Le Leader de L'Algorithmique

Chahid Khichane

Cours et Exercices avec Corrections

Le leader de l'algorithmique

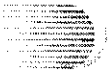
Cours et Exercices avec leur corrigés

Auteur: **Chahid KHICHANE**

el M A A R I F A

Editions

ISBN : **9961-48-197-6**
Dépot Légal : **2873/2004**



Société EL MAARIFA

22 BD Said Touafdit
(ex. Deux Moulins) 16090 **Bologhine** ALGER

Service commercial :
Tél : 021.96.76.65
Service editions :
Tél : 021 70.93.71.
Tél /Fax : 021.70.98.30
http//www.el-maarifa.com
e mail : el maarifa @ el-maarifa.com

Tous Droits Réservés

Préambule

Cet ouvrage est destiné aux étudiants débutants en algorithmique ou ayant déjà quelques notions d'algorithmique ou de programmation et souhaitant approfondir leur connaissance.

Les notions fondamentales (types de données, opérateurs, instructions de contrôle, fonctions, procédures, *tableaux*, *fichiers* ...) sont exposées avec un grand soin pédagogique, le lecteur étant conduit progressivement vers la maîtrise de ces concepts.

Chaque notion importante est illustrée d'exemples d'algorithmes complets. De nombreux exercices, dont la solution est fournie en fin de chapitre, vous permettront de tester vos connaissances fraîchement acquises et de les approfondir.

A qui s'adresse ce livre ?

Aux étudiants futurs ingénieurs, aux autodidactes ou professionnels de tous horizons souhaitant s'initier à la programmation, aux techniciens ou techniciens supérieurs étudiant dans les instituts ou dans les centres de formation, aux enseignants et formateurs à la recherche d'une méthode pédagogique et d'un support de cours structuré pour enseigner l'algorithmique.

Sommaire

Historique des ordinateurs et notion de codage 1

1- Introduction 1

2- Les langages 4

3- Le codage 6

3.1- La numérotation binaire 6

3.2- Changement de base et programmation 6

3.3- Un autre système de numérotation 8

3.4- Pourquoi l'ordinateur est-il binaire ? 8

4- Stockage informatique 9

4.1- Unités de mémoire 9

4.2- Les disquettes 10

4.3- Les disque durs 12

4.4- Les Cédéroms 13

4.5- Les DVD 14

4.6- Les disquettes ZIP 16

4.7- Les mémoires flash (ou disques flash) 16

Exercices 19

Corrigés des exercices 21

Chapitre I : Introduction à l'algorithmique 29

1- Historique 29

2- Notion d'algorithme 29

3- Utilité des algorithmes 30

4- Faut- il être matheux pour être bon en algorithmique ? 31

5- La forme générale d'un algorithme 31

6- Caractéristiques d'un bon algorithme 32

7- La modularité 33

8- Place de l'algorithme dans la résolution d'un problème informatique 34

9- Notion de pseudo-code (ou pseudo-langage) 35

10- Des problèmes sans solution 36

11- Méthodes d'analyse 37

11.1- Analyse descendante 37

11.2- Analyse ascendante 38

11.3- Les méthodes dirigées par les données 38

12- Evolution de la programmation impérative vers la programmation objet informatique 39

1ère étape : la programmation procédurale 39

2ème étape : la programmation modulaire 39

3ème étape : la programmation par type générique (ou abstraction) 39

4ème étape : les objets 39

Chapitre II : Les variables 41

1- Notion de variable et de constante 41

2- A quoi servent les variables ? 42

3- Déclaration des variables 43

4- Types de variables 43

4.1- Types numériques classiques 44

4.2- Types non numériques 45

a- Type alphanumérique (ou type caractère) 45

b- Type booléen 45

5- L'instruction d'affectation 45

5.1- Syntaxe et signification 45

5.2- Ordre des instructions 46

5.3- Expressions 47

5.4- Opérateurs 47

a- Opérateurs numériques (opérateurs arithmétiques) 47
b- Opérateur alphanumérique 48
c- Opérateurs logiques 48
5.5- Remarques 49
6- Forme générale d'un algorithme avec variables 49
7- Lecture et écriture 50
7.1- Les instructions Lire et Ecrire 50
7.2- Exemple de programme utilisant les instructions Lire et Ecrire 51
Exercices 53
Corrigés des exercices 59
Chapitre III : Les tests 65
1- Structure d'un test 65
2- Une condition, c'est quoi ? 66
3- Conditions composées 67
4- Tests imbriqués 68
5- Variables booléennes 69
6- Jeux logiques 70
7- Choix multiples : la structure Selon 71
Exercices 73
Corrigés des exercices 79
Chapitre IV : Les boucles 95
1- Les boucles, c'est quoi au juste ? 95
2- Intérêt des boucles 95
3- La boucle « Tant que » 96
4- La boucle « Répéter » 97
5- La boucle « Pour » 97
6- Des boucles dans des boucles 98

7- Comparaisons des boucles 99
Exercices 101
Corrigés des exercices 105
Chapitre V : Les tableaux 115
1- *Pourquoi les tableaux ?* 115
2- Les tableaux à deux dimensions 122
3- Les tableaux à n dimensions 123
4- *Trier un tableau à une dimension* 123
4.1- Qu'est ce que le tri ? 123
4.2- La procédure échanger 124
4.3- Le tri par minimum successif 124
4.4- Le tri par insertion 128
4.5- Le tri Bulle 131
5- La recherche dichotomique 137
Exercices 141
Corrigés des exercices 151
Chapitre VI : Les fonctions et procédures 167
1- Rappels 167
2- Notion de sous-programme 168
3- Nom d'un sous-programme 168
4- Portée des variables dans un sous-programme 169
5- Structure d'un programme 169
6- Les paramètres d'un sous-programme 170
7- Le passage de paramètres 170
7.1- Le passage de paramètres en entrée 170
7.2- Le passage de paramètres en sortie 171
7.3- Le passage de paramètres en entrée / sortie 171
8- Les fonctions 171

9- Les fonctions prédéfinies 174
9.1- Structure générale des fonctions prédéfinies 175
9.2- Les fonctions de texte 175
9.3- Deux fonctions classiques 176
10- Les procédures 177
Exercices 181
Corrigés des exercices 187
Chapitre VII : Les fichiers 197
A- Les fichiers de données 197
1- Généralités sur l'organisation des informations 197
2- Notion de fichier et d'enregistrement 198
2.1- Définition 198
2.2- Notion d'enregistrement 198
3- Les accès aux fichiers 199
3.1- Accès séquentiel 199
3.2- Accès direct 200
4- Les fichiers en algorithmique 200
4.1- Déclaration d'une variable de type fichier 200
4.2- Assignation d'un nom de fichier logique 200
4.3- Actions et opérations sur les fichiers 201
B- Les fichiers texte 202
1- Organisation des fichiers texte 202
2- Types d'accès 203
3- Instructions 204
4- Stratégies de traitement 206
Exercices 209
Corrigés des exercices 213

Historique des ordinateurs et notion de codage

1- Historique :

Il est difficile de faire commencer l'histoire des ordinateurs à une date bien précise. Comme l'ordinateur est un outil d'aide au traitement de l'information, nous citerons les principales innovations qui ont facilité ou automatisé le calcul (les nombres étant longtemps la principale source d'information).

L'idée d'utiliser des supports matériels pour manipuler les nombres d'une manière répétitive est très ancienne. Vers 2500 avant J-C, apparaissait déjà le boulier qui permettait d'effectuer des opérations arithmétiques élémentaires. Six siècles plus tard, une tablette babylonnienne en argile (1900-1600 avant J-C) permettra à partir de deux côtés d'un triangle rectangle, de trouver le troisième.

Les premiers dispositifs mécaniques d'aide au calcul apparaissent seulement à la Renaissance: en 1642, Blaise Pascal invente une machine permettant d'additionner et de soustraire, pour simplifier la tâche de son père, commissaire pour la levée des impôts. En 1671, le mathématicien allemand Gottfried Wilhem Leibniz conçoit une machine permettant les quatre opérations arithmétiques. Jusqu'au XIX ème siècle, ces machines seront copiées sans qu'on y apporte d'améliorations significatives.

Au cours du XIX ème siècle, quelques nouvelles machines capables d'effectuer les quatre opérations élémentaires apparaissent: l'arithmomètre (1820) de Thomas de Colmar conçu à partir des idées de Leibniz, la machine à curseur du suédois Odhner (1875), le comptomètre à clavier de l'américain Felt (1885). Le développement des techniques de réalisation d'automates (par exemple, les horloges astronomiques) allait être stimulé par celui des industries naissantes, en particulier de l'industrie textile pour laquelle Jacquard, perfectionnant en 1801 une invention de Vaucanson (1745) crée la carte perforée destinée à commander des métiers à tisser.

Avec plus d'un siècle d'avance, l'anglais Charles Babbage propose en 1822 sa "machine différentielle" permettant d'élever un nombre à la puissance n et en 1833, sa "machine analytique" où on retrouve les trois éléments essentiels de nos calculateurs: un organe d'introduction des données (cartes perforées ou cadrans), un organe de sortie des résultats (cartes perforées, cadrans ou papier) et un organe de contrôle et de calcul qui utilisait des dispositifs mécaniques. Une mémoire était réalisée par l'intermédiaire de roues dentées tandis que les opérations à effectuer étaient introduites à l'aide de cartes perforées. La machine de Babbage, limitée par les possibilités techniques de l'époque (elle aurait demandé plus de 50000 pièces mobiles), restera cependant à l'état de plans, et ses idées seront presque toutes "redécouvertes" indépendamment dans les années 40.

L'ingénieur américain Herman Hollerith développe pour le recensement de 1890 une machine électromécanique, plus élémentaire que celle de Babbage, capable de trier des cartes perforées et de les compter. Les opérations étaient toujours mécaniques mais les commandes étaient réalisées par l'intermédiaire de relais électromagnétiques, actionnés par des impulsions électriques obtenues par contact entre des aiguilles et un bac de mercure sur lequel reposaient les cartes perforées. Hollerith fonde en 1896 la "Tabulating Machine Company" reprise en 1911 par Thomas Watson qui crée en 1924 la firme IBM (International Business Machine).

La guerre 1940-1945, suite à l'effort de plusieurs pays, va donner l'impulsion décisive à la naissance des premiers ordinateurs dignes de ce nom :

- les Z2 et Z3 de l'allemand Zuse prêts en 1939 et 1941
- la série des "Automatic Sequence Controlled Computer Mark" conçus par Howard Aiken. Le Mark I fonctionnera en 1944. Les temps de calcul étaient de 1/3 de seconde pour une addition, de 6 secondes pour une multiplication et de 11 secondes pour une division. Les organes de commande et de calcul utilisaient des relais électromagnétiques.
- l'ENIAC (1943-1946), destiné initialement au calcul de tables d'artillerie, de Prosper Eckert et John Mauchly, utilisait des tubes à vide. Les temps de calcul sont divisés par 1000 mais cette machine qui comporte 18000 tubes à vide, 1500 relais, 10000 condensateurs et 70000 résistances, prend une place considérable (270 m3 - 30 tonnes) et consomme 150 kw. Il fallait changer le cablage et les switches pour modifier le programme. L'ENIAC était une machine décimale dont les entrées-sorties et la mémoire auxiliaire étaient réalisées par cartes perforées.

Dès 1944, des théoriciens de Princeton tels que J. Von Neumann, A.Buks et H. Goldstine se penchent sur les problèmes logiques posés par la réalisation des ordinateurs et établissent les bases des futurs développements, notamment l'utilisation du binaire, les notions de programmes stockés en mémoire, d'ordinateurs à usages multiples ...

L'EDSAC (Cambridge University 1949) et l'EDVAC (University of Pennsylvania 1950) utilisent des mémoires à ligne de retard à mercure (inventées par Eckert) qui permettent de stocker programmes et nombres sous forme digitale.

Vers 1951, apparaît le premier UNIVAC (Eckert et Mauchly) utilisant des diodes à cristal et des bandes magnétiques comme mémoire de masse et l'IAS (Von Neumann) où une mémoire à tube de Williams (1947) permet l'accès parallèle à tous les bits. C'est une machine binaire où les instructions modifiables comportent une adresse.

Entre 1953 et 1960 apparaissent de nombreuses innovations (2ème génération d'ordinateurs): les tores de ferrite utilisés comme mémoire (d'après les travaux de J.W. Forrester en 1951 au MIT), les circuits imprimés, les disques magnétiques, les transistors (inventés en 1948),... Les vitesses de traitement et les capacités de mémoire augmentent considérablement. Les "ordinateurs" (mot inventé en 1953 par M. Perret pour traduire "computer") acquièrent une plus grande sécurité de fonctionnement et une facilité d'emploi permettant d'effectuer des tâches de plus en plus vastes pour un nombre plus important d'utilisateurs. La production en série commence et les coûts de production diminuent rapidement.

En 1963, les circuits imprimés sont remplacés par des circuits intégrés (3ème génération). Les équipements se miniaturisent et on voit apparaître les mini- et micro-ordinateurs. Les vitesses d'exécution diminuent: quelques dizaines de picosecondes (1 picoseconde=0.000000000001 s = 10^{-12} s).

L'ordinateur commence à se banaliser. En 1963, il n'est déjà plus un matériel extraordinairement coûteux, même s'il est encore très cher. Il n'est plus en la possession exclusive de quelques centres de recherche aux énormes budgets, déjà de nombreuses entreprises le possèdent. Il est devenu fiable, rapide et performant.

En prenant comme référence la multiplication de deux nombres tenant chacun dans un mot, on constate une division du temps de calcul par 100 tous les 7 ans. Cette amélioration des performances s'accompagne d'une diminution des prix, d'un facteur 100 tous les 10 ans environ. Cette diminution continue aujourd'hui où, pour quelques dizaines de milliers de dinars, on trouve des ordinateurs, dits microordinateurs, qui, en dépit de leur petite taille, sont bien

plus puissants que ne le fut, il n'y a guère plus de 30 ans, l'ENIAC, malgré son poids de plusieurs tonnes. C'est que la microélectronique a entraîné à son tour, au cours des ans, une diminution considérable de la taille des ordinateurs et, ce qui est bien plus curieux, selon une tendance voisine de celles que nous avons observées dans les autres domaines. Les unités centrales réalisées sous de très faibles volumes qui forment ce qu'on appelle les microprocesseurs ont en effet été largement diffusées, en 1976, lorsque fut mise au point la fabrication de masse des composants de très haute intégration (Very Large Scale Integration ou VLSI). Cette technologie, encore aujourd'hui en pleine évolution, a permis par exemple de construire les unités centrales des IBM 4300 avec des plaquettes de silicium de 20 millimètres carrés, ayant 704 circuits logiques.

Notons d'ailleurs qu'il ne faut pas confondre microprocesseur et microordinateur. Un microordinateur est construit autour d'un microprocesseur, mais les unités d'entrée et de sortie, écrans de visualisation, claviers, imprimantes, ..., ne pouvant être miniaturisés, le volume d'un microordinateur est essentiellement occupé par ces dispositifs qui en sont la partie la plus coûteuse.

L'amélioration du rapport prix/performance a permis aussi de trouver de nouveaux utilisateurs et de construire des ordinateurs mieux adaptés aux besoins des utilisateurs. C'est ainsi que la gestion (paye et facturation, applications administratives et bancaires, réservations, ...) devient le domaine principal d'utilisation des ordinateurs.

2- Les langages :

Parallèlement aux modifications techniques, les moyens de communiquer à l'ordinateur ce qui lui est demandé de faire ont fortement changé. On appelle langages de première génération les langages-machine ou "codes-machine" utilisés initialement (1945). Un programme en "code-machine" est une suite d'instructions élémentaires, composées uniquement de 0 et de 1, exprimant les opérations de base que la machine peut physiquement exécuter: instructions de calcul (addition, soustraction, ...) ou de traitement ("et" logique, ...), instructions d'échanges entre la mémoire principale et l'unité de calcul ou entre la mémoire principale et une mémoire externe, des instructions de test qui permettent par exemple de décider de la prochaine instruction à effectuer.

Voici en code machine de l'IBM 370 l'ordre de charger 293 dans le registre "3":

01011000 0011 00000000000100100100

=> charger =3 =293

Ce type de code binaire est le seul que la machine puisse directement comprendre et donc réellement exécuter. Tout programme écrit dans un langage évolué devra par conséquent être d'abord traduit en code-machine avant d'être exécuté.

La deuxième génération est caractérisée par l'apparition de langages d'assemblage (1950) où chaque instruction élémentaire est exprimée de façon symbolique. Un programme dit "assembleur" assure la traduction en code exécutable. L'instruction de l'exemple précédent s'écrira :

constante X = 293 charger X R3

La troisième génération débute en 1957 avec le 1er langage dit évolué: FORTRAN (acronyme de mathematical FORmula TRANslating system). Apparaissent ensuite ALGOL (ALGOrithmic Language en 1958), COBOL (COmmon Business Oriented Language en 1959), BASIC (Beginner's All-purpose Symbolic Instruction Code en 1965), Pascal (1968), ... Les concepts employés par ces langages sont beaucoup plus riches et puissants que ceux des générations précédentes et leur formulation se rapproche du langage mathématique. Il existe deux méthodes pour les rendre exécutables :

- **la compilation** : l'ensemble du programme est globalement traduit par un autre programme appelé compilateur et le code-machine produit est optimisé. Ce code-machine est ensuite exécuté autant de fois qu'on le veut.
- **l'interprétation** : un programme (interpréteur) décode et effectue une à une et au fur et à mesure les instructions du programme-source.

La quatrième génération qui commence au début des années 80 devait mettre l'outil informatique à la portée de tous, en supprimant la nécessité de l'apprentissage d'un langage évolué. Ses concepts fondamentaux sont "convivialité" et "non-procéduralité" (il suffit de "dire" à la machine ce qu'on veut obtenir sans avoir à préciser comment le faire). Cet aspect a été rencontré avec les langages du type Visual qui prennent en charge l'élaboration de l'interface graphique.

3- Le codage :

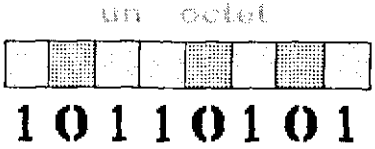
3.1- La numérotation binaire :

L'informatique est entièrement construite sur la logique « binaire », c'est à dire sur la numération en base « 2 ». Dans cette base il n'y a que DEUX chiffres, le « 0 » et le « 1 » !.

On appelle chiffre binaire le zéro ou le un. Les anglais disent ***Binary digiT ou bit*** pour parler d'un chiffre binaire zéro ou un. Un nombre à 4 bits sera donc une succession de 4 chiffres binaires soit par exemple 1001 qui est la représentation en binaire du chiffre 9 ($1x2^0 + 0x2^1 + 0x2^2 + 1x2^3$, c'est-à-dire 8+1 qui renvoie comme résultat le chiffre 9).

Du moment, qu'on ne peut pas représenter tous les caractères sur un bit (lettres alphabétiques majuscules et minuscules, chiffres, espace, caractères de ponctuation, ...), de telle sorte que les codes soient uniques (deux caractères différents ne peuvent avoir le même code), on a donc utilisé une grandeur du bit pour représenter les caractères : l'octet qui permet de stocker simultanément 8 bits. Par exemple, le mot "BIT" occupe 3 octets (il est composé de 3 lettres) => le mot "BIT" occupe 24 bits (3 x 8).

La figure ci-contre illustre la représentation d'un tel objet (l'octet).

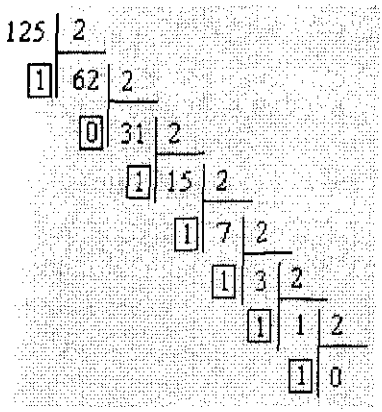


La valeur qu'il représente, exprimée en base 10 (le système de numérotation qu'on utilise dans la vie quotidienne), est **181**, le résultat de l'expression : $1x2^0 + 0x2^1 + 1x2^2 + 0x2^3 + 1x2^4 + 1x2^5 + 0x2^6 + 1x2^7$ (1 + 0 + 4 + 0 + 16 + 32 + 128).

3.2- Changement de base et programmation :

Soit N un nombre représenté dans la numération décimale : nous souhaitons connaître son équivalent en numération binaire. Pour cela il suffit de savoir effectuer la division entière de N par 2, (on obtient un quotient Q et un reste R) et de recommencer l'opération sur le quotient obtenu jusqu'à ce que le quotient soit nul. Voyons les opérations dans le détail :

Par exemple, pour écrire en binaire le nombre 125, il faut procéder comme suit :



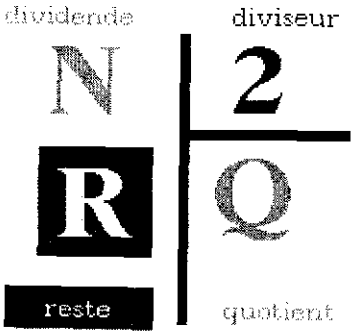
En effectuant la **division entière** de **125** par **2**, le **quotient** est **62** et le **reste** est **1**

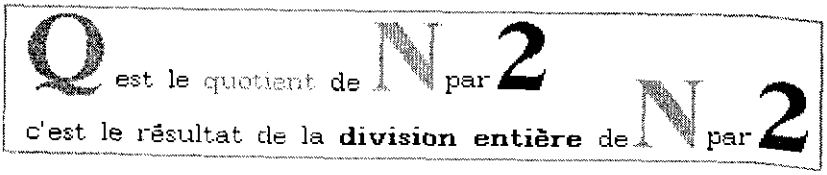
Autrement dit :
Si N = 125 on trouve Q = 62 et R = 1

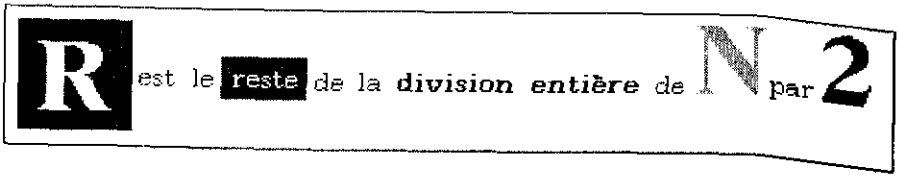
On divise ensuite 62 par 2, on trouve Q = 31 et R = 0, et ainsi de suite

Le nombre 125 s'écrit donc en binaire comme suit : **1111101** (il faut inverser l'ordre d'écriture, de haut en bas dans le schéma => de droite à gauche dans l'écriture de la représentation du nombre).

De façon plus générale, il est possible d'effectuer la **division entière** d'un nombre **N** (le **dividende**) par un nombre appelé le **diviseur** : ici, c'est **2** . Le résultat **Q** est appelé **quotient**, et **R** est le **reste** de la **division entière**...







3.3- Un autre système de numérotation :

En informatique, les données sont également codées en hexadécimal utilisant les dix chiffres habituels auxquels on ajoute les six premières lettres de l'alphabet en majuscules : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F. Dans ce système de numérotation, la lettre **A** représente le chiffre 10, la lettre **B** le chiffre 11, et ainsi de suite jusqu'à arriver à la lettre **F** qui représente le chiffre 15.

La base utilisée est 16 : la méthode est donc similaire au décimal en remplaçant 10 par 16.

Exemple : $FAC8 = F \times 16^3 + A \times 16^2 + C \times 16^1 + 8 \times 16^0$

$= 15 \times 16^3 + 10 \times 16^2 + 12 \times 16^1 + 8 \times 16^0$

$= 64\,200$

Voici pourquoi l'hexadécimal a été choisi : un processeur n'utilise seulement que des 0 et des 1 (binaire). Mais, c'est lourd à traduire : 1000 en hexadécimal => 1000000000000 en binaire.

On a choisi une écriture plus condensée : il s'agit de grouper les quartets (donc des blocs de 4 symboles consécutifs) du code binaire 4 par 4.

Exemple : le décimal 125 est codé en mémoire par l'octet 01111101

Séparons-le en 2 quartets : 0111 – 1101. Chaque quartet binaire représente un nombre décimal compris entre 0 et 15. Ici les deux quartets sont : 7 – 13 (tableau Bibinaire), ce qui donne 7D en hexadécimal

Application pratique : la couleur « vert d'eau » sur un moniteur est codée en hexadécimal 82DAB7 soit 8-2-13-10-11-7 (tableau Bibinaire) c'est à dire 1000-0010-1101-1010-1011-0111

=> en binaire 100000101101101010110111 en mémoire, ce qui correspond au décimal 8 575 671

3.4- Pourquoi l'ordinateur est-il binaire ?

La notion d'information utilisée en informatique est définie comme le support de connaissances pouvant être exprimées par des textes, des nombres, des images, des enregistrements sonores ou vidéo...

L'ordinateur pour représenter et mémoriser ces informations, utilise toujours le codage binaire (qui ne contient que les 2 valeurs : 0 et 1). Quand on parle de

"technologie numérique" ou "digitale", de "son numérique" ou de "photos numériques", cela fait référence au codage binaire qui est alors utilisé pour représenter les informations.

L'informatique a introduit massivement ces technologies dites "numériques", en opposition à la "technologie analogique" selon laquelle une grandeur (le volume d'un son par exemple) est représentée - par analogie - par une autre grandeur continue proportionnelle à la première (l'intensité d'un champ magnétique par exemple, dans un magnétophone à cassettes).

Le numérique a l'avantage sur l'analogique de ne pas craindre de petites altérations : un chiffre binaire vaut 0 ou 1 mais jamais 0,99. Une information numérisée (codée en binaire) peut ainsi être copiée à l'identique.

Rappelez-vous qu'un octet contient 8 chiffres binaires (en abréviation bit, pour BInary digiT): on a ainsi $2^8 = 256$ valeurs différentes pour un octet : 00000000, 00000001, 00000010, 00000011, 00000100 ...

Pour représenter moins de 256 informations différentes, on peut donc se contenter d'un octet : il suffit alors de choisir arbitrairement quel code attribuer à quelle information, c'est ce qui est fait pour les caractères. Bien que totalement arbitraire, un codage a cependant intérêt à être partagé par le plus grand nombre possible d'ordinateurs; en effet ce sont les codes qui sont transmis lors de communications entre ordinateurs, d'où l'intérêt des normalisations.

Pour représenter plus d'informations ou des informations plus complexes, la solution consiste à utiliser autant d'octets que nécessaire, rangés les uns après les autres dans la mémoire de l'ordinateur.

4- Stockage informatique :

4.1- Unités de mémoire :

Un ordinateur possède plusieurs types de mémoires. On s'intéresse ici aux supports qui permettent un stockage de l'information en quantité importante. Cette information peut être conservée entre deux utilisations de l'ordinateur. Cette mémoire, appelée **mémoire de masse**, consiste en des **disques ou bandes** que l'on peut conserver.

Rappelez-vous que l'unité d'enregistrement est le **BIT**, constitué par un chiffre zéro ou un chiffre un. Cependant, pour coder une information, on utilise l'**octet**, constitué de **huit BITS**.

1100 1011 0110 0101

La quantité d'informations présentes sur un disque se mesure en octets. Comme un fichier comporte un grand nombre d'octets, on utilise des multiples de l'octet:

- le **kilo-octet** : 1 ko = 2^{10} octets (le Ko représente donc les milliers)
- le **méga-octet** : 1 Mo = 2^{10} ko = 2^{20} octets (le Mo représente donc les millions)
- le **giga-octet** : 1 Go = 2^{10} Mo = 2^{20} ko = 2^{30} octets (le Go représente donc les milliards)

4.2- Les disquettes :

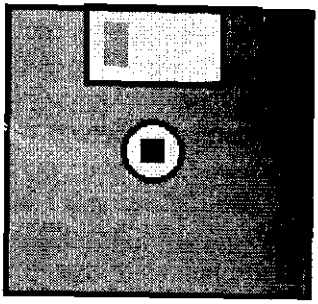
Les **disquettes** contiennent un disque en plastique souple protégé par un étui.

Les disquettes sont des **supports magnétiques.** Cela signifie que leur surface est aimantable et peut ainsi mémoriser des zéros et des uns. On distingue les types de disquettes suivants :

- Disquettes 5"1/4 (5 pouces un quart), aujourd'hui abandonnées,
- Disquettes 3"1/2, équipant les ordinateurs vendus actuellement (ci-contre).

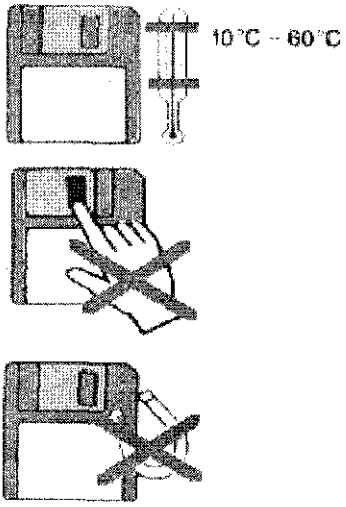
On voit au dos de la disquette :

- au centre, l'anneau d'entraînement,
- en haut, la partie métallique qui protège la disquette et qui s'ouvre au moment où la disquette est introduite dans son lecteur, (éviter d'ouvrir)
- en bas, le clapet de protection qui, abaissé (trou visible) , empêche toute écriture et donc l'effacement de la disquette. Pour pouvoir enregistrer dans la disquette un nouveau fichier, ou si vous souhaitez modifier son contenu, il faut lever le clapet de protection, afin de fermer le trou.



