « RAM ») est formée de millions de composants électroniques qui peuvent retenir ou relâcher une charge électrique. La surface d'un disque dur, d'une bande ou d'une disquette est recouverte de particules métalliques qui peuvent, grâce à un aimant, être orientées dans un sens ou dans l'autre. Et sur un CD-ROM, on trouve un long sillon étroit irrégulièrement percé de trous.

Toutefois, la coutume veut qu'on symbolise une information binaire, quel que soit son support physique, sous la forme de 1 et de 0. Il faut bien comprendre que ce n'est là qu'une représentation, une image commode, que l'on utilise pour parler de toute information binaire. Dans la réalité physique, il n'y a pas plus de 1 et de 0 qui se promènent dans les ordinateurs qu'il n'y a écrit, en lettres géantes, « Océan Atlantique »sur la mer quelque part entre la Bretagne et les Antilles. Le 1 et le 0 dont parlent les informaticiens sont des signes, ni plus, ni moins, pour désigner une information, indépendamment de son support physique.

Les informaticiens seraient-ils des gens tordus, possédant un goût immodéré pour l'abstraction, ou pour les jeux intellectuels alambiqués ? Non, pas davantage en tout cas que le reste de leurs contemporains non-informaticiens. En fait, chacun d'entre nous pratique ce genre d'abstraction tous les jours, sans pour autant trouver cela bizarre ou difficile. Simplement, nous le faisons dans la vie quotidienne sans y penser. Et à force de ne pas y penser, nous ne remarquons même plus quel mécanisme subtil d'abstraction est nécessaire pour pratiquer ce sport. Par exemple, le feu de signalisation a deux états vert et rouge. Le vert veut dire passer, le feu rouge, ne pas passer[1]. De même,lorsque nous disons que 4+3=7 (ce qui reste, normalement, dans le domaine de compétence mathématique de tous ceux qui lisent ce cours!), nous manions de pures abstractions, représentées par de non moins purs symboles! Un être humain d'il y a quelques millénaires se serait demandé longtemps qu'est-ce que c'est que « quatre »ou « trois », sans savoir quatre ou trois « quoi ? ». Mine de rien, le fait même de concevoir des nombres, c'est-à-dire de pouvoir considérer, dans un ensemble, la quantité indépendamment de tout le reste, c'est déjà une abstraction très hardie, qui a mis très longtemps avant de s'imposer à tous comme une évidence. Et le fait de faire des additions sans devoir préciser des additions « de quoi ? », est un pas supplémentaire qui a été encore plus difficile à franchir. Le concept de nombre, de quantité pure, a donc constitué un immense progrès (que les ordinateurs n'ont quant à eux, je le répète, toujours pas accompli). Mais si concevoir les nombres, c'est bien, posséder un système de notation performant de ces nombres, c'est encore mieux. Et là aussi, l'humanité a mis un certain temps (et essayé un certain nombre de pistes qui se sont révélées être des impasses) avant de parvenir au système actuel, le plus rationnel. Ceux qui ne sont pas convaincus des progrès réalisés en ce domaine peuvent toujours essayer de résoudre une multiplication comme 587 x 644 en chiffres romains, on leur souhaite bon courage!

Donc, on peut conclure que les ordinateurs sont binaires (1) parce que cela fonctionne, (2) parce que cela se stocke plus facilement.

1.2.2 La numérotation de position en base décimale

L'humanité actuelle, pour représenter n'importe quel nombre, utilise un système de numérotation de position, à base décimale. Qu'est-ce qui se cache derrière cet obscur jargon ? Commençons par la numérotation de position. Pour représenter un nombre, aussi grand soit-il, nous disposons d'un alphabet spécialisé : une série de 10 signes qui s'appellent les chiffres. Et lorsque nous écrivons un nombre en mettant certains de ces chiffres les uns derrière les autres, l'ordre dans lequel nous mettons les chiffres est capital. Ainsi, par exemple, 2 569 n'est pas du tout le même nombre que 9 562. Et pourquoi ? Quel opération, quel décodage mental effectuons-nous lorsque nous lisons une suite de chiffres représentant un nombre ? Le problème, c'est que nous sommes tellement habitués à faire ce décodage de façon instinctive que généralement nous n'en connaissons plus les règles. Mais ce n'est pas très compliqué de les reconstituer... Et c'est là que nous mettons le doigt en plein dans la deuxième caractéristique de notre système de notation numérique : son caractère décimal. Lorsque j'écris 9562, de quel nombre est-ce que je parle ? Décomposons la lecture chiffre par chiffre, de gauche à droite : 9562, c'est 9000 + 500 + 60 + 2. Allons plus loin, même si cela paraît un peu bébête :

- 9000, c'est 9 x 1000, parce que le 9 est le quatrième chiffre en partant de la droite
- 500, c'est 5 x 100, parce que le 5 est le troisième chiffre en partant de la droite
- 60, c'est 6 x 10, parce que le 6 est le deuxième chiffre en partant de la droite
- 2, c'est 2 x 1, parce que le 5 est le premier chiffre en partant de la droite

On peut encore écrire ce même nombre d'une manière légèrement différente. Au lieu de :
9 562 = 9 x 1 000 + 5 x 100 + 6 x 10 + 2,
On écrit que :
9 562 = (9 x 10 x 10 x 10) + (5 x 10 x 10) + (6 x 10) + (2)
Arrivés à ce stade de la compétition, je prie les allergiques de m'excuser, mais il nous faut employer un petit peu de jargon mathématique. Ce n'est pas grand-chose, et on touche au but. Alors, courage! En fait, ce jargon se résume au fait que les matheux notent la ligne ci-dessus à l'aide du symbole de « puissance ». Cela donne :
$9\ 562 = 9x10^3 + 5x10^2 + 6x10^1 + 2x10^0$ Et voilà, nous y sommes. Nous avons dégagé le mécanisme général de la représentation par numérotation de position en base décimale.

[1] Il semblerait, à ce propos, que les gens qui habitent Paris n'ont pas entièrement capté cette dernière information.

Alors, nous en savons assez pour conclure sur les conséquences du choix de la base décimale. Il y en a deux, qui n'en forment en fin de compte qu'une seule :

- parce que nous sommes en base décimale, nous utilisons un alphabet numérique de dix symboles. Nous nous servons de dix chiffres, pas un de plus, pas un de moins.
- toujours parce nous sommes en base décimale, la position d'un de ces dix chiffres dans un nombre désigne la puissance de dix par laquelle ce chiffre doit être multiplié pour reconstituer le nombre. Si je trouve un 7 en cinquième position à partir de la droite, ce 7 ne représente pas 7 mais 7 fois 104, soit 70 000.

