

Programmation partie 01 : Logiques de base de l'algorithmique

A. Introduction

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.

1. Ou'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 – et fait exécuter – 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. 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 on 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. 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 juger.

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. En effet, 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.

3. L'ADN et les ordinateurs

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

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 :

- l'affectation de variables
- la lecture / écriture
- les tests
- 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 de gestion. 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).

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 !

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.

B. Les variables

1. A quoi servent les variables ?

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

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

En réalité, dans la mémoire vive de l'ordinateur, il n'y a bien sûr pas une vraie boîte, et pas davantage de vraie étiquette collée dessus (j'avais bien prévenu que la boîte et l'étiquette, c'était une image). Dans l'ordinateur, physiquement, il y a un emplacement de mémoire, repéré par une adresse binaire. Si on programmait dans un langage directement compréhensible par la machine, on devrait se fader de désigner nos données par de superbes 1001 1001 et autres 01001001 (enchanté !). Mauvaise nouvelle : de tels langages existent ! Ils portent le doux nom d'assembleur. Bonne nouvelle : ce ne sont pas les seuls langages disponibles.

Les langages informatiques plus évolués (ce sont ceux que presque tout le monde emploie) se chargent précisément, entre autres rôles, d'épargner au programmeur la gestion fastidieuse des emplacements mémoire et de leurs adresses. Et, comme vous commencez à le comprendre, il est beaucoup plus facile d'employer les étiquettes de son choix, que de devoir manier des adresses binaires.

2.1 Types numériques classiques

Commençons par le cas très fréquent, celui d'une variable destinée à recevoir des nombres.

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).

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.

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 dépend du langage utilisé.

Un groupe de caractères (y compris un groupe de un, ou de zéro caractères), qu'il soit ou non stocké dans une variable, d'ailleurs, est donc souvent appelé chaîne de caractères.

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

Pourquoi diable ? Pour éviter deux sources principales de possibles confusions :

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

2.4 Type booléen

Le dernier type de variables est le type booléen : on y stocke uniquement les valeurs logiques VRAI et FAUX.

On peut représenter ces notions abstraites de VRAI et de FAUX par tout ce qu'on veut : de l'anglais (TRUE et FALSE) ou des nombres (0 et 1). Peu importe. Ce qui compte, c'est de comprendre que le type booléen est très économique en termes de place mémoire occupée, puisque pour stocker une telle information binaire, un seul bit suffit.

3. L'instruction d'affectation

3.1 Syntaxe et signification

Ouf, après tout ce baratin préliminaire, on aborde enfin nos premières véritables manipulations d'algorithmique. Pas trop tôt, certes, mais pas moyen de faire autrement !

En fait, la variable (la boîte) n'est pas un outil bien sorcier à manipuler. A la différence du couteau suisse ou du superbe robot ménager vendu sur Télé Boutique Achat, on ne peut pas faire trente-six mille choses avec une variable, mais seulement une et une seule.

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

En pseudo-code, l'instruction d'affectation se note avec le signe ←

Ainsi :

Toto ← 24

Attribue la valeur 24 à la variable Toto.

Ceci, soit dit en passant, sous-entend impérativement que Toto soit une variable de type numérique. Si Toto a été défini dans un autre type, il faut bien comprendre que cette instruction provoquera une erreur. C'est un peu comme si, en donnant un ordre à quelqu'un, on accolait un verbe et un complément incompatibles, du genre « Epluchez la casserole ». Même dotée de la meilleure volonté du monde, la ménagère lisant cette phrase ne pourrait qu'interrompre dubitativement sa tâche. Alors, un ordinateur, vous pensez bien...

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

Tutu ← Toto

Signifie que la valeur de Tutu est maintenant celle de Toto.

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

Tutu ← Toto + 4

Si Toto contenait 12, Tutu vaut maintenant 16. De même que précédemment, Toto vaut toujours 12.

Tutu ← Tutu + 1

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

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

Exemple n°1

- Début
- Riri ← "Loulou"
- Fifi ← "Riri"
- Fin

Exemple n°2

- Début
- Riri ← "Loulou"
- Fifi ← Riri
- Fin

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

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

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

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

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

3.2 Ordre des instructions

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

Exemple 1

- Variable A en Numérique
- Début
- $A \leftarrow 34$
- $A \leftarrow 12$
- Fin

Exemple 2

- Variable A en Numérique
- Début
- $A \leftarrow 12$
- $A \leftarrow 34$
- 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 1 km, puis à droite » n'envoie pas les gens au même endroit que si l'on dit « prenez à droite puis tout droit pendant 1 km ».