Précautions : (voir logos ci-contre)

- éviter de soumettre une disquette à des températures inférieures à 10°C ou supérieures à 60°C,
- éviter de les soumettre à un champ magnétique (aimant, électro-aimants, haut-parleurs ...)
- ne pas ouvrir la fenêtre (la partie métallique, en haut de la disquette), ce qui ferait entrer de la poussière, en plus on risque de la toucher avec les doigts.



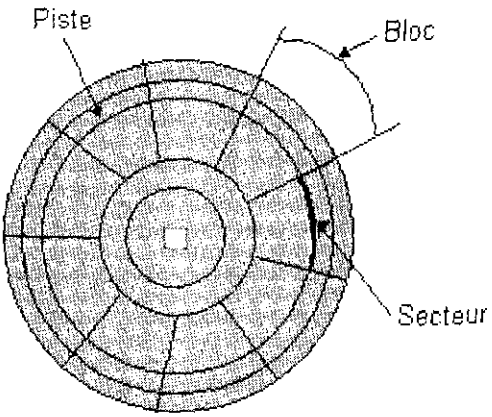
Pratique : Démonter une disquette défectueuse et observer le disque magnétique en plastique souple sur lequel s'inscrivent les données.

Les disquette sont **double face**, mais on en trouve deux types : DD (abandonnées) et HD :

- DD (Double Densité) : 80 **pistes**, 9 **secteurs** (720 Ko)
- HD (Haute Densité) : 80 **pistes**, 18 **secteurs** (1,44 Mo)

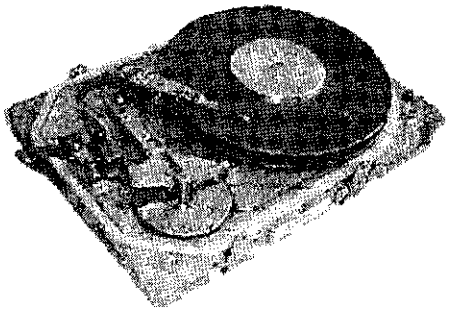
Chaque piste est donc divisée en 18 **blocs**, un secteur pouvant contenir 512 octets.

Par ailleurs, le début de la disquette est réservée à la "**FAT**", sorte de table des matières permettant de retrouver facilement tous les fichiers de la disquette. Si la piste 0 contenant la FAT est endommagée, c'est l'ensemble de la disquette qui l'est.



4.3- Les disque durs :

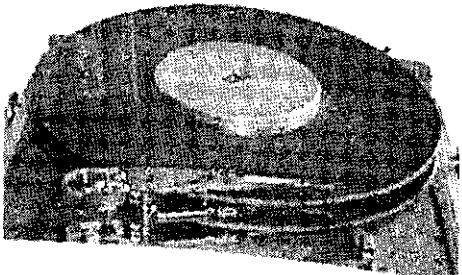
Le **disque dur**, comme son nom l'indique, est rigide, au contraire de la disquette. Sa surface est recouverte d'**oxyde de fer**. Le disque dur est capable de s'aimanter et par là même d'enregistrer les zéros et les uns.



Les disques durs ont vu leur capacité passer de 4 Mo à 80 Go en moins de 10 ans.

Le **temps d'accès** aux données est d'environ 10 ms pour un disque dur, ce qui signifie que le bras de lecture met 10 ms pour se positionner. Le débit d'un disque dur est fonction de la vitesse de rotation et de la densité des informations du disque dur. On améliore cette densité par la qualité des têtes de lecture et d'écriture.

Pour des raisons d'encombrement et de rapidité de lecture, le disque dur comporte plusieurs plateaux superposés avec une tête de lecture (et d'écriture) sur chaque face. Lorsque le disque tourne, la tête de lecture frôle la surface du disque à moins d'un



micromètre pour en lire les données.

Les poussières pourraient occasionner des dégâts en détruisant localement la surface du disque. Le disque dur est dans une boîte bien étanche.

Comme pour la disquette, le disque dur est **formaté** pour y délimiter les **pistes**, secteurs et **blocs** qui permettent de localiser les données. Mais, aujourd'hui les ordinateurs sont livrés avec des disques durs formatés et stockant même des programmes, tel qu'un système d'exploitation. Les disquettes également sont vendues formatées et prêtes donc à être utilisées.

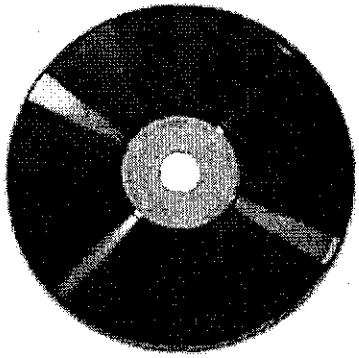
Les fichiers peuvent être enregistrés en plusieurs parties sur le disque dur. On dit alors qu'ils sont **fragmentés**. Si beaucoup de fichiers sont fragmentés, le temps de lecture est augmenté. Il est alors nécessaire de "**défragmenter**" le disque dur.

4.4- Les Cédéroms :

Les cédéroms (CD-Rom) ne sont pas, contrairement aux disquettes et disques durs, des supports magnétiques.

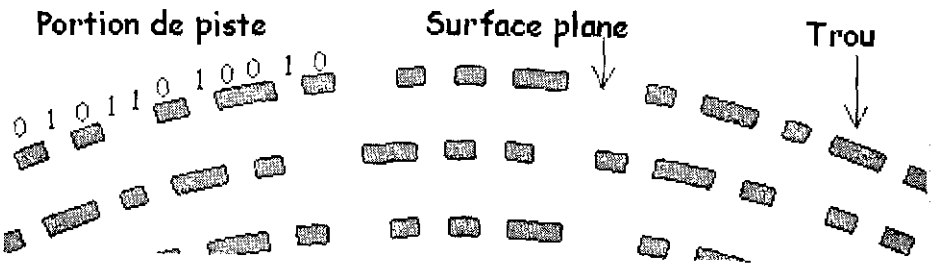
Ils comportent une piste unique. La zone où se trouve la piste, large de 37 mm, décompose la lumière.

Entre deux tours successifs de la piste, on trouve une distance de 1,6 µm.



La surface du CD-Rom est parcourue par une **piste de lecture** qui comporte une alternance de **trous** et de surfaces planes.

La lecture se fait par un petit faisceau **LASER**. Ce dernier est réfléchi s'il arrive sur une portion plane de la piste (1) et dispersé s'il parvient sur un trou (0). La surface est métallisée pour bien réfléchir le faisceau.



La piste a une largeur de 0,5 µ m et chaque bit codé correspond à un trou (0) ou une surface lisse (1) de 0,83 µ m.

Les cédéroms possèdent une capacité de 650 Mo ou 700 Mo. Ils peuvent donc stocker une masse importante d'information.

4.5- Les DVD :

Supplantant la disquette, le CD-Rom s'est imposé comme le support de stockage de référence. Mais alors que les vitesses de gravure semblent avoir atteint leur limite, les besoins en espace deviennent de plus en plus importants. Le successeur logique du CD devrait être le DVD, inscriptible et réinscriptible. Mais alors que les formats CD (CD audio, CD-R, CD-RW, VCD) sont parfaitement standardisés et reconnus par l'ensemble des fabricants, ceux concernant les DVD ne sont absolument pas figés.

Pourtant, le DVD-Forum, l'organisme de réglementation et de nomalisation du monde du DVD, a labellisé trois formats pour la gravure de DVD: DVD-R (inscriptible), DVD-RW (réinscriptible) et le DVD-Ram (réinscriptible). Ce dernier étant dédié à la sauvegarde en entreprise. Cela n'a toutefois pas empêché Hitachi, Samsung et Matsushita d'essayer de l'introduire sur le marché des graveurs de salon et des caméscopes à ce format. En même temps, une dizaine de constructeurs, conduits par Philips et rejoints par Microsoft, entrent en dissidence et proposent un autre standard, le DVD+RW.

DVD-R / RW :

Le DVD-R et le DVD-RW sont deux des formats validés par le DVD-Forum. Techniquement, la couche d'enregistrement d'un DVD-RW se compose, en fonction de la marque, d'un alliage d'argent et d'indium ou de germanium. Le DVD-R se voit doté d'une couche de polycarbonate. Comme un CD-RW, un DVD-RW autorise jusqu'à 1 000 enregistrements et conserve les données pendant 100 ans, en théorie.

Un support au format DVD-RW affiche une vitesse de lecture de 1x (soit un débit de 1 385 ko/seconde). Ainsi, remplir les 4,37 Go de données d'un DVD vierge prend une heure, contre 30 minutes pour un DVD-R qui offre un taux de tranfert de 2x. Enfin, ce format offre une large compatibilité. Le DVD-R / RW est reconnu par environ 90% des lecteurs de DVD-Rom et des platines de salon.

DVD+R / RW :

Dernier arrivé sur le marché, le DVD+RW est très proche physiquement et chimiquement du DVD-RW. Ainsi, il se compose lui aussi d'une couche d'enregistrement composée d'un alliage d'argent et d'indium ou de germanium. Les différences chimiques et physiques entre les deux formats sont minimes (profondeur de gravure).

En revanche, la vitesse de gravure de 2,4x autorise l'enregistrement de 4,37 Go de données en 25 minutes. Les opérations de gravure étant plus rapides, le format DVD+RW s'avère plus agréable à utiliser. Le choix de graveurs l'exploitant est plus important.

Néanmoins, il ne faut pas oublier que les premiers graveurs de DVD+RW sont sortis 6 à 8 mois après le premier graveur de DVD-R/RW. L'avantage de la vitesse n'est donc que temporaire. Car le DVD+RW présente aussi son lot de défauts. Parfaits pour réaliser des essais, des masters avant une gravure finale, ils sont aussi plus chers que les médias inscriptibles une seule fois.

DVD-Ram :

Le DVD-Ram est le premier type de DVD réinscriptible à avoir été normalisé. A l'origine dédié au monde professionnel, ce média dispose de nombreux avantages. Il est réinscriptible 100 000 fois contre 1 000 pour le DVD-RW et le DVD+RW. De plus, il dispose d'un caddie protecteur, augmentant ainsi sa durée de vie. Enfin, le DVD-Ram existe en plusieurs versions: 2,6 Go, 3,9 Go, 4,7 Go et même 9,4 Go.

Ce media est véritablement une excellente alternative aux produits de stockage magnéto-optiques déjà utilisés dans le monde de l'entreprise. Malheureusement, il n'est pas non plus exempt de défauts. Tout d'abord, il s'agit du plus cher d'entre tous. Enfin, sa compatibilité est très limitée. Il doit être sorti de son caddie pour être lu ailleurs que dans un lecteur dédié.

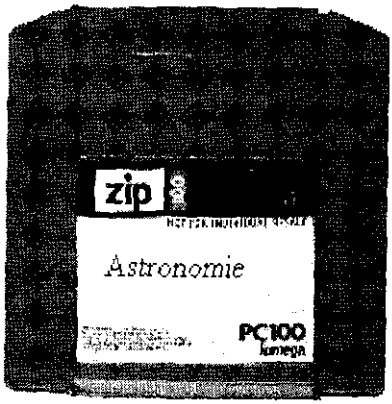
La question de la compatibilité se pose pour la lecture de disque avec un autre lecteur/graveur que celui qui a servi à le fabriquer. En particulier, avec un périphérique non informatique.

La notion de compatibilité est essentielle pour ceux dont l'utilisation d'un graveur de DVD se limite à la création de DVD-Vidéo. Le problème de compatibilité n'est pas posé avec la même intensité, si l'usage principal de votre graveur est le stockage de données purement informatique, qu'importe que le format DVD-Ram ne soit lisible que par un lecteur/graveur de DVD de ce type, puisqu'en général vos disques n'ont pas vocation à être lus ailleurs que sur le PC, ni même par un autre lecteur.

4.6- Les disquettes ZIP :

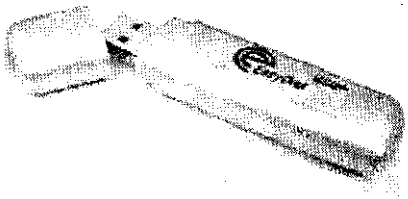
Les disquettes de 1.44 Mo sont bien souvent insuffisantes pour transférer des données d'un ordinateur à l'autre.

Les lecteurs "**zip**" fonctionnent avec des disquettes de 100 Mo ou 250 Mo. Ces lecteurs peuvent être externes et donc transportables. Les disquettes peuvent être effacées comme des disquettes ou disques durs habituels.



4.7- Les mémoires flash (ou disques flash) :

Une mémoire flash, appelée aussi disque flash ou EasyDisk, est une unité de stockage USB pour stocker et transporter des données et applications. Léger et compacte, la mémoire flash peut tenir dans votre poche!



elle ne nécessite pas de câble, adaptateur de courant, ou de piles.

La mémoire flash est reconnue comme disque amovible lorsqu'elle est connectée sur le port USB. Elle est réellement plug-and-play sur les systèmes d'exploitation supportant l'USB 1.1, comme Windows ME, 2000, et XP. Windows 98 nécessite l'installation d'un driver afin de fonctionner normalement.

La mémoire flash utilise le stockage 'durable solid state', étant ainsi résistant aux chocs.

La mémoire flash est fournie avec un cable prolongateur USB pour faciliter son utilisation sur des systèmes dont le port USB est difficile d'accès. Elle est également fournie avec une attache qui permet de l'accrocher sur un sac ou votre ceinture.

Spécifications :

Capacités	16Mo, 32Mo, 64Mo, 128Mo, 256Mo, 512Mo
Matériel	PC avec une interface USB
Système d'Exploitation (OS)	Windows 98/SE, Windows ME, Windows 2000, Windows XP, Mac OS 9.X / Mac OS X
Drivers	Requis uniquement avec Windows 98/SE
Alimentation	Pas d'alimentation requise - alimentation au travers du port USB (4.5V à 5.5V)
Interface USB	Universal Serial Bus 1.0/1.1 (USB 1.0/1.1)
Courant (actif)	< 50mA
Courant (Veille)	< 300uA
Rétention de Données	Supérieur à 10 ans
Indicateur lumineux LED	LED allumée: Connecté et fonctionnant Clignotant: lecteur et écriture de données
Vitesse de lecture	950Ko/S (3Mo/S avec le cache Windows)
Vitesse d'écriture	600Ko/S (2Mo/S avec le cache Windows)
Température (Actif)	-40 degrées Celsius à +70 degrées Celsius
Température (Inactif)	-50 degrées Celsius à +85 degrées Celsius
Humidité Relative (Actif)	5% à 95%
Humidité Relative (Inactif)	1% to 98%

Dimensions	Long x Larg x Haut: 81mm x 23mm x 12.5mm
Poids	15g approx

Applications :

- Accéder aux données, applications, fichiers MP3, et photos sur n'importe quel ordinateur possédant un port USB.
- Emporter des fichiers volumineux de son travail chez soi.
- Remplace les disquettes ZIP, disquettes 3"5, et CDRoms.
- Il n'y a pas besoin d'alimentation ou de câble.
- Résistant aux chocs.
- Une Led s'illumine au travers du boîtier indiquant le statut de la mémoire Flash.

EXERCICES

Exercice 1

1- Ecrire en base « 2 » la suite des nombres de 0 à 15 !

2- Pour stocker ces nombres, on veut fabriquer une « mémoire ».

Combien de bits faut-il prévoir pour stocker indifféremment l'un de ces nombres ?

Exercice 2

1- Combien de nombres différents peut-on stocker sur une mémoire de 8 bits ?

2- Codez sur 8 bits les valeurs 32, 64, 65, 125, 250 et 255.

Exercice 3

Voici des informations, telles que l'on pourrait les voir à l'intérieur de la mémoire d un ordinateur :

01000010010100100100000101010110010011110010000 0

01001100010001010101001100100000010100 00 01010010

01001111010100110010000000100001001000010010000 1

Décodez le "message" ci-dessus en considérant que les entiers naturels, les lettres alphabétiques et tout autre caractère sont codés sur 8 bits. Pensez à vous servir du code ASCII.

Exercice 4

Le noir est codé FFFFFF, ce qui correspond à quel nombre binaire ? à quel nombre décimal ? Justifier l'appellation 16 millions de couleurs pour certains de nos « micros » actuels.

Exercice 5

Un écran ancien comporte une palette de 256 couleurs. Combien faut – il d'octet pour les représenter ?

CORRIGES DES EXERCICES

Exercice 1

1- Ecriture en base « 2 » de la suite des nombres de 0 à 15 :

Nombre en décimal	Représentation en binaire
0	0
1	01
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

2- Pour stocker ces nombres, il faut 4 bits (la plus grande valeur est 15 qui s'écrit en binaire 1111 : ce nombre nécessite donc 4 bits).

Exercice 2

1- Sur une mémoire de 8 bits, on peut stocker 2^8, donc 256 nombres différents.

2- Codification des valeurs sur huit bits :

Nombre en décimal	Représentation en binaire
32 (32 = 2^5, donc 1 et 5 zéros à droite, puis on rajoute 2 zéros à gauche du 1 pour avoir au total 8 bits)	00100000
64 (64 = 2^6 => 1 suivi de 6 zéros à droite et 1 zéro à gauche du 1)	01000000
65 (65 = 64 +1)	01000001
125 (125 = 64 + 32 + 16 + 8 + 4 + 1 => 125 = $2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$)	01111101
250 (250 = 128 + 64 + 32 + 16 + 8 + 2 => 250 = $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$)	11111010
255 (255 = 128 + 64 + 32 + 16 + 8 + 4 + 2 +1 => 255 = $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$)	11111111

Exercice 3

Il faut regrouper par groupes de 8 bits pour trouver la valeur en décimal de chaque octet, ce qui nous permet de décoder chacune des informations. Evidemment, il faut utiliser la table des codes ASCII suivante pour trouver le code de chaque groupe de 8 bits.

Car	Dec	Hex	Car	Dec	Hex
Nul	0	0	SP	32	20
SOH	1	1	!	33	21
STX	2	2	"	34	22
ETX	3	3	#	35	23
EOT	4	4	$	36	24
ENQ	5	5	%	37	25
ACK	6	6	&	38	26
BEL	7	7	'	39	27
BS	8	8	(	40	28
HT	9	9	)	41	29
LF	10	A	*	42	2A
VT	11	B	+	43	2B
FF	12	C	,	44	2C
CR	13	D	-	45	2D
SO	14	E	.	46	2E
SI	15	F	/	47	2F
DLE	16	10	0	48	30
DC1	17	11	1	49	31
DC2	18	12	2	50	32
DC3	19	13	3	51	33
DC4	20	14	4	52	34
NAK	21	15	5	53	35
SYN	22	16	6	54	36
ETB	23	17	7	55	37
CAN	24	18	8	56	38
EM	25	19	9	57	39
SUB	26	1A	:	58	3A
ESC	27	1B	;	59	3B
FS	28	1C	<	60	3C
GS	29	1D	=	61	3D
RS	30	1E	>	62	3E
US	31	1F	?	63	3F

Car	Dec	Hex	Car	Dec	Hex
@	64	40	`	96	60
A	65	41	a	97	61
B	66	42	b	98	62
C	67	43	c	99	63
D	68	44	d	100	64
E	69	45	e	101	65
F	70	46	f	102	66
G	71	47	g	103	67
H	72	48	h	104	68
I	73	49	i	105	69
J	74	4A	j	106	6A
K	75	4B	k	107	6B
L	76	4C	l	108	6C
M	77	4D	m	109	6D
N	78	4E	n	110	6E
O	79	4F	o	111	6F
P	80	50	p	112	70
Q	81	51	q	113	71
R	82	52	r	114	72
S	83	53	s	115	73
T	84	54	t	116	74
U	85	55	u	117	75
V	86	56	v	118	76
W	87	57	w	119	77
X	88	58	x	120	78
Y	89	59	y	121	79
Z	90	5A	z	122	7A
[	91	5B	{	123	7B
\	92	5C	|	124	7C
]	93	5D	}	125	7D
^	94	5E	~	126	7E
_	95	5F	Del	127	7F

Table ASCII étendue

Car (Times)	Dec	Hex	Car (Times)	Dec	Hex
Ä	128	80	†	160	A0
Å	129	81	°	161	A1
Ç	130	82	¢	162	A2
É	131	83	£	163	A3
Ñ	132	84	§	164	A4
Ö	133	85	•	165	A5
Ü	134	86	¶	166	A6
á	135	87	ß	167	A7
à	136	88	®	168	A8
â	137	89	©	169	A9
ä	138	8A	™	170	AA
ã	139	8B	´	171	AB
å	140	8C	¨	172	AC
ç	141	8D	≠	173	AD
é	142	8E	Æ	174	AE
è	143	8F	Ø	175	AF
ê	144	90	∞	176	B0
ë	145	91	±	177	B1
í	146	92	≤	178	B2
ì	147	93	≥	179	B3
î	148	94	¥	180	B4
ï	149	95	µ	181	B5
ñ	150	96	∂	182	B6
ó	151	97	∑	183	B7
ò	152	98	∏	184	B8
ô	153	99	π	185	B9
ö	154	9A	∫	186	BA
õ	155	9B	ª	187	BB
ú	156	9C	º	188	BC
ù	157	9D	Ω	189	BD
û	158	9E	æ	190	BE
ü	159	9F	ø	191	BF

Car (Times)	Dec	Hex	Car (Times)	Dec	Hex
¿	192	C0	‡	224	E0
¡	193	C1	·	225	E1
¬	194	C2	‚	226	E2
√	195	C3	„	227	E3
ƒ	196	C4	‰	228	E4
≈	197	C5	Â	229	E5
Δ	198	C6	Ê	230	E6
«	199	C7	Á	231	E7
»	200	C8	Ë	232	E8
…	201	C9	È	233	E9
(espace)	202	CA	Í	234	EA
À	203	CB	Î	235	EB
Ã	204	CC	Ï	236	EC
Õ	205	CD	Ì	237	ED
Œ	206	CE	Ó	238	EE
œ	207	CF	Ô	239	EF
–	208	D0		240	F0
—	209	D1	Ò	241	F1
“	210	D2	Ú	242	F2
”	211	D3	Û	243	F3
‘	212	D4	Ù	244	F4
’	213	D6	ı	245	F5
÷	214	D6	ˆ	246	F6
◊	215	D7	˜	247	F7
ÿ	216	D8	¯	248	F8
Ÿ	217	D9	˘	249	F9
/	218	DA	˙	250	FA
¤	219	DB	˚	251	FB
‹	220	DC	¸	252	FC
›	221	DD	˝	253	FD
fi	222	DE	˛	254	FE
fl	223	DF	ˇ	255	FF

On trouve donc que les valeurs des informations en décimal sont respectivement :

66-82-65-86-79-32

76-69-83-32-80-82

79-83-32-33-33-33

Ainsi, le codage du message est comme suit :

01000010-01010010-01000001-01010110-01001111-00100000
=> BRAVO

01001100-01000101-01010011-00100000-01010000-01010010
=> LES PR

et, enfin, 01001111-01010011-00100000-00100001-00100001-00100001
=> OS !!!
Le message complet est donc BRAVO LES PROS !!!

Exercice 4

- Le noir qui est codé FFFFFF correspond au nombre binaire suivant :

11111111	11111111	11111111	11111111	11111111	11111111
F	F	F	F	F	F

- Pour trouver, en décimal, le nombre qui correspond au code en hexadécimal FFFFFF, il faut effectuer le calcul suivant :

  F x 165 + F x 164 + F x 163 + F x 162 + F x 161 + F x 160

  Ce qui donne : 15 x 1 048 576 + 15 x 65 536 + 15 x 4 096 + 15 x 256 + 15 x 16 + 15 x 1

  On trouve donc : 16 777 215

- Le résultat précédent (16 777 215) justifie l'appellation 16 millions de couleurs ; puisque la codification avec 6 positions hexadécimales nous permet de coder des nombres de 0 (000000 en hexa) à 16 777 215 (FFFFFF), c'est-à-dire plus de 16 millions de couleurs.

Exercice 5

L'écran comporte une palette de 256 couleurs. Il faut donc un octet (8 bits) pour les représenter. Les codes sont de 00000000 (qui représente la valeur zéro) à 11111111 (qui représente la valeur 255).

Introduction à l'algorithmique

Pourquoi apprendre l'algorithmique pour apprendre à programmer ?

Parce que l'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage.

1- Historique :

Un algorithme est la «spécification d'un schéma de calcul, sous forme d'une suite [finie] d'opérations élémentaires obéissant à un enchaînement déterminé» (Encyclopedia Universalis). Cependant le terme d'algorithme ne concerne pas que l'informatique, et la notion d'algorithme a précédé celle d'ordinateur.

On connaît depuis l'antiquité des algorithmes sur les nombres. Par exemple, l'algorithme d'Euclide qui permet de calculer le PGDC de 2 nombres entiers. Mais, ce n'est qu'au neuvième siècle que la notion d'algorithme a été introduite. Son nom vient d'un mathématicien musulman : ***El Khawarismi***.

2- Notion d'algorithme :

Dans la vie courante, un algorithme peut, par exemple, prendre la forme :

- d'une recette de cuisine,
- d'un mode d'emploi,
- d'une notice de montage,
- d'une partition musicale,

- d'un itinéraire routier qu'on explique à un touriste perdu,
- ...

Un algorithme décrit un traitement sur un certain nombre fini de données. C'est la composition d'un ensemble fini d'étapes, qui exécutée correctement, conduit à un résultat donné et voulu.

Chaque étape étant formée d'un nombre fini d'opérations, appelées instructions, dont chacune est :

- définie de façon rigoureuse et non ambiguë;
- effective, c'est à dire pouvant être effectivement réalisée par une machine. Par exemple, la division entière est une opération effective, mais pas la division avec un nombre infini de décimales.

Un algorithme informatique se ramène toujours à la combinaison de 4 briques de base :

- l'affectation de variables
- la lecture / écriture
- les tests
- les boucles

Notons qu'un algorithme peut être constitué de quelques instructions, comme il peut être composé de milliers d'instructions. Aussi, la taille ne reflète pas toujours la complexité d'un algorithme. En fait, de longs algorithmes peuvent être assez simples, et de petits très compliqués.

Enfin, notons également qu'un programme est généralement la description d'un algorithme dans un langage accepté par l'ordinateur. Alors qu'un algorithme est indépendant du langage de programmation utilisé.

3- Utilité des algorithmes :

Pour traiter l'information, on a développé des algorithmes opérant sur des données non numériques

- les algorithmes de tri : permettent, par exemple, de ranger par ordre alphabétique une suite de noms,
- les algorithmes de recherche d'une chaîne de caractères dans un texte

- les algorithmes d'ordonnancement, qui permettent de décrire la coordination entre différentes tâches, nécessaire pour mener à bien un projet.

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

Non, l'algorithmique demande deux qualités :

- **Rigueur**

  Chaque fois qu'on écrit une série d'instructions, il faut se mettre à la place de la machine qui va les exécuter, pour vérifier si le résultat obtenu est bien celui escompté.

- **Intuition**

  - Aucune recette ne permet de savoir à priori quelles instructions permettront d'obtenir le résultat voulu.
  - Alors on est plus ou moins intuitifs, mais les réflexes, cela s'acquiert, et l'expérience finit par compenser largement des intuitions.

5- La forme générale d'un algorithme :

Algorithme NomAlgorithme	- profil (donne le nom de l'algorithme)
Début	- délimiteur de début
... actions	- les différentes actions
Fin	- délimiteur de fin.

Ainsi, l'algorithme suivant est valide :

```
Algorithme bonjour
Début
    Afficher( 'Salut Omar')
    Passer à la ligne
    Afficher( 'Ecrit moi. J'attend de tes nouvelles')
Fin
```

6- Caractéristiques d'un bon algorithme :

Un bon algorithme doit être :

- **Déterministe** :

  Toute exécution d'un algorithme sur les mêmes données donne lieu à la même suite d'opérations et au même résultat.

- Systématique
- **Bien structuré** (et si possible simple à comprendre)
- **Non ambiguë**
- **Juste**
- **Eventuellement élégant**.
- **Lisible** :
  - Retours à la ligne
  - Indentation
  - Commentaires
    - Courts
    - Se limitent à l'explication des idées essentielles ou des points délicats.
- **Pas trop long** (décomposition en modules);
- **Eviter les branchements**
- **Efficace**. Quelques critères d'efficacité :

- Temps d'exécution
- Quantité d'espace mémoire
- Quantité d'espace disque
- Quantité d'information à lire / écrire
- Quantité d'information à transférer
- Modulaire, s'il s'agit d'un algorithme qui peut être décomposé (voir ci-après la définition de la modularité).

7- La modularité :

La résolution d'un problème (= module) doit être découpée en sous-problèmes (= sous- modules) plus petits et plus simples à résoudre :

- C'est plus clair,
- Les sous-modules sont plus facilement réutilisables,
- Les sous-modules peuvent devoir être utilisés plusieurs fois par module. On appelle le sous- module à chaque fois que c'est nécessaire.
- Les modules doivent être aussi indépendants que possible.
- L'interface des différents modules avec l'extérieur doit être décrite précisément :
  - variables d'entrée- sorties,
  - variables globales utilisées et/ ou modifiées.
- Les liaisons entre les différents modules doivent être clairs :
  - quel module utilise quel autre module,
  - quels modules partagent une même variable globale.
- Le rôle de chaque module doit être explicité clairement,
  - soit sous forme de formules,
  - soit sous forme d'une phrase en langage naturel: on donne la spécification de ce module.

8- Place de l'algorithme dans la résolution d'un problème informatique :

La résolution d'un problème informatique se décompose en cinq phases :

- Analyse du problème
- Expression d'une solution en langage courant
- Expression d'une solution en pseudo-langage
- Tests et Vérification de l'adéquation de la solution
- Phase de traduction

Analyse du problème :

Bien comprendre l'énoncé du problème: il est inutile et dangereux de passer à la phase suivante si vous n'avez pas bien discerné le problème.