Un dernier mot concernant le choix de la base dix. Pourquoi celle-là et pas une autre? Après tout, la base dix n'était pas le seul choix possible. Les babyloniens, qui furent de brillants mathématiciens, avaient en leur temps adopté la base 60 (dite sexagésimale). Cette base 60 impliquait certes d'utiliser un assez lourd alphabet numérique de 60 chiffres. Mais c'était somme toute un inconvénient mineur, et en retour, elle possédait certains avantages non négligeables. 60 étant un nombre divisible par beaucoup d'autres (c'est pour cette raison qu'il avait été choisi), on pouvait, rien qu'en regardant le dernier chiffre, savoir si un nombre était divisible par 2, 3, 4, 5, 6, 10, 12, 15, 20 et 30. Alors qu'en base 10, nous ne pouvons immédiatement répondre à la même question que pour les diviseurs 2 et 5. La base sexagésimale a certes disparu en tant que système de notation des nombres. Mais Babylone nous a laissé en héritage sa base sexagésimale dans la division du cercle en soixante parties (pour compter le temps en minutes et secondes), et celle en 6 x 60 parties (pour les degrés de la géométrie et de l'astronomie). Alors, pourquoi avons-nous adopté la base décimale, moins pratique à bien des égards? Nul doute que cela tienne au dispositif matériel grâce auquel tout être humain normalement constitué stocke spontanément une information numérique : ses doigts!

1.2.3 La numérotation de position en base binaire

Les ordinateurs, eux, comme on l'a vu, ont un dispositif physique fait pour stocker (de multiples façons) des informations binaires. Alors, lorsqu'on représente une information stockée par un ordinateur, le plus simple est d'utiliser un système de représentation à deux chiffres : les fameux 0 et 1. Mais une fois de plus, je me permets d'insister, le choix du 0 et du 1 est une pure convention, et on aurait pu choisir n'importe quelle autre paire de symboles à leur place.

Dans un ordinateur, le dispositif qui permet de stocker de l'information est donc rudimentaire, bien plus rudimentaire que les mains humaines. Avec des mains humaines, on peut coder dix choses différentes (en fait bien plus, si l'on fait des acrobaties avec ses doigts, mais écartons ce cas). Avec un emplacement d'information d'ordinateur, on est limité à deux choses différentes seulement. Avec une telle information binaire, on ne va pas loin. Voilà pourquoi, dès leur invention, les ordinateurs ont été conçus pour manier ces informations par paquets de 0 et de 1. Et la taille de ces paquets a été fixée à 8 informations binaires. Une information binaire (symbolisée couramment par 0 ou 1) s'appelle un bit (en anglais... bit). Un groupe de huit bits s'appelle un octet (en anglais, byte) Donc, méfiance avec le byte (en abrégé, B majuscule), qui vaut un octet, c'est-à-dire huit bits (en abrégé, b minsucule). Dans combien d'états différents un octet peut-il se trouver? Le calcul est assez facile (mais il faut néanmoins savoir le refaire). Chaque bit de l'octet peut occuper deux états. Il y a donc dans un octet : 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 = 28 = 256 possibilités Cela signifie qu'un octet peut servir à coder 256 nombres différents : ce peut être la série des nombres entiers de 1 à 256, ou de 0 à 255, ou de Ű127 à +128. C'est une pure affaire de convention, de choix de codage. Mais ce qui n'est pas affaire de choix, c'est le nombre de possibilités : elles sont 256, pas une de plus, pas une de moins, à cause de ce qu'est, par définition, un octet.

Si l'on veut coder des nombres plus grands que 256, ou des nombres négatifs, ou des nombres décimaux, on va donc être contraint de mobiliser plus d'un octet. Ce n'est pas un problème, et c'est très souvent que les ordinateurs procèdent ainsi. En effet, avec deux octets, on a 256 x 256 = 65 536 possibilités. En utilisant trois octets, on passe à 256 x 256 x 256 = 16 777 216 possibilités. Et ainsi de suite, je ne m'attarderai pas davantage sur les différentes manières de coder les nombres avec des octets. On abordera de nouveau brièvement le sujet un peu plus loin. Cela implique également qu'un octet peut servir à coder autre chose qu'un nombre : l'octet est très souvent employé pour coder du texte. Il y a 26 lettres dans l'alphabet. Même en comptant différemment les minuscules et les majuscules, et même en y ajoutant les chiffres et les signes de ponctuation, on arrive à un total inférieur à 256. Cela veut dire que

pour coder convenablement un texte, le choix d'un caractère par octet est un choix pertinent. Se pose alors le problème de savoir quel caractère doit être représenté par quel état de l'octet. Si ce choix était librement laissé à chaque informaticien, ou à chaque fabricant d'ordinateur, la communication entre deux ordinateurs serait un véritable casse-tête. L'octet 10001001 serait par exemple traduit par une machine comme un T majuscule, et par une autre comme une parenthèse fermante ! Aussi, il existe un standard international de codage des caractères et des signes de ponctuation. Ce standard stipule quel état de l'octet correspond à quel signe du clavier. Il s'appelle l'ASCII (pour American Standard Code for Information Interchange). Et fort heureusement, l'ASCII est un standard universellement reconnu et appliqué par les fabricants d'ordinateurs et de logiciels. Bien sûr, se pose le problème des signes propres à telle ou telle langue (comme les lettres accentuées en français, par exemple). L'ASCII a paré le problème en réservant certains codes d'octets pour ces caractères spéciaux à chaque langue. En ce qui concerne les langues utilisant un alphabet non latin, un standard particulier de codage a été mis au point. Quant aux langues non alphabétiques (comme le chinois), elles payent un lourd tribut à l'informatique pour n'avoir pas su évoluer vers le système alphabétique... Revenons-en au codage des nombres sur un octet. Nous avons vu qu'un octet pouvait coder 256 nombres différents, par exemple (c'est le choix le plus spontané) la série des entiers de 0 à 255. Comment faire pour, à partir d'un octet, reconstituer le nombre dans la base décimale qui nous est plus familière ? Ce n'est pas sorcier ; il suffit d'appliquer, si on les a bien compris, les principes de la numérotation de position, en gardant à l'esprit que là, la base n'est pas décimale, mais binaire. Prenons un octet au hasard : 1 1 0 1 0 0 1 1 D'après les principes vus plus haut, ce nombre représente en base dix, en partant de la gauche :

$$\begin{aligned} & 1\times 2^7+1\times 2^6+0\times 2^5+1\times 2^4+0\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0 \\ = \; & 1\times 128+1\times 64+1\times 16+1\times 2+1\times 1 \\ = \; & 128+64+16+2+1=211 \end{aligned}$$

Et voilà ! Ce n'est pas plus compliqué que cela ! Inversement, comment traduire un nombre décimal en codage binaire ? Il suffit de rechercher dans notre nombre les puissances successives de deux. Prenons, par exemple, 186.

- Dans 186, on trouve 1 x 128, soit 1 x 27. Je retranche 128 de 186 et j'obtiens 58.
- Dans 58, on trouve 0 x 64, soit 0 x 26. Je ne retranche donc rien.
- Dans 58, on trouve 1 x 32, soit 1 x 25. Je retranche 32 de 58 et j'obtiens 26.
- Dans 26, on trouve 1 x 16, soit 1 x 24. Je retranche 16 de 26 et j'obtiens 10.
- Dans 10, on trouve 1 x 8, soit 1 x 23. Je retranche 8 de 10 et j'obtiens 2.
- Dans 2, on trouve 0 x 4, soit 0 x 22. Je ne retranche donc rien.
- Dans 2, on trouve 1 x 2, soit 1 x 21. Je retranche 2 de 2 et j'obtiens 0.
- Dans 0, on trouve 0 x 1, soit 0 x 20. Je ne retranche donc rien.