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

Exemple 1

- Variable A en Numérique
- Début
- $A \leftarrow 12$
- Fin

Exemple 2

- Variable A en Numérique
- Début
- $A \leftarrow 34$
- Fin

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

Réaliser les exercices de 1.1 à 1.9

C. Lecture et écriture

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, à la fin de la foire, on peut tout de même se demander à quoi ça sert.

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

- Variable A en Numérique
- Début
- $A \leftarrow 12^2$
- Fin

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

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

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

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 ça 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.

2. Les instructions de lecture et d'écriture

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

- Lire Titi

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

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

- **Ecrire** Toto

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

- **Ecrire** "Entrez votre nom : "
- **Lire** NomFamille

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

Et ça y est, vous savez d'ores et déjà sur cette question tout ce qu'il y a à savoir...

Réaliser les exercices de 2.1 à 2.4

D. Les tests

1. De quoi s'agit-il ?

Reprenons le cas de notre « programmation algorithmique du touriste égaré ».
Normalement, l'algorithme ressemblera à quelque chose comme : « Allez tout droit jusqu'au prochain carrefour, puis prenez à droite et ensuite la deuxième à gauche, et vous y êtes ».

Mais en cas de doute légitime de votre part, cela pourrait devenir : « Allez tout droit jusqu'au prochain carrefour et là regardez à droite. Si la rue est autorisée à la circulation, alors prenez la et ensuite c'est la deuxième à gauche. Mais si en revanche elle est en sens interdit, alors continuez jusqu'à la prochaine à droite, prenez celle-là, et ensuite la première à droite ».

Ce deuxième algorithme a ceci de supérieur au premier qu'il prévoit, en fonction d'une situation pouvant se présenter de deux façons différentes, deux façons différentes d'agir. Cela suppose que l'interlocuteur (le touriste) sache analyser la condition que nous avons fixée à son comportement (« la rue est-elle en sens interdit ? ») pour effectuer la série d'actions correspondante.

Eh bien, croyez le ou non, mais les ordinateurs possèdent cette aptitude, sans laquelle d'ailleurs nous aurions bien du mal à les programmer. Nous allons donc pouvoir parler à notre ordinateur comme à notre touriste, et lui donner des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre. Cette structure logique répond au doux nom de test. Toutefois, ceux qui tiennent absolument à briller en société parleront également de structure alternative.

2. Structure d'un test

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

- Si booléen Alors
- Instructions
- Finsi

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

Ceci appelle quelques explications.

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

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

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

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

Dans le cas de la structure complète, c'est à peine plus compliqué. Dans le cas où le booléen est VRAI, et après avoir exécuté la série d'instructions 1, au moment où elle arrive au mot « Sinon », la machine saute directement à la première instruction située après le « Finsi ». De même, au cas où le booléen a comme valeur « Faux », la machine saute directement à la première ligne située après le « Sinon » et exécute l'ensemble des « instructions 2 ». Dans tous les cas, les instructions situées juste après le FinSi seront exécutées normalement.

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

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

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

3. Qu'est ce qu'une condition ?

Une condition est une comparaison

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

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

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

Les opérateurs de comparaison sont :

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

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

À noter que ces opérateurs de comparaison peuvent tout à fait s'employer avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique, 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.

Réaliser l'exercice 3.1

4. Conditions composées

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

Comme on l'a évoqué plus haut, l'informatique met à notre disposition quatre opérateurs logiques : ET, OU, NON, et XOR.