Expression du raisonnement :

Bien souvent, quelques lignes écrites en langage courant suffisent pour décrire l'essentiel du problème. L'intérêt de cette étape est qu'elle permet de vérifier rapidement que l'on se trouve sur la bonne voie. De plus, ces quelques lignes seront un support efficace lors de l'écriture de l'algorithme.

Il est donc toujours recommandé, lorsque vous écrivez un algorithme, de le commenter en spécifiant brièvement son rôle et ses objectifs. Entre autres, il faut :

- décrire ce que l'algorithme fait, sans détailler comment (rôle des commentaires souvent).
- le commentaire (ou les commentaires) doit être concis, au risque d'être imprécis
- écrire le commentaire en italique, afin de le distinguer des instructions.

Expression d'une solution en pseudo-langage :

Il peut arriver que plusieurs solutions répondent à un problème donné. Il faudra choisir la solution la plus judicieuse et rester cohérent jusqu'au bout.

Attention : la comparaison d'algorithmes n'a de sens que si les spécifications (buts et objectifs) sont les mêmes.

Tests et Vérification de l'adéquation de la solution :

Vérifier l'exactitude du comportement de l'algorithme, son bon déroulement. Si l'algorithme ne répond pas parfaitement à toutes les requêtes exprimées dans l'énoncé du problème, retournez à la phase n°1.

En général, la vérification consiste à prendre un exemple et lui appliquer les instructions de l'algorithme. Si le résultat renvoyé est faux, c'est qu'il y a un problème quelque part. Mais, si le résultat est bon on ne peut confirmer l'exactitude de l'algorithme.

Phase de traduction :

Mettre en oeuvre les algorithmes en les traduisant en un langage de programmation adapté à la machine utilisée.

9- Notion de pseudo-code (ou pseudo-langage) :

Aujourd'hui, on dispose d'une grande variété de langages de programmation. Certains langages très connus du grand public sont largement utilisés dans des domaines très divers (PASCAL, C, LISP, ADA, COBOL etc...).

On distingue généralement les langages de bas niveau (proches de la machine : Assembleur) et les langages évolués (dits de haut niveau).

De tout temps, les chercheurs ont essayé de mettre au point des langages permettant de se détacher le plus possible de la machine sur laquelle les programmes seront exécutés.

Certains algorithmes très anciens (Crible d'Erathostène par exemple) ont été décrits en utilisant un langage peu formalisé (proche du langage naturel).

D'un autre côté, un algorithme n'a d'intérêt que s'il peut être compris et utilisé par un grand nombre de programmeurs.

Il a donc fallu élaborer un langage de description suffisamment formel, pour permettre des implantations dans différents langages de programmation peu fastidieuses, et d'un niveau suffisant pour qu'il soit un outil de communication efficace. Un tel langage s'appelle pseudo-langage.

En résumé, l'avantage du pseudo-langage est qu'il permet d'écrire tout algorithme de façon formelle, c'est-à-dire suffisamment précise, tout en restant compréhensible pour l'ensemble des informaticiens.

La phase de programmation se trouvera nécessairement allégée, puisqu'elle se résumera à adapter l'ensemble des opérations décrites aux spécificités du langage utilisé.

10- Des problèmes sans solution :

Tous les problèmes ne se résolvent pas grâce à un algorithme. On distingue deux cas :

a- Complexité algorithmique exponentielle :

- Algorithmes qui ne permettent pas le traitement du problème dans un temps raisonnable
- Ressources nécessaires à leur exécution en temps et en mémoire trop importante

Exemple : le jeu d'échec :

- Possible de faire un programme calculant toutes les conséquences de tous les coups possibles => meilleur que tout joueur humain.
- Mais, il faudrait considérer de l'ordre de 10^{19} coups possibles pour décider de chaque déplacement. (10^{19} ms est de l'ordre de 300 millions d'années).
- Complexité trop importante => pas envisageable de mettre un tel algorithme en pratique.

b- Indécidabilité :

Parfois il n'existe aucun algorithme pour certains problèmes.

Exemple : le paradoxe du barbier

Dans une ville où les gens se font tous raser par le barbier et personne ne doit se raser lui-même. Mais, qui rase le barbier ?

Dans cette ville, il existe un seul barbier, et les autorités de cette ville ont décidé que toute personne habitant cette ville doit se faire raser la barbe.

Qui rase donc le barbier ?

- Soit il se fait raser par le barbier, donc il se rase lui- même
- Soit il ne se pas rase lui- même et, dans ce cas, il n'est pas rasé par le barbier,
- Donc, il doit se raser lui-même et il ne doit pas se raser lui-même.

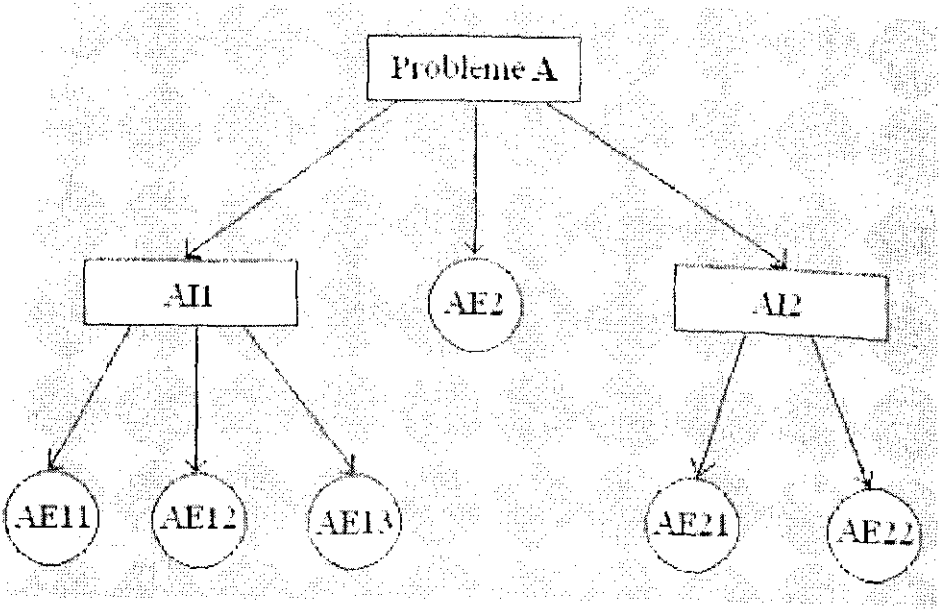
=> On doit donc conclure qu'une telle ville avec de tels habitants ne peut exister et, de la même manière, l'algorithme n'existe pas non plus.

11- Méthodes d'analyse :

11.1- Analyse descendante :

C''est une méthode de décomposition des problèmes.

Soit le problème A à résoudre



On a tout d'abord décomposé le problème A en deux actions intermédiaires (AI1 et AI2) et une action élémentaire AE2. AI1 est ensuite décomposé en trois actions élémentaires AE11, AE12 et AE13. De même, AI2 est ensuite décomposé en deux actions élémentaires AE21 et AE22.

Une telle situation peut sembler idéale, mais il y a une contrepartie: l'analyse descendante ne permettra que rarement de définir des sous-programmes identiques appelés de points différents.

L'analyse Top-Down (l'analyse descendante) doit être le guide général, mais il peut être profitable, dans le cas de traitements complexes, d'osciller quelque peu entre analyse descendante et montante (analyse ascendante, que nous allons définir dans le paragraphe suivant).

Les buts de l'analyse descendante sont les suivants :

- Obtenir une analyse et une programmation de haut en bas selon une décomposition arborescente qui autorise le fractionnement des

programmes en modules de taille facilement lisible (par exemple de la taille d'une page de listing, au maximum).

- Rendre possible la lecture des modules de haut niveau par des non-spécialistes.
- Faciliter la maintenance et l'extensibilité en donnant la possibilité de modifier une arborescence, sans affecter ce qui est au niveau supérieur.
- Diminuer les risques d'erreurs de programmation.
- Fournir aux programmeurs un mode commun de décomposition des problèmes.

11.2- Analyse ascendante :

Avec la méthode descendante appliquée de façon rigide, il est possible de rencontrer ou de résoudre plusieurs fois le même problème ou des problèmes voisins.

La méthode d'analyse ascendante consiste à identifier dès le début les actions élémentaires qu'il faudra savoir résoudre.

Cette méthode consiste donc à déterminer les objets (ou classes d'objets) de l'application, une classe d'objets étant caractérisée:

- par un ensemble de valeurs ou d'états (on parle aussi d'attributs ou de propriétés),
- et par un ensemble d'actions ou d'opérations sur ces objets d'autre part.

11.3- Les méthodes dirigées par les données :

Dans la méthode descendante, l'accent est mis sur la décomposition en actions. Les données manipulées et nécessaires sont affinées au fur et à mesure mais ne jouent pas un rôle stratégique dans la décomposition.

Dans un système de gestion de base de données (SGBD), le programmeur commence par décrire les données de l'application (parce que les données jouent le rôle central). Il utilise pour cela un langage de description de données (LDD) qui permet de décrire les objets ou les classes d'objets qu'il utilise ainsi que les associations entre classes d'objets et les contraintes d'intégrité éventuelles. On construit ainsi le modèle conceptuel des données (MCD). On écrit ensuite les programmes qui permettront la création et la mise à jour de la base de données en utilisant un langage de manipulation de données (LMD).

12- Evolution de la programmation impérative vers la programmation objet :

Dans ce paragraphe, nous allons définir les différents types de programmation. Il ne s'agit pas ici de donner des définitions assez complètes, puisque cela sort largement du cadre de cet ouvrage. Mais, il s'agit tout juste de vous donner une petite idée sur ces types de programmation.

1ère étape : la programmation procédurale

Dans ce type de programmation, on distingue les langages Algol, Pascal et le langage C. Elle se base sur les notion de types et de structures de contrôles.

2ème étape : la programmation modulaire

A un module correspond un sous-système, ce sous-système rend au reste de l'application un service. Pour l'utilisateur, il suffit de savoir ce que fait le module et comment on y accède. C'est le premier pas vers le composant logiciel réutilisable.

3ème étape : la programmation par type générique (ou abstraction)

Une abstraction fait ressortir les caractéristiques essentielles d'un objet (qui le distinguent de tous les autres genres d'objets) et donc procure des frontières conceptuelles rigoureusement définies par rapport au point de vue de l'observateur.

4ème étape : les objets

L'accent est mis sur la réutilisation et l'adaptation en utilisant les mécanismes d'héritage (ou de sous-typage) et la programmation par interfaces. On décrit le monde comme étant formé d'objets ainsi que les relations existant entre les objets.

Les variables

1- Notion de variable et de constante :

La programmation utilise des variables, mais en fait une variable c'est quoi ? Voici un exemple : si vous dîtes : a = 5, b = 7, c = a + b..., vous comprenez rapidement que c = 12, n'est-ce pas ? a, b et c sont des variables.

Pour l'ordinateur, il faut le prévenir (avant de commencer) que a, b et c existent et sont des nombres entiers. On appelle ça la déclaration des variables.

Une constante est une "variable" dont la valeur ne change jamais dans le programme. Ici, a et b peuvent être considérés comme constantes.

Le programme ressemblera donc à ça (plusieurs solutions) :

```
Programme addition1
Variables
   a : Entier
   b : Entier
   c : Entier
   (ou a, b, c : Entier)
Début
   a <--- 5
   b <--- 7
   c <--- a + b
Fin
```

```
Programme addition2
Variables
    a <--- 5, b <--- 7, c : Entier
Début
    c <--- a + b
Fin
```

```
Programme addition3
Constantes
    a = 5, b = 7
Variables
    c : Entier
Début
    c <--- a + b
Fin
```

La partie déclaration est une partie avant le programme pour définir les variables et les constantes. Dans les programmes 2 et 3, a et b obtiennent leurs valeurs (on dit qu'elles sont initialisées) dans la partie déclaration.

Dans la suite de ce chapitre, nous aborderons plus en détails la notion de variable, la déclaration des variables, les types de variables, l'instruction d'affectation, les instructions de lecture et d'écriture et les opérateurs arithmétiques, alphanumérique et logiques.

2- A quoi servent les variables ?

Dans un programme, on a besoin de stocker provisoirement des valeurs

- Données issues du disque dur, frappées au clavier ou obtenues par le programme (intermédiaires ou définitives).

- Ces données peuvent être de plusieurs types : nombre, texte,...

Dès que l'on a besoin de stocker une information dans un programme, on utilise une **variable**.

On peut simuler une variable à une boîte, repérée par une étiquette. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette, c'est-à-dire son nom.

Les variables sont des éléments fondamentaux d'un programme, elles décrivent l'état du système à un instant donné. Affecter une valeur à une variable consiste à lui associer cette valeur jusqu'à une nouvelle affectation. Si une variable doit toujours avoir la même valeur, nous utilisons une constante.

3- Déclaration des variables :

Avant de pouvoir utiliser une variable il faut la créer et lui coller une étiquette

- C'est la **déclaration des variables**.
- Se fait tout au début de l'algorithme,
- Avant même les instructions proprement dites.

Le **nom** de la variable obéit à des impératifs dépendant du langage, mais en général :

- Commence impérativement par une lettre.
- Peut comporter lettres et chiffres,
- Mais pas la plupart des signes de ponctuation, les caractères spéciaux et les espaces en particulier.

4- Types de variables :

Une fois la boîte créée (emplacement mémoire réservé), il faut préciser ce que l'on veut mettre dedans, car cela influence :

- La taille de la boîte (de l'emplacement mémoire)
- Le **type** de codage utilisé.
  - Type numérique
  - Type alphanumérique (type caractère)
  - Type booléen

4.1- Types numériques classiques :

Une variable de type numérique est destinée à recevoir des nombres (cas le plus fréquent). Le type de codage (= 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 offrent plusieurs types numériques, généralement :

Type	Plage des valeurs
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	-3,40E38 à -1, 40E- 45 pour les valeurs négatives
	1, 40E- 45 à 3, 40E38 pour les valeurs positives
Réel double	1, 79E308 à -4, 94E- 324 pour les valeurs négatives
	4,94E- 324 à 1, 79E308 pour les valeurs positives

Pourquoi ne pas déclarer toutes les variables numériques en réel double?

- A cause du principe de l'économie de moyens.
- Un bon algorithme marche tout 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.

Une déclaration algorithmique de variables aura ainsi cet en-tête :

Variable quantité **en Entier Long**

Variables prixHt, tauxTva, prixTtc **en Réel Simple**

4.2- Types non numériques :

En plus du type numérique, un langage de programmation met à votre disposition d'autres types, en général le type alphanumérique et le type booléen.

a- Type alphanumérique (ou type caractère) :

Une variable de type caractère stocke des caractères (lettres, signes de ponctuation, espaces, chiffres, ou une combinaison de ces caractères). Une série de caractères s'appelle une **chaîne** de caractères.

Une chaîne de caractères (ou un caractère) est toujours notée entre guillemets, parce que 423 peut représenter le nombre 423, ou la suite de caractères 4, 2, et 3, selon le type de variable qui a été utilisé pour le stocker. Les guillemets permettent d'éviter toute ambiguïté à ce sujet.

b- Type booléen :

Une variable de type booléen stocke uniquement les valeurs logiques VRAI ou FAUX.

5- L'instruction d'affectation :

5.1- Syntaxe et signification :

L'affectation de variable (= attribution d'une valeur) est la seule chose qu'on puisse faire avec une variable.

En algorithmique, cette instruction se note avec le signe <---. Ce signe signifie que la variable se trouvant à gauche reçoit la valeur (ou l'expression) qui se trouve à droite.

Exemple :

toto <--- 24

Attribue la valeur *24* à la variable *Toto* (*Toto* est donc une variable numérique).

On peut attribuer à une variable la valeur d'une autre variable, telle quelle ou modifiée.

Exemple :

tutu <--- *toto*

Signifie que la valeur de *tutu* est maintenant celle de *toto*. Cette instruction d'affectation ne modifie pas la valeur de *toto* : une instruction d'affectation ne modifie que ce qui est situé à gauche de la flèche.

Questions :

- *tutu* <--- *toto* + 4
  - si *toto* contenait 12, *tutu* vaut combien ?
  - et *toto* maintenant ?

  Réponse : *tutu* vaut 16 (12 + 4), alors que *toto* garde la même valeur (la valeur 12).

- *tutu* <--- *tutu* + 1
  - La valeur de *tutu* est-elle modifiée après exécution de l'instruction ?
  - Si *tutu* valait 6, il vaut combien maintenant ?

  Réponse :
  - La valeur de *tutu* est modifiée, puisque *tutu* est la variable située à gauche de la flèche.
  - Après exécution de l'instruction, *tutu* aura la valeur 7 (le résultat de l'addition 6 + 1).

5.2- Ordre des instructions :

L'ordre dans lequel les instructions sont écrites joue un rôle essentiel dans le résultat final. Considérons les deux algorithmes suivants :

```
Variable a en Entier
Début
   a <--- 34
   a <--- 12
Fin
```

```
Variable a en Entier
Début
   a <--- 12
   a <--- 34
Fin
```

Question :

Quelle est la valeur finale de A dans le premier cas ? Et dans le second ?

Réponse :

Dans le premier cas, A aura à la fin la valeur 12, et dans le second cas elle aura la valeur 34. Dans les deux cas, la seconde valeur écrasera la première.

5.3- Expressions :

Dans une instruction d'affectation, on trouve :

- à gauche de la flèche : uniquement un nom de variable.
- à droite de la flèche :
  - une **valeur**
  - ou une **expression** : ensemble de valeurs liées par des opérateurs.

La valeur (ou le résultat final de l'expression) est du même type que la variable à gauche.

5.4- Opérateurs :

Un opérateur est un signe qui peut relier deux valeurs pour produire un résultat. Les opérateurs possibles dépendent du type des valeurs qui sont en jeu. On distingue les opérateurs arithmétiques, les opérateurs alphanumériques et les opérateurs logiques.

a- Opérateurs numériques (opérateurs arithmétiques) :

+	addition
-	soustraction
*	multiplication
/	division
^	"puissance"

Notez que vous pouvez utiliser des parenthèses (mêmes règles qu'en math) et qu'il existe une priorité entre les opérateurs :

- L'opérateur puissance a la priorité sur tous les opérateurs.

- La multiplication et la division ont naturellement priorité sur l'addition et la soustraction.
- Les parenthèses ne servent qu'à modifier cette priorité naturelle.
  - Exemple : en informatique, 12 * 3 + 5 et (12 * 3) + 5 valent 41.
  - mais 12 * (3 + 5) vaut 12 * 8 soit 96.

b- Opérateur alphanumérique :

L'opérateur alphanumérique **&** permet de **concaténer** (= agglomérer) deux chaînes de caractères.

Exemples :

- "Nawel" & "IDJGA" renvoie "NawelIDJGA"
- "Nawel " & "IDJGA" renvoie "Nawel IDJGA"

  Remarquez la présence du séparateur (l'espace) entre le prénom et le nom, puisque nous avons mis un espace à la fin de la première chaîne de caractère.

- Si la variable prenom vaut "Nawel" et la variable nom stocke la valeur "IDJGA", prenom & nom renvoie "NawelIDJGA". Pour séparer le prénom du nom, il faut écrire prenom & " " & nom

c- Opérateurs logiques :

En informatique, on distingue surtout les opérateurs logiques ET, OU et NON :

- **ET** (même sens en informatique que dans le langage courant).

  Pour que : C1 ET C2 soit VRAI, il faut que C1 soit VRAIE et que C2 soit VRAIE.

- **OU** (Le OU informatique ne veut pas dire " ou bien ").

  Pour que : C1 OU C2 soit VRAI, Il suffit que C1 soit VRAIE ou que C2 soit VRAIE.

  De plus, si C1 est VRAIE et que C2 est VRAIE aussi, C1 OU C2 est VRAIE.

- NON : inverse une condition :

  Condition VRAI ⇔ NON (Condition) FAUX

Par exemple, NON (X > 15) revient à écrire X <= 15

On représente ceci dans des tables de vérité :

C2 \ C1	Vrai	Faux
Vrai	Vrai	Faux
Faux	Faux	Faux

C1 ET C2

C2 \ C1	Vrai	Faux
Vrai	Vrai	Vrai
Faux	Vrai	Faux

C1 OU C2

C	Non(C)
Vrai	Faux
Faux	Vrai

Non(condition)

5.5- Remarques :

- En mathématiques, une "variable" est généralement une inconnue. En informatique, une variable a toujours une valeur et une seule.

  Les variables non encore affectées sont considérées comme valant zéro. Et cette valeur ne varie que lorsqu'elle est l'objet d'une instruction d'affectation.

- En algorithmique, le signe de l'affectation est <---. Mais en pratique, la quasi-totalité des langages emploient le signe égal. La confusion est facile avec les maths.

  Alors on dit souvent non pas « égal » mais « reçoit » ou « prend pour valeur ».

6- Forme générale d'un algorithme avec variables :

Il faut d'abord distinguer les variables internes à l'algorithme et les paramètres externes qui sont passés par un programme appelant au sous-programme (l'algorithme en cours). Ainsi, la forme générale d'un algorithme qui utilise des variables est la suivante :

```
Algorithme nomAlgorithme (paramètres...)
Variable ...
Début
   ... actions
Fin
```

Par exemple,

```
Algorithme dessineEtoiles (n : entier)
Variable i : entier
Début
   Pour i allant de 1 à n Faire
      Afficher( "*")
   Fin pour
Fin
```

7- Lecture et écriture :

7.1- Les instructions Lire et Ecrire :

L'instruction **lire** signifie simplement "Lire ce que l'utilisateur tape sur le clavier". L'instruction **écrire** signifie écrire à l'écran.

Si vous écrivez l'instruction lire (a) cela revient à dire : la variable "a" est égal à ce que l'utilisateur rentre au clavier.

7.2- Exemple de programme utilisant les instructions Lire et Ecrire :

Voici un exemple de programme utilisant les instructions de lecture et d'écriture :

```
Programme addition
Variables
   a, b, c : Entier
Début
   Ecrire "Donnez la valeur de a : "
   Lire a
   Ecrire "Donnez la valeur de b : "
   Lire b
   c <--- a + b
   Ecrire c
Fin
```

Il est facile de voir que le résultat sera : "Donnez la valeur de a" s'affichera à l'écran. L'utilisateur donne un entier dont la valeur va dans a. Idem pour b. Puis, la somme des deux s'affiche à l'écran.

Note : vous pouvez remplacer les deux instructions **c <--- a + b** et **Ecrire c** par une seule instruction, l'instruction **Ecrire(a + b)**. Si on écrit directement Ecrire (a + b), ce n'est pas la peine de déclarer "c" comme variable, "c" n'est plus utilisée. Il est également à noter que si l'utilisateur ne rentre pas un entier mais par exemple une lettre, il y a une erreur...

EXERCICES

Exercice 2.1

Quelles seront les valeurs des variables a et b après exécution des instructions suivantes ?

```
Variables a, b en Entier
Début
  a <--- 1
  b <--- a + 3
  a <--- 3
Fin
```

Exercice 2.2

Quelles seront les valeurs des variables a, b et c après exécution des instructions suivantes ?

```
Variables a, b, c en Entier
Début
  a <--- 5
  b <--- 3
  c <--- a + b
  a <--- 2
  c <--- b - a
Fin
```

Exercice 2.3

Quelles seront les valeurs des variables a et b après exécution des instructions suivantes ?

```
Variables a, b en Entier
Début
  a <--- 5
  b <--- a + 4
  a <--- a + 1
  b <--- a - 4
Fin
```

Exercice 2.4

Quelles seront les valeurs des variables a, b et c après exécution des instructions suivantes ?

```
Variables a, b, c en Entier
Début
  a <--- 3
  b <--- 10
  c <--- a + b
  b <--- a + b
  a <--- c
Fin
```

Exercice 2.5

Quelles seront les valeurs des variables a et b après exécution des instructions suivantes ?

```
Variables a, b en Entier
Début
   a <--- 5
   b <--- 2
   a <--- b
   b <--- a
Fin
```

Questions : les deux dernières instructions permettent-elles d'échanger les deux valeurs de b et a ? Si l'on inverse les deux dernières instructions, cela change-t-il quelque chose ?

Exercice 2.6

Plus difficile, mais c'est un classique absolu, qu'il faut absolument maîtriser : écrire un algorithme permettant d'échanger les valeurs de deux variables a et b, et ce quel que soit leur contenu préalable.

Exercice 2.7

Une variante du précédent : on dispose de trois variables a, b et c. Ecrivez un algorithme transférant à b la valeur de a, à c la valeur de b et à a la valeur de c (toujours quels que soient les contenus préalables de ces variables).

Exercice 2.8

Que produit l'algorithme suivant ?

```
Variables a, b, c en Caractères
Début
```

```
a <--- "423"
  b <--- "12"
  c <--- a * b
Fin
```

Exercice 2.9

Que produit l'algorithme suivant ?

```
Variables a, b en Caractères
Début
  a <--- "423"
  b <--- "12"
  c <--- a & b
Fin
```

Exercice 2.10

Quel résultat produit le programme suivant ?

```
Variables val, double : Entier
Début
  Val <--- 231
  Double <--- Val * 2
  Ecrire Val
  Ecrire Double
Fin
```

Exercice 2.11

Ecrire un programme qui demande un nombre à l'utilisateur, puis calcule et affiche le carré de ce nombre.

Exercice 2.12

Ecrire un programme qui lit le prix HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant. Faire en sorte que des libellés apparaissent clairement.

Exercice 2.13

Ecrire un algorithme utilisant des variables de type chaîne de caractères, et affichant trois variantes possibles de la phrase «cher ami ta présence à mes côtés m'est très chère». On ne se soucie pas de la ponctuation, ni des majuscules.

CORRIGES DES EXERCICES

Exercice 2.1

Après :	La valeur des variables est :	
a <--- 1	a = 1	b = ?
b <--- a + 3	a = 1	b = 4
a <--- 3	a = 3	b = 4

Exercice 2.2

Après :	La valeur des variables est :		
a <--- 5	a = 5	b = ?	c = ?
b <--- 3	a = 5	b = 3	c = ?
c <--- a + b	a = 5	b = 3	c = 8
a <--- 2	a = 2	b = 3	c = 8
c <--- b - a	a = 2	b = 3	c = 1

Exercice 2.3

Après :	La valeur des variables est :	
a <--- 5	a = 5	b = ?
b <--- a + 4	a = 5	b = 9
a <--- a + 1	a = 6	b = 9
b <--- a - 4	a = 6	b = 2

Exercice 2.4

Après :	La valeur des variables est :		
a <--- 3	a = 3	b = ?	c = ?
b <--- 10	a = 3	b = 10	c = ?
c <--- a + b	a = 3	b = 10	c = 13
b <--- a + b	a = 3	b = 13	c = 13
a <--- c	a = 13	b = 13	c = 13

Exercice 2.5

Après :	La valeur des variables est :	
a <--- 5	a = 5	b = ?
b <--- 2	a = 5	b = 2
a <--- b	a = 2	b = 2
b <--- a	a = 2	b = 2

Les deux dernières instructions ne permettent donc pas d'échanger les deux valeurs de b et a, puisque l'une des deux valeurs (celle de a) est ici écrasée.

Si l'on inverse les deux dernières instructions, cela ne changera rien du tout, hormis le fait que cette fois c'est la valeur de b qui sera écrasée.

Exercice 2.6

```
Début
Variables a, b, c : Entier
  c <--- a
  a <--- b
  b <--- c
Fin
```

On est obligé de passer par une variable dite temporaire (la variable c). On note aussi qu'on peut déclarer les variables a, b et c d'un autre type : réel, booléen ou autre.

Exercice 2.7

```
Début
Variables a, b, c, d : Entier
  d <--- c
  c <--- b
  b <--- a
  a <--- d
Fin
```

En fait, quel que soit le nombre de variables, une seule variable temporaire suffit...

Exercice 2.8

Il ne peut produire qu'une erreur d'exécution, puisqu'on ne peut pas multiplier des chaînes de caractères.

Exercice 2.9

...En revanche, on peut les concaténer. A la fin de l'algorithme, c vaudra donc "42312".

Exercice 2.10

On verra apparaître à l'écran 231, puis 462 (qui vaut 231 * 2)

Exercice 2.11

```
Variables nb, carr en Entier
Début
  Ecrire "Entrez un nombre :"
  Lire nb
  carr <--- nb*nb
  Ecrire "Son carré est : ", carr
Fin
```

En fait, on pourrait tout aussi bien économiser la variable carr en remplaçant les deux avant-dernières lignes par :

```
Ecrire "Son carré est : ", nb*nb
```

C'est une question de style ; dans un cas, on privilégie la lisibilité de l'algorithme, dans l'autre, on privilégie l'économie d'une variable. Mais, si le carré est utilisé une autre fois quelque part dans l'algorithme, il vaudra mieux dans ce cas opter pour l'écriture des deux lignes : les instructions `carr <--- nb*nb` et `Ecrire ("Son carré est : ", carr)`

Exercice 2.12

```
Variables pHt, tTva, pTtc en Réel
Variable nb en Entier
Début
  Ecrire "Entrez le prix hors taxes :"
  Lire pHt
  Ecrire "Entrez le nombre d'articles :"
  Lire nb
  Ecrire "Entrez le taux de TVA (valeur entre 0 et 1) :"
  Lire tTva
  pTtc <--- nb * pHt * (1 + tTva)
  Ecrire ("Le prix toutes taxes est : ", pTtc)
Fin
```

Là aussi, on pourrait économiser une variable et une ligne en écrivant directement. :

```
Ecrire " Le prix toutes taxes est : ", nb * pHt * (1 + tTva)
```

C'est plus rapide, plus léger en mémoire, mais plus difficile à relire (et à écrire !)

