

# Introduction à l'algorithmique

## Master IC

V. Berry, M. Huchard, T. Libourel, J.F. Pons

2000-2009

# Table des matières

<b>1</b>	<b>Premiers éléments</b>	<b>3</b>
1.1	Introduction à l’algorithmique et à la programmation . . . . .	3
1.2	Présentation d’un algorithme . . . . .	4
1.3	Constantes, variables, types, expressions . . . . .	5
1.3.1	Le type <b>booléen</b> (ou logique) . . . . .	6
1.3.2	Le type <b>entier</b> . . . . .	6
1.3.3	Le type <b>réel</b> . . . . .	7
1.3.4	Le type <b>caractère</b> . . . . .	7
1.3.5	Le type <b>chaîne de caractères</b> . . . . .	8
1.4	Instructions simples . . . . .	8
1.4.1	Affectation . . . . .	8
1.4.2	Saisie de valeur . . . . .	9
1.4.3	Affichage de valeur . . . . .	9
1.4.4	Exemple d’algorithme . . . . .	10
<b>2</b>	<b>Instructions de contrôle</b>	<b>11</b>
2.1	Instructions conditionnelles . . . . .	11
2.1.1	Manipuler les conditions . . . . .	11
2.1.2	Conditionnelle à une possibilité . . . . .	13
2.1.3	Conditionnelle à deux possibilités (alternative) . . . . .	14
2.1.4	Conditionnelle à possibilités multiples . . . . .	14
2.2	Instructions répétitives . . . . .	16
2.2.1	Répétition contrôlée par un compteur . . . . .	16
2.2.2	Répétition contrôlée par une condition (forme 1) . . . . .	18
2.2.3	Répétition contrôlée par une condition (forme 2) . . . . .	19
<b>3</b>	<b>Fonctions</b>	<b>20</b>
3.1	Motivation . . . . .	20
3.2	Déclaration d’une fonction . . . . .	20
3.2.1	Entête . . . . .	21
3.2.2	Partie Déclarations de variables d’une fonction . . . . .	21
3.2.3	Partie Instructions d’une fonction . . . . .	21
3.2.4	Partie Appels de fonctions . . . . .	21
3.2.5	Exemple complet de déclaration de fonction . . . . .	22
3.2.6	Appel de la fonction . . . . .	22
3.3	Exécution d’une fonction . . . . .	22
3.4	Exemple : c’est les J.O.!!!! . . . . .	24
3.5	Modularité . . . . .	25

<b>4</b>	<b>Tableaux - Récursivité</b>	<b>26</b>
4.1	Motivation . . . . .	26
4.2	Définition et syntaxe . . . . .	26
4.2.1	Fonctions classiques sur les tableaux . . . . .	27
4.3	Exemples d'utilisation . . . . .	28
4.3.1	Initialisation d'un tableau . . . . .	28
4.3.2	Inversion d'un tableau . . . . .	28
4.3.3	Tri d'un tableau . . . . .	29
4.4	Tableaux à plusieurs dimensions . . . . .	30
4.4.1	Déclaration . . . . .	30
4.4.2	Exemple 1 : initialisation à zéro d'un tableau à deux dimensions . . . . .	31
4.4.3	Exemple 2 : carré "faiblement magique" . . . . .	31
4.5	Fonctions récursives . . . . .	31
4.5.1	Définition . . . . .	31
4.5.2	Exemple : la fonction <i>factorielle(n)</i> . . . . .	31

# Cours 1

## Premiers éléments

### 1.1 Introduction à l'algorithmique et à la programmation

Très schématiquement, un ordinateur peut être utilisé :

- de manière "passive", en utilisant des programmes (logiciels) existants, tels que traitement de textes, tableur, jeux, etc.
- de manière "active" en écrivant soi-même un programme pour résoudre un problème donné.

Cette distinction est modérée par le fait que l'utilisation de certains logiciels (tableurs, par exemple) peut également nécessiter la rédaction de programmes.