- Le ET a le même sens en informatique que dans le langage courant. Pour que "Condition1 ET Condition2" soit VRAI, il faut impérativement que Condition1 soit VRAI et que Condition2 soit VRAI. Dans tous les autres cas, "Condition 1 et Condition2" sera faux.
- Il faut se méfier un peu plus du OU. Pour que "Condition1 OU Condition2" soit VRAI, il suffit que Condition1 soit VRAIE ou que Condition2 soit VRAIE. Le point important est que si Condition1 est VRAIE et que Condition2 est VRAIE aussi, Condition1 OU Condition2 reste VRAIE. Le OU informatique ne veut donc pas dire «ou bien»
- Le XOR (ou OU exclusif) fonctionne de la manière suivante. Pour que "Condition1 XOR Condition2" soit VRAI, il faut que soit Condition1 soit VRAI, soit que Condition2 soit VRAI. Si toutes les deux sont fausses, ou que toutes les deux sont VRAI, alors le résultat global est considéré comme FAUX. Le XOR est donc l'équivalent du "ou bien" du langage courant.
J'insiste toutefois sur le fait que le XOR est une rareté, dont il n'est pas strictement indispensable de s'encombrer en programmation.
- Enfin, le NON inverse une condition : NON(Condition1) est VRAI si Condition1 est FAUX, et il sera FAUX si Condition1 est VRAI. C'est l'équivalent pour les booléens du signe "moins" que l'on place devant les nombres.
Alors, vous vous demandez peut-être à quoi sert ce NON. Après tout, plutôt qu'écrire NON(Prix > 20), il serait plus simple d'écrire tout bonnement Prix <= 20.
Dans ce cas précis, c'est évident qu'on se complique inutilement la vie avec le NON. Mais si le NON n'est jamais indispensable, il y a tout de même des situations dans lesquelles il s'avère bien utile.

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

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

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

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

Non C1	
C1 Vrai	Faux
C1 Faux	Vrai

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 !)

Réalise les exercices 3.1 et 3.2.

5. Tests imbriqués

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

Une première solution serait la suivante :

- Variable Temp en Entier

- Début
- Ecrire "Entrez la température de l'eau :"
- Lire Temp
- Si Temp ≤ 0 Alors
 - Ecrire "C'est de la glace"
- FinSi
- Si Temp > 0 Et Temp < 100 Alors
 - Ecrire "C'est du liquide"
- Finsi
- Si Temp > 100 Alors
 - Ecrire "C'est de la vapeur"
- Finsi
- Fin

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

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

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

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

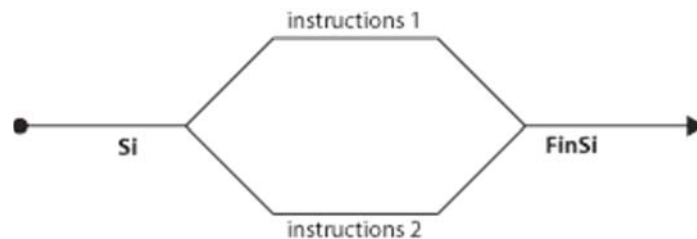
Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

Réaliser les exercices de 3.4 à 3.6.

6. De l'aiguillage à la gare de tri

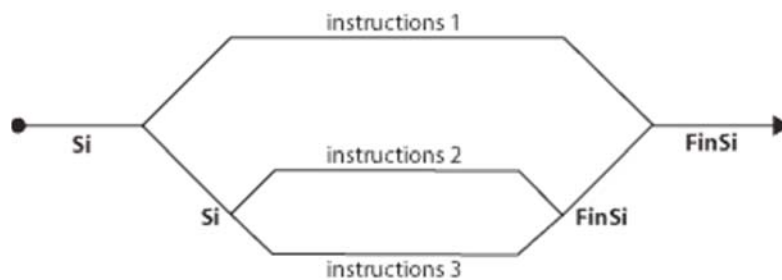
Cette citation n'apporte peut-être pas grand chose à cet exposé, mais je l'aime bien, alors c'était le moment ou jamais.

En effet, dans un programme, une structure SI peut être facilement comparée à un aiguillage de train. La voie principale se sépare en deux, le train devant rouler ou sur l'une, ou sur l'autre, et les deux voies se rejoignant tôt ou tard pour ne plus en former qu'une seule, lors du FinSi. On peut schématiser cela ainsi :



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

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



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

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

- Finsi
- Fin

devienne :

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

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

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

7. Variables Booléennes

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

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

- Variable Temp en Entier
- Variables A, B en Booléen
- Début
- Ecrire "Entrez la température de l'eau :"
- Lire Temp
- $A \leftarrow \text{Temp} \leq 0$
- $B \leftarrow \text{Temp} < 100$
- Si A Alors
 - Ecrire "C'est de la glace"
- SinonSi B Alors
 - Ecrire "C'est du liquide"
- Sinon
 - Ecrire "C'est de la vapeur"
- Finsi
- Fin

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

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

E. Les boucles

Et ça y est, on y est, on est arrivés, la voilà, c'est Broadway, la quatrième et dernière structure : ça est les boucles. Si vous voulez épater vos amis, vous pouvez également parler de structures répétitives, voire carrément de structures itératives. Ça calme, hein ? Bon, vous faites ce que vous voulez, ici on est entre nous, on parlera de boucles.