Exercice 2.13

```
Variables chaine1, chaine2, chaine3 en Caractère
Début
  chaine1 <--- "cher ami"
  chaine2 <--- "ta présence à mes côtés"
  chaine3 <--- "m'est très chère"
  Ecrire (chaine1 & " " chaine2 & " " & chaine3)
  Ecrire (chaine2 & " " & chaine3 & " " & chaine1)
  Ecrire (chaine2 & " " & chaine1 & " " & chaine3)
Fin
```

Les tests

Une structure conditionnelle (test) est une série d'instructions que l'ordinateur doit effectuer selon que la situation se présente d'une manière ou d'une autre.

1- Structure d'un test :

Il existe deux formes pour un test : la forme complète et la forme simplifiée.

Si booléen **Alors**

Instructions 1

Sinon

Instructions 2

Finsi

Forme complète d'un test

Si booléen **Alors**

Instructions

Finsi

Forme simplifié d'un test

Note : Un booléen est une expression dont la valeur est VRAI ou FAUX. Cela peut donc être :

- une variable de type booléen
- une condition (comparaison)

Explications :

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 « instructions 1 ». A la fin de cette série d'instructions, au moment où elle arrive au mot "Sinon", la machine sautera directement à la première instruction située après le "Finsi".

- Au cas où le booléen avait comme valeur "Faux", la machine saute directement à la première ligne située après le "Sinon" et exécute l'ensemble des "instructions 2".

Si l'une des deux "branches" du Si est vide, plutôt qu'écrire "sinon ne rien faire du tout", il est plus simple de ne rien écrire. Cela revient donc à utiliser le deuxième schéma de l'instruction conditionnelle Si (la forme simplifiée).

Exprimé sous forme de pseudo- code, la programmation d'un touriste perdu donnerait :

Exemple :

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

Fin si

2- Une condition, c'est quoi ?

Une condition est une comparaison. Elle comporte trois éléments :

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

Les valeurs peuvent être à priori de n'importe quel type (numériques, caractères...). Les opérateurs de comparaison sont : = (égal à) < (inférieur à) > (supérieur à) <> (différent de) >= (supérieur ou égal à) <= (inférieur ou égal à).

L'ensemble constitue donc une condition, qui à un moment donné est VRAIE ou FAUSSE.

Les opérateurs de comparaison s'emploient aussi avec des caractères.

- codés par la machine dans l'ordre alphabétique,
- les majuscules placées avant les minuscules

  - "t" < "w" VRAI
    La lettre T est avant la lettre W, donc on utilise l'opérateur < (inférieur) qui signifie ici avant.
  - "Maman" > "Papa" FAUX
    Maman n'est pas après Papa mais avant, puisque la lettre M est avant la lettre P (comme dans un dictionnaire).
  - "maman" > "Papa" VRAI
    Malgré que Maman est avant Papa, mais en informatique les lettres minuscules sont situées après les lettres majuscules (le code des lettres minuscules est supérieur à celui des lettres majuscules). De ce fait, le mot "maman" (en minuscule) est supérieur au mot "Papa" (première lettre en majuscule).

Certains raccourcis du langage peuvent mener à des non-sens informatiques. Par exemple, la condition "Toto est compris entre 5 et 8" nous pouvons être tenté de la traduire par : $5 \leq \text{Toto} \leq 8$ qui a un sens en mathématiques, mais ne veut rien dire en programmation. Il faudra dans ce cas utiliser une condition composée, comme nous le verrons ci-après.

3- Conditions composées :

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci- dessus.

Pour "Toto est inclus entre 5 et 8", il y a en fait 2 conditions : "Toto >= 5" et "Toto <= 8", reliées par un *opérateur logique*, le mot ET.

Rappelez-vous que vous pouvez faire appel, dans une condition, à trois opérateurs logiques : ET, OU, et NON et que le rôle de ces opérateurs est comme suit :

- ET (même sens en informatique que dans le langage courant).

  Pour que C1 ET C2 soit VRAI, il faut que C1 soit VRAI et que C2 soit VRAI.

- OU (Le OU informatique ne veut pas dire "ou bien").

  Pour que C1 OU C2 soit VRAI, Il suffit que C1 soit VRAI ou que C2 soit VRAI.

  Bien entendu, si C1 est VRAI et que C2 est VRAI aussi, C1 OU C2 est VRAI.

- NON : inverse une condition :
  Condition VRAI ⇔ NON (Condition) FAUX

  Par exemple, NON (réussi) revient à écrire échec

4- Tests imbriqués :

SI peut ouvrir deux voies, ou plus... Par exemple, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles. Voici l'algorithme correspondant :

```
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
```

L'utilisation de Si imbriqués (tests imbriqués) présente les avantages suivants :

- Economies au niveau de la frappe du programme : au lieu de devoir taper trois conditions, dont une composée, on a plus que deux conditions simples.

=> plus simple et plus lisible

- Economies sur le temps d'exécution de l'ordinateur : si la première possibilité est la bonne, le programme passe directement à la fin sans tester le reste, forcément faux.

  => plus performant.

5- Variables booléennes :

Les variables booléennes stockent les valeurs VRAI ou FAUX. on peut donc entrer des conditions dans ces variables, et tester ensuite la valeur de ces variables.

On peut réécrire l'exemple de l'eau ainsi :

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

A priori, cela n'a guère d'intérêt : on a alourdi l'algorithme de départ, en lui faisant recourir à 2 variables supplémentaires, mais :

- Une variable booléenne n'a besoin que d'un seul bit pour être stockée

  => alourdissement moindre.

- Dans le cas de conditions composées très lourdes (avec plein de ET et de OU)

=> facilite le travail de programmation.

- On peut réutiliser à volonté ces variables

6- Jeux logiques :

Dans le cas de conditions composées (ET + OU), les parenthèses jouent un rôle important.

Exemple :

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

Question :

Si X = 3, que vaut D ? et que vaut E ?

Réponse :

X = 3 => A <--- 3 < 2, donc A reçoit FAUX

B <--- 3 > 12, donc B reçoit FAUX

C <--- 3 < 6, donc C reçoit VRAI

D <--- (FAUX **ET** FAUX) **OU** VRAI, donc D reçoit FAUX OU VRAI c'est-à-dire D reçoit VRAI

E <--- FAUX **ET** (FAUX **OU** VRAI) , donc E reçoit FAUX ET VRAI c'est-à-dire E reçoit FAUX

Notes :

- S'il n'y a que des ET ou que des OU, les parenthèses ne changent rien.
- Toute structure de test avec ET peut être exprimée de manière équivalente avec OU, et réciproquement.

La règle d'équivalence :

Si A **ET** B **Alors**
Instructions 1
Sinon
Instructions 2
Finsi

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

En fait, nous avons les relations d'équivalence suivantes :

Non(A ET B) ⇔ Non(A) OU Non(B)

Non(A OU B) ⇔ Non(A) ET Non(B)

7- Choix multiples : la structure Selon

La structure Selon, appelée aussi "Etude de cas" et *Case* en anglais, est assez simple à comprendre. Elle permet de faire quelque chose en fonction de la valeur d'une variable.

Imaginons une variable Note comprise entre 0 et 20. En fonction de la note, on peut écrire un commentaire différent. Le tableau suivant illustre le commentaire à affecter à la note :

Valeur de Note	Commentaire
De 0 à 6,99	C'est nul!
De 7 à 9,99	Insuffisant
10	C'est la moyenne
De 10,01 à 11,99	Passable
De 12 à 15,99	Bien
De 16 à 20	Excellent
<0 ou >20	Erreur! La note n'est pas entre 0 et 20

En pseudo-code, cela donne :

```
Variable note : Réel
Selon note
  0..6,99 : Ecrire ( "C'est nul!" )
  7..9,99 : Ecrire ( "Insuffisant" )
  10 : Ecrire ( "C'est la moyenne" )
  10,01..11,99 : Ecrire ( "Passable" )
  12..15,99 : Ecrire ( "Bien" )
  16..20 : Ecrire ( "Excellent" )
  Default : Ecrire ( "Erreur! La note n'est pas entre 0 et 20" )
Fin Selon
```

Note : Default est utilisé si aucune valeur ne correspond à la variable.

EXERCICES

Exercice 3.1

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou *négatif* (on laisse de côté le cas où le nombre vaut zéro).

Exercice 3.2

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul). Attention toutefois : on ne doit **pas** calculer le produit des deux nombres.

Exercice 3.3

Ecrire un algorithme qui demande trois noms à l'utilisateur et l'informe ensuite s'ils sont rangés ou non dans l'ordre alphabétique.

Exercice 3.4

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on inclut cette fois le traitement du cas où le nombre vaut zéro).

Exercice 3.5

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si le produit est négatif ou positif (on inclut cette fois le traitement du cas où le produit peut être nul). Attention toutefois, on ne doit pas calculer le *produit* !

Exercice 3.6

Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :

- « Poussin » de 6 à 7 ans
- « Pupille » de 8 à 9 ans
- « *Minime* » de 10 à 11 ans
- « Cadet » après 12 ans

Peut-on concevoir plusieurs algorithmes équivalents menant à ce *résultat* ?

Exercice 3.7

Formulez un algorithme équivalent à l'algorithme suivant :

```
Si Tutu > Toto + 4  OU Tata = "OK" Alors
     Tutu <--- Tutu + 1
Sinon
     Tutu <--- Tutu - 1
Finsi
```

Exercice 3.8

Cet algorithme est destiné à prédire l'avenir, et il doit être infaillible !

Il lira au clavier l'heure et les minutes, et il affichera l'heure qu'il sera une minute plus tard. Par exemple, si l'utilisateur tape 21 puis 32, l'algorithme doit répondre : "Dans une minute, il sera 21 heure(s) 33".

NB : on suppose que l'utilisateur entre une heure valide. Pas besoin donc de la vérifier.

Exercice 3.9

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 heure valide.

Exercice 3.10

Une compagnie de distribution d'eau facture 7,80 DA les quarante premiers m^3, 12,20 DA les vingt suivants et 21,30 au-delà.

Ecrivez un algorithme qui demande à l'utilisateur le nombre de m^3 consommés puis affiche la facture correspondante.

Exercice 3.11

Les habitants d'une ville 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 de l'habitant, et se prononcera donc ensuite sur le fait que l'habitant est imposable ou non.

Exercice 3.12

Ecrivez un algorithme qui après 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.

Exercice 3.13

Les élections législatives dans un pays asiatique 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).

Exercice 3.14

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 (pour deux accidents ou plus)
- 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 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 en a l'air (cela s'appelle faire une analyse !)

CORRIGES DES EXERCICES

Exercice 3.1

```
Variable n en Entier
Début
  Ecrire "Entrez un nombre : "
  Lire n
  Si n > 0 Alors
        Ecrire "Ce nombre est positif"
     Sinon
        Ecrire "Ce nombre est négatif"
  Finsi
Fin
```

Exercice 3.2

```
Variables m, n en Entier
Début
  Ecrire "Entrez deux nombres : "
  Lire m, n
  Si (m > 0 ET n > 0) OU (m < 0 ET n < 0) Alors
       Ecrire "Leur produit est positif"
     Sinon
       Ecrire "Leur produit est négatif"
  Finsi
Fin
```

Exercice 3.3

```
Variables a, b, c en Caractère
Début
  Ecrire "Entrez successivement trois noms : "
  Lire a, b, c
  Si a < b et b < c Alors
        Ecrire "Ces noms sont classés alphabétiquement"
     Sinon
        Ecrire "Ces noms ne sont pas classés"
  Finsi
Fin
```

Exercice 3.4

```
Variable n en Entier
Début
  Ecrire "Entrez un nombre : "
  Lire n
  Si n < 0 Alors
       Ecrire "Ce nombre est négatif"
     Sinon
       Si n = 0 Alors
           Ecrire "Ce nombre est nul"
         Sinon
           Ecrire "Ce nombre est positif"
       FinSi
  FinSi
Fin
```

Exercice 3.5

```
Variables m, n en Entier
Début
  Ecrire "Entrez deux nombres : "
  Lire m, n
  Si m = 0 OU n = 0 Alors
      Ecrire "Le produit est nul"
    Sinon
      Si (m < 0 ET n < 0) OU (m > 0 ET n > 0) Alors
          Ecrire "Le produit est positif"
        Sinon
          Ecrire "Le produit est négatif"
      Finsi
..Finsi
Fin
```

Si on souhaite simplifier l'écriture de la condition lourde du SinonSi, on peut toujours passer par des variables booléennes intermédiaires. Une astuce de sioux consiste également à employer un Xor (c'est l'un des rares cas dans lesquels il est pertinent).

Exercice 3.6

```
Variable age en Entier
  Début
  Ecrire "Entrez l'âge de l'enfant : "
  Lire age
  Si age >= 12 Alors
      Ecrire "Catégorie Cadet"
    Sinon
      Si age >= 10 Alors
          Ecrire "Catégorie Minime"
        Sinon
```

```
            Si age >= 8 Alors
                Ecrire "Catégorie Pupille"
              Sinon
                Si age >= 6 Alors
                     Ecrire "Catégorie Poussin"
                Finsi
            Finsi
        Finsi
    Finsi
Fin
```

On peut évidemment écrire cet algorithme de différentes façons, ne serait-ce qu'en commençant par la catégorie la plus jeune.

On peut aussi simplifier l'algorithme, en utilisant l'instruction Selon, comme c'est illustré ci-dessous :

```
Variable age en Entier
  Début
  Ecrire "Entrez l'âge de l'enfant : "
  Lire age
  Selon age
    6..7 : Ecrire "Catégorie Poussin"
    8..9 : Ecrire "Catégorie Pupille"
    10..11 : Ecrire "Catégorie Minime"
    >=12 : Ecrire "Catégorie Cadet"
    >=12 : Ecrire "l'enfant est très jeune. Il ne peut être
                   accepté"
  FinSelon
Fin
```

Exercice 3.7

Aucune difficulté, il suffit d'appliquer la règle de la transformation du OU en ET vue en cours. Attention toutefois à la rigueur dans la transformation des conditions en leur contraire...

```
Si Tutu <= Toto + 4  et Tata <> "OK" Alors
     Tutu <--- Tutu - 1
  Sinon
     Tutu <--- Tutu + 1
Finsi
```

Exercice 3.8

```
Variables h, m en Entier
Début
  Ecrire "Entrez les heures, puis les minutes : "
  Lire h, m
  m <--- m + 1
  Si m = 60 Alors
     m <--- 0
     h <--- h + 1
  FinSi
  Si h = 24 Alors
     h <--- 0
  FinSi
  Ecrire "Dans une minute il sera ", h, "heure(s) ", m,
         "minute(s)"
Fin
```

Exercice 3.9

```
Variables h, m, s en Entier
Début
  Ecrire "Entrez les heures, puis les minutes, puis les
          secondes : "
  Lire h, m, s
  s <--- s + 1
  Si s = 60 Alors
      s <--- 0
      m <--- m + 1
  FinSi
  Si m = 60 Alors
      m <--- 0
      h <--- h + 1
  FinSi
  Si h = 24 Alors
      h <--- 0
  FinSi
  Ecrire "Dans une seconde il sera ", h, "heure(s)", m,
          "minute(s) et ", s, "seconde(s)"
Fin
```

Exercice 3.10

```
Variable n en Entier
Variable montant en Réel
Début
  Ecrire "Nombre de m3 : "
  Lire n
  Si n <= 40 Alors
      montant <--- n * 7,80
```

```
    Sinon
      Si n <= 60 Alors
          montant <--- 40 * 7,80 + (n - 40) * 12,20
        Sinon
          montant <--- 40 * 7,80 + 20 * 12,20 + (n - 60) *
                       21,30
      FinSi
  Finsi
  Ecrire ("Le prix total est : ", montant, " DA ")
Fin
```

Exercice 3.11

```
Variable sex en Caractère
Variable age en Entier
Début
  Ecrire "Entrez le sexe (M/F) : "
  Lire sex
  Ecrire "Entrez l'âge : "
  Lire age
  C1 <--- sex = "M" et age > 20
  C2 <--- sex = "F" et (age > 18 et age < 35)
  Si C1 ou C2 Alors
      Ecrire "Imposable"
    Sinon
      Ecrire "Non Imposable"
  FinSi
Fin
```

Exercice 3.12

En ce qui concerne le début de cet algorithme, il n'y a aucune difficulté. C'est de la saisie bête et même pas méchante :

```
Variables J, M, A, jMax en Entier
Variables jValide, mValide, aBisex en Booleen
Début
  Ecrire "Entrez le numéro du jour"
  Lire J
  Ecrire "Entrez le numéro du mois"
  Lire M
  Ecrire "Entrez l'année"
  Lire A
```

C'est évidemment ensuite que les ennuis commencent... La première manière d'aborder la chose consiste à se dire que fondamentalement, la structure logique de ce problème est très simple. Si nous créons deux variables booléennes jValide et mValide, représentant respectivement la validité du jour et du mois entrés, la fin de l'algorithme sera d'une simplifié (l'année est valide par définition) :

```
Si jValide et mValide Alors
    Ecrire "La date est valide"
  Sinon
    Ecrire "La date n'est pas valide"
FinSi
```

Toute la difficulté consiste à affecter correctement les variables jValide et mValide, selon les valeurs des variables J, M et A. Dans l'absolu, jValide et mValide pourraient être les objets d'une affectation monstrueuse, avec des conditions atrocement composées. Mais franchement, écrire ces conditions en une seule fois est un travail sans grand intérêt. Pour éviter d'en arriver à une telle extrémité, on peut simplifier la chose en créant deux variables supplémentaires : **aBisex**, une variable booléenne qui indique s'il s'agit d'une année bissextile (donc divisible par 4), et jMax, variable numérique qui indiquera le dernier jour valable pour le mois entré.

Avec tout cela, on peut y aller et en ressortir vivant.

On commence par initialiser nos variables booléennes, puis on traite les *années, puis les mois, puis les jours.*

On note "divPar" la condition "divisible par" :

```
aBisex <--- A divPar 4
jMax <--- 0
mValide <--- M >= 1 et M =< 12
Si mValide Alors
    Si M = 2 et aBisex Alors
        jMax <--- 29
      Sinon
        Si M = 2 Alors
            jMax <--- 28
          Sinon
            Si M = 4 ou M = 6 ou M = 9 ou M = 11 Alors
                jMax <--- 30
              Sinon
                JMax <--- 31
            FinSi
        FinSi
    FinSi
    jValide <--- J >= 1 et J =< Jmax
FinSi
```

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

Composons alors les différentes parties pour avoir l'algorithme final, qui est le suivant :

```
Variables J, M, A, jMax en Entier
Variables jvalide, mValide, aBisex en Booleen
Début
  Ecrire "Entrez le numéro du jour"
  Lire J
```

```
Ecrire "Entrez le numéro du mois"
Lire M
Ecrire "Entrez l'année"
Lire A
aBisex <--- A divPar 4
jMax <--- 0
mValide <--- M >= 1 et M =< 12
Si mValide Alors
    Si M = 2 et aBisex Alors
        jMax <--- 29
      Sinon
        Si M = 2 Alors
            jMax <--- 28
          Sinon
            Si M = 4 ou M = 6 ou M = 9 ou M = 11 Alors
                jMax <--- 30
              Sinon
                JMax <--- 31
            FinSi
        FinSi
    FinSi
    jValide <--- J >= 1 et J =< Jmax
FinSi
Si jValide et mValide Alors
    Ecrire "La date est valide"
  Sinon
    Ecrire "La date n'est pas valide"
FinSi
```

Fin

Il convient enfin de citer une solution très simple et élégante, un peu plus difficile peut-être à imaginer du premier coup, mais qui avec le recul apparaît comme très immédiate. Sur le fond, cela consiste à dire qu'il y a quatre cas pour qu'une date soit valide : celui d'un jour compris

entre 1 et 31 dans un mois à 31 jours, celui d'un jour compris entre 1 et 30 dans un mois à 30 jours, et celui d'un jour compris entre 1 et 28 pour le mois de Février et une année qui n'est bissextile, et enfin le 4ème cas où le jour est compris entre 1 et 29 pour le mois de Février d'une année bissextile.

```
aBisex <--- A divPar 4

cas1 <--- (m=1 ou m=3 ou m=5 ou m=7 ou m=8 ou m=10 ou m=12)
          et (J>=1 et J=<31)

cas2 <--- (m=4 ou m=6 ou m=9 ou m=11) et (J>=1 et J=<30)

cas3 <--- m=2 et aBisex et J>=1 et J=<29

cas4 <--- m=2 et Non(aBisex) et J>=1 et J=<28
Si cas1 ou cas2 ou cas3 ou cas4 Alors
    Ecrire "Date valide"
  Sinon
    Ecrire "Date non valide"
FinSi
```

Tout est alors réglé avec quelques variables booléennes et quelques conditions composées, en un minimum de lignes de code.

Ce que nous retenons de ce long exercice - et non moins long corrigé, c'est qu'un problème de test un peu compliqué admet une pléiade de solutions justes.

Exercice 3.13

Cet exercice, du pur point de vue algorithmique, n'est pas très méchant. En revanche, il représente dignement la catégorie des énoncés piégés. En effet, rien de plus facile que d'écrire : si le candidat a plus de 50%, il est élu, sinon s'il a plus de 12,5 %, il est au deuxième tour, sinon il est éliminé. Hé hé hé... mais il ne faut pas oublier que le candidat peut très bien avoir eu 20 % mais être tout de même éliminé, tout simplement parce que l'un des autres a fait plus de 50 % et donc qu'il n'y a pas de deuxième tour !... Moralité : ne jamais se jeter sur la programmation avant d'avoir soigneusement mené l'analyse du problème à traiter.

```
Variables A, B, C, D en Caractère
```

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

Exercice 3.14

Là encore, on illustre l'utilité d'une bonne analyse. Je propose deux corrigés différents. Le premier suit l'énoncé pas à pas. C'est juste, mais c'est vraiment lourd. La deuxième version s'appuie sur une vraie compréhension d'une situation pas si embrouillée qu'elle n'en a l'air.

Dans les deux cas, un recours aux variables booléennes aère sérieusement l'écriture.

Donc, premier corrigé, on suit le texte de l'énoncé pas à pas :

```
Variables age, perm, acc, assur en Entier
Variable situ en Caractère
Début
  Ecrire "Entrez l'âge : "
  Lire age
```

```
Ecrire "Entrez le nombre d'années de permis : "
Lire perm
Ecrire "Entrez le nombre d'accidents : "
Lire acc
Ecrire "Entrez le nombre d'années d'assurance : "
Lire assur
C1 <--- age >= 25
C2 <--- perm >= 2
C3 <--- assur > 1
Si Non(C1) et Non(C2) Alors
    Si acc = 0 Alors
        situ <--- "Rouge"
      Sinon
        situ <--- "Refusé"
    FinSi
  Sinon
      Si ((Non(C1) et C2) ou (C1 et Non(C2)) Alors
          Si acc = 0 Alors
              situ <--- "Orange"
            Sinon
              Si acc = 1 Alors
                  situ <--- "Rouge"
                Sinon
                  situ <--- "Refusé"
              FinSi
          FinSi
        Sinon
          Si acc = 0 Alors
              situ ← "Vert"
            Sinon
              Si acc = 1 Alors
                  situ ← "Orange"
                Sinon
```

```
                    Si acc = 2 Alors
                        situ ← "Rouge"
                    Sinon
                        situ ← "Refusé"
                    FinSi
                FinSi
            Finsi
        Finsi
    Finsi
    Si C3 Alors
        Si situ = "Rouge" Alors
            situ = "Orange"
        Sinon
            Si situ = "Orange" Alors
                situ = "Vert"
            Sinon
                Si situ = "Vert" Alors
                    situ = "Bleu"
                FinSi
            FinSi
        Finsi
    Finsi
    Ecrire ("Votre situation : ", situ)
Fin
```

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

```
nbPoints <--- 0
Si Non(C1) Alors
   nbPoints <--- nbPoints + 1
FinSi

Si Non(C2) Alors
  nbPoints <--- nbPoints + 1
FinSi
nbPoints <--- nbPoints + acc
Si P < 3 et C3 Alors
   P <--- P - 1
FinSi
Si P = -1 Alors
    situ <--- "Bleu"
  Sinon
    Si P = 0 Alors
        situ <--- "Vert"
      Sinon
        Si P = 1 Alors
             situ <--- "Orange"
          Sinon
            Si P = 2 Alors
                situ <--- "Rouge"
              Sinon
                situ <--- "Refusé"
            FinSi
         Finsi
     Finsi
 Finsi
 Ecrire ("Votre situation : ", situ)
 Fin
```

Les boucles

1- Les boucles, c'est quoi au juste ?

On appelle boucle une structure où un ensemble d'instructions est exécuté plusieurs fois. Il y a trois types de boucle, la boucle **Pour**, la boucle **Tant Que**, et la boucle **Répéter - Jusqu'à**.

2- Intérêt des boucles :

On pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Mais l'utilisateur risque de taper autre chose. Dès lors, le programme peut planter ou produire des résultats fantaisistes.

=> Alors, on peut mettre en place un **contrôle de saisie** (pour vérifier que les données entrées correspondent bien à celles attendues par l'algorithme).

On peut faire cela avec une structure de condition SI, comme c'est illustré par l'algorithme suivant :

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

Si l'utilisateur ne se trompe qu'une seule fois => OK. Mais pour prévoir le cas de deuxième erreur, il faut rajouter un SI. Et ainsi de suite....

=> impasse...

3- La boucle « Tant que » :

La seule issue pour le cas précédent est donc une structure de boucle, qui se présente ainsi :

```
Algorithme JusquAuMur
Début
Tant que Non(DevantMur) Faire
   Avancer
Fin tant que
Fin
```

Le principe:

i- Le programme arrive sur la ligne du Tant que.

ii- Il examine alors la valeur du booléen (variable booléenne ou une condition).

iii- Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne Fin Tant que.

iv- Il retourne ensuite sur la ligne du Tant que, procède au même examen, et ainsi de suite.

v- Ça ne s'arrête que lorsque le booléen prend la valeur FAUX.

Illustration avec notre problème de contrôle de saisie :

```
Variable rep en Caractère
         correct en Booléen
Début
Ecrire "Voulez vous un café ? (O/ N)"
correct <--- FAUX
Tant que Non(correct) Faire
   Lire rep
   Si rep <> "O" ET rep <> "N" Alors
      Ecrire "Saisie erronée. Recommencez"
```

```
    Sinon
        Correct <--- VRAI
    FinSi
Fin Tant que
Fin
```

4- La boucle « Répéter » :

Si nous reprenons l'exemple de l'algorithme JusquAuMur, on peut l'écrire en utilisant une autre forme de boucle : la *boucle Répéter*. Voici l'algorithme :

```
Algorithme JusquAuMurVersionRépéter
Début
Répéter
    Avancer
jusqu'à DevantMur
Fin
```

Contrairement à la boucle *tant que*, on passe toujours au moins une fois dans une boucle *répéter*. Ainsi, dans l'exemple ci- dessus, on avancera forcément, si on est pas heurté au mur ... Il conviendrait donc de tester si l'on n'est pas devant un mur avant d'utiliser cet algorithme.

5- La boucle « Pour » :

Avec variables, on peut écrire la boucle *pour*. Lorsque l'on sait exactement combien de fois on doit itérer un traitement, c'est cette boucle qui doit être privilégiée.

```
Algorithme CompteJusqueCent
Début
Pour i de 1 à 100
  Afficher(i)
```

```
    ALaLigne
  i Suivant
  Fin
```

6- Des boucles dans des boucles :

Une boucle peut contenir d'autres boucles. Pour résoudre certains problèmes, vous serez amené à écrire un algorithme qui comporte des boucles imbriqués : une boucle à l'intérieur d'une autre boucle. L'algorithme suivant en est un exemple :

```
Variables i, j en Entier
Pour i de 1 à 15
  Ecrire "Il est passé par ici"
  Pour j de 1 à 6
    Ecrire "Il repassera par là"
  j Suivant
i Suivant
```

Dans cet exemple,

- le programme écrira une fois "il est passé par ici"
- puis 6 fois de suite "il repassera par là",
- et ceci 15 fois en tout.

A la fin, il y aura donc eu 15 x 6 = 90 passages dans la 2ème boucle (celle du milieu).

Des boucles peuvent être imbriquées ou successives, mais elles ne peuvent jamais être croisée => aucun sens logique.

Par exemple, l'écriture suivante est fausse puisque cet algorithme comporte deux boucles croisées :

```
Variables i, j en Entier
Pour i de 1 à 10
    Ecrire "Il est passé par ici"
    Pour j de 1 à 6
        Ecrire "Il repassera par là"
    i Suivant
j Suivant
```

7- Comparaisons des boucles :

Prenons un même problème (Affichage des 100 premiers nombres entiers) dont on écrit plusieurs algorithmes différents, en changeant la boucle utilisée :

Boucle « *pour* »

```
Algorithme compteJusqueCentVersionPour
Variable i : entier
Début
Pour i de 1 à 100
   Afficher( i)
   ALaLigne
i Suivant
Fin
```

Boucle « tant que »

(il faut initialiser *i* avant la boucle, et l'augmenter de 1 à chaque passage) :

```
Algorithme compteJusqueCentVersionTantQue
Variable i : entier
Début
```

```
i <--- 1
Tant que (i <= 100) Faire
   Afficher( i)
   ALaLigne
   i <--- i + 1
 Fin TantQue
Fin
```

Boucle « répéter »

(noter que la condition d'arrêt est ici la négation de la condition du *tant que*):

```
Algorithme compteJusqueCentVersionRepeter
Variable i : entier
Début
i <--- 1
Répéter
   Afficher( i)
   ALaLigne
   i <--- i + 1
Jusqu'à (i > 100)
Fin
```

Exercices

Exercice 4.1

Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 4.2

Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

Exercice 4.3

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

Exercice 4.4

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) :

```
Table de 7 :

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
...
```

```
7 x 10 = 70
```

Exercice 4.5

Ecrire un algorithme qui demande un nombre de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer :

```
1 + 2 + 3 + 4 + 5 = 15
```

Exercice 4.6

Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle.