Il ne me reste plus qu'à reporter ces différents résultats (dans l'ordre !) pour reconstituer l'octet. J'écris alors qu'en binaire, 186 est représenté par : 1 0 1 1 1 0 1 0 C'est bon ? Alors on passe à la suite.

Chapitre 2

Introduction à l'algorithmique

> Un langage de programmation est une convention pour donner des ordres à un ordinateur. Ce n'est pas censé être obscur, bizarre et plein de pièges subtils. Ça, ce sont les caractéristiques de la magie.
>
> Dave Small

L'algorithmique est un terme d'origine arabe, comme algèbre, amiral ou zénith. Ce n'est pas une excuse pour massacrer son orthographe, ou sa prononciation. Ainsi, l'algo n'est pas « rythmique », à la différence du bon rock'n roll. L'algo n'est pas non plus « l'agglo ». Alors, ne confondez pas l'algorithmique avec l'agglo rythmique, qui consiste à poser des parpaings en cadence.

2.1 Qu'est-ce que l'algomachin ?

Avez-vous déjà ouvert un livre de recettes de cuisine ? Avez vous déjà déchiffré un mode d'emploi traduit directement du coréen pour faire fonctionner un magnétoscope ou un répondeur téléphonique réticent ? Si oui, sans le savoir, vous avez déjà exécuté des algorithmes.

Plus fort : avez-vous déjà indiqué un chemin à un touriste égaré ? Avez vous fait chercher un objet à quelqu'un par téléphone ? Ecrit une lettre anonyme stipulant comment procéder à une remise de rançon ? Si oui, vous avez déjà fabriqué Ũ et fait exécuter Ũ des algorithmes.

Comme quoi, l'algorithmique n'est pas un savoir ésotérique réservé à quelques rares initiés touchés par la grâce divine, mais une aptitude partagée par la totalité de l'humanité. Donc, pas d'excuses...

Un algorithme, c'est une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné. Si l'algorithme est juste, le résultat est le résultat voulu, et le touriste se retrouve là où il voulait aller. Si l'algorithme est faux, le résultat est, disons, aléatoire, et décidément, cette saloperie de répondeur ne veut rien savoir.

Complétons toutefois cette définition. Après tout, en effet, si l'algorithme, comme on vient de le dire, n'est qu'une suite d'instructions menant celui qui l'exécute à résoudre un problème, pourquoi ne pas donner comme instruction unique : « résous le problème », et laisser l'interlocuteur se débrouiller avec ça ? A ce tarif, n'importe qui serait champion d'algorithmique sans faire aucun effort. Pas de ça Lisette, ce serait trop facile.

Le malheur (ou le bonheur, tout dépend du point de vue) est que justement, si le touriste vous demande son chemin, c'est qu'il ne le connaît pas. Donc, si on n'est pas un goujat intégral, il ne sert à rien de lui dire de le trouver tout seul. De même les modes d'emploi contiennent généralement (mais pas toujours) un peu plus d'informations que « débrouillez vous pour que ça marche ».

Pour fonctionner, un algorithme doit donc contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter. C'est d'ailleurs l'un des points délicats pour les rédacteurs de modes d'emploi :

les références culturelles, ou lexicales, des utilisateurs, étant variables, un même mode d'emploi peut être très clair pour certains et parfaitement abscons pour d'autres. En informatique, heureusement, il n'y a pas ce problème : les choses auxquelles ont doit donner des instructions sont les ordinateurs, et ceux-ci ont le bon goût d'être tous strictement aussi idiots les uns que les autres.

2.2 Faut-il être matheux pour être bon en algorithmique ?

Je consacre quelques lignes à cette question, car cette opinion aussi fortement affirmée que faiblement fondée sert régulièrement d'excuse : « moi, de toute façon, je suis mauvais(e) en algo, j'ai jamais rien pigé aux maths ». Faut-il être « bon en maths »pour expliquer correctement son chemin à quelqu'un ? Je vous laisse juge.

La maîtrise de l'algorithmique requiert deux qualités, très complémentaires d'ailleurs : il faut avoir une certaine intuition, car aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu. C'est là, si l'on y tient, qu'intervient la forme « d'intelligence »requise pour l'algorithmique. Alors, c'est certain, il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant, et j'insiste sur ce point, les réflexes, cela s'acquiert. Et ce qu'on appelle l'intuition n'est finalement que de l'expérience tellement répétée que le raisonnement, au départ laborieux, finit par devenir « spontané ».

il faut être méthodique et rigoureux, et c'est probablement la raison de cette idée que les mathématiques sont un préalable. Chaque fois qu'on écrit une série d'instructions qu'on croit justes, il faut systématiquement se mettre mentalement à la place de la machine qui va les exécuter, armé d'un papier et d'un crayon, afin de vérifier si le résultat obtenu est bien celui que l'on voulait. Cette opération ne requiert pas la moindre once d'intelligence. Mais elle reste néanmoins indispensable, si l'on ne veut pas écrire à l'aveuglette.

Et petit à petit, à force de pratique, vous verrez que vous pourrez faire de plus en plus souvent l'économie de cette dernière étape : l'expérience fera que vous « verrez »le résultat produit par vos instructions, au fur et à mesure que vous les écrirez. Naturellement, cet apprentissage est long, et demande des heures de travail patient. Aussi, dans un premier temps, évitez de sauter les étapes : la vérification méthodique, pas à pas, de chacun de vos algorithmes représente plus de la moitié du travail à accomplir... et le gage de vos progrès.

2.3 L'ADN, les Shadoks, et les ordinateurs

Quel rapport me direz-vous? Eh bien le point commun est : quatre mots de vocabulaire.

L'univers lexical Shadok, c'est bien connu, se limite aux termes « Ga », « Bu », « Zo », et « Meu ». Ce qui leur a tout de même permis de formuler quelques fortes maximes, telles que : « Mieux vaut pomper et qu'il ne se passe rien, plutôt qu'arrêter de pomper et risquer qu'il se passe quelque chose de pire »(pour d'autres fortes maximes Shadok, n'hésitez pas à visiter leur site Internet, il y en a toute une collection qui vaut le détour).

L'ADN, qui est en quelque sorte le programme génétique, l'algorithme à la base de construction des êtres vivants, est une chaîne construite à partir de quatre éléments invariables. Ce n'est que le nombre de ces éléments, ainsi que l'ordre dans lequel ils sont arrangés, qui vont déterminer si on obtient une puce ou un éléphant. Et tous autant que nous sommes, splendides réussites de la Nature, avons été construits par un « programme »constitué uniquement de ces quatre briques, ce qui devrait nous inciter à la modestie.

Enfin, les ordinateurs, quels qu'ils soient, ne sont fondamentalement capables de comprendre que quatre catégories d'ordres (en programmation, on n'emploiera pas le terme d'ordre, mais plutôt celui d'instructions).

Ces quatre familles d'instructions sont :

1. l'affectation de variables
2. la lecture / écriture
3. les tests
4. les boucles

Un algorithme informatique se ramène donc toujours au bout du compte à la combinaison de ces quatre petites briques de base. Il peut y en avoir quelques unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes. Rassurez-vous, dans le cadre de ce cours, nous n'irons pas jusque là (cependant, la taille d'un algorithme ne conditionne pas en soi sa complexité : de longs algorithmes peuvent être finalement assez simples, et de petits très compliqués).