Nous nous intéressons ici à l'écriture de programmes. Pour mener à bien cette activité, il faut parvenir à :

- s'exprimer dans le langage de l'ordinateur (le "langage machine"), qui n'est constitué que de 0 et de 1,
- travailler avec des données dans sa mémoire. Ces données sont aussi des suites de 0 et de 1, et les placer ou les modifier en mémoire s'effectue par l'intermédiaire d'un système d'adressage à base de 0 et de 1.

Il est évident que l'on ne peut pas résoudre des problèmes complexes de cette manière. Le passage d'un problème à traiter à un programme qui le résolve comporte ainsi différentes étapes, que nous résumons ici.

1. **Analyse** : *du problème informel à la spécification.*

La spécification du problème, qui consiste à définir avec précision :

- le résultat à obtenir,
- les données nécessaires.

2. **Conception** : *de la spécification à une description abstraite de la manière de résoudre le problème et des moyens mis en jeux.*

La description des données et d'un "algorithme", c'est-à-dire d'une méthode pour résoudre le problème. Cette description présuppose un langage d'expression des données et des algorithmes, mais qui reste relativement abstrait (proche du langage naturel) comme nous le verrons. Même s'il n'est pas destiné à directement à la machine, ce langage demande un mode de raisonnement particulier. C'est cet aspect que traite ce cours.

3. **Codage** : *du papier au programme*

La traduction de l'algorithme et de la description des données dans un programme (souvent appelé "programme source") que l'ordinateur peut comprendre.

- Le programme est écrit sur un support accessible à l'ordinateur (fichier sur un support mémoire quelconque : disquette, disque, mémoire vive). Le fichier est réalisé à l'aide d'un

éditeur de textes.

- Il demande une grande rigueur d'écriture contrairement au langage d'algorithmique.
- Parmi les langages qui seront étudiés cette année, nous verrons notamment PERL et Java. Nous décrirons à ce moment-là les différentes sortes de langages de programmation qui existent.

#### 4. *Mise en œuvre : du programme au code machine*

L'ordinateur se charge du reste du travail (convertir le programme en langage machine de 0 et de 1), mais il faut le lui demander en utilisant différents outils (compilateurs, interpréteurs) qu'il propose. Il y a différentes stratégies d'utilisation du programme source suivant les langages et les environnements, dont nous reparlerons ensuite.

Dans la suite de ce cours introductif à l'algorithmique, nous nous concentrons sur le point 2 de la liste précédente, *i.e.* la conception.

## 1.2 Présentation d'un algorithme

Tout ce que nous décrirons dans ce cours sont des conventions "humaines" mais qui, de par leur présentation, seront relativement faciles à utiliser ensuite pour écrire des programmes sources dans le paradigme de programmation dit "impératif". Nous proposons ci-dessous un schéma général de présentation d'un algorithme.

**Algorithme** *nom\_de\_l'algorithme*

**But** *spécification du problème à résoudre*

**Données :** *données extérieures utilisées par l'algorithme*

**Résultats :** *résultats produits par l'algorithme vers l'extérieur*

**Principe :** *brève description de la méthode de résolution*

<b>Objets utilisés</b>	<b>Instructions</b>	<b>Actions non-élémentaires</b>
<i>objets et données internes à l'algorithme</i>	<i>suite des actions ou instructions à exécuter</i>	<i>rappel des actions non élémentaires utilisées et décrites dans d'autres algorithmes (procédures, fonctions)</i>

Nous donnons ci-après un aperçu de la manière dont cette structure serait utilisée, avec la recette de l'omelette.

**Algorithme** *PreparerOmelette*  
**But** *Préparation d'une omelette*

**Données :** *b est une boîte de 6 oeufs, p est une poêle*  
**Résultats :** *p contient une omelette préparée à l'aide des oeufs de b*

**Principe :** *casser et mélanger les oeufs dans un saladier, assaisonner, verser dans la poêle chaude, et cuire quelques minutes*