NB : la factorielle de 8, notée 8 !, vaut 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8

Exercice 4.7

Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dise ensuite quel était le plus grand parmi ces 20 nombres :

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

Exercice 4.8

Réécrire l'algorithme précédent, mais cette fois-ci on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un zéro. Nous voulons aussi que cet algorithme renvoie le nombre de valeurs que l'utilisateur a saisi.

Exercice 4.9

Ecrire un algorithme qui lit la suite des prix des achats d'un client, qu'on terminera par zéro, puis calcule la somme qu'il doit ; ensuite lire la somme qu'il paye, et simuler la remise de la monnaie en affichant les textes 500 DA, 200 DA et 100 DA autant de fois qu'il y a de coupures de chaque sorte à rendre (on supposera que le coût de chaque article vendu dans le magasin est un multiple de 100 DA).

Exercice 4.10

L'algorithme d'Euclide : Trouver le pgdc (le Plus Grand Diviseur Commun) de 2 nombres entiers

Pour cela, il vous est demandé de suivre la démarche suivante :

i- Comprendre le problème et savoir le résoudre

ii- Rédiger proprement la méthode

iii- Rédiger un algorithme

Outils à disposition : papier, crayon, cerveau.

Exercice 4.11

Ecrire l'algorithme qui affiche a l'écran si un nombre entier est premier

---

Exercice 4.12

Ecrire l'algorithme qui approxime la valeur de 1/(1-x) a l'aide du développement limite :

$1/(1-x) = 1 + x + x^2 + \ldots + x^n + o(x^n)$

Exercice 4.13

Ecrire l'algorithme de la procédure dessinerLignes du type Sortie. Cette procédure reçoit un paramètre n, elle dessine ensuite n lignes formées du symbole '*' comme l'indique la figure ci-dessus.

pour n = 5, le dessin est le suivant :

```
    *
   ***
  *****
 *******
*********
```

CORRIGES DES EXERCICES

Exercice 4.1

```
Variable N en Entier
         correct en Booléen
Début
correct <--- FAUX
N <--- 0
Ecrire "Entrez un nombre entre 1 et 3"
Tant Que Non(correct) Faire
    Lire N
    Si N < 1 ou N > 3 Alors
        Ecrire "Saisie erronée. Recommencez"
      Sinon
       correct <--- VRAI
    FinSi
Fin TantQue
Fin
```

Exercice 4.2

```
Variable N en Entier
Debut
N <--- 0
Ecrire "Entrez un nombre entre 10 et 20"
Tant Que N < 10 ou N > 20 Faire
    Lire N
    Si N < 10 Alors
        Ecrire "Plus grand !"
      Sinon
```

```
        Si N > 20 Alors
          Ecrire "Plus petit !"
        FinSi
    FinSi
Fin TantQue
Fin
```

Exercice 4.3

```
Variables n, i en Entier
Debut
Ecrire "Entrez un nombre : "
Lire n
Ecrire "Les 10 nombres suivants sont : "
Pour i de n + 1 à n + 10
    Ecrire i
i Suivant
Fin
```

Exercice 4.4

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

```
i Suivant
Fin
```

Exercice 4.5

```
Variables N, i, somme en Entier
Debut
Ecrire "Entrez un nombre : "
Lire N
somme <--- 0
Pour i de 1 à N
    Somme <--- Somme + i
i Suivant
Ecrire ("La somme est : ", Somme)
Fin
```

Exercice 4.6

```
Variables N, i, fact en Entier
Début
Ecrire "Entrez un nombre : "
Lire N
fact <--- 1
Pour i de 2 à N
    fact <--- fact * i
i Suivant
Ecrire ("La factorielle est : ", fact)
Fin
```

Exercice 4.7

```
Variables N, i, PG en Entier
Debut
PG <--- 0
Pour i de 1 à 20
    Ecrire ("Entrez le nombre : ", i, " : ")
    Lire N
    Si i = 1 ou N > PG Alors
          PG <--- N
    FinSi
i Suivant
Ecrire ("Le nombre le plus grand était : ", PG)
Fin
```

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

Le premier nombre, cas où i = 1, est affecté à PG pour écraser la valeur d'initialisation (la valeur zéro). Ceci est utile dans le cas où on donne à l'utilisateur la possibilité d'introduire des nombres négatifs. Si les nombres saisis sont tous positifs, nous pouvons alors remplacer le test **Si i = 1 ou N > PG** par **Si N > PG**. Mais, dans ce cas nous sommes obligés d'initialiser la variable PG à 0 (nous n'avons pas le choix).

Pour la version améliorée qui doit permettre d'afficher, en plus du plus grand nombre, en quelle position avait été saisi ce nombre, il faut écrire l'algorithme suivant :

```
Variables N, i, PG, IPG en Entier
Debut
PG <--- 0
Pour i de 1 à 20
   Ecrire "Entrez un nombre : "
   Lire N
   Si i = 1 ou N > PG Alors
      PG <--- N
      IPG <--- i
```

```
    FinSi
i Suivant
Ecrire ("Le nombre le plus grand était : ", PG)
Ecrire ("Il a été saisi en position numéro ", IPG)
Fin
```

Exercice 4.8

```
Variables N, i, PG, IPG en Entier
Debut
N <--- 1
i <--- 0
PG <--- 0
Tant Que N <> 0 Faire
    Ecrire "Entrez un nombre : "
    Lire N
    i <--- i + 1
    Si i = 1 ou N > PG Alors
        PG <--- N
        IPG <--- i
    FinSi
Fin TantQue
Ecrire ("Vous avez saisi au total ", i, " nombres")
Ecrire ("Le nombre le plus grand était : ", PG)
Ecrire ("Il a été saisi en position numéro ", IPG)
Fin
```

Exercice 4.9

```
Variables montantAchat, somDue, montantVerse, IPG, reste,
          Nb500, Nb200 : Entier
Debut
```

```
montantAchat <--- 1
somDue <--- 0
TantQue montantAchat <> 0
    Ecrire "Entrez le montant : "
    Lire montantAchat
    somDue <--- somDue + montantAchat
Fin TantQue
Ecrire ("Vous devez : ", somDue, " DA"
Ecrire "Montant versé : "
Lire montantVerse
reste <--- montantVerse - somDue
Nb500 <--- 0
TantQue reste >= 500 Faire
    Nb500 <--- Nb500 + 1
    reste <--- reste - 500
Fin TantQue
Nb200 <--- 0
Tant que reste >= 200 Faire
    Nb200 <--- 1
    reste <--- reste - 200
Fin TantQue
Si reste >= 100 Alors
    Nb100 <--- 1
  Sinon
    Nb100 <--- 0
FinSi
Ecrire "Rendu de la monnaie :"
Ecrire ("Billets de 500 DA : ", Nb500)
Ecrire ("Billets de 200 DA : ", Nb200)
Ecrire ("Billets ou pièces de 100 DA : ", Nb100)
Fin
```

Exercice 4.10

i- Etape 1 : **Comprendre le problème et savoir le résoudre**

Trouver le pgdc de 2 nombres entiers. Pour résoudre ce problème, on se base sur :

- si a> b alors pgdc(a, b) = pgdc(a- b, b)
- si b> a alors pgdc(a, b) = pgdc(a, b-a)
- pgdc(a, a) = a
- Exemple : pgdc(144, 96)

  (144, 96) => (48, 96) => (48, 48)

  => pgdc = 48

ii- Etape 2 : **Rédiger proprement la méthode**

Pour calculer pgdc de a et b :

- Tant que a est différent de b exécuter les 2 lignes suivantes :
  - Si a > b, retrancher b à a
  - Sinon, retrancher a à b
- Affirmer que le pgdc est a

iii- Etape 3 : **Rédiger un algorithme**

```
Algorithme pgdc(a,b : entiers strictements positifs) : Entier
Variables N1, N2 en entier
N1 <--- a
N2 <--- b
Tant que N1 <> N2 Faire
   Si N1 > N2 Alors
        N1 <--- N1 – N2
      Sinon
        N2 <--- N2 – N1
   FinSi
Fin TantQue
```

Ecrire ("Le pgcd est : ", N1)

Fin

Exercice 4.11

```
Algorithme affichePremier(x : entier)
Variable y : ENTIER
DEBUT
   y <-- 2
   TANT QUE ((y * y) < x) ET  (x Mod y) <> 0 FAIRE
       y <-- y + 1
   Fin TantQue
   SI (y * y) >= x ALORS
         Ecrire(x,' est un Premier')
       Sinon
         Ecrire(x,' n'est pas premier')
   FinSi
FIN
```

Exercice 4.12

```
Fonction devUnSurUnMoinsX(x : Réel ; n : Entier) : Réel
Variables y, res : Réel
          i : Entier
DEBUT
   res <-- 1.0
   i <-- 1
   y <-- x
   TANT QUE i <= n FAIRE
      res <-- res + y
      y <-- y * x
```

```
      i <-- i + 1
   Fin TantQue
   Retourner res
FIN
```

Exercice 4.13

```
Procédure dessinerLignes(n : Entier)
Variables ligne en Caractères
          i, j : Entier
DEBUT
   i <-- 1
   TANT QUE i <= n FAIRE
      ligne <-- "  "          // On affecte 3 espaces à la variable ligne
      j <-- 1
      TANT QUE j <= (n-i) FAIRE
          Ligne & " ")        // On place un espace après la chaîne Ligne
          j <-- j+1
      Fin TantQue
      j <-- 1
      TANT QUE j <= (2*i-1) FAIRE
          concatener(ligne,'*')
          j <-- j+1
      Fin TantQue
      ecrire(ligne)
      i <-- i+1
   Fin TantQue
FIN
```

Les tableaux

1- Pourquoi les tableaux ?

Imaginons que l'on veuille calculer la moyenne des notes d'une promotion, quel algorithme allons nous utiliser ?

Pour l'instant on pourrait avoir la procédure suivante :

```
Procédure calculerMoyenne ()
Variables somme, nbEleves, uneNote, i : Entier
Début
  somme <--- 0
  Ecrire(" Nombre d'eleves : ")
  Lire(nbEleves)
  Pour i de 1 à nbEleves
    Ecrire("Note de l'élève numero ", i," : ")
    Lire(uneNote)
    somme <--- somme + uneNote
  i Suivant
  Ecrire("La moyenne est de : ", somme/nbEleves)
Fin
```

Dans l'exemple ci-dessus, nous avons défini une procédure, sans qu'on entre dans les détails de cette définition : déclaration d'une procédure ou fonction, déclaration des variables et des paramètres, passage des paramètres, ... D'autres exemples dans le chapitre en cours montrent comment déclarer et définir une procédure ou une fonction. De plus, nous reviendrons à ce sujet avec tous les détails qu'il faut. Un chapitre complet est réservé à l'étude des fonctions et procédures.

Imaginons maintenant que l'on veuille toujours calculer la moyenne des notes d'une promotion mais en gardant en mémoire toutes les notes des étudiants (pour par exemple faire d'autres calculs tels que l'écart type, la note minimale, la note maximale, etc.).

Il faudrait alors déclarer autant de variables qu'il y a d'étudiants, par exemple en supposant qu'il y ait trois étudiants, on aurait la procédure suivante : procédure calculerMoyenne3Etudiants ().

```
Procédure calculerMoyenne3Etudiants ()
Variables somme, note1, note2, note3 : Entier
Début
  Ecrire("Les notes des trois étudiants : ")
  Lire(note1) ;
  Lire(note2) ;
  Lire(note3)
   somme <--- note1 + note2 + note3
   Ecrire("La moyenne est de : ", somme/ 3)
Fin
```

Le problème est que cet algorithme ne fonctionne que pour 3 étudiants. Si on en a dix, il faut déclarer 10 variables. Si on en a n, il faut déclarer n variables.

=> ce n'est pas réaliste.

Il *faudrait pouvoir par l'intermédiaire* d'une seule variable stocker plusieurs valeurs de même type

=> c'est le rôle des tableaux

C'est ce que l'on nomme un type complexe (en opposition aux types simples vus précédemment).

Le type défini par un tableau est fonction :

- du nombre d'éléments maximal que peut contenir le tableau
- du type des éléments que peut contenir le tableau

Par exemple un tableau d'entiers de taille 10 et un tableau d'entiers de taille 20 sont deux types différents.

On peut utiliser directement des variables de type tableau, ou définir de nouveau type à partir du type tableau.

On utilise un type tableau via la syntaxe suivante :

Tableau[*intervalle*] de *Type des éléments stockés par le tableau*

Où *intervalle* est un intervalle sur un type simple dénombrable avec des bornes constantes.

Par exemple :

- **Type Notes = Tableau[1.. 26] de Entier**

  défini un nouveau type appelé Notes, qui est un tableau de 26 naturels

- **a : Notes**

  déclare une variable de type Notes

- **b : Tableau[1.. 26] de Entier**

  déclare une variable de type tableau de 26 naturels.
  a et b sont de même type.

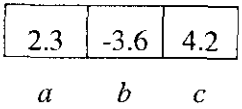
- **c : Tableau['a'.. 'z'] de Entier**

  déclare une variable de type tableau de 26 entiers.
  a et c sont de types différents

Ainsi l'extrait suivant :

```
tab : Tableau[ 'a'.. 'c'] de Réel
tab['a'] <--- 2.3
tab['b'] <--- -3.6
tab['c'] <--- 4.2
```

peut être présenté graphiquement par :



On accède (en lecture ou en écriture) à la i ème valeur d'un tableau en utilisant la syntaxe suivante :

nom de la variable [indice]

Par exemple, si *tab* est un tableau de 10 entiers (tab : Tableau[1..10] : Entier), alors

- tab[2] <--- -5 met la valeur -5 dans la 2 ème case du tableau.
- En considérant le cas où *a* est une variable de type Entier, a <--- tab[2] met la valeur de la 2 ème case du tableau tab dans a, c'est- à- dire -5.
- Lire(tab[1])
  met l'entier saisi par l'utilisateur dans la première case du tableau
- Ecrire(tab[1])
  affiche la valeur de la première case du tableau

Exemple :

Ecrire la procédure *afficherMoyenneUnePromotion* permettant d'afficher les notes d'une promotion, notes saisies par un enseignant :

L'analyse descendante de ce problème dit qu'il faut des opérations pour :

- Demander à l'enseignant quel est le nombre d'étudiants
- Demander à l'enseignant les notes des étudiants
- Calculer la moyenne des notes
- Afficher la moyenne des notes

L'algorithme est donc décomposé en quatre parties qui sont les suivantes :

- fonction obtenirNbEleves () : TaillePromotion

  Cette fonction a pour rôle de récupérer le nombre des étudiants

- procédure saisirNotesEleves (lesNotes : Notes, E nbEleves : TaillePromotion)

  Cette procédure permet de récupérer les notes des étudiants

- fonction calculerMoyenne (lesNotes : Notes, nbEleves : TaillePromotion) : Réel

  permet de récupérer la moyenne des notes

- procédure afficherMoyenne (laMoyenne : Réel)

  permet d'afficher la moyenne des notes

On défini les types TaillePromotion et Notes de la façon suivante :

Type TaillePromotion = 1.. MAX

Type Notes = Tableau[TaillePromotion] : Réel

avec Constante MAX = 100

a- 1er algorithme : Obtenir le nombre d'élèves

```
Fonction obtenirNbEleves () : TaillePromotion
Variable nb : Entier
Début
  Répéter
    Ecrire("Nombre d'élèves (compris entre 1 et ", MAX,") : ")
    Lire(nb)
  Jusqu'à ce que nb >= 1 et nb <= MAX
  Retourner nb
Fin
```

b- 2ème algorithme : Saisie des notes

```
Procédure saisirNotesEleves (lesNotes : Notes, nbEleves : TaillePromotion)
  Variable i : Entier
  Début
  Pour i de 1 à nbEleves
      Ecrire("Note de l'étudiant numéro ", i, " : ")
      Lire( lesNotes[i])
  i Suivant
Fin
```

c- 3ème algorithme : Calcul de la moyenne

```
Fonction calculerMoyenne (lesNotes : Notes, nbEleves : TaillePromotion) :
        Réel
Variables somme : Réel
          i : Entier
Début
   somme <--- 0
   Pour i de 1 à nbEleves
      somme <--- somme + lesNotes[i]
   i Suivant
   Retourner somme / nbEleves
Fin
```

d- 4ème algorithme : Affichage de la moyenne

```
Procédure afficherMoyenne (laMoyenne : Réel )
Début
   Ecrire("La moyenne de la promotion est de : ", laMoyenne)
Fin
```

e- 5ème algorithme : algorithme principal, à partir duquel on appelle les sous-programmes (fonctions et procédures)

```
Procédure afficherMoyenneUnePromotion ()
Variables notes : Notes
          nb : TaillePromotion
          moyenne : Réel
Début
   Nb <--- obtenirNbEleves()
   saisirNotesEleves(notes, nb)
   moyenne <--- calculerMoyenne(notes, nb)
   afficherMoyenne(moyenne)
Fin
```

Remarques :

- Un tableau possède un nombre maximal d'éléments défini lors de l'écriture de l'algorithme (les bornes sont des constantes implicites, par exemple 10, ou explicites, par exemple MAX). Ce nombre d'éléments ne peut être fonction d'une variable.
- Par défaut, si aucune initialisation n'a été effectuée les cases d'un tableau possèdent des valeurs aléatoires.
- Le nombre d'éléments maximal d'un tableau est différent du nombre d'éléments *significatifs dans un tableau*. *Dans l'exemple précédent* le nombre maximal d'éléments est de MAX, mais le nombre significatif d'éléments est référencé par la variable nb.
- L'accès aux éléments d'un tableau est direct (temps d'accès constant).
- Il n'y a pas conservation de l'information d'une exécution du programme à une autre.
- Les opérations de bases sur des tableaux de même type sont :
  - L'affectation (←) qui copie tous les éléments du tableau (opérande droite) dans un autre (opérande gauche).
  - L'égalité (=) qui permet de savoir si deux tableaux de même type possèdent des éléments de même valeur ($\forall i$, a[i]= b[i]).

- L'inégalité (≠ symbolisé aussi par <>) qui permet de savoir si deux tableaux de même type possèdent au moins un élément différent (∃ i, a[i] ≠ b[i]).

- Attention, on ne peut comparer deux tableaux que si ces derniers sont totalement remplis.

2- Les tableaux à deux dimensions :

On peut aussi avoir des tableaux à deux dimensions (permettant ainsi de représenter par exemple des matrices à deux dimensions).

On déclare une matrice à deux dimensions de la façon suivante :

Tableau[*intervallePremièreDimension*][*intervalleDeuxièmeDimension*] de *type des éléments*

On accéde (en lecture ou en écriture) à la i ème, j ème valeur d'un tableau en utilisant la syntaxe suivante :

nom de la variable [i][j]

Par exemple, si *tab* est défini par tab : Tableau[1..3][1..2] de Réel

- tab[2][1] <--- -1.2 met la valeur -1.2 dans la case 2,1 du tableau
- En considérant le cas où *a* est une variable de type Réel, a <--- tab[2][1] met -1.2 dans a

	1	2
1	7.2	5.4
2	-1.2	2
3	4	-8.5

Attention, le sens que vous donnez à chaque dimension est important et il ne faut pas en changer lors de l'utilisation du tableau.

Par exemple, le tableau tab défini de la façon suivante :

tab : Tableau[1..3][1..2] : Réel

tab[1][1] <--- 2.0; tab[2][1] <--- -1.2; tab[3][1] <--- 3.4

tab[1][2] <--- 2.6; tab[2][2] <--- -2.9; tab[3][2] <--- 0.5

peut permettre de représenter l'une des deux matrices suivantes :

$$\begin{pmatrix} 2.0 & -1.2 & 3.4 \\ 2.6 & -2.9 & 0.5 \end{pmatrix} \quad \begin{pmatrix} 2 & 2.6 \\ -1.2 & -2.9 \\ 3.4 & 0.5 \end{pmatrix}$$

3- Les tableaux à n dimensions :

Par extension, on peut aussi utiliser des tableaux à plus grande dimension.

- Leur déclaration est à l'image des tableaux à deux dimensions, c'est- à-dire :

  tableau [*intervalle1*][*intervalle2*] ... [*intervallen*] de *type des valeurs*

  Par exemple, tab : tableau[1..10][0.. 9]['a'..'e'] d'Entier

- Ainsi que leur utilisation :

  tab[2][1]['b'] <--- 10

  a <--- tab[2][1]['b']

4- Trier un tableau à une dimension :

4.1- Qu'est ce que le tri ?

On désigne par "**tri**" l'opération consistant à ordonner un ensemble d'éléments en fonction de *clés* sur lesquelles est définie une relation d'ordre.

Les algorithmes de tri ont une grande importance pratique. Ils sont fondamentaux dans certains domaines, comme l'informatique de gestion où l'on tri de manière quasi-systématique des données avant de les utiliser.

L'étude du tri est également intéressante en elle-même car il s'agit sans doute du domaine de l'algorithmique qui a été le plus étudié et qui a conduit à des résultats remarquables sur la construction d'algorithmes et l'étude de leur complexité.

Les tableaux permettent de stocker plusieurs éléments de même type au sein d'une seule entité. Lorsque ces éléments ne sont pas ordonnés, on peut donc les ranger en ordre croissant ou décroissant.

Trier un tableau c'est donc ranger les éléments d'un tableau en ordre croissant ou décroissant.

Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension pour le programmeur. Dans cet ouvrage, nous avons sélectionné pour vous trois méthodes de tri : le tri par sélection, le tri par insertion et le tri Bulle. Mais, avant de donner les algorithmes correspondants, il est plus judicieux de commencer par une procédure, à laquelle on fait toujours appel dans les algorithmes de tri : c'est la procédure *Echanger*.

4.2- La procédure échanger :

Tous les algorithmes de tri utilisent une procédure qui permet d'échanger (de permuter) la valeur de deux variables.

Dans le cas où les variables sont entières, la procédure échanger est la suivante :

```
Procédure échanger (a, b : Entier )
Variable temp : Entier
Début
  temp <--- a
  a <--- b
  b <--- temp
Fin
```

Si les variables sont d'un autre type, il suffit de changer l'instruction de déclaration de ces variables.

Remarquez que nous avons utilisé dans cet algorithme une variable **temp** pour stocker temporairement la valeur de l'une des deux variables (dans l'algorithme la variable a). Ainsi, nous ne risquons pas de perdre le contenu de cette variable lors de l'exécution de l'affectation a <--- b.

4.3- Le tri par minimum successif :

Le principe du tri par minimum successif, qu'on appelle aussi tri par sélection, est que, pour une place donnée, on sélectionne l'élément qui doit y être positionné.

De ce fait, si on parcourt la tableau de gauche à droite, on positionne à chaque fois le plus petit *élément qui se trouve* dans le sous tableau droit.

Ou plus généralement, pour trier le sous-tableau tab[i .. nbElements] il suffit de positionner au rang i le plus petit élément de ce sous-tableau et de trier le sous-tableau tab[i+1 .. nbElements].

Il nous faut donc une fonction qui soit capable de déterminer le plus petit élément (en fait l'indice du plus petit élément) d'un tableau à partir d'un certain rang. La définition de la fonction *indiceDuMinimum* qui réalise cette tâche est donnée juste après l'exemple suivant.

Exemple :

5	12	2	7	31	17	2	24	19	12	38	5

Rappelons-le que pour trier ce tableau avec la *méthode du tri par minimum* successif, il faut pour chaque position chercher le minimum dans la liste des éléments du tableau, à partie l'élément en cours jusqu'au dernier élément du tableau. Les tableaux suivants illustrent ce processus :

i=1

2	12	5	7	31	17	2	24	19	12	38	5

Au cours de la première étape, nous avons permuté entre la valeur du premier élément qui était égal à 5 avec la plus petite valeur stockée dans le tableau.

i=2

2	2	5	7	31	17	12	24	19	12	38	5

Au cours de la deuxième étape, nous avons permuté entre la valeur du deuxième élément qui était égal à 12 avec la plus petite valeur, du deuxième jusqu'au dernier élément tableau.

i=3

2	2	5	7	31	17	12	24	19	12	38	5

Au cours de cette étape, il n'y a pas eu de permutation puisque la valeur 5 est le minimum des éléments du tableau, du troisième jusqu'au dernier.

i=4

2	2	5	5	31	17	12	24	19	12	38	7

Permutation entre le quatrième élément (qui avait pour valeur 7) et le dernier élément du tableau dont la valeur 5 représentait le minimum du quatrième au dernier élément du tableau.

i=5

2	2	5	5	7	17	12	24	19	12	38	31

Permutation entre le cinquième et le dernier élément du tableau.

i=6

2	2	5	5	7	12	17	24	19	12	38	31

Permutation entre le sixième et le septième élément du tableau.

i=7

2	2	5	5	7	12	12	24	19	17	38	31

Permutation entre le septième et le dixième élément du tableau.

i=8

2	2	5	5	7	12	12	17	19	24	38	31

Permutation entre le huitième et le dixième élément du tableau.

i=9

2	2	5	5	7	12	12	17	19	24	38	31

Au cours de cette étape, il n'y a pas eu de permutation puisque aucune valeur à partir du neuvième élément du tableau ne lui est plus petite.

i=10

2	2	5	5	7	12	12	17	19	24	38	31

Pas de permutation pour le dixième élément du tableau.

i=11

2	2	5	5	7	12	12	17	19	24	31	38

Au cours de la onzième et dernière étape, il y a eu permutation entre le onzième et le dernier élément du tableau.

Nous sommes arrivés à la dernière étape, nous obtenons donc le tableau suivant qui est trié par ordre croissant :

2	**2**	**5**	**5**	**7**	**12**	**12**	**17**	**19**	**24**	**31**	**38**

Fonction *indiceDuMinimum*

```
Fonction indiceDuMinimum (tab : Tableau[1..MAX] d'Entier; rang,
                          nbElements : Entier) : Entier
Variables i, indiceCherche : Entier
Début
   indiceCherche <--- rang
   Pour i de rang + 1 à nbElements
```

```
        Si tab[i]< tab[indiceCherche] Alors
            indiceCherche <--- i
        Finsi
    i Suivant
    Retourner indiceCherche
Fin
```

L'algorithme de tri par minimum successif est donc :

```
Procédure effectuerTriParMinimumSuccessif (t : Tableau[1..MAX] d'Entier;
                                           nbElements : Entier )
Variables i, indice : Entier
Début
  Pour i de 1 à nbElements - 1
     indice <--- indiceDuMinimum(t, i, nbElements)
     Si i ≠ indice Alors
        echanger(t[i], t[indice])
     Fin si
  i Suivant
Fin
```

Enfin, rappelons que **echanger** est la procédure qui permet de permuter deux valeurs. Si cette procédure n'a pas été défini et que vous souhaitez écrire directement les instructions qui permettent d'échanger les valeurs de deux variables, sans faire appel à la procédure **echanger**, il faudra alors écrire les instructions suivantes à la place de l'instruction **echanger(t[i], t[indice])** :

```
temp <--- t[i]
t[i] <--- t[indice]
t[indice] <--- temp
```

Il ne faut oublier bien sûr de déclarer la variable **temp** de même type que les éléments du tableau, c'est-à-dire de type Entier. Cette variable sert, rappelons-le, à sauvegarder la valeur de l'élément t[i] avant qu'on lui affecte la valeur de

t[indice]. Sans cela, la valeur de l'élément t[i] sera perdue après exécution de l'affectation : t[i] <--- t[indice].

Note : Dans tous les cas, le nombre d'exécutions de la boucle interne étant $n(n-1)/2$. Par exemple, pour un tableau de 20 éléments, il faut exécuter 190 fois la boucle.

4.4- Le tri par insertion :

Soit à trier une suite $a_1, \ldots, a_n$ en ordre croissant.

Le principe du tri par insertion consiste à insérer le i_ème élément dans la suite des i - 1 éléments déjà triés.

Supposons que la valeur du i ème élément est comprise entre celle du j ème et j+1 ème élément. J et J +1 sont des nombres entiers positifs, compris entre 1 et i-1.

Il faut donc placer la valeur du i ème élément dans la case du j+1 ème élément, en faisant décaler les éléments du tableau, à partir du j+1 ème élément jusqu'au i ème élément.

Enfin, il faut noter que le processus du tri commence à partir du 2ème élément du tableau.