2.4 Algorithmique et programmation

Pourquoi apprendre l'algorithmique pour apprendre à programmer ? En quoi a-t-on besoin d'un langage spécial, distinct des langages de programmation compréhensibles par les ordinateurs ? Parce que l'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage. Pour prendre une image, si un programme était une dissertation, l'algorithmique serait le plan, une fois mis de côté la rédaction et l'orthographe. Or, vous savez qu'il vaut mieux faire d'abord le plan et rédiger ensuite que l'inverse... Apprendre l'algorithmique, c'est apprendre à manier la structure logique d'un programme informatique. Cette dimension est présente quelle que soit le langage de programmation ; mais lorsqu'on programme dans un langage (en C, en Visual Basic, etc.) on doit en plus se colleter les problèmes de syntaxe, ou de types d'instructions, propres à ce langage. Apprendre l'algorithmique de manière séparée, c'est donc sérier les difficultés pour mieux les vaincre. A cela, il faut ajouter que des générations de programmeurs, souvent autodidactes (mais pas toujours, hélas !), ayant directement appris à programmer dans tel ou tel langage, ne font pas mentalement clairement la différence entre ce qui relève de la structure logique générale de toute programmation (les règles fondamentales de l'algorithmique) et ce qui relève du langage particulier qu'ils ont appris. Ces programmeurs, non seulement ont beaucoup plus de mal à passer ensuite à un langage différent, mais encore écrivent bien souvent des programmes qui même s'ils sont justes, restent laborieux. Car on n'ignore pas impunément les règles fondamentales de l'algorithmique... Alors, autant l'apprendre en tant que telle ! Bon, maintenant que j'ai bien fait l'article pour vendre ma marchandise, on va presque pouvoir passer au vif du sujet...

2.5 Avec quelles conventions écrit-on un algorithme ?

Historiquement, plusieurs types de notations ont représenté des algorithmes. Il y a eu notamment une représentation graphique, avec des carrés, des losanges, etc. qu'on appelait des organigrammes. Aujourd'hui, cette représentation est quasiment abandonnée, pour deux raisons. D'abord, parce que dès que l'algorithme commence à grossir un peu, ce n'est plus pratique du tout du tout. Ensuite parce que cette représentation favorise le glissement vers un certain type de programmation, dite non structurée (nous définirons ce terme plus tard), que l'on tente au contraire d'éviter. C'est pourquoi on utilise généralement une série de conventions appelée « pseudo-code », qui ressemble à un langage de programmation authentique dont on aurait évacué la plupart des problèmes de syntaxe. Ce pseudo-code est susceptible de varier légèrement d'un livre (ou d'un enseignant) à un autre. C'est bien normal : le pseudo-code, encore une fois, est purement conventionnel ; aucune machine n'est censée le reconnaître. Donc, chaque cuisinier peut faire sa sauce à sa guise, avec ses petites épices bien à lui, sans que cela prête à conséquence.

Comme je ne suis pas du tout assez créatif pour inventer mon propre langage de pseudo-code, et que j'ai l'inavouable habitude de faire tout en MATLAB, le pseudo-code que je vais vous proposer s'inspire, avec un manque d'originalité navrant, du MATLAB, ce qui en fait un code pas si pseudo que ça.

Chapitre 3

Les variables

N'attribuez jamais à la malveillance ce qui s'explique très bien par l'incompétence.

Napoléon Bonaparte

A l'origine de toute erreur attribuée à l'ordinateur, vous trouverez au moins deux erreurs humaines. Dont celle consistant à attribuer l'erreur à l'ordinateur.

Anonyme

3.1 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.

3.2 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. Les gens civilisés, dont je ne suis pas, choisissent en outre des noms de variables qui leur rappeleront confusément quelque chose... Ça peut aider le débugage. Il est en outre malvenu d'utiliesr comme nom de variable le nom d'une fonction prédéfine de l'ordinateur. De deux choses l'une, ou il vous insultera copieusement, ou il vous laissera faire, et cela entraînera des catastrophes du genre explosion thermonucléaire.

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.

En pratique, MATLAB que nous utiliserons ici s'en moque de la déclaration de variable, puisque tout cela est géré dynamiquement. D'un autre côté, il est bon, lorsque l'on débute la programmation, de s'habituer à se demander ce qu'on met où, et donc nous déclarerons gaillardement toutes les variables utilisées, mais en ligne de commentaire.

3.2.1 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 Ũ127 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	-3,40x1038 à -1,40x1045 pour les valeurs négatives 1,40x10-45 à 3,40x1038 pour les valeurs positives
Réel double	1,79x10308 à -4,94x10-324 pour les valeurs négatives 4,94x10-324 à 1,79x10308 pour les valeurs positives

Pourquoi ne pas déclarer toutes les variables numériques en réel double, histoire de bétonner et d'être certain qu'il n'y aura pas de problème ? En vertu du principe de l'économie de moyens. Un bon algorithme ne se contente pas de « marcher » ; il marche en évitant de gaspiller les ressources de la machine. Sur certains programmes de grande taille, l'abus de variables surdimensionnées peut entraîner des ralentissements notables à l'exécution, voire un plantage pur et simple de l'ordinateur. Alors, autant prendre dès le début de bonnes habitudes d'hygiène.

En algorithmique, on ne se tracassera pas trop avec les sous-types de variables numériques (sachant qu'on aura toujours assez de soucis comme ça, allez). On se contentera donc de préciser qu'il s'agit d'un nombre, en gardant en tête que dans un vrai langage, il faudra être plus précis.

En pseudo-code, une déclaration de variables aura cette tête :
% Variable g en Numérique
ou encore
% Variables RayonTerre, TrucMuch, bof en Numérique

3.2.2 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 pourriez les rencontrer en programmation proprement dite.

3.2.3 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 Ű mais ne fantasmez pas trop vite, c'est loin d'être aussi excitant que le nom le suggère...). 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 apostrophe. En MATLAB aussi... Pourquoi diable ? Pour éviter deux sources principales de 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 Ű- 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.

3.2.4 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). En MATLAB, en particulier, c'est 0 et 1 qui sont utilisé... 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. Le type booléen est très souvent négligé par les programmeurs, à tort. Ca permet de faire des tas de truc rigolo avec une économie de moyen qui fait furieusement penser à un film d'Alain Resnais, alors...

Il est vrai qu'il n'est pas à proprement parler indispensable, et qu'on pourrait écrire à peu près n'importe quel programme en l'ignorant complètement. Pourtant, si le type booléen est mis à disposition des programmeurs dans tous les langages, ce n'est pas pour rien. Le recours aux variables booléennes s'avère très souvent un puissant instrument de lisibilité des algorithmes : il peut faciliter la vie de celui qui écrit l'algorithme, comme de celui qui le relit pour le corriger.

Alors, maintenant, c'est certain, en algorithmique, il y a une question de style : c'est exactement comme dans le langage courant, il y a plusieurs manières de s'exprimer pour dire sur le fond la même chose. Nous verrons plus loin différents exemples de variations stylistiques autour d'une même solution. En attendant, vous êtes prévenus : l'auteur de ce cours est un adepte fervent (mais pas irraisonné) de l'utilisation des variables booléennes.