Les boucles, c'est généralement le point douloureux de l'apprenti programmeur. C'est là que ça coince, car autant il est assez facile de comprendre comment fonctionnent les boucles, autant il est souvent long d'acquérir les réflexes qui permettent de les élaborer judicieusement pour traiter un problème donné.

On peut dire en fait que les boucles constituent la seule vraie structure logique caractéristique de la programmation. Si vous avez utilisé un tableur comme Excel, par exemple, vous avez sans doute pu manier des choses équivalentes aux variables (les cellules, les formules) et aux tests (la fonction SI...). Mais les boucles, ça, ça n'a aucun équivalent. Cela n'existe que dans les langages de programmation proprement dits.

Le maniement des boucles, s'il ne différencie certes pas l'homme de la bête (il ne faut tout de même pas exagérer), est tout de même ce qui sépare en informatique le programmeur de l'utilisateur, même averti.

Alors, à vos futures – et inévitables – difficultés sur le sujet, il y a trois remèdes : de la rigueur, de la patience, et encore de la rigueur !

1. A quoi cela sert-il donc ?

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

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

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

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

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

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

- TantQue booléen
 - o ...
 - o Instructions
 - o ...
- FinTantQue

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

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

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

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

Son principal défaut est de provoquer une erreur à chaque exécution. En effet, l'expression booléenne qui figure après le TantQue interroge la valeur de la variable Rep.

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

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

Une autre possibilité, fréquemment employée, consiste à ne pas lire, mais à affecter arbitrairement la variable avant la boucle. Arbitrairement ? Pas tout à fait, puisque cette affectation doit avoir pour résultat de provoquer l'entrée obligatoire dans la boucle. L'affectation doit donc faire en sorte que le booléen soit mis à VRAI pour déclencher le premier tour de la boucle. Dans notre exemple, on peut donc affecter Rep avec n'importe quelle valeur, hormis « O » et « N » : car dans ce cas, l'exécution sauterait la boucle, et Rep ne serait pas du tout lue au clavier. Cela donnera par exemple :

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

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

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

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

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

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

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

Quant à la deuxième solution, elle pourra devenir :

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

Le(s) gag(s) 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 baveux, le poil à gratter ou le bleu de méthylène : c'est éculé, mais ça fait toujours rire.

Cette faute de programmation grossière – mais fréquente - ne manquera pas d'égayer l'ambiance collective de cette formation... et accessoirement d'étancher la soif proverbiale de vos enseignants.

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

Réaliser les exercices 5.1, 5.2 et 5.3

2. Boucler en comptant, ou compter en bouclant

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

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

- Variable Truc en Entier
- Début
- $\text{Truc} \leftarrow 0$
- TantQue $\text{Truc} < 15$
 - $\text{Truc} \leftarrow \text{Truc} + 1$
 - Ecrire "Passage numéro : ", Truc
- FinTantQue
- Fin

Equivalent à :

- Variable Truc en Entier
- Début
- Pour $\text{Truc} \leftarrow 1 \text{ à } 15$
 - Ecrire "Passage numéro : ", Truc
- Truc Suivant
- Fin

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

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

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

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

Naturellement, quand on stipule un pas négatif dans une boucle, la valeur initiale du compteur doit être supérieure à sa valeur finale si l'on veut que la boucle tourne ! Dans le

cas contraire, on aura simplement écrit une boucle dans laquelle le programme ne rentrera jamais.

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

- **Pour** Compteur ← Initial à Final **Pas** ValeurDuPas
- ...
- Instructions
- ...
- Compteur **suivant**

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

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

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.

Recommencer l'exercice 5.3 en utilisant la boucle **pour**.