Exemple :

5	12	2	7	31	17	2	24	19	12	38	5

Nous allons trier ce tableau en utilisant la méthode du tri par insertion. Il s'agit donc de suivre les étapes suivantes :

i=2

5	12	2	7	31	17	2	24	19	12	38	5

On ne déplace pas le deuxième élément du tableau, puisque sa valeur est supérieure à celle du 1er élément.

i=3

2	5	12	7	31	17	2	24	19	12	38	5

Le troisième élément du tableau (qui avait pour valeur 2) a été déplacé pour prendre la place du 1er élément ; le 1er et le 2ème élément du tableau, qui avaient pour valeur 5 et 12 respectivement, ont été décalé vers la droite.

i=4

2	5	7	12	31	17	2	24	19	12	38	5

Le quatrième élément du tableau a été déplacé pour prendre la place du 3ème élément ; celui-ci a été déplacé d'une position vers la droite.

i=5

2	5	7	12	31	17	2	24	19	12	38	5

On ne déplace pas le cinquième élément du tableau, puisque sa valeur est supérieure à celle des éléments qui le précèdent.

i=6

2	5	7	12	17	31	2	24	19	12	38	5

Le sixième élément du tableau a été déplacé pour prendre la place de celui qui le précède; le cinquième élément a été déplacé d'une position vers la droite.

i=7

2	2	5	7	12	17	31	24	19	12	38	5

Le septième élément du tableau a été déplacé pour prendre la place du 2ème élément ; le 2ème, 3ème, 4ème, 5ème et 6ème élément ont été déplacé chacun d'une position vers la droite.

i=8

2	2	5	7	12	17	24	31	19	12	38	5

Le huitième élément du tableau a été déplacé pour prendre la place de celui qui le précède; le septième élément a été déplacé d'une position vers la droite.

i=9

2	2	5	7	12	17	19	24	31	12	38	5

Le neuvième élément a été déplacé pour prendre la place du 7ème élément ; le 7ème et le 8ème élément du tableau ont été déplacé chacun d'une position vers la droite.

i=10

2	2	5	7	12	12	17	19	24	31	38	5

Le dixième élément a été déplacé pour prendre la place du 6ème élément ; le 6ème, 7ème, 8ème et 9ème élément du tableau ont été déplacé chacun d'une position vers la droite.

i=11

2	2	5	7	12	12	17	19	24	31	38	5

On ne déplace pas le onzième élément du tableau, puisque sa valeur est supérieure à celle de tous les éléments qui le précèdent.

i=12

2	2	5	5	7	12	12	17	19	24	31	38

Au cours de cette étape (qui est la dernière), le douzième et dernier élément a été déplacé pour prendre la place du 4ème élément ; le 4ème, 5ème, 6ème, 7ème, 8ème, 9ème, 10ème et 11ème élément du tableau ont été déplacé chacun d'une position vers la droite.

Nous somme arrivés à la dernière étape, nous obtenons donc le tableau suivant qui est trié par ordre croissant :

2	2	5	5	7	12	12	17	19	24	31	38

Voici en pseudo-code l'algorithme du tri par insertion :

```
Procédure Tri-insertion(Tableau tab[1..n] d'Entier) : Réel
Variables i, j : Entier
          temp : Réel
Début
  Pour i de 2 à n
     temp <--- tab[ i ]
     j <--- i - 1
     Tant que j > 0 et tab[ j ] > temp Faire
        tab[ j + 1 ] <--- tab[ j ]
        j  <--- j -1
     Fin TantQue
     tab [ j + 1 ] <--- temp
  i Suivant
Fin
```

Dans la boucle sur i , on insère tab [*i*] dans le tableau, trié entre 0 et i - 1. La variable temp permet de ne pas perdre la valeur que l'on échange.

Note : Dans le pire des cas, les parcours successifs du tableau imposent d'effectuer *(n-1)+(n-2)+(n-3)..+1* comparaisons. Le nombre total de comparaisons est donc majoré par *n*(n-1)/2* Le nombre de comparaisons est donc *n*(n-1)/4* en moyenne et *n*(n-1)/2* en valeur maximale, c'est à dire quand le tableau est initialement trié "à l'envers".

Par exemple, pour un tableau de 20 éléments, il faut effectuer 190 comparaisons au maximum et 95 comparaisons en moyenne.

4.5- Le tri Bulle :

Le principe du tri bulle (*bubble sort*) est de comparer deux à deux les éléments E1 et E2 consécutifs d'un tableau et d'effectuer une permutation si E1 > E2. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

En d'autres termes, il faut balayer le tableau, puis :

- s'il n'y a eu aucune permutation, on arrête le processus de tri : le tableau est trié du moment qu'il n'y a pas eu de permutation.
- s'il y a eu au moins une permutation, il faudra dans ce cas balayer une seconde fois le tableau. Et ainsi de suite, jusqu'à tomber sur le cas précédent (balayage du tableau sans qu'il y ait une permutation).

Exemple :

5	12	2	7	31	17	2	24	19	12	38	5

Nous allons trier ce tableau en utilisant la méthode du tri Bulle. Il s'agit donc de suivre les étapes suivantes :

1er balayage du tableau :

i=1

5	12	2	7	31	17	2	24	19	12	38	5

Il n'y a pas eu de permutation, puisque la valeur du premier élément est inférieure à celle du deuxième élément du tableau.

i=2

5	2	12	7	31	17	2	24	19	12	38	5

Il y a eu permutation entre le deuxième et le troisième élément du tableau, puisque la valeur du deuxième élément (qui était de 12) est supérieure à celle du troisième élément (qui était de 2).

i=3

5	2	7	12	31	17	2	24	19	12	38	5

Il y a eu permutation entre le troisième et le quatrième élément du tableau.

i=4

5	2	7	12	31	17	2	24	19	12	38	5

Il n'y a pas eu de permutation.

i=5

5	2	7	12	17	31	2	24	19	12	38	5

Il y a eu permutation entre le cinquième et le sixième élément du tableau.

i=6

5	2	7	12	17	2	31	24	19	12	38	5

Il y a eu permutation entre le sixième et le septième élément du tableau.

i=7

5	2	7	12	17	2	24	31	19	12	38	5

Il y a eu permutation entre le septième et le huitième élément du tableau.

i=8

5	2	7	12	17	2	24	19	31	12	38	5

Il y a eu permutation entre le huitième et le neuvième élément du tableau.

i=9

5	2	7	12	17	2	24	19	12	31	38	5

Il y a eu permutation entre le neuvième et le dixième élément du tableau.

i=10

5	2	7	12	17	2	24	19	12	31	38	5

Il n'y a pas eu de permutation.

i=11

5	2	7	12	17	2	24	19	12	31	5	38

Il y a eu permutation entre le onzième et le douzième élément du tableau.

Nous avons balayé le tableau, mais comme il y a eu au moins une permutation il faut donc le balayer une seconde fois.

2ème balayage du tableau :

i=1

2	5	7	12	17	2	24	19	12	31	5	38

Il y a eu permutation entre le premier et le deuxième élément du tableau.

i=2

2	5	7	12	17	2	24	19	12	31	5	38

Il n'y a pas eu de permutation.

i=3

2	5	7	12	17	2	24	19	12	31	5	38

Il n'y a pas eu de permutation.

i=4

2	5	7	12	17	2	24	19	12	31	5	38

Il n'y a pas eu de permutation.

i=5

2	5	7	12	2	17	24	19	12	31	5	38

Il y a eu permutation entre le cinquième et le sixième élément du tableau.

i=6

2	5	7	12	2	17	24	19	12	31	5	38

Il n'y a pas eu de permutation.

i=7

2	5	7	12	2	17	19	24	12	31	5	38

Il y a eu permutation entre le septième et le huitième élément du tableau.

i=8

2	5	7	12	2	17	19	12	24	31	5	38

Il y a eu permutation entre le huitième et le neuvième élément du tableau.

i=9

2	5	7	12	2	17	19	12	24	31	5	38

Il n'y a pas eu de permutation.

i=10

2	5	7	12	2	17	19	12	24	5	31	38

Il y a eu permutation entre le dixième et le onzième élément du tableau.

i=11

2	5	7	12	2	17	19	12	24	5	31	38

Il n'y a pas eu de permutation.

Il y a eu au moins une permutation, il faut donc continuer à balayer le tableau.

3ème balayage du tableau :

i=1	2	5	7	12	2	17	19	12	24	5	31	38

Il n'y a pas eu permutation.

i=2	2	5	7	12	2	17	19	12	24	5	31	38

Il n'y a pas eu permutation.

i=3	2	5	7	12	2	17	19	12	24	5	31	38

Il n'y a pas eu permutation.

i=4	2	5	7	2	12	17	19	12	24	5	31	38

Il y a eu permutation entre le quatrième et le cinquième élément du tableau.

i=5	2	5	7	2	12	17	19	12	24	5	31	38

Il n'y a pas eu permutation.

i=6	2	5	7	2	12	17	19	12	24	5	31	38

Il n'y a pas eu permutation.

i=7	2	5	7	2	12	17	12	19	24	5	31	38

Il y a eu permutation entre le septième et le huitième élément du tableau.

i=8	2	5	7	2	12	17	12	19	24	5	31	38

Il n'y a pas eu permutation.

i=9	2	5	7	2	12	17	12	19	5	24	31	38

Il y a eu permutation entre le neuvième et le dixième élément du tableau.

i=10	2	5	7	2	12	17	12	19	5	24	31	38

Il n'y a pas eu permutation.

i=11	2	5	7	2	12	17	12	19	5	24	31	38

Il n'y a pas eu permutation.

Il y a eu au moins une permutation, il faut donc continuer à balayer le tableau.

4ème balayage du tableau :

i=1

2	5	7	2	12	17	12	19	5	24	31	38

Il n'y a pas eu permutation.

i=2

2	5	7	2	12	17	12	19	5	24	31	38

Il n'y a pas eu permutation.

i=3

2	5	2	7	12	17	12	19	5	24	31	38

Il y a eu permutation entre le troisième et le quatrième élément du tableau.

i=4

2	5	2	7	12	17	12	19	5	24	31	38

Il n'y a pas eu permutation.

i=5

2	5	2	7	12	17	12	19	5	24	31	38

Il n'y a pas eu permutation.

i=6

2	5	2	7	12	12	17	19	5	24	31	38

Il y a eu permutation entre le sixième et le septième élément du tableau.

i=7

2	5	2	7	12	12	17	19	5	24	31	38

Il n'y a pas eu permutation.

i=8

2	5	2	2	12	12	17	5	19	24	31	38

Il y a eu permutation entre le huitième et le neuvième élément du tableau.

i=9

2	5	2	7	12	12	17	5	19	24	31	38

Il n'y a pas eu permutation.

i=10

2	5	2	7	12	12	17	5	19	24	31	38

Il n'y a pas eu permutation.

i=11

2	5	2	7	12	12	17	5	19	24	31	38

Il n'y a pas eu permutation.

Il y a eu trois permutations, on continue donc à balayer le tableau pour d'éventuelles permutations. On procède donc ainsi jusqu'à ce qu'on balaye le tableau et qu'il n'y ait plus de permutation.

A la fin, nous obtenons le tableau suivant qui est trié par ordre croissant :

2	2	5	5	7	12	12	17	19	24	31	38

Voici, en pseudo-code, l'algorithme du tri bulle :

```
Procédure Tri_bulle (Tableau tab[1..n]) : Entier
Variable permut : Booleen;
         Temp : Booleen;
Début
  Répéter
      permut = FAUX
      Pour i de 1 à n-1
          Si tab[i] > tab[i+1] Alors
              echanger a[i] et a[i+1]
              permut = VRAI
          Fins si
      i Suivant
  Jusqu'à permut = FAUX
Fin
```

Note : Dans le pire des cas (quand le tableau est initialement trié "à l'envers"), il faut effectuer *n* fois le parcours du tableau, *n * (n-1)* comparaisons et *n * (n-1)* permutations. Donc, il faut n * (n-1) comparaisons au maximum et n * (n-1) / 2 comparaisons en moyenne.

Par exemple, pour un tableau de 20 éléments, il faut effectuer 380 comparaisons au maximum et 190 comparaisons en moyenne.

5- La recherche dichotomique :

Soit un vecteur A (tableau mono-dimensionnel) de dimension N (il comporte N éléments) dans lequel nous devons rechercher la position d'un élément y apparaissant éventuellment. La méthode la plus rudimentaire consiste à passer tous les éléments du tableau en revue en commençant par le premier et en s'arrêtant lorsque l'élément est trouvé ou lorsque le dernier élément est atteint: il s'agit d'une recherche séquentielle. Ce type de recherche impose un nombre de tests compris entre 1 (si l'élément recherché est en 1ère position) et N (si l'élément recherché est en dernière position), pour une moyenne de (1+2+...+N)/N = [(1+N)N/2]/N = (1+N)/2 tests. Par exemple, pour un tableau à 25 éléments, il faut en moyenne 13 tests.

Lorsque l'information contenue dans le tableau n'est pas triée ou triée suivant un critère qu'il n'est pas possible d'exploiter, cette recherche est la seule possible mais si le tableau est trié sur un critère exploitable, nous allons pouvoir diminuer notablement le nombre d'opérations à effectuer par une recherche dichotomique.

Le principe de la recherche dichotomique est de comparer le nombre à chercher avec l'élément se trouvant au milieu du tableau. Trois cas peuvent se présenter :

- s'ils sont égaux, c'est que la recherche a abouti, il faut donc retourner (ou mémoriser) l'indice de l'élément dans le tableau.
- si le nombre à chercher est inférieur à l'élément se trouvant au milieu du tableau, il faut procéder comme suit :
  - scinder le tableau en deux,
  - ignorer la partie droite du tableau,
  - et refaire la recherche dans la partie gauche
- si le nombre à chercher est supérieur à l'élément se trouvant au milieu du tableau, il faut :
  - scinder le tableau en deux,
  - ignorer la partie gauche du tableau,
  - et refaire la recherche dans la partie droite

On doit continuer à chercher jusqu'à trouver la valeur à chercher ou jusqu'à ce qu'on puisse plus scinder le tableau en deux, c'est-à-dire lorsque la partie du tableau dans laquelle on doit effectuer la recherche comporte un seul élément.

Imaginons que nous devions trouver le nombre 4 dans le tableau trié ci-dessous:

1	2	3	4	5	6	7	8	9	10	11
1	3	3	5	7	**8**	8	8	9	9	10

Si nous comparons le nombre à chercher à l'élément (en gras) se trouvant au milieu du tableau, nous constatons qu'il doit se trouver (s'il est dans le tableau) dans la partie se trouvant devant l'élément en gras. Et nous pouvons même préciser qu'il doit se trouver entre les positions 1 (le début du tableau) et 5, puisqu'il n'est pas en position 6 qui stocke l'élément qu'on vient de comparer avec le nombre à chercher.

1	2	3	4	5	6	7	8	9	10	11
1	3	**3**	5	7	8	8	8	9	9	10

Si nous comparons le nombre à chercher à l'élément se trouvant au milieu de la partie du tableau dont les limites sont indiquées par les cases ombrées (le fond grisé), nous constatons qu'il doit se trouver (s'il est dans le tableau) dans la 2ème partie (celle se trouvant après l'élément en gras). Et nous pouvons même préciser qu'il doit se trouver entre les positions 4 (puisqu'il n'est pas en position 3) et 5.

1	2	3	4	5	6	7	8	9	10	11
1	3	3	**5**	7	8	8	8	9	9	10

Si nous comparons le nombre à chercher à l'élément (en gras) se trouvant au milieu de la partie du tableau dont les limites sont indiquées par les cases ombrées, nous constatons qu'il doit se trouver (s'il est dans le tableau) dans la 1ère partie (celle se trouvant avant l'élément en gras). Et nous pouvons même préciser qu'il doit se trouver entre les positions 3 et 3 (puisqu'il n'est pas en position 4).

1	2	3	4	5	6	7	8	9	10	11
1	3	3	5	7	8	8	8	9	9	10

Si nous comparons le nombre à chercher à l'élément (en gras) se trouvant à la troisième position, nous constatons qu'il est différent à cette valeur => nous sommes donc arrivés à la conclusion que 4 ne se trouve pas dans le tableau.

Nous pouvons constater qu'à la première étape, nous devions prendre le milieu d'un tableau de N éléments, à l'étape suivante le milieu d'un tableau de N/2 éléments, puis de N/4 éléments, ...

Dans le plus mauvais des cas, la recherche s'arrête quand le nombre d'éléments est réduit à 1. Si k correspond au nombre de recherches (c'est-à-dire le nombre de tests), la recherche s'arrête donc, au pire des cas, lorsque $N/2^k \leq 1$, c'est-à-dire quand $N \leq 2^k$. Nous pouvons donc facilement déduire que la recherche s'arrête lorsque k est le plus petit entier supérieur ou égal à $\log_2 N$ (logarithme à base 2 du nombre N).

Pour 1024 éléments, nous nous en tirerons donc avec un maximum de 10 étapes, à rapprocher des 1024 étapes de la recherche séquentielle!

Voici l'algorithme en pseudo-code :

```
Fonction chercheDicho(eltAChercher, n : Entier ; tab: Tableau[1..n] à valeur
                      entier)
Variable min, max, posInter, position : Entier
Debut
  min <--- 1
  max <--- n
  posInter <--- (min + max)\2   (\ symbolise une division entière)
  Tant que Non ((max - min ≤ 1) ou (eltAChercher=tab[posInter]) Faire
     Si eltAChercher < tab[posInter] Alors
        max <--- posInter
     Sinon
        min <--- posInter
     Fin si
     posInter <--- (min + max) \ 2
  Fin TantQue
  Si eltAChercher = tab[posInter] Alors
     position <--- posInter
  Sinon
     position <--- 0
```

```
    Fin si
    Si position = 0 Alors
        Ecrire (eltAChercher," n'a pas été trouvé dans le tableau")
    Sinon
        Ecrire (eltAChercher," a été trouvé en position ", position, " dans le
                tableau"
    Fin si
Fin
```

Exercices

Exercice 5.1

Ecrire un algorithme qui déclare et remplit un tableau de 7 valeurs numériques en les mettant toutes à zéro.

Exercice 5.2

Ecrire un algorithme qui déclare et remplit un tableau contenant les six voyelles de l'alphabet latin.

Exercice 5.3

Ecrire un algorithme qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur.

Exercice 5.4

Que produit l'algorithme suivant ?

```
Tableau Nb[1..6] en Entier
Variable i en Entier
Début
Pour i de 1 à 6
    Nb[i] <--- i * i
i Suivant
Pour i de 1 à 6
```

```
    Ecrire Nb(i)
i Suivant
Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?

Exercice 5.5

Que produit l'algorithme suivant ?

```
Tableau N[1..7] en Entier
Variables i, k en Entier
Début
N[1] <--- 1
Pour k de 2 à 7
    N[k] <--- N[k-1] + 2
k Suivant
Pour i de 1 à 7
    Ecrire N[i]
i Suivant
Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?

Exercice 5.6

Que produit l'algorithme suivant ?

```
Tableau Suite[1..8] en Entier
Variable i en Entier
Début
```

```
Suite[1] <--- 1
Suite[2] <--- 1
Pour i de 3 à 8
     Suite[i] <--- Suite[i-1] + Suite[i-2]
i suivant
Pour i de 1 à 8
     Ecrire Suite[i]
i suivant
Fin
```

Exercice 5.7

Ecrivez la fin de l'algorithme 5.3 afin que le calcul de la moyenne des notes soit effectué et affiché à l'écran.

Exercice 5.8

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

Exercice 5.9

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

Exercice 5.10

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

Exemple :

Tableau 1 :	4	8	7	9	1	5	4	6
Tableau 2 :	7	6	5	2	1	3	7	4
Tableau à constituer :	11	14	12	11	2	8	11	10

Exercice 5.11

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

Exemple :

Tableau 1 :	4	8	7	12

Tableau 2 :	3	6

Le Schtroumpf :

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

Exercice 5.12

Écrivez un algorithme qui permet la saisie d'un nombre quelconque de valeurs, sur le principe de l'ex 5.8. Toutes les valeurs doivent être ensuite augmentées de 1, et le nouveau tableau sera affiché à l'écran.

Exercice 5.13

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

Exercice 5.14

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

Exercice 5.15

Ecrivez un algorithme qui permet de saisir un nombre quelconque de valeurs, et qui les range au fur et à mesure dans un tableau. Le programme, une fois la saisie terminée, doit dire si les éléments du tableau sont tous consécutifs ou non.

Par exemple, si le tableau est : 12 - 13 - 14 - 15 - 16 - 17 - 18, ses éléments sont tous consécutifs. Si le tableau est : 9 - 10 - 11 - 15 - 16 - 17, ses éléments ne sont pas tous consécutifs.

Exercice 5.16

Ecrivez un algorithme qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement saisi (« les premiers seront les derniers... »)

Exercice 5.17

Ecrivez un algorithme qui permet à l'utilisateur de supprimer une valeur d'un tableau préalablement saisi. L'utilisateur donnera l'indice de la valeur qu'il souhaite supprimer. Attention, il ne s'agit pas de remettre une valeur à zéro, mais bel et bien de la supprimer du tableau lui-même ! Si le tableau de départ était

12 – 8 – 4 – 45 – 64 – 9 – 2 – 7

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

12 – 8 – 4 – 45 – 9 – 2 - 7

Exercice 5.18

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

Exercice 5.19

Ecrivez un algorithme remplissant un tableau de 6 sur 13, avec des zéros.

Exercice 5.20

Quel résultat produira cet algorithme ?

```
Tableau X[1..2, 1..3] en Entier
Variables i, j, val en Entier
Début
Val <--- 1
Pour i de 1 à 2
    Pour j de 1 à 3
        X[i, j] <--- Val
        Val <--- Val + 1
    j Suivant
i Suivant
Pour i de 1 à 2
    Pour j de 1 à 3
        Ecrire X[i, j]
    j Suivant
i Suivant
Fin
```

Exercice 5.21

Quel résultat produira cet algorithme ?

```
Tableau X[1..2, 1..3] en Entier
Variables i, j, val en Entier
Début
Val <--- 1
Pour i de 1 à 2
   Pour j de 1 à 3
        X[i, j] <--- Val
        Val <--- Val + 1
```

```
   j Suivant
i Suivant
Pour j de 1 à 3
    Pour i de 1 à 2
        Ecrire X[i, j]
    i Suivant
j Suivant
Fin
```

Exercice 5.22

Quel résultat produira cet algorithme ?

```
Tableau T[1..4, 1..2] en Entier
Variables k, m en Entier
Début
Pour k de 1 à 4
    Pour m de 1 à 2
        T[k, m] <--- k + m
        m Suivant
k Suivant
Pour k de 1 à 4
    Pour m de 1 à 2
        Ecrire T[k, m]
        m Suivant
k Suivant
Fin
```

Exercice 5.23

Mêmes questions, en remplaçant la ligne :

```
T[k, m] <--- k + m
```

Par

a) `T[k, m] <--- 2 * k + (m + 1)`

b) `T[k, m] <--- (k + 1) + 4 * m`

Exercice 5.24

Soit un tableau T à deux dimensions [1..13, 1..9) préalablement rempli de valeurs numériques.

Ecrire un algorithme qui recherche la plus grande valeur au sein de ce tableau.

CORRIGES DES EXERCICES

Exercice 5.1

```
Tableau tab[1..7] en Entier
Variable i en Entier
Debut
Pour i de 1 à 7
     tab[i] <--- 0
i Suivant
Fin
```

Exercice 5.2

```
Tableau tab[1..6] en Caractère
Debut
   tab[1] <--- "a"
   tab[2] <--- "e"
   tab[3] <--- "i"
   tab[4] <--- "o"
   tab[5] <--- "u"
   tab[6] <--- "y"
Fin
```

Exercice 5.3

```
Tableau Notes[1..9] en Entier
Variable i en Entier
Début
Pour i de 1 à 9
   Ecrire "Entrez la note numéro ", i + 1
   Lire Notes[i]
i Suivant
Fin
```

Exercice 5.4

Cet algorithme remplit un tableau avec six valeurs : 1, 4, 9, 16, 25 et 36. Il les écrit ensuite à l'écran.

On peut simplifier l'algorithme en l'écrivant comme suit :

```
Tableau Nb[1..6] en Entier
Variable i en Entier
Début
Pour i de 1 à 6
    Nb[i] <--- i * i
    Ecrire Nb[i]
i Suivant
Fin
```

Exercice 5.5

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

L'algorithme peut être simplifié en l'écrivant comme suit :

```
Tableau N[1..7] en Entier
Variables i, k en Entier
Début
N[1] <--- 1
Ecrire N[1]
Pour k de 2 à 7
    N[k] <--- N[k-1] + 2
    Ecrire N[k]
k Suivant
Fin
```

Exercice 5.6

Cet algorithme remplit un tableau de 8 valeurs : 1, 1, 2, 3, 5, 8, 13, 21 puis les affiche à l'écran.

Exercice 5.7

```
Variable S en Entier
Tableau Notes[1..9] en Entier
Debut
S <--- 0
Pour i de 1 à 9
     Ecrire ("Entrez la note n° ", i)
     Lire Notes[i]
     S <--- S + Notes[i]
i Suivant
Ecrire ("Moyenne : ", S/9)
Fin
```

Exercice 5.8

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

Exercice 5.9

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

Exercice 5.10

```
Variables i, N en Entier
Tableaux T1[1..N], T2[1..N], T3[1..N] en Entier
Debut
... (on suppose que T1 et T2 comptent N éléments, et qu'ils sont déjà saisis)
Pour i de 1 à N
    T3[i] <--- T1[i] + T2[i]
i Suivant
Fin
```

Exercice 5.11

```
Variables i, j, N1, N2, S en Entier
```

... On ne programme pas la saisie des tableaux T1 et T2.

On suppose que T1 possède N1 éléments, et que T2 en possède N2)

```
Tableaux T1[1..N1], T2[1..N2] en Entier
Debut
S <--- 0
Pour i de 1 à N1
     Pour j de 1 à N2
          S <--- S + T1[i] * T2[j]
     j Suivant
i Suivant
Ecrire ("Le schtroumpf est : ", S)
Fin
```

Exercice 5.12

```
Variables Nb, i en Entier
Tableau T[] en Entier
Debut
Ecrire "Entrez le nombre de valeurs : "
Lire Nb
Redim T[1..Nb]
Pour i de 1 à Nb
     Ecrire ("Entrez le nombre n° ", i)
     Lire T[i]
i Suivant
Ecrire "Nouveau tableau : "
Pour i de 1 à Nb
     T[i] <--- T[i] + 1
     Ecrire T[i]
i Suivant
Fin
```

Exercice 5.13

```
Variables Nb, Posmaxi en Entier
Tableau T[] en Entier
Ecrire "Entrez le nombre de valeurs : "
Lire Nb
Redim T[1..Nb]
Pour i de 1 à Nb
     Ecrire ("Entrez le nombre n° ", i)
     Lire T[i]
i Suivant
Posmaxi <--- 1
Pour i de 1 à Nb
     Si T[i] > T[Posmaxi] alors
          Posmaxi <--- i
     Finsi
i Suivant
Ecrire ("Element le plus grand : ", T[Posmaxi])
Ecrire ("Position de cet élément : ", Posmaxi)
Fin
```

Exercice 5.14

```
Variables Nb, i, Som, Moy, Nbsup en Entier
Tableau T[] en Entier
Debut
Ecrire "Entrez le nombre de notes à saisir : "
Lire Nb
Redim T[1..Nb]
Pour i de 1 à Nb
     Ecrire ("Entrez le nombre n° ", i)
     Lire T[i]
```

```
i Suivant
Som <--- 0
Pour i de 1 à Nb
     Som <--- Som + T[i]
i Suivant
Moy <--- Som / Nb
NbSup <--- 0
Pour i de 1 à Nb
     Si T[i] > Moy Alors
          NbSup <--- NbSup + 1
     FinSi
i Suivant
Ecrire (NbSup, " élèves dépassent la moyenne de la
        classe")
Fin
```

Exercice 5.15

```
Variables Nb, i en Entier
Variable Flag en Booleen
Tableau T[] en Entier
Debut
Ecrire "Entrez le nombre de valeurs : "
Lire Nb
Redim T[1..Nb]
Pour i de 1 à Nb
     Ecrire ("Entrez le nombre n° ", i)
     Lire T[i]
i Suivant
Flag <--- Vrai
Pour i de 2 à Nb
     Si T[i] <> T[i - 1] + 1 Alors
          Flag <--- Faux
     FinSi
```

```
i Suivant
Si Flag Alors
    Ecrire "Les nombres sont consécutifs"
Sinon
    Ecrire "Les nombres ne sont pas consécutifs"
FinSi
Fin
```

Cette programmation est sans doute la plus spontanée, mais elle présente le défaut d'examiner la totalité du tableau, même lorsqu'on découvre dès le départ deux éléments non consécutifs. Une autre manière de procéder serait de sortir de la boucle dès que deux éléments non consécutifs sont détectés. L'algorithme deviendrait donc :

```
Variables Nb, i en Entier
Variable Flag en Booleen
Tableau T[] en Entier
Debut
Ecrire "Entrez le nombre de valeurs : "
Lire Nb
Redim T[1..Nb]
Pour i de 1 à Nb
     Ecrire ("Entrez le nombre n° ", i)
     Lire T[i]
i Suivant
i <--- 2
TantQue (T[i] = T[i - 1] + 1) et i < Nb Faire
     i <--- i + 1
FinTantQue
Si T[i] = T[i - 1] + 1 Alors
     Ecrire "Les nombres sont consécutifs"
   Sinon
     Ecrire "Les nombres ne sont pas consécutifs"
FinSi
Fin
```

Exercice 5.16

On suppose que n est le nombre d'éléments du tableau préalablement saisi

...

```
//Calculer la division entière de N par 2
M <--- n \ 2
Pour i de 1 à m
     temp <--- T[i]
     T[i] <--- T[N+1-i]
     T[N+1-i] <--- temp
i suivant
Fin
```

Exercice 5.17

Pour un tel traitement, on n'aura pas le choix ; il va falloir passer par un deuxième tableau, temporaire. L'algorithme qui suit suppose que le tableau T, comptant N éléments, a déjà été saisi.