3.3 L'instruction d'affectation

3.4 Préambule à la con

Comment met-on un éléphant dans un frigo en trois étapes ?

1. On ouvre le frigo,
2. on rentre l'éléphant,
3. on ferme le frigo.

Comment met-on une giraphe dans un frigo en quatre étapes ?

1. on ouvre le frigo,
2. on retire l'éléphant,
3. on met la giraphe,
4. on ferme le frigo.

3.4.1 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 ou le frigo) 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.

Dans notre pseudo-code, nous utiliserons le signe = pour l'instruction d'affectation. Ce n'est certe pas criant d'originalité, et cela peut préter à confusion, mais tant pis. 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 du signe égal. Cette instruction n'est pas une question. Quand l'ordinateur la reçoit, il efface la valeur qu'il y avait dans `Tutu` (s'il y en avait une) et met à la place celle qui est dans Toto. Cependant Toto reste inchangé. `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 du signe égal. 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.

3.5 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 n°1

```
%Variable A en Numérique
%Début
A=34
A=12
%Fin
```

Exemple n°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 n°1

```
%Variable A en Numérique
%Début
A=12
%Fin
```

Exemple n°2

```
%Variable A en Numérique
%Début
A=34
%Fin
```

3.6 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.

3.6.1 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.

3.6.2 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'. Notons en passant que l'opérateur de concaténation en MATLAB est remplacé par la fonction strcat, ou par C=[A,B].

3.6.3 Opérateurs logiques (ou booléens)

Il s'agit du ET, du OU, du NON et du mystérieux (mais rarissime XOR). Nous les laisserons de côté... provisoirement, soyez-en sûrs.

3.6.4 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 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 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!!!

3.7 Exercices corrigés

1. Quels sont les valeurs de A, B, et C à la fin des algorithmes suivants :
   (a)
   ```
   % Debut
   % Variables A, B, C en numériques
   A=3
   B=1
   C=2+A*B
   % Fin
   ```
   A=3, B=1, C=2+3*1=5.

(b)
```
% Variables A, B en numerique
% Début
A=1
B=3
B=3+B-A
A=-B-A
%Fin
```
B=3+3-1=5, A=-5-1=-6.

2. Ecrire un algorithme qui initialise A et B à 1, met dans B la valeur de la somme de A et de B, puis dans A la valeur de la somme de A et de B. Quels sont les valeurs de A et de B a la fin du calcul ?

```
% Variables A, B en Entier
% Début
A=1
B=1
B=A+B
A=A+B
%Fin
```

B=1+1=2, A=1+2=3.

3.8 Exercices non corrigés

1. Quels sont les valeurs de A, B, et C à la fin des algorithmes suivants :

(a)
```
% Debut
% Variables A, B en numériques
A=3
B=A-1
B=2
% Fin
```

(b)
```
% Variables A, B en Entier
% Début
A=1
B=A + 3
A=3
%Fin
```

(c)
```
% Variables A, B en Entier
% Début
A=5
B=2
A=B
B=A
%Fin
```

(d)
```
% Variables A, B en Entier
% Début
A=1
B=A + 3
A=3
%Fin
```

(e)
```
% Variables A, B, C en Entier
% Début
A=3
B=10
```

```
C=A + B
B=A + B
A=C
%Fin
```

2. Écrire un algorithme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable.
3. 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).
4. Que produit l'algorithme suivant ?

```
% Variables A, B, C en Caractères
% Début
A='423'
B='12'
C=A+B
%Fin
```

5. Que produit l'algorithme suivant ?

```
% Variables A, B en Caractères
% Début
A='423'
B='12'
C='A & B'
% Fin
```

6. Que produit l'algorithme suivant ?

```
% Variables A, B en Caractères
% Début
A='423'
B='12'
C=A & B
% Fin
```

7. Ecrire un algorithme qui fait la différence entre un nombre de 2 chiffres et le meme nombre inversé (le chiffre des dizaines et celui des unités sont permutés).

Chapitre 4

Lecture et écriture

Un programme est un sort jeté sur un ordinateur, qui transforme tout texte saisi au clavier en message d'erreur.

Anonyme

Un clavier Azerty en vaut deux

Anonyme

4.1 De quoi parle-t-on ?

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, si l'informatique se limitait à cela, ce serait aussi vain qu'un aphorisme de Jean-Claude Van Damme[1].

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 calcule le carré de 12. C'est très gentil à lui (on vous l'avait dit, un ordinateur, c'est intrinsequement gentil et docile...). 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 »).

[1] On doit à ce philosophe contemporain quelques-unes des pensées les plus denses de notre temps, telles que *T'as pas besoin d'un flash quand tu photographies un lapin qui a déjà les yeux rouges* ou encore *Si tu travailles avec un marteau-piqueur pendant un tremblement de terre, désynchronise-toi, sinon tu travailles pour rien*, et *Je crois au moment. S'il n'y a pas le moment, à ce moment-là, il faut arriver à ce moment-là, au moment qu'on veut.*, sans oublier *Les cacahuetes c'est le mouvement perpétuel à la portée de l'homme*

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.

4.2 Les instructions de lecture et d'écriture

Tout bêtement, pour que l'utilisateur entre la (nouvelle) valeur de Titi, on mettra :
Titi=input,
parce que c'est quand même plus classe de parler en anglais.

Dès que le programme rencontre une instruction input, 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 :
display(Titi)
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!) :
nom=input('Entrer votre nom')
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). Et ça y est, vous savez d'ores et déjà sur cette question tout ce qu'il y a à savoir...

4.3 Exercices corrigés

1. Ecrire un algorithme qui demande le prénom à l'utilisateur et qui lui dit bonjour.

```
% variable nom en caractère
% Début
nom=input('Quel est votre nom?')
display('Bonjour '& nom & ', ça va bien?')
% fin
```

2. Ecrire l'algorithme qui devine l'âge des gens. On demande à l'utilisateur de prendre le nombre de son mois de naissance, de multiplier par 2, d'ajouter 5, de multiplier par 50, d'ajouter 1753, de retrancher l'année de naissance, puis de rentrer le résultat trouvé. Comme l'utilisateur est bête, on va lui donner un exemple en même temps. L'ordinateur prend le premier chiffre pour le mois de naissance, et les deux derniers pour l'âge.

```
% variables res, age, mois en numérique
% début
display('Prenez le mois de votre naissance en nombre (exemple: juin=6)')
display('Multipliez par 2 (exemple: 6x2=12)')
display('Ajoutez 5 (exemple: 12+5=17)')
display('Multipliez par 50 (exemple: 17x5=850)')
display('Ajoutez 1756 (exemple: 850+1756=2606)')
display('Retranchez votre année de naissance (exemple: 2606-1976=630)')
```

```
res=input('Entrez le résultat trouvé: ')
mois=fix(res/100)
age=res-100*mois
display(age)
% fin
```

Notons que l'instruction fix prend la partie entière d'un nombre, donc fix(res/100) prend le chiffre des centaines dans le résultat.

4.4 Exercices non corrigés

1. Quel résultat produit le programme suivant ?

```
% Variables val, double  en numériques
% Début
Val=231
Double=Val * 2
display(Val)
display(Double)
% Fin
```

2. Ecrire un programme qui demande un nombre à l'utilisateur, puis qui calcule et affiche le carré de ce nombre.
3. Ecrire un programme qui demande un rayon à l'utilisateur et calcule et affiche le volume et la surface de la sphère correspondante, avec un libellé précis.
4. Ecrire un algorithme utilisant des variables de type chaîne de 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.

Chapitre 5

Les tests

> Il est assez difficile de trouver une erreur dans son code quand on la cherche. C'est encore bien plus dur quand on est convaincu que le code est juste.
>
> Steve McConnell

> Il n'existe pas, et il n'existera jamais, de langage dans lequel il soit un tant soit peu difficile d'écrire de mauvais programmes
>
> Anonyme

> Si le déboguage est l'art d'enlever les bogues, alors la programmation doit être l'art de les créer.
>
> Anonyme

> Les ordinateurs, plus on s'en sert moins, moins ça a de chance de mal marcher.
>
> Jacques Rouxel

Je vous avais dit que l'algorithmique, c'est la combinaison de quatre structures élémentaires. Nous en avons déjà vu deux, voici la troisième. Autrement dit, on a quasiment fini le programme.
Mais non, je rigole.

> La programmation peut être un plaisir ; de même que la cryptographie. Toutefois, il faut éviter de combiner les deux.
>
> Kreitzberg et Sneidermann

5.1 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.

5.2 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.

```
if booléen
  Instructions
end
```

ou alors

```
if booléen
  Instructions 1
else
  Instructions 2
end
```

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 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 end. 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 else, la machine saute directement à la première instruction située après le end. De même, au cas où le booléen a comme valeur « Faux », la machine saute directement à la première ligne située après le else et exécute l'ensemble des « instructions 2 ». Dans tous les cas, les instructions situées juste après le end seront exécutées normalement. En fait, la forme simplifiée correspond au cas où l'une des deux « branches »du if est vide. Dès lors, plutôt qu'écrire « sinon ne rien faire du tout », il est plus simple de ne rien écrire. Et laisser un if... 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
if(la rue à droite est autorisée à la circulation)
  Tournez à droite
  Avancez
  Prenez la deuxième à gauche
else
  Continuez jusqu'à la prochaine rue à droite
  Prenez cette rue
  Prenez la première à droite
end
```

5.3 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

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

5.4 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), $\sim$ (NON), et XOR. Le & a le même sens en informatique que dans le langage courant. Pour que Condition1 & Condition2 soit VRAI, il faut impérativement que Condition1 soit VRAI et que Condition2 soit VRAI. Dans tous les autres cas, Condition1 & Condition2 sera faux.

TAB. 5.1 – Tables de vérité

C1&C2	C2 Vrai	C2 Faux
C1 Vrai	Vrai	Faux
C1 Faux	Faux	Faux

C1|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

Il faut se méfier un peu plus du |. Pour que Condition1 | 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 | 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 ~ inverse une condition : ~(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)

5.5 LE GAG DE LA JOURNÉE...

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

5.6 Tests imbriqués

Graphiquement, on peut très facilement représenter un if 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 numérique
```

```
%Début
Temp=input('Entrez la température de l'eau :')
if (Temp <= 0)
  display('C est de la glace')
end
if (Temp > 0&Temp < 100)
  display('C est du liquide')
end
if (Temp > 100)
  display('C est de la vapeur')
end
%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 numérique
% Début
Temp=input('Entrez la température de l'eau :')
if(Temp <= 0)
  display('C est de la glace')
else
  if(Temp < 100)
    display('C est du liquide')
  else
    display('C est de la vapeur')
  end
end
%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.

5.7 De l'aiguillage à la gare de tri

Dans un programme, une structure if 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 end.

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.

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 numérique
% Début
Temp=input('Entrez la température de l'eau :')
if(Temp <= 0)
  display('C est de la glace')
else
  if(Temp < 100)
    display('C est du liquide')
  else
    display('C est de la vapeur')
  end
end
%Fin
```

devienne :

```
% Variable Temp en numérique
% Début
Temp=input('Entrez la température de l'eau :')
if(Temp <= 0)
  display('C est de la glace')
elseif(Temp < 100)
  display('C est du liquide')
else
  display('C est de la vapeur')
end
%Fin
```

Dans le cas de tests imbriqués, le else et le if peuvent être fusionnés en un elseif. On considère alors qu'il s'agit d'un seul bloc de test, conclu par un seul end. Le elseif permet en quelque sorte de créer (en réalité, de simuler) des aiguillages à plus de deux branches. On peut ainsi enchaîner les elseif les uns derrière les autres pour simuler un aiguillage à autant de branches que l'on souhaite.

5.8 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 numérique
% Variables A, B en Booléen
% Début
Temp=input('Entrez la température de l'eau :')
A=(Temp<=0)
B=(Temp<100)
if A
  display('C est de la glace')
elseif B
  display('C est du liquide')
else
  display('C est de la vapeur')
end
% 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).

5.9 Faut-il mettre un & ? Faut-il mettre un | ?

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
X=input('X=?')
A=(X>12)
B=(X>2)
C=(X<6)
D=(A&B)|C
E=A&(B|C)
display([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 &, ou que des |, en revanche, les parenthèses ne changent strictement rien. Dans une condition composée employant à la fois des opérateurs & et des opérateurs |, 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 :

```
if (il fait trop chaud) & (il ne pleut pas)
  Ouvrir la fenêtre
else
  Fermer la fenêtre
end
```

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

```
if (il ne fait pas trop chaud) | (il pleut)
  Fermer la fenêtre
else
  Ouvrir la fenêtre
end
```

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 oeil 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) :

```
if (A \& B)
  Instructions 1
else
  Instructions 2
end
```

équivaut à :

```
if (\non A | \non B)
  Instructions 2
else
  Instructions 1
end
```

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

5.10 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 le |. 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 :

```
if (il fait trop chaud) & (il ne pleut pas)
  Ouvrir la fenêtre
else
  Fermer la fenêtre
end
```

Possède un parfait équivalent algorithmique sous la forme de :

```
if (il fait trop chaud)
  if (il ne pleut pas)
    Ouvrir la fenêtre
  else
    Fermer la fenêtre
  end
else
  Fermer la fenêtre
end
```

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 & peut aussi être exprimé par un |, 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. Si vous avez compris ce qui précède, et que les exercices qui suivent ne vous pose plus aucun problème, alors vous savez tout ce qu'il y a à savoir sur les tests pour affronter n'importe quelle situation. Non, ce n'est pas de la démagogie ! Malheureusement, nous ne sommes pas tout à fait au bout de nos peines ; il reste une dernière structure logique à examiner, et pas des moindres...

5.11 Exercices corrigés

1. Ecrire la table de vérité de A|(∼B). Si A est vrai, c'est vrai quel que soit le résultat du test sur B. Si B est faux, c'est vrai quel que soit le résultat du test sur A. Donc, le seul cas ou c'est faux, c'est A faux et B vrai.