...

```
Tableau temp[1 .. N - 1]
Ecrire "Rang de la valeur à supprimer ? "
Lire S
Pour i de 1 à N
     Si i < S Alors
          temp[i] <--- T[i]
       Sinon
          Si i > S Alors
               temp[i-1] <--- T[i]
          Finsi
    Finsi
i suivant
// On recopie temp dans T
```

```
Redim T[1 .. N - 1]
Pour i de 1 à N - 1
     T[i] <--- temp[i]
i suivant
Fin
```

Exercice 5.18

N est le nombre d'éléments du tableau DICO, contenant les mots du dictionnaire, tableau préalablement rempli. Pour rechercher le mot, nous allons utiliser la méthode de recherche dichotomique, puisque le tableau est trié par ordre croissant et, dans ce cas, la recherche serait plus rapide.

```
Tableau dico[1..n] en Entier

Variables sup, inf, comp, n en Entier
Variables fini en Booléen
Variables mot en Caractère
Début
Ecrire "Entrez le mot à vérifier "
Lire mot
```

On définit les bornes de la partie du tableau à considérer

```
sup <--- n
inf <--- 1
fini <--- Faux
TantQue Non fini
```

Comp désigne l'indice de l'élément à comparer. En bonne rigueur, il faudra veiller à ce que Comp soit bien un nombre entier, ce qui pourra s'effectuer de différentes manières selon les langages.

```
     Comp <--- (sup + inf) \ 2
```

Rappelez-vous que le symbole \ désigne la division entière

Si le mot se situe avant le point de comparaison, alors la borne supérieure change, la borne inférieure ne bouge pas.

```
     Si mot < dico(comp) Alors
```

```
            sup <--- comp - 1
        Sinon, c'est l'inverse ( c'est la borne inférieure qui change)
          Sinon
            inf <--- comp + 1
        FinSi
        fini <--- mot = dico(comp) ou sup < inf
Fin TantQue
Si Mot = dico(comp) Alors
      Ecrire "Le mot existe"
   Sinon
      Ecrire "Le mot n'existe pas"
Finsi
Fin
```

Exercice 5.19

```
Tableau tab[1..6, 1..13] en Entier
Debut
Pour i de 1 à 6
     Pour j de 1 à 13
          tab[i, j] <--- 0
     j Suivant
i Suivant
Fin
```

Exercice 5.20

Cet algorithme remplit un tableau de la manière suivante :

```
X[1, 1] = 1
X[1, 2] = 2
X[1, 3] = 3
X[2, 1] = 4
X[2, 2] = 5
X[2, 3] = 6
```

Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Exercice 5.21

Cet algorithme remplit un tableau de la manière suivante :

```
X[1, 1] = 1
X[1, 2] = 2
X[1, 3] = 3
X[2, 1] = 4
X[2, 2] = 5
X[2, 3] = 6
```

Il écrit ensuite ces valeurs à l'écran, dans l'ordre suivant : X[1, 1], X[2, 1], X[1, 2], X[2, 2], X[1, 3], X[2, 3]. Cet algorithme affiche donc les valeurs suivantes dans cet ordre :

1 – 4 – 2 – 5 – 3 – 6

Exercice 5.22

Cet algorithme remplit un tableau de la manière suivante :

```
T[1, 1] = 2
T[1, 2] = 3
T[2, 1] = 3
T[2, 2] = 4
T[3, 1] = 4
T[3, 2] = 5
T[4, 1] = 5
T[4, 2] = 6
```

Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Exercice 5.23

Version a : cet algorithme remplit un tableau de la manière suivante :

```
T[1, 1] = 4
T[1, 2] = 5
T[2, 1] = 6
T[2, 2] = 7
T[3, 1] = 8
T[3, 2] = 9
T[4, 1] = 10
T[4, 2] = 11
```

Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Version b : cet algorithme remplit un tableau de la manière suivante :

```
T[1, 1] = 6
T[1, 2] = 10
T[2, 1] = 7
T[2, 2] = 11
T[3, 1] = 8
T[3, 2] = 12
T[4, 1] = 9
T[4, 2] = 13
```

Il écrit ensuite ces valeurs à l'écran, dans cet ordre.

Exercice 5.24

```
Variables i, j, iMax, jMax en Numérique
Tableau T[1..13, 1..9] en Numérique
```

Le principe de la recherche dans un tableau à deux dimensions est strictement le même que dans un tableau à une dimension, ce qui ne doit pas nous étonner. La seule chose qui change, c'est qu'ici le balayage requiert deux boucles imbriquées, au lieu d'une seule.

```
Debut
...
iMax <--- 1
jMax <--- 1
Pour i de 1 à 13
  Pour j de 1 à 9
    Si T[i,j] > T[iMax,jMax] Alors
        iMax <--- i
        jMax <--- j
    FinSi
  j Suivant
```

```
i Suivant
Ecrire ("Le plus grand élément est ", T[iMax, jMax])
Ecrire ("Il se trouve aux indices ", iMax, " , ", jMax)
Fin
```

Les fonctions et procédures

Dans ce cours nous allons parler de "programme" et de "sous- programme". Il faut comprendre ces mots comme "programme algorithmique" indépendant de toute implantation.

1- Rappels :

La méthodologie de base de l'informatique est :

- **Abstraire**

  Retarder le plus longtemps possible l'instant du codage

- **Décomposer**

  "... diviser chacune des difficultés que j'examinerai en autant de parties qu'il se pourrait et qu'il serait requis pour les mieux résoudre."

  Descartes

- **Combiner**

  Résoudre le problème par combinaison d'abstractions

Par exemple, résoudre le problème suivant :

Ecrire un programme qui affiche en ordre croissant les notes d'une promotion suivies de la note la plus faible, de la note la plus élevée et de la moyenne

Revient à résoudre les problèmes suivants :

- Remplir un tableau de naturels avec des notes saisies par l'utilisateur
- Afficher un tableau de naturels
- Trier un tableau de naturel en ordre croissant
- Trouver le plus petit naturel d'un tableau
- Trouver le plus grand naturel d'un tableau

- Calculer la moyenne d'un tableau de naturels

Chacun de ces sous- problèmes devient un nouveau problème à résoudre. Si on considère que l'on sait résoudre ces sous- problèmes, alors on sait "quasiment" résoudre le problème initial

2- Notion de sous-programme :

De ce qui a précédé, nous pouvons donc affirmer qu'écrire un programme qui résout un problème revient souvent à écrire des sous- programmes qui résolvent des sous parties du problème initial.

En algorithmique, il existe deux types de sous- programmes :

- Les fonctions
- Les procédures

Un sous- programme est obligatoirement caractérisé par un nom (un identifiant) unique. Lorsqu'un sous programme a été explicité (on a donné l'algorithme), son nom devient une nouvelle instruction, qui peut être utilisé dans un autre programme ou sous-programme.

Le programme (ou sous-programme) qui utilise un sous- programme est appelé **programme** (resp sous-programme) **appelant**.

3- Nom d'un sous-programme :

Nous savons maintenant que les variables, les constantes, les types définis par l'utilisateur (comme les énumérateurs) et que les sous- programmes possèdent un nom.

Ces noms doivent suivre certaines règles et obéir à des conventions afin de clarifier la lecture et la compréhension de l'algorithme :

- Les noms doivent être explicites (à part quelques cas particuliers, comme par exemple les variables *i* et *j* pour les boucles)
- Ils ne peuvent contenir que des lettres et des chiffres
- Ils commencent obligatoirement par une lettre
- Les variables et les sous- programmes commencent toujours par une miniscule
- Les types commencent toujours par une majuscule

- Les constantes ne sont composées que de majuscules
- Lorsqu'ils sont composés de plusieurs mots, on utilise les *majuscules* (sauf pour les constantes) pour séparer les mots (par exemple jourDeLaSemaine).

4- Portée des variables dans un sous-programme :

La **portée** d'une variable est l'ensemble des sous- programmes où cette variable est connue (les instructions de ces sous- programmes peuvent utiliser cette variable).

Une variable définie au niveau du programme principal (celui qui résoud le problème initial, le problème de plus haut niveau) est appelée **variable globale**. Sa portée est totale : **tout** sous- programme du programme principal peut utiliser cette variable, en plus du programme principal bien sur.

Une variable définie au sein d'un sous programme est appelée **variable locale**. La portée d'une variable locale est uniquement le sous- programme qui la déclare.

Lorsque le nom d'une variable locale est identique à une variable globale, la variable globale est localement masquée. Dans ce sous- programme la variable globale devient inaccessible.

5- Structure d'un programme :

Un programme doit suivre la structure suivante :

Programme *nom du programme*
Définition des constantes
Définition des types
Déclaration des variables globales
Définition des sous- programmes
Début
instructions du programme principal
Fin

6- Les paramètres d'un sous-programme :

Un paramètre d'un sous- programme est une variable locale particulière qui est associée à une variable ou constante du programme (ou sous-programme) appelant :

- Puisqu'un paramètre est une variable locale, un paramètre admet un type.
- Lorsque le programme (ou sous-programme) appelant appelle le sous-programme il doit indiquer la variable (ou la constante), de même type, qui est associée au paramètre.

Par exemple, si le sous- programme *sqr* permet de calculer la racine carrée d'un réel :

- Ce sous- programme admet un seul paramètre de type réel positif
- Le programme (ou sous-programme) qui utilise *sqr* doit donner le réel positif dont il veut calculer la racine carrée, cela peut être :
  - une variable, par exemple *a*
  - une constante, par exemple *5.25*

7- Le passage de paramètres :

Il existe trois types d'association (que l'on nomme **passage de paramètre**) entre le paramètre et la variable (ou la constante) du programme (ou sous-programme) appelant :

- Le **passage de paramètre en entrée**
- Le **passage de paramètre en sortie**
- Le **passage de paramètre en entrée/ sortie**

7.1- Le passage de paramètres en entrée :

Les instructions du sous- programme ne peuvent pas modifier l'entité (variable ou constante) du programme (ou sous-programme) appelant :

- En fait, c'est la valeur de l'entité du programme (ou sous-programme) appelant qui est copiée dans le paramètre (à part cette copie il n'y a pas de relation entre le paramètre et l'entité du programme ou sous-programme appelant).

- C'est le seul passage de paramètres qui admet l'utilisation **d'une** constante.

Par exemple,

- le sous- programme *sqr* permettant de calculer la racine carrée d'un nombre admet un paramètre en entrée
- le sous- programme **écrire** qui permet d'afficher des informations admet **n** paramètres en entrée

7.2- Le passage de paramètres en sortie :

Les instructions du sous- programme affecte obligatoirement une valeur à ce paramètre (valeur qui est donc aussi affectée à la variable associée du programme ou sous-programme appelant).

Il y a donc une liaison forte entre la paramètre et l'entité du programme (ou sous-programme) appelant. C'est pour cela que nous ne pouvons pas utiliser de constante pour ce type de paramètre.

La valeur que pouvait posséder la variable associée du programme (ou sous-programme) appelant n'est pas utilisée par le sous- programme.

Par exemple, le sous- programme **lire** qui permet de mettre dans des variables des valeurs saisies par l'utilisateur admet **n** paramètres en sortie.

7.3- Le passage de paramètres en entrée / sortie :

Passage de paramètre qui combine les deux précédentes. A utiliser lorsque le sous- programme doit utiliser et/ ou modifier la valeur de la variable du programme (ou sous-programme) appelant. Comme pour le passage de paramètre en sortie, on ne peut pas utiliser de constante.

Par exemple, le sous-programme **échanger** qui permet d'échanger les valeurs de deux variables utilise les deux variables comme paramètres entrée / sortie.

8- Les fonctions :

Les fonctions sont des sous- programmes admettant des paramètres et retournant un **seul** résultat (comme les fonctions mathématiques y= f(x, y, ...)).

Les fonctions sont caractérisées par :

- Les paramètres sont en nombre fixe (≥ 0)
- Une fonction possède un seul type, qui est le type de la valeur retournée
- Le passage de paramètre est **uniquement en entrée** : c'est pour cela qu'il n'est pas précisé. Lors de l'appel, on peut donc utiliser comme paramètre des variables, des constantes mais aussi des résultats de fonction.
- La valeur de retour est spécifiée par l'instruction **retourner**.

Généralement, le nom d'une fonction est soit un nom (par exemple *minimum*), soit une question (par exemple *estVide*).

On déclare une fonction de la façon suivante :

Fonction *nom de la fonction* (*paramètre(s) de la fonction*) **:** *type de la valeur retournée*

Déclaration *variable locale 1 : type 1; . . .*

Début

instructions de la fonction avec au moins une fois l'instruction ***retourner***

Fin

On utilise une fonction en précisant son nom suivi des paramètres entre parenthèses. Les parenthèses sont toujours présentes même lorsqu'il n'y a pas de paramètre.

Exemple de déclaration de fonction :

```
Fonction abs (unEntier : Entier) : Entier
Début
   Si unEntier ≥ 0 Alors
      retourner unEntier
   Sinon
      retourner –unEntier
   Fin si
Fin
```

Exemple de programme :

Voici un exemple de programme qui définit puis fait appel à une fonction (ici la fonction **ABS**)

```
Programme exemple1
Déclaration a : Entier, b : Naturel
Fonction abs (unEntier : Entier) : Naturel
   Déclaration valeurAbsolue : Naturel
   Début
   Si unEntier ≥ 0 Alors
        valeurAbsole ←unEntier
     Sinon
        valeurAbsolue ←-unEntier
   Fin si
   retourner valeurAbsolue
Fin fonction
Début
   Ecrire("Entrez un entier : ")
   Lire(a)
   b ←abs(a)
   Ecrire("La valeur absolue de ", a," est ", b)
Fin
```

Lors de l'exécution de la fonction *abs*, la variable *a* et le paramètre *unEntier* sont associés par un passage de paramètre en entrée : la valeur de *a* est copiée dans *unEntier*.

Un autre exemple...

```
Fonction minimum2 (a, b : Entier) : Entier
Début
   Si a ≥ b Alors
        retourner b
     Sinon
        retourner a
   Fin si
Fin
```

```
Fonction minimum3 (a, b, c : Entier) : Entier
Début
    retourner minimum2(a, minimum2(b, c))
Fin
```

9- Les fonctions prédéfinies :

Certains traitements *ne peuvent pas* être effectués par un algorithme, par exemple le calcul du sinus d'un angle.

Tous les langages ont un certain nombre de **fonctions** qui permettent de connaître immédiatement ce genre de résultat :

- Certaines sont indispensables : elles permettent d'effectuer des traitements impossibles sans elles.
- D'autres servent à soulager le programmeur, en lui épargnant de longs algorithmes.

9.1- Structure générale des fonctions prédéfinis :

Pour illustrer la structure d'une fonction prédéfinie, nous prendrons l'exemple de la fonction SIN qui permet de calculer le sinus d'un angle. Par exemple, pour stocker le sinus de 60 dans la variable A, nous écrirons : A ← Sin(60).

Une fonction est donc constituée de trois parties :

- Le **nom** de la fonction.
  - correspond à une fonction *proposée par le langage*
  - ne s'invente pas, il faut utiliser le nom que le langage a attribué à la fonction
- Deux parenthèses
- Une liste de valeurs (arguments, ou paramètres)
  - indispensables à la bonne exécution de la fonction.
  - certaines fonctions exigent un seul argument, d'autres 2,... Mais, il existe aussi des fonctions qui n'admettent aucun argument. Si tel est le cas, il faut écrire le nom de la fonction suivie des parenthèses, la fonction **alea**(), par exemple.
  - le nombre d'arguments nécessaire pour une fonction donnée est fixé par le langage.
  - les arguments doivent être d'un certain **type** qu'il faut respecter.

9.2- Les fonctions de texte :

Cette catégorie de fonctions nous permet de manipuler des chaînes de caractères. Tous les langages proposent les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

- **Len**(chaîne) renvoie la longueur d'une chaîne de caractères, c'est à dire le nombre de caractères d'une chaîne.

  Ex : Len("Bonjour, ça va ?") vaut 16 (les espaces sont aussi comptabilisés).

- **SubStr**(chaîne, n1, n2) renvoie un extrait de la chaîne commençant au caractère n1 et faisant n2 caractères de long.

  Ex : SubStr("Zorro is back", 4, 6) vaut ("ro is "

- **Left**(chaîne, n) renvoie les n caractères les plus à gauche dans chaîne.

  Ex : Left("Et pourtant...", 8) vaut "Et pourt"

- **Right**(chaîne, n) renvoie les n caractères les plus à droite dans chaîne

  Ex : Right("Et pourtant...", 4) vaut "t..."

- **Trouve**(chaîne1, chaîne2) renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie zéro.

  Ex : Trouve("Un pur bonheur", "pur") vaut 4

- Une fonction renvoyant un nombre représentant le code Ascii du caractère spécifié : fonction **Asc**

  Ex : Asc("A") vaut 65, Asc("Z") vaut 90 et Asc("a") vaut 97.

  En fait, le code ASCII d'une lettre en majuscule est différent du code ASCII de la même lettre écrite en minuscule.

9.3- Deux fonctions classiques :

- Récupérer la partie entière d'un nombre :

  A ←Ent(3,228)

  A vaut 3

- Générer un nombre choisi au hasard.
  - c'est la fonction ALEA()
  - sert dans les jeux
    - simuler un lancer de dés
    - simuler le déplacement d'un vaisseau
    - ...
  - sert dans la modélisation
    - physique,
    - géographique,
    - économique,
    - ...

10- Les procédures :

Les procédures sont des sous- programmes qui ne retournent **aucun** résultat. Par contre, elles admettent des paramètres avec des passages :

- en entrée, préfixés par **Entrée** (ou E)
- en sortie, préfixés par **Sortie** (ou S)
- *en entrée / sortie, préfixés par* **Entrée / Sortie** (ou **E/S**)

Généralement, le nom d'une procédure est un verbe.

On déclare une procédure de la façon suivante :

Procédure *nom de la procédure* (**E** *paramètre(s) en entrée;* **S** *paramètre(s) en sortie;* **E/S** *paramètre(s) en entrée / sortie*)
Déclaration *variable(s) locale(s)*
Début
instructions de la procédure
Fin

Et on appelle une procédure comme une fonction, en indiquant son nom suivi des paramètres entre parenthèses.

Notez qu'une procédure peut admettre des paramètres avec un seul type de passage, 2 types ou 3 types de passage.

Exemple de déclaration de procédure :

Procédure calculerMinMax3 (**E** a, b, c : **Entier**; **S** min, max : **Entier**)
Début
min ←minimum3(a, b, c)
max ←maximum3(a, b, c)
Fin

Exemple de programme :

Voici un exemple de programme qui définit puis fait appel à une procédure (ici la procédure **échanger** qui permet d'échanger les valeurs de deux variables).

```
Programme exemple2
Déclaration a, b : Entier
Procédure echanger (E/ S val1 : Entier; E/ S val2 : Entier;)
   Déclaration temp : Entier
   Début
      temp ←val1
      val1 ←val2
      val2 ←temp
   Fin procédure
Début
   Ecrire("Entrez deux entiers : ")
   Lire(a, b)
   echanger(a, b)
   Ecrire(" a = ", a," et b = ", b)
Fin
```

Lors de l'exécution de la procédure *echanger* , la variable *a* et le paramètre *val1* sont associés par un passage de paramètre en entrée / sortie : toute modification sur val1 est effectuée sur a (de même pour b et val2).

Autre exemple de programme...

```
Programme exemple3
Déclaration entier1, entier2, entier3, min, max : Entier
Fonction minimum3 (a, b, c : Entier) : Entier
...
Fonction maximum3 (a, b, c : Entier) : Entier
```

```
...
Procédure calculerMinMax3 ( E a, b, c : Entier ; S min3, max3 : Entier )
  Début
    min3 ←minimum3(a, b, c)
    max3 ←maximum3(a, b, c)
  Fin procédure

Début
  Ecrire("Entrez trois entiers : ")
  Lire(entier1)
  Lire(entier2)
  Lire(entier3)
  calculerMinMax3(entier1, entier2, entier3, min, max)
  Ecrire("La valeur la plus petite est ", min," et la plus grande est ", max)
Fin
```

Exercices

Exercice 6.1

Parmi ces affectations (considérées indépendamment les unes des autres), lesquelles provoqueront des erreurs, et pourquoi ?

```
Variables A, B, C en Numérique
Variables D, E en Caractère
...
A <-- Sin(B)
...
A <-- Sin(A + B * C)
...
B <-- Sin(A) - Sin(D)
...
D <-- Sin(A / B)
...
C <-- Cos(Sin(A)
```

Exercice 6.2

Ecrivez un algorithme qui demande un mot à l'utilisateur puis qui affiche à l'écran le nombre de lettres de ce mot.

Exercice 6.3

Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de mots de cette phrase. On suppose que les mots ne sont séparés que par des espaces.

Exercice 6.4

Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de voyelles contenues dans cette phrase. On supposera que toutes les lettres de la phrase sont en minuscule.

On pourra écrire deux solutions. La première déploie une condition composée bien fastidieuse. La deuxième, en utilisant la fonction Trouve, allège considérablement l'algorithme.

Exercice 6.5

Ecrivez un algorithme qui demande une phrase à l'utilisateur. Celui-ci entrera ensuite le rang d'un caractère à supprimer, et la nouvelle phrase doit être affichée (on doit réellement supprimer le caractère dans la variable qui stocke la phrase, et pas uniquement à l'écran).

Exercice 6.6 : Cryptographie 1

Un des plus anciens systèmes de cryptographie (aisément déchiffrable) consiste à décaler les lettres d'un message pour le rendre illisible. Ainsi, les A deviennent des B, les B des C, etc. Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui la code selon ce principe. On supposera que les lettres de la phrase sont en majuscule. Comme dans le cas précédent, le codage doit s'effectuer au niveau de la variable stockant la phrase, et pas seulement à l'écran.

Exercice 6.7 - Cryptographie 2 : *le chiffre de César*

Une amélioration (relative) du principe précédent consiste à opérer avec un décalage non de 1, mais d'un nombre quelconque de lettres. Ainsi, par exemple, si l'on choisit un décalage de 12, les A deviennent des M, les B des N, etc.

Réalisez un algorithme sur le même principe que le précédent, mais qui demande en plus quel est le décalage à utiliser. Votre sens de l'élégance des algorithmes vous interdira bien sûr une série de vingt-six "Si...Alors".

Exercice 6.8 - Cryptographie 3

Une technique ultérieure de cryptographie consista à opérer non avec un décalage systématique, mais par une substitution aléatoire. Pour cela, on utilise un alphabet-clé, dans lequel les lettres se succèdent de manière désordonnée, par exemple :

HYLUJPVREAKBNDOFSQZCWMGITX

C'est cette clé qui va servir ensuite à coder le message. Selon notre exemple, les A deviendront des H, les B des Y, les C des L, etc.

Ecrire un algorithme qui effectue ce cryptage (l'alphabet-clé sera saisi par l'utilisateur, et on suppose qu'il effectue une saisie correcte).

Exercice 6.9

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

Exercice 6.10

Ecrivez les algorithmes qui génèrent un nombre Glup aléatoire tel que ...

a) `0 =< Glup < 2`
b) `-1 =< Glup < 1`
c) `1,35 =< Glup < 1,65`
d) `-10,5 =< Glup < +6,5`

Exercice 6.11

Ecrivez une fonction qui renvoie la somme de cinq nombres fournis en argument.

Exercice 6.12

Ecrivez une fonction qui renvoie le nombre de voyelles contenues dans une chaîne de caractères passée en argument. Au passage, notez qu'une fonction a tout à fait le droit d'appeler une autre fonction.

Exercice 6.13

Réécrivez la fonction Trouve à l'aide des fonctions sousChaine et Len (comme quoi, Trouve, à la différence de sousChaine et Len, n'est pas une fonction indispensable dans un langage)

Exercice 6.14 : Fonction ChoixDuMot

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

CORRIGES DES EXERCICES

Exercice 6.1

`A <-- Sin(B)`	Aucun problème
`A <-- Sin(A + B * C)`	Aucun problème
`B <-- Sin(A) - Sin(D)`	Erreur ! D est en caractère
`D <-- Sin(A / B)`	Aucun problème... si B est différent de zéro
`C <-- Cos(Sin(A)`	Erreur ! Il manque une parenthèse fermante

Exercice 6.2

Cet algorithme est très simple. En fait, il suffit de se servir de la fonction Len qui renvoie la longueur d'une chaîne de caractères, et c'est réglé :

```
Variable Mot en Caractère
Variable Nb en Entier
Debut
Ecrire "Entrez un mot : "
Lire Mot
Nb <-- Len(Mot)
Ecrire "Ce mot compte ", Nb, " lettres"
Fin
```

Exercice 6.3

Là, on est obligé de compter par une boucle le nombre d'espaces de la phrase, et on en déduit le nombre de mots. La boucle examine les caractères de la phrase un par un, du premier au dernier, et les compare à l'espace.

Cette solution marche seulement si les mots sont séparés par un seul espace et un seul.

```
Variable phrase en Caractère
```

```
Variables Nb, i en Entier
Debut
Ecrire "Entrez une phrase : "
Lire phrase
Nb <-- 0
Pour i de 1 à Len(phrase)
     Si sousChaine(phrase, i , 1) = " " Alors
            Nb <-- Nb + 1
     FinSi
i suivant
Ecrire ("Cette phrase compte ", Nb + 1, " mots")
Fin
```

Exercice 6.4

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

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

Note : la plupart des programmes comportent un opérateur **Dans** qui teste si une lettre (ou sous chaîne) est incluse dans une chaîne de caractères. Si tel est le cas, il suffit d'écrire :

```
Si lettre dans "aeiouy" Alors ...
```

au lieu de la longue instruction :

```
lettre = "a" ou lettre = "e" ou lettre = "i" ou
lettre = "o" ou lettre = "u" ou lettre = "y" Alors ...
```

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

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

Exercice 6.5

Il n'existe aucun moyen de supprimer directement un caractère d'une chaîne... autrement qu'en procédant par collage. Il faut donc concaténer ce qui se trouve à gauche du caractère à supprimer, avec ce qui se trouve à sa droite. Attention aux paramètres des fonctions sousChaine, ils n'ont rien d'évident !

```
Variable phrase en Caractère
```

```
Variables rang, longueur, i, j en Entier
Début
Ecrire "Entrez une phrase : "
Lire phrase
Ecrire "Entrez le rang du caractère à supprimer : "
Lire rang
longueur <-- Len(phrase)
phrase <-- sousChaine(phrase, 1, rang - 1) &
           sousChaine(phrase, rang + 1, longueur - rang)
Ecrire ("La nouvelle phrase est : ", phrase)
Fin
```

Exercice 6.6

Pour cet exercice, il y a une règle générale : pour chaque lettre, on détecte sa *position dans l'alphabet*, et on la remplace par la lettre occupant la position suivante. Seul cas particulier, la vingt-sixième lettre (le Z) doit être codée par la première (le A), et non par la vingt-septième, qui n'existe pas !

```
Variables lettre, phrase, phraseCodee, lettresAlphabet en
          Caractère
Variables i, position en Entier
Début
Ecrire "Entrez la phrase à coder : "
Lire phrase
lettresAlphabet <-- "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
phraseCodee <-- ""
Pour i de 1 à Len(phrase)
    lettre <-- sousChaine(phrase, i, 1)
    Si Lettre <> "Z" Alors
        position <-- Trouve(lettresAlphabet, lettre)
        phraseCodee <-- phraseCodee &
         sousChaine(lettresAlphabet, position + 1, 1)
      Sinon
        phraseCodee <-- phraseCodee & "A"
    FinSi
```

```
i Suivant
phrase <-- phraseCodee
Ecrire ("La phrase codée est : ", phrase)
Fin
```

Exercice 6.7

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

Il y a une petite ruse supplémentaire à appliquer, puisque 26 doit rester 26 et ne pas devenir 0.

```
Variables lettre, phrase, phraseCodee, lettresAlphabet en
          Caractère
Variables i, position, nouvPos, décalage en Entier
Début
Ecrire "Entrez le décalage à appliquer : "
Lire décalage
Ecrire "Entrez la phrase à coder : "
Lire phrase
lettresAlphabet <-- "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
phraseCodee <-- ""
Pour i de 1 à Len(phrase)
    lettre <-- sousChaine(phrase, i, 1)
    position <-- Trouve(lettresAlphabet, lettre)
     nouvPos <-- Mod(position + décalage, 26)
    // MOD est la fonction qui calcule le modulo, c'est-à-dire le reste de la
       division entière, d'un nombre (ici position + décalage) sur un autre
       nombre (ici 26)
     Si nouvPos = 0 Alors
```

```
            nouvPos <-- 26
        FinSi
        phraseCodee <-- phraseCodee & sousChaine(lettresAlphabet,
                        NouvPos, 1)
    i Suivant
    phrase <-- phraseCodee
    Ecrire ("La phrase codée est : ", phrase)
    Fin
```

Exercice 6.8

Là, c'est assez direct.

```
Variables lettre, phrase, phraseCodee, lettresAlphabet,
          cléCodage en Caractère
Variables i, position, décalage en Entier
Début
Ecrire "Entrez l'alphabet clé : "
Lire cléCodage
Ecrire "Entrez la phrase à coder : "
Lire phrase
lettresAlphabet <-- "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
phraseCodee <-- ""
Pour i de 1 à Len(phrase)
    lettre <-- sousChaine(phrase, i, 1)
    position <-- Trouve(lettresAlphabet, lettre)
    phraseCodee <-- phraseCodee & sousChaine(cléCodage,
                    position, 1)
i Suivant
phrase <-- phraseCodee
Ecrire ("La phrase codée est : ", phrase)
Fin
```

Exercice 6.9

On en revient à des choses plus simples...