A|(∼B)	A Vrai	A Faux
B Vrai	Vrai	Faux
B Faux	Vrai	Vrai

2. Ecrire un algorithme qui demande deux nombres à l'utilisateur et dit quel est le nombre le plus grand, ou s'ils sont égal.

```
% Variables x1, x2 en numérique
% début
x1=input('Entrer le nombre 1')
x2=input('Entrer le nombre 2')
if (x1>x2)
   display('Le premier nombre est plus grand que le second')
elseif(x1<x2)
   display('Le second nombre est plus grand que le premier')
else
   display('Les deux nombres sont égaux')
end
% fin
```

3. Formulez un algorithme équivalent à l'algorithme suivant :

```
if (Tutu > Toto + 4|Tata=='OK')
  Tutu=Tutu + 1
else
  Tutu=Tutu - 1
end
```

Par exemple, on peut faire

```
if (Tutu <= Toto + 4&Tata~='OK')
  Tutu=Tutu - 1
else
  Tutu=Tutu + 1
end
```

5.12 Exercices non corrigés

1. Ecrire la table de vérité pour les conditions suivantes :
   (a) ∼A|B
   (b) ∼A|A
   (c) A|(B&∼A)
   (d) A&∼B

(e) A & ∼A

2. Soient A=3, B=2, C=1. Quelles conditions sont vraies ?
   (a) A>B>C
   (b) C=1
   (c) A<3&B==2
   (d) A>C&B<A
   (e) A<C|B>=B/2+1
3. Quelles sont les conditions contraires à celles écrites ci-dessous ?
   (a) A&B&C
   (b) A|B
   (c) A&(B|∼A)
   (d) a>b|b>c
4. 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). Ecrire cet algorithme en utilisant des variables boléennes.
5. 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.
6. Ecrire un algorithme qui demande trois noms à l'utilisateur et l'informe ensuite s'ils sont rangés ou non dans l'ordre alphabétique. Ecrire cet algorithme en utilisant des variables boléennes.
7. 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).
8. 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
9. 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.
10. 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.
11. Un magasin de reprographie facture 0,10 € les dix premières photocopies, 0,09 € les vingt suivantes et 0,08 € au-delà. Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.
12. 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.

13. 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 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).

14. 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.
    - 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!)

15. 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 ».

Chapitre 6

Les Boucles

> Les premiers 90% du code prennent les premiers 90% du temps de développement. Les 10% restants prennent les autres 90% du temps de développement
>
> Tom Cargill

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 **if**). 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 !

6.1 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 **if**. Voyons voir ce que ça donne :

```
% Variable Rep en Caractère
% Début
```

```
Rep=input('Voulez vous un café ? (O/N)')
if (Rep~='O'&Rep~='N')
  Rep=input('Saisie erronnée. Recommencez (O/N)')
end
%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 if en pagaille est donc une impasse. La seule issue est donc de flanquer une structure de boucle, qui se présente ainsi :

```
while booléen
  ...
  Instructions
  ...
end
```

while veut dire "tant que".

Le principe est simple : le programme arrive sur la ligne du while. 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 end. Il retourne ensuite sur la ligne du while, 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
display('Voulez vous un café ? (O/N)')
while (Rep~='O'&Rep~='N')
  Rep=input('Saisie erronnée. Recommencez (O/N)')
end
%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
Rep=input('Voulez vous un café ? (O/N)')
while (Rep~='O'&Rep~='N')
  Rep=input('Saisie erronnée. Recommencez (O/N)')
end
%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='pif'
display('Voulez vous un café ? (O/N)')
while (Rep~='O'&Rep~='N')
  Rep=input('Saisie erronnée. Recommencez (O/N)')
end
%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
Rep=input('Voulez vous un café ? (O/N)')
while (Rep~='O'&Rep~='N')
  Rep=input('Saisie erronnée. Recommencez (O/N)')
end
display('Saisie acceptée')
%Fin
```

6.2 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... Bon, eh bien vous allez pouvoir faire de chouettes algorithmes, déjà rien qu'avec ça...

6.3 Boucler en comptant, ou compter en bouclant

Une boucle peut 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 while 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 numérique
% Début
Truc=0
while(Truc < 15)
  Truc=Truc + 1
  display(['Passage numéro :', Truc])
end
%Fin
```

Equivaut à :

```
% Variable Truc en Entier
% Début
for Truc=1:15
  display(['Passage numéro : ', Truc])
end
%Fin
```

Insistons : la structure for ... end n'est pas du tout indispensable ; on pourrait fort bien programmer toutes les situations de boucle uniquement avec un while. Le seul intérêt du for 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 for ... end est un cas particulier de while : celui où le programmeur peut dénombrer à l'avance le nombre de tours de boucles nécessaires.

Il faut noter que dans une structure for ... end, 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 for ; 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 for il suffit de le lui spécifier.

```
for Truc=1:3:15
```

fera prendre à Truc les valeurs 1, 4, 7, 10 et 13, successivement.

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.

Les structures while 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

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

Nous verrons dans les chapitres suivants des séries d'éléments appelés tableaux et chaînes de caractères. Selon les cas, le balayage systématique des éléments de ces séries pourra être effectué par un for ou par un while : 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.

6.4 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 **if** peut contenir d'autres structures **if**, une boucle peut tout à fait contenir d'autres boucles. Y a pas de raison.

```
% Variables Truc, Trac en Entier
% Début
for Truc=1:15
  display('Il est passé par ici')
  for Trac=1:6
    display('Il repassera par là')
  end
end
% 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
for Truc=1:15
  display('Il est passé par ici')
end
for Trac=1:6
  display('Il repassera par là')
end
%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 1) ou successives (cas 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 penser cette structure aberrante.

```
% Variables Truc, Trac en Entier
% Début
for Truc = ...
  instructions
  for Trac = ...
    instructions
end %(lié à Truc)
  instructions
end %(lié à Trac)
%Fin
```

En fait, c'est tellement impossible que l'on ne sécifie même pas ce que le **end** ferme. On sait que cela ne peut qu'être le dernier ouvert, sinon, le code est monstrueux.

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.

6.5 Et encore une bêtise à ne pas faire !

Examinons l'algorithme suivant :

```
% Variable Truc en Entier
% Début
for Truc=1:15
  Truc=Truc*2
  display(['Passage numéro : ', Truc])
end
%Fin
```

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

```
for Truc=1:15
```

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. Le Gag De La Journée consiste donc à manipuler, au sein d'une boucle for, la variable qui sert de compteur à cette boucle. Cette technique est à proscrire absolument... sauf bien sûr, si vous cherchez un prétexte pour régaler tout le monde au bistrot. Mais dans ce cas, n'ayez aucune inhibition, proposez-le directement, pas besoin de prétexte.

Une dernière remarque... l'instruction
`display(['Passage numéro : ', Truc])`
ne saurait fonctionner. En effet, il est incorrect de concaténer une variable caractère (en l'occurence 'Passage numéro : ') avec un nombre... Cela peinera sûrement l'ordinateur que ne manquera pas soit de vous abreuver d'un flot d'insultes, soit de vous donner un résultat absurde, tout en vous maudissant, vous et vos descendants, jusqu'à la septième génération. Il faut donc convertir Truc en caractère, chose que l'on fait au moyen de la fonction num2str qui convertit, comme son nom très poétique le laisse entendre, des numériques en strings. L'instruction correcte sera donc
`display(['Passage numéro : ', num2str(Truc)])`

6.6 Exercices corrigés

1. Que font les algorithmes suivants ? Quand vous le pouvez, écrire un algorithme plus simple qui fait la même action.

   (a)
   ```
   % Variable i,x en Entier
   % Début
   x=0
   for i=1:15
     x=x+1
     display(x)
   end
   ```

```
display(['Au total, ',num2str(x),' passages'])
%Fin
```

Au départ, x=0, on entre dans une boucle pour i qui va de zéro à 15. A chaque passage, la variable x est incrémentée de 1, et on indique le numéro du passage. A la fin, on écrit le nombre total de passages. Une fois exécuté, cet algorithme produira :

```
Passage numéro : 1
Passage numéro : 2
Passage numéro : 3
Passage numéro : 4
Passage numéro : 5
Passage numéro : 6
Passage numéro : 7
Passage numéro : 8
Passage numéro : 9
Passage numéro : 10
Passage numéro : 11
Passage numéro : 12
Passage numéro : 13
Passage numéro : 14
Passage numéro : 15
Au total, 15 passages
```

(b)
```
% Variable i,x en numérique
% Début
x=0;
for i=1:5
   if (x>1&i<3)
     x=x-i
   elseif(x>0&i>2)
     x=x-2*i
   else
     x=x+3*i
   end
end
% Fin
```

Au premier passage, x=0, i=1, la première condition est fausse, la seconde aussi, donc on applique la troisième, et l'on change x=0 pour x=x+3*i=3. Au second passage, x=3, i=2, la première condition est vraie, donc on change x=3 pour x=x-2=1. Au troisième passage, x=1, i=3, la première condition est fausse, et la seconde est vraie, donc x=1 devient x=1-6=-5. Au quatrième passage, x=-5,i=4, les deux premières conditions sont fausses (puisque x est négatif), et par conséquent, x devient x=-5+3*4=7. Au cinquième et dernier passage, x=7, i=5, la première condition est fausse et la seconde est vraie. Par conséquent, x=7-2*4=-1. Le programme génèrera donc la suite : 0, 3, 1, -5 ,7 ,-1, pour les valeurs de x.

(c)
```
% Variable i,x en numérique
% Début
x=0;
i=0;
while (x<5)
   i=i+1
   if (x>0&i<10)
      i=i-1
      x=x-i
   else
      x=x+i
```

```
    end
  end
  % Fin
```

Au début, quand on arrive au while, x=0, i=0, la condition du while est bien respectée, et on exécute la boucle. On met 1 dans i, on a par conséquent : x=0, i=1, la condition est donc fausse, et on change x pour x=x+1=1. Ensuite, x=1, i=1, on respecte toujours la condition, on change donc i pour le mettre à 2. Dans ce cas, la condition est vraie, et l'on change donc i=i-1=1, x=x-1=0. Au troisième tour, la condition du while est toujours vraie, et l'on remet donc i à 2. La condition n'est pas vérifiée, et l'on a donc x=x+2=2. Au début de l'itération suivante, on a donc i=2+1=3, x=2, la condition est donc vraie, et l'on remet donc i=2 et l'on a x=x-2=1. Ensuite, on change i pour remettre 3, la condition est vraie, donc i=2 et l'on a x=x-2=-1. L'itération suivante donne i=3 et la condition est fausse, donc x=x+3=2. Au tour suivant, i=4 et x=2, donc la condition est vraie et l'on fait x=x-2=0, et i=3. L'itération suivante, vérifiant toujours la condition du while, remet donc 4 dans i, et la condition est vraie, donc i=3x=x-3=-1. Ensuite, i=4 et la condition est fausse, puisque x<0, donc on remet x=x+4=3. Par suite, on met i=4, la condition est vérifiée, donc i=3 et x=0. Par conséquent, l'itération suivante donne i=4, condition fausse, donc x=4. Puis i=5, condition vraie, donc i=4, x=0. Dès lors, i=5 et x=0, la condition est fausse, et l'on ajoute gaillardement 5 à x, ce qui nous fait sortir de la boucle while. La suite des valeurs de x est donc 0, 0, 1, 0, 2, -1, 2, 0, -1, 3, 0, 4, 0, 5.

2. Quelle sont les erreurs dans l'algorithme suivant ?

```
% Variable x, i en numérique
% Début
for i=1:10
   x=x+2*i-1
   if (x>0&2<i<=3)
     i=i+1;
     x=x-i;
  end
end
% Fin
```

(1) La valeur de x n'est pas initialisée, donc problème pour calculer x=x+2*i-1. (2) La condition n'a pas de sens. (3) On tripote la variable du for.

3. Ecrire un algorithme qui décompose un nombre en facteur premier.
Rappelons que la décomposition en facteur premier correspond à écrire le nombre sous la forme d'un produit de nombres premiers. Donc, l'algorithme passe en revue tous les nombres et teste si le nombre est divisible par ce nombre-là, si oui, il le divise, et reteste, sinon, il passe au suivant. Prenons 12 par exemple. On commence par 2, 12 est divisible par 2, le quotient est 6, toujours divisible par 2, le quotient est 3, plus divisible par 2. Donc, les facteurs sont donc 2 2 3.

```
% Variable i, ind, n en Entier
% Début
ind=1
for i=2:sqrt(n)
   while (n/i==fix(n/i))
       display(i)
       n=n/i
       ind=ind+1
   end
end
if (n~=1)
    display(n)
end
%Fin
```

6.7 Exercices non corrigés

1. Pourquoi les programmes suivant vont-ils poser problème? (attention, parfois, j'ai poussé le vice jusqu'à mettre plusieurs erreurs dans un même code).

   (a)
   ```
   % Variable i en Entier
   % Début
   for i=1:15
     i=i-1
     display(['Passage numéro : ', i])
   end
   %Fin
   ```

   (b)
   ```
   % Variable i en Entier
   % Début
   i=1
   while (i<0)
     i=i+1
   end
   %Fin
   ```

   (c)
   ```
   % Variable i en Entier
   % Début
   i=1
   while (i>0)
     i=i+1
   end
   %Fin
   ```

2. Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.
3. 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.
4. 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.
5. 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 3 = 21
   ...
   7 x 10 = 70
   ```

6. 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.
7. 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.
8. Ecrire un algorithme qui demande successivement 10 nombres à l'utilisateur, et qui lui dise ensuite quel était le plus grand parmi ces 10 nombres :

```
Entrez le nombre numéro 1 : 12
Entrez le nombre numéro 2 : 14
etc.
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
```

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 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.
10. Ecrire 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 :

    ```
    X = n ! / (n - p) !
    Y = n ! / (p ! * (n - p) !)
    ```

11. Ecrire un algorithme qui fais deviner un nombre entre 1 et 1000 à l'utilisateur. D'abord, il faut entrer le nombre à deviner. Puis, l'utilisateur est prié d'entrer un nombre entre 1 et 1000 (la validité de l'entrée est testée), puis l'ordinateur dis si le nombre est plus petit ou plus grand jusqu'à ce que le nombre soit trouvé.