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

Une autre solution :

On peut faire appel à la fonction MOD qui, rappelons-le, calcule le reste de la division entière d'un nombre par un autre. Voici l'algorithme dans une autre forme mais qui, bien sur, renvoie le même résultat c'est-à-dire qu'il est équivalent au précédent :

```
Variable Nb en Entier
Ecrire "Entrez votre nombre : "
Lire Nb
Si Nb mod 2 = 0 Alors
    Ecrire "Ce nombre est pair"
  Sinon
    Ecrire "Ce nombre est impair"
FinSi
Fin
```

Exercice 6.10

Il faut d'abord rappeler que Glup est un nombre aléatoire, ce nombre est réel et il est compris entre 0 et 1.

a) `Glup <-- Alea() * 2`

b) `Glup <-- Alea() * 2 - 1`

c) `Glup <-- Alea() * 0.30 + 1.35`

```
d) Glup <-- Alea() * 17 - 10,5
```

Exercice 6.11

```
Fonction Sum(a, b, c, d, e)
Renvoyer a + b + c + d + e
Fin
```

Exercice 6.12

```
Fonction NbVoyelles(mot en Caractère)
Variables i, nb en Numérique
Pour i de 1 à Len(mot)
   Si Trouve("aeiouy", sousChaine(mot, i, 1)) <> 0 Alors
     nb <-- nb + 1
   FinSi
i suivant
Renvoyer nb
Fin
```

Exercice 6.13

```
Fonction trouve(a, b)
Variable i en Entier
Début
i <-- 1
TantQue i < Len(a) - Len(b) et b <> sousChaine(a, i, Len(b))
     Faire
       i <-- i + 1
Fin TantQue
Si b = sousChaine(a, i, Len(b)) Alors
```

```
        Renvoyer i
Sinon
        Renvoyer 0
FinSi
Fin
```

Exercice 6.14

```
Fonction choixDuMot()
Tableau liste[] en Caractère
Variable nbMots en Numérique
Variable choisi en Entier
Ouvrir "Dico.txt" sur 1 pour Lecture
nbMots <-- 0
Tantque Non EOF(1) Faire
        nbMots <-- nbMots + 1
        Redim liste[1..nbMots]
        LireFichier 1, liste[nbMots]
Fin TantQue
Fermer 1
choisi <-- 0
Tantque choisi = 0 Faire
        choisi <-- Ent(Alea() * nbMots)
        // Ent est la fonction qui calcule la partie entière d'un nombre
Fin TantQue
Renvoyer liste[Choisi]
Fin
```

Les fichiers

Lorsqu'on parle de fichier, il faut distinguer entre fichiers de données et fichiers texte. Nous allons donc les étudier séparément, puisque les méthodes et instructions qu'on applique à un type de fichier ne peuvent pas toujours être appliquées à l'autre type. Cependant, il existe d'autres types de fichiers dont l'étude en algorithmique dépasse le cadre de cet ouvrage : les fichiers images, les fichiers son, les vidéos, etc.

A- Les fichiers de données :

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

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

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

1- Généralités sur l'organisation des informations :

On a souvent besoin de stocker différentes informations sur des supports physiques pour une utilisation éventuelle ultérieure. La notion d'accès à une information rangée devient fondamentale.

Les différentes primitives sur les supports d'information sont :

- accès à un élément.
- recherche d'un élément.
- insertion d'un nouvel élément.
- suppression d'un élément.
- copie d'une partie des informations.
- éclatement de l'ensemble en plusieurs sous-ensemble.
- fusion de plusieurs sous-ensembles en un seul.
- tri d'un ensemble.

2- Notion de fichier et d'enregistrement :

Les différentes informations que l'on aura à traiter seront donc stockées sur un support physique. Il va donc falloir organiser les différents transferts entre la mémoire et ce support physique grâce à différentes primitives d'accès.

L'organisation d'un fichier est la structure logique permanente établie au moment où le fichier est crée.

2.1- Définition :

Un fichier est un ensemble organisé d'informations (articles ou enregistrements) de même nature susceptibles de faire l'objet de traitements divers.

2.2- Notion d'enregistrement :

Un enregistrement (ou record) est une structure constituée d'un nombre fixe de composants appelés **champs**. Les champs peuvent être de différents types et chaque champ comporte un identificateur de champ permettant de le sélectionner.

Exemple :

```
TYPE
T_abonne = ENREG
    nom : CHAINE
    prenom : CHAINE
    age : ENTIER
    salaire : REEL
Fin ENREG
```

Une variable du type T_abonne sera déclarée et utilisée de la façon suivante :

```
VAR
    abonne1 : T_abonne
```

Dans le programme utilisant cette variable, *abonne1.nom* identifie le champ *nom* de l'enregistrement *abonne1* de type *T_abonne*. Les opérations permises sur le type chaîne seront possibles sur *abonne1.nom* (lecture, écriture, affectation, comparaison, etc...).

Si on déclare une autre variable *abonne*2 du type *T_abonne*, les opérations d'affectation seront possibles entre *abonne1* et *abonne*2.

3- Les accès aux fichiers :

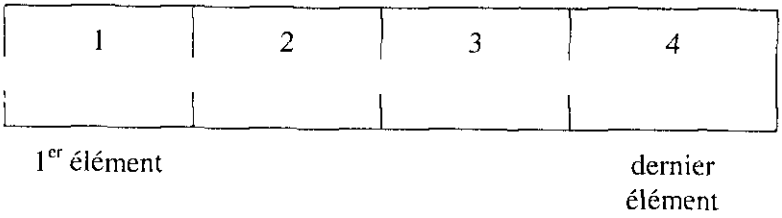
On accède généralement aux fichiers par deux méthodes principales :

- accès séquentiel.
- accès direct.

3.1- Accès séquentiel :

Soit F un fichier: ce fichier est dit à accès séquentiel si l'on ne peut accéder à un élément quelconque de rang n qu'après avoir accédé aux n-1 éléments qui le précèdent.

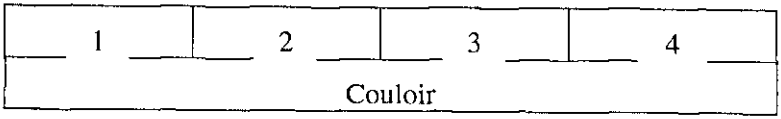
On se déplace dans ce type de fichier en passant d'un élément à son suivant. Il existe deux éléments particuliers: le premier qui nous permet d'accéder au fichier et le dernier qui nous signale qu'il n'a pas de suivant.



3.2- Accès direct :

Soit F un fichier: ce fichier est dit à accès direct lorsqu'il suffit de connaître le rang n de l'élément recherché pour y accéder, sans parcourir le reste du fichier.

Cette méthode est en général plus intéressante en temps et en efficacité.



4- Les fichiers en algorithmique :

4.1- Déclaration d'une variable de type fichier :

```
VAR
fic : FICHIER
```

où fic est l'identificateur (= nom logique).

4.2- Assignation d'un nom de fichier logique :

```
ASSIGNER (nom_de_fichier_logique, nom_de_fichier_physique)
```

où nom_de_fichier_physique est soit une constante chaîne, soit une variable chaîne.

Le nom du fichier physique correspond au nom du fichier sur le support physique (en général disque magnétique, bande, etc...). Ce nom peut comprendre le chemin (absolu ou relatif) du fichier.

L'opération d'assignation est obligatoire, tous les accès au fichier se feront à l'aide du nom de fichier logique.

4.3- Actions et opérations sur les fichiers :

a- Ouverture :

- en écriture : **OUVREECR(nom_de_fichier_logique)**
- en lecture : **OUVRELEC(nom_de_fichier_logique)**

b- Fermeture :

FERMER(nom_de_fichier_logique)

c- Ecriture dans un fichier :

ECRIRE(nom_de_fichier_logique, expression)

ECRIRE(fic, expression) écrit à partir de la position courante de la tête d'écriture sur le fichier *fic* la valeur de l'expression. La tête d'écriture est ensuite positionnée sur l'élément suivant.

d- Lecture dans un fichier :

LIRE(nom_de_fichier_logique, variable)

LIRE(fic,variable) : comme pour une lecture clavier, le type de la variable doit être le même que celui de l'élément lu. La tête de lecture est ensuite déplacée sur l'élément suivant à la fin de l'opération.

e- Fin de fichier :

On peut trouver également : EOF(fic) ou encore FF(fic).

EOF : acronyme pour **E**nd **O**f **F**ile, qui signifie fin de fichier.

B- Les fichiers texte :

1- Organisation des fichiers texte :

Il existe deux grandes variantes pour structurer les données au sein d'un fichier texte : la délimitation, ou les champs de largeur fixe. Reprenons le cas du carnet d'adresse ; Nous nous limiterons aux renseignements suivants : Nom, prénom, téléphone, e-mail. Les données, sur le fichier texte, peuvent être organisées ainsi :

Structure n°1

"Boumedous";"Kamel";021510012;"kamelAmine@hotmail.com"

"Chahi";"Mohamed";021781098 ;"mchahi@yahoo.fr"

"Zemmouri";"Omar";024765194; "omarzemou@yahoo.fr"

"Mebarki";"Toufik";021907531;"mebtoufik@yahoo.fr"

ou ainsi :

Structure n°2

Boumedous	Kamel	021510012	kamelAmine@hotmail.com
Chahi	Mohamed	021781098	mchahi@yahoo.fr
Zemmouri	Omar	024765194	omarzemou@yahoo.fr
Mebarki	Toufik	021907531	mebtoufik@yahoo.fr

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

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

- L'avantage de la structure n°1 est son faible encombrement en place mémoire ; il n'y a aucun espace perdu, et un fichier texte codé de cette

manière occupe le minimum de place possible. Mais elle possède en revanche un inconvénient majeur, qui est la lenteur de la lecture. En effet, chaque fois que l'on récupère une ligne dans le fichier, il faut alors parcourir un par un tous les caractères pour repérer chaque occurrence du caractères de séparation avant de pouvoir découper cette ligne en différents champs.

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

A l'époque où la place mémoire coûtait cher, la structure délimitée était privilégiée. Mais depuis bien des années, la quasi-totalité des logiciels – et des programmeurs – optent pour la structure en champs de largeur fixe. Aussi, sauf mention contraire, nous ne travaillerons qu'avec des fichiers bâtis sur cette structure.

2- Types d'accès :

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

On distingue :

- *L'accès séquentiel* : on ne peut accéder qu'à l'enregistrement suivant celui qu'on vient de lire.
- *L'accès direct* (ou *aléatoire*) : on peut accéder directement à l'enregistrement de son choix, en précisant le numéro de cet enregistrement.

A la différence de la précédente, cette typologie ne se reflète pas dans la structure elle-même du fichier. En fait, tout fichier, quelle que soit sa structure (délimité ou en largeur fixe) peut être utilisé avec l'un ou l'autre des deux grands types d'accès. Le choix du type d'accès n'est pas un choix qui concerne le fichier lui-même, mais uniquement la manière dont il va être traité par la machine. C'est donc dans le programme, et seulement dans le programme, que l'on choisit le type d'accès souhaité.

A première vue, il est clair que l'accès direct possède tous les avantages et aucun inconvénient. Sa programmation est immensément plus simple et moins

fastidieuse que celle de l'accès séquentiel. Mais... vous vous doutez bien qu'il y a un mais.

Le gros problème de l'accès direct, c'est qu'il n'est pas géré de la même manière selon le type de machine. Car la gestion de l'accès direct suppose des éléments de langage " annexes ", qui diffèrent selon les ordinateurs et qui ne seront pas forcément présents. Pour résumer, disons que si vous écrivez un programme C utilisant des accès directs à des fichiers, vous n'êtes pas absolument certains que votre programme pourra tourner sur tous les ordinateurs.

En revanche, l'accès séquentiel, lui, est entièrement piloté par le langage lui-même. Il est donc laborieux à programmer, rustique, tout ce que vous voulez, mais il offre une sécurité absolue de portabilité. Voilà pourquoi la plupart des programmeurs s'astreignent encore à utiliser ce type d'accès, et voilà pourquoi c'est celui-là que nous allons étudier.

Et de toute manière, qui peut le plus peut le moins, et une fois que vous aurez maîtrisé la programmation de l'accès séquentiel, celle de l'accès direct vous apparaîtra comme un jeu d'enfant.

3- Instructions :

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

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

- Si on ouvre un fichier pour lecture, on ne pourra que récupérer les informations qu'il contient, sans les modifier en aucune manière.
- Si on ouvre un fichier pour écriture, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront intégralement écrasées.
- Si on ouvre un fichier pour ajout, on ne peut ni lire, ni modifier les infos existantes. Mais on pourra, comme vous commencez à vous en douter, ajouter de nouveaux enregistrements.

A première vue, ces limitations peuvent sembler infernales. Mais avec un peu d'habitude, on se rend compte qu'en fait on peut faire tout ce qu'on veut avec ces fichiers séquentiels.

Pour ouvrir un fichier texte, on écrira par exemple :

Ouvrir "Exemple.txt" sur 4 **pour Lecture**

Ici, "Exemple.txt" est le nom du fichier sur le disque dur, 4 est le numéro de canal, et ce fichier a donc été ouvert en lecture.

```
Ouvrir "Exemple.txt" sur 4 en Lecture
Variables chaine, nom, prénom, tel, mail en Caractères
```

```
Lire 4, chaine
Nom ←sousChaine(chaine, 1, 20)
Prénom ←sousChaine(chaine, 21, 15)
Tel ←sousChaine(chaine, 36, 9)
Mail ←sousChaine(chaine, 45, 20)
```

L'instruction *Lire* récupère donc dans la variable spécifiée (ici chaine) l'enregistrement suivant dans un fichier. Suivant par rapport à quoi ? Par rapport au dernier enregistrement lu. C'est en cela que le fichier est dit séquentiel.

Lire un fichier séquentiel de bout en bout suppose donc de programmer une boucle. Comme on sait rarement combien d'enregistrements comporte le fichier, la combine consiste à utiliser la fonction EOF (acronyme pour End Of File). Cette fonction est vraie si on a atteint la fin du fichier (auquel cas une lecture supplémentaire déclencherait une erreur). L'algorithme ultra classique en pareil cas est donc :

```
Variable chaine en Caractère
Ouvrir "Exemple.txt" sur 5 en Lecture
Tant que Non EOF(5)
   Lire 5, chaine
   ...
Fin TantQue
```

Pour une opération d'écriture, ou d'ajout, il faut d'abord constituer une chaîne équivalente à la nouvelle ligne du fichier. Cette chaîne doit donc être " calibrée " de la bonne manière, avec les différents champs qui " tombent " aux emplacements corrects. Le moyen le plus simple pour s'épargner de longs

traitements est de procéder avec des chaînes correctement dimensionnées dès leur déclaration (la plupart des langages offrent cette possibilité) :

```
Ouvrir "Exemple.txt" sur 3 en Ajout
Variable chaine en Caractère
Variables nom*20, prénom*15, tel*9, mail*15 en Caractère
...
nom ←"Mebarki"
prénom ←"Ali"
tel ←"021946532"
mail ←"alimeb@yahoo.fr"
chaine ←nom & prénom & tel & mail
Ecrire 3, chaine
```

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

4- Stratégies de traitement :

Il existe globalement deux manières de traiter les fichiers textes :

- l'une consiste à s'en tenir au fichier proprement dit. C'est parfois un peu acrobatique, lorsqu'on veut supprimer un élément d'un fichier : on programme alors une boucle avec un test, qui recopie dans un deuxième fichier tous les éléments du premier sauf un ; et il faut ensuite recopier intégralement le deuxième dans le premier... C'est infernal, n'est ce pas !!!
- l'autre consiste à passer par un ou plusieurs tableaux. On recopie l'intégralité du fichier de départ dans un tableau (en mémoire vive) et on ne manipule ensuite que ce tableau, qu'on recopie à la fin dans le fichier d'origine.

Les avantages de la seconde technique sont nombreux :

- la **rapidité** : les accès en mémoire vive sont des milliers de fois plus rapides (nanosecondes) que les accès aux mémoires de masse (millisecondes au mieux pour un disque dur).
- la **facilité de programmation** : *bien qu'il faut écrire en plus les* instructions de recopie du fichier dans le tableau, c'est largement plus *facile de faire cela avec un tableau* qu'avec des fichiers.

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

- la recopie d'un gros fichier en *mémoire vive consomme* des ressources qui peuvent atteindre des dimensions considérables. Et au final, le gain *de rapidité d'exécution* risque de ne pas être évident.
- si le fichier contient des données de type non homogènes (chaînes, numériques, etc.) cela risque de vous compliquer la tâche pour stocker ce fichier dans un tableau : il va falloir déclarer plusieurs tableaux, dont le maniement au *final peut être* aussi lourd que celui des fichiers de départ. Ou alors, il faut utiliser une ruse qu'offrent certains langages (mais pas tous) : créer des types de variables personnalisés, composés d'un " collage " de plusieurs types existants (10 caractères, puis un numérique, puis 15 caractères, etc.). Cette technique est un peu acrobatique. Bien qu'elle ne soit pas *vraiment difficile, elle exige tout* de même une certaine aisance.

Rappelons qu'à priori, les fichiers textes ne sont pas là pour servir de bases de données (en tout cas dans des langages de programmation récents). Donc, on aura peu de chances (ou de risques) de se trouver devant des immenses fichiers textes, ne pouvant pas être mis en tableau. Mais ce n'est pas une règle : il faut donc bien comprendre les tenants et les aboutissants de chaque stratégie. Et pour cela, quoi de mieux que la pratique ?

Exercices

Exercice 7.1

Quel résultat cet algorithme produit-il ?

```
Variable chaine en Caractère
Ouvrir "Exemple.txt" sur 5 en Lecture
Tantque Non EOF(5)
    LireFichier 5, chaine
    Ecrire chaine
FinTantQue
Fermer 5
```

Exercice 7.2

Ecrivez l'algorithme qui produit un résultat similaire au précédent, mais le fichier texte "Exemple.txt" est cette fois de type délimité (caractère de délimitation : /). On produira à l'écran un affichage où pour des raisons esthétiques, ce caractère sera remplacé avec des espaces.

Exercice 7.3

On travaille avec le fichier du carnet d'adresses en champs de largeur fixe.

Ecrivez un algorithme qui permet à l'utilisateur de saisir au clavier un nouvel individu qui sera ajouté à ce carnet d'adresses.

Exercice 7.4

Même question, mais cette fois le carnet est supposé être trié par ordre alphabétique. L'individu doit donc être inséré au bon endroit dans le fichier.

Exercice 7.5

Ecrivez un algorithme qui permet de modifier un renseignement (pour simplifier, disons uniquement le nom de famille) d'un membre du carnet d'adresses. Il faut donc demander à l'utilisateur quel est le nom à modifier, puis quel est le nouveau nom, et mettre à jour le fichier. Si le nom recherché n'existe pas, le programme devra le signaler.

Exercice 7.6

Ecrivez un algorithme qui trie les individus du carnet d'adresses par ordre alphabétique.

Exercice 7.7

Soient Toto.txt et Tata.txt deux fichiers dont les enregistrements ont la même structure. Ecrire un algorithme qui recopie tout le fichier Toto dans le fichier Tutu, puis à sa suite, tout le fichier Tata (concaténation de fichiers).

Exercice 7.8

Ecrire un algorithme qui supprime dans notre carnet d'adresses tous les individus dont le mail est invalide (pour employer un critère simple, on considèrera que sont invalides les mails ne comportant aucune arobase, ou plus d'une arobase).

Exercice 7.9

Les enregistrements d'un fichier contiennent les deux champs Nom (chaîne de caractères) et Montant (Entier). Chaque enregistrement correspond à une vente conclue par un commercial d'une société.

On veut mémoriser dans un tableau, puis afficher à l'écran, le total de ventes par vendeur. Pour simplifier, on suppose que le fichier de départ est déjà trié alphabétiquement par vendeur.

CORRIGES DES EXERCICES

Exercice 7.1

Cet algorithme écrit l'intégralité du fichier "Exemple.txt" à l'écran

---

Exercice 7.2

```
Variable chaine en Caractère
Variable i en Entier
Debut
Ouvrir "Exemple.txt" sur 5 en Lecture
Tantque Non EOF(5) Faire
    LireFichier 5, chaine
    Pour i de 1 à Len(chaine)
    // La fonction LEN renvoie la longueur de la chaine
        Si sousChaine(chaine, i, 1) = "/" Alors
            Ecrire " "
        Sinon
            Ecrire sousChaine(chaine, i, 1)
        FinSi
    Suivant
Fin TantQue
Fermer 5
```

Exercice 7.3

```
Variables nom*20, prénom*15, tel*9, mail*20, lig en Caractère
Debut
Ecrire "Entrez le nom : "
Lire nom
Ecrire "Entrez le prénom : "
Lire prénom
Ecrire "Entrez le téléphone : "
Lire tel
Ecrire "Entrez le mail : "
Lire mail
lig <-- nom & prénom & tel & mail
Ouvrir "Adresse.txt" sur 1 pour Ajout
EcrireFichier 1, lig
Fermer 1
Fin
```

Exercice 7.4

Là, comme indiqué dans le cours, on passe par un tableau de strutures en mémoire vive, ce qui est la technique la plus fréquemment employée. Le tri - qui est en fait un simple test - sera effectué sur le premier champ (nom).

```
personne = ENREG
   nom en Caractère * 20
   prénom en Caractère * 15
   tel en caractère * 9
   mail en Caractère * 20
Fin ENREG
Tableau mesContacts[] en Personne
Variables contact, nouveau en Personne
Variables i, j en Numérique
Debut
```

```
Ecrire "Entrez le nom : "
Lire nouveau.nom
Ecrire "Entrez le prénom : "
Lire nouveau.prénom
Ecrire "Entrez le téléphone : "
Lire nouveau.tel
Ecrire "Entrez le mail : "
Lire nouveau.mail
```

On recopie l'intégralité du fichier "Adresses.txt" dans le tableau mesContacts[]. Quand on tombe au bon endroit, on insère subrepticement notre nouveau contact dans le tableau.

```
Ouvrir "Adresse.txt" sur 1 pour Lecture
i <-- 0
inséré <-- Faux
Tantque Non EOF(1) Faire
    i <-- i + 1
    Redim mesContact[1..i]
    LireFichier 1, contact
    Si contact.nom > nouveau.nom Et Non inséré Alors
        mesContacts[i] <-- nouveau
        inséré <-- Vrai
        i <-- i + 1
        Redim mesContacts[1..i]
    FinSi
    mesContacts[i] <-- contact
Fin TantQue
Fermer 1
```

Et le tour est quasiment joué. Il ne reste plus qu'à rebalancer tel quel l'intégralité du tableau mesContacts dans le fichier, en écrasant l'ancienne version.

```
Ouvrir "Adresse.txt" sur 1 pour Ecriture
Pour j de 1 à i
```

```
    EcrireFichier 1, mesContacts[j]
j suivant
Fermer 1
Fin
```

Exercice 7.5

Cet algorithme ressemble beaucoup au précédent, à quelques variantes près. Il y a essentiellement une petite gestion de flag pour faire bonne mesure.

```
personne = ENREG
   nom en Caractère * 20
   prénom en Caractère * 15
   tel en caractère * 10
   mail en Carctère * 20
Fin ENREG
Tableau mesContacts[] en personne
Variables contact en personne
Variables ancien, nouveau en Caractère*20
Variables i, j en Numérique
Variable trouvé en Booléen
Debut
Ecrire "Entrez le nom à modifier : "
Lire ancien
Ecrire "Entrez le nouveau nom : "
Lire nouveau
```

On recopie l'intégralité du fichier "Adresses.txt" dans le tableau mesContacts, tout en recherchant le contact dont on veut modifier le nom. Si on le trouve, on procède à la modification.

```
Ouvrir "Adresse.txt" sur 1 pour Lecture
i <-- 0
trouvé <-- Faux
Tantque Non EOF(1) Faire
     i <-- i + 1
```

```
    Redim mesContacts[1..i]
    LireFichier 1, contact
     Si contact.nom = ancien.nom Alors
          trouvé <-- Vrai
          contact.nom <-- nouveau
     FinSi
    mesContacts[i] <-- contact
Fin TantQue
Fermer 1
```

On recopie ensuite l'intégralité du tableau mesContacts dans le fichier "Adresse.txt"

```
Ouvrir "Adresse.txt" sur 1 pour Ecriture
Pour j de 1 à i
    EcrireFichier 1, mesContacts[j]
j Suivant
Fermer 1
```

Et un petit message pour finir !

```
Si trouvé Alors
  Ecrire "Modification effectuée"
Sinon
  Ecrire "Nom inconnu. Aucune modification effectuée"
FinSi
Fin
```

Exercice 7.6

Là, c'est un tri sur un tableau de structures (d'enregistrements), rien de plus facile.

```
personne = ENREG
    nom en Caractère * 20
    prénom en Caractère * 15
    tel en caractère * 9
    mail en Carctère * 20
Fin ENREG

Tableau mesContacts[] en personne
Variables mini en personne
Variables i, j, k, posMini en Numérique
Debut
```

On recopie l'intégralité du fichier "Adresses.txt" dans le tableau mesContacts...

```
Ouvrir "Adresse.txt" sur 1 pour Lecture
i <-- 0
Tantque Non EOF(1) Faire
     i <-- i + 1
    Redim mesContacts[1..i]
    LireFichier 1, mesContacts[i]
Fin TantQue
Fermer 1
```

On procède maintenant au tri du tableau selon l'algorithme de tri par insertion, en utilisant le champ Nom de l'enregistrement :

```
Pour j de 1 à i
     mini <-- mesContacts[j]
     posmini <-- j
    Pour k de (j + 1) à i Faire
```

```
        Si mesContacts[k].nom < mini.nom Alors
            mini <-- mesContacts[k]
            posMini <-- k
        Finsi
    k suivant
   mesContacts[posMini] <-- mesContacts[j]
  mesContacts[j] <-- mini
j suivant
```

On recopie ensuite l'intégralité du tableau dans le fichier "Adresse.txt"

```
Ouvrir "Adresse.txt" sur 1 pour Ecriture
Pour j de 1 à i
    EcrireFichier 1, mesContacts[j]
j suivant
Fermer 1
Fin
```

Exercice 7.7

Bon, celui-là est tellement simple qu'on n'a même pas besoin de passer par des tableaux en mémoire vive.

```
Variable lig en Caractère
Debut
Ouvrir "Tutu.txt" sur 1 pour Ajout
Ouvrir "Toto.txt" sur 2 pour Lecture
Tantque Non EOF(2) Faire
    LireFichier 2, lig
    EcrireFichier 1, lig
Fin TantQue
Fermer 2
Ouvrir "Tata.txt" sur 3 pour Lecture
LireFichier 3, lig
Tantque Non EOF(3) Faire
    LireFichier 3, Lig
```

```
    EcrireFichier 1, Lig
Fin TantQue
Fermer 3
Fermer 1
```

Exercice 7.8

On va éliminer les mauvaises entrées dès la recopie : si l'enregistrement ne présente pas un mail valide, on l'ignore, sinon on le copie dans le tableau.

```
personne = ENREG
   nom : Caractère * 20
   prénom : Caractère * 15
   tel : caractère * 9
   mail : Caractère * 20
Fin ENREG
Tableau mesContacts[] : personne
Variable contact : personne
Variables i, j : Numérique
Debut
```

On recopie le fichier "Adresses.txt" dans le tableau mesContacts en testant le mail...

```
Ouvrir "Adresse.txt" sur 1 pour Lecture
i <-- 0
Tant que Non EOF(1) Faire
    LireFichier 1, contact
    nb <-- 0
    Pour i de 1 à Len(contact.mail)
        Si sousChaine(contact.mail, i, 1) = "@" Alors
            nb <-- nb + 1
        FinSi
    i suivant
    Si nb = 1 Alors
       i <-- i + 1
```

```
        Redim mesContacts[1..i]
        mesContacts[i] <-- contact
    FinSi
Fin TantQue
Fermer 1
```

On recopie ensuite l'intégralité du tableau mesContacts dans le fichier "Adresse.txt"

```
Ouvrir "Adresse.txt" sur 1 pour Ecriture
Pour j de 1 à i
    EcrireFichier 1, mesContacts[j]
j Suivant
Fermer 1
Fin
```

Exercice 7.9

Une fois de plus, le passage par un tableau de structures est une stratégie commode. Attention toutefois, comme il s'agit d'un fichier texte, tout est stocké en caractère. Il faudra donc convertir en numérique les caractères représentant les ventes, pour pouvoir effectuer les calculs demandés.

Pour le traitement, il y a deux possibilités :

- soit on recopie le fichier à l'identique dans un premier tableau, et on traite ensuite ce tableau pour faire la somme par vendeur.
- soit on fait le traitement directement, dès la lecture du fichier. C'est cette option qui est choisie dans ce corrigé.

```
Vendeur = ENREG
   Nom en Caractère * 20
   Montant en Numérique
Fin ENREG
Tableau mesVendeurs[] : Vendeur
Variables ligne, nom : caractère
Variables nomPrec : caractère * 20
Variables i, j, somme, vente : Numérique
```

On balaye le fichier en faisant nos additions. Dès que le nom a changé (on est passé au vendeur suivant), on range le résultat et on remet tout à zéro

```
Debut
Ouvrir "Ventes.txt" sur 1 pour Lecture
i <-- 0
LireFichier 1, ligne
nom <-- sousChaine(ligne, 1, 20)
vente <-- CNum(sousChaine(ligne, 21, 10))
// La fonction CNUM convertit une chaîne de caractères composée de chiffres
   en valeur numérique
Tantque Non EOF(1) Faire
    nomPrec <-- nom
    somme <-- 0
    // On va maintenant parcourir les ventes du même vendeur pour calculer
       le montant total de ses ventes
    Tantque nom = nomPrec Et Non EOF(1) Faire
        somme <-- somme + vente
        LireFichier 1, ligne
        Si Non EOF(1) Alors
            nom <-- sousChaine(ligne, 1, 20)
            vente <-- CNum(sousChaine(ligne, 21, 10))
        FinSi
    Fin TantQue
    // Remplir le tablea : on ajoute l'élément en cours
    i <-- i + 1
    Redim mesVendeurs[1..i]
    mesVendeurs[i].nom <-- nomPrec
    mesVendeurs[i].montant <-- somme
Fin TantQue
Fermer 1
// Pour terminer, on affiche le tableau à l'écran
Pour j de 1 à i
    Ecrire mesVendeurs[j]
j suivant        Fin
```