Objets utilisés	Instructions	Actions non-élémentaires
	/* — ceci est un commentaire, qui ne fait pas partie des instructions du programme — */	
entier <i>v</i> compte les oeufs	<b>pour</b> ( $v \leftarrow 1; v \leq 6; v \leftarrow v + 1$ ) <b>faire</b> casserOeuf( <i>v</i> , <i>s</i> )	casserOeuf( <i>v</i> , <i>s</i> ) consiste à casser le <i>vième</i> oeuf dans le saladier <i>s</i>
saladier <i>s</i>	<b>fpour</b> mélanger( <i>s</i> ) assaisonner( <i>s</i> )	assaisonner( <i>s</i> ) consiste à ajouter dans <i>s</i> du sel et du poivre
poêle <i>p</i>	verser( <i>s</i> , <i>p</i> )  cuire( <i>p</i> ,4)	verser( <i>s</i> , <i>p</i> ) consiste à verser dans <i>p</i> le contenu de <i>s</i> cuire( <i>p</i> , <i>t</i> ) consiste à faire chauffer <i>p</i> et à faire cuire son contenu pendant <i>t mn</i>

*Remarques :*

- certaines des rubriques du schéma ci-dessus ne figureront pas dans tous les exemples de la suite du cours : quand le contexte sera clair, nous ne préciserons que les rubriques pertinentes afin d'alléger l'écriture des algorithmes.
- Le langage utilisé n'est pas encore formalisé, c'est l'objet des sections suivantes.

### 1.3 Constantes, variables, types, expressions

Dans cette partie, nous étudions les données utilisées dans les algorithmes, et les opérations qui peuvent leur être appliquées. Ces données sont classées suivant des types et se manipulent grâce à des variables et des expressions.

**Type.** Un type peut se définir comme :

- un domaine, c'est-à-dire un ensemble de valeurs. Ces valeurs sont dénotées par des symboles que nous appellerons des **constants** dans le langage algorithmique. Le type limite les valeurs qui peuvent être prises par une variable.
- un ensemble d'opérations que l'on peut effectuer sur les valeurs du type. Ces opérations sont dénotées dans les algorithmes par des symboles que nous appellerons des **opérateurs**.

Nous verrons dans ce cours cinq types et les opérations les plus courantes qui leur sont associées : logique (booléen), entier, réel, caractère, chaîne de caractères.

**Variable.** Une variable est un triplet (*identificateur,type,valeur*). Comme en mathématiques, les variables permettent d'écrire de manière plus générale que si on ne s'autorisait que l'utilisation de constantes. En informatique, elles correspondent aussi à une abstraction de la notion d'emplacement mémoire, ainsi nous les représenterons quelquefois par de petites boîtes, ou des cases d'un tableau dans les traces d'algorithme. L'identificateur doit débuter par une lettre, cette lettre peut être suivie d'une combinaison quelconque de lettres, chiffres et du caractère '\_' (*souligné*). Exemples : *mon\_compteur*, *n274*, *monCompteur*.

**Expression.** Une expression est une combinaison bien formée de variables, constantes, opérateurs et parenthèses. Elle a également un type, et une valeur. Exemple :  $(2 * 3)$ .

### 1.3.1 Le type booléen (ou logique)

#### Domaine.

Deux valeurs, notées **V** (vrai) et **F** (faux)

#### Opérations.

**non**, **egal**, **non\_egal**, **et**, **ou**, qui se définissent par les tables suivantes, pour des variables booléennes *a* et *b*. Les opérateurs binaires sont utilisés en notation infixé (placés entre leurs opérandes).

<i>a</i>	<b>non a</b>
V	F
F	V

<i>a</i>	<i>b</i>	<b>a egal b</b>	<b>a non_egal b</b>	<b>a et b</b>	<b>a ou b</b>
V	V	V	F	V	V
V	F	F	V	F	V
F	V	F	V	F	V
F	F	V	F	F	F

#### Expressions.

*V ou F* (résultat booléen)

$(a \text{ ou } b) \text{ et } (c \text{ ou } V)$  (résultat booléen) si *a*, *b*, *c* sont des variables booléennes.

### 1.3.2 Le type entier

#### Domaine.

Un intervalle de  $\mathcal{Z}$ , ensemble des entiers relatifs, de la forme  $[-p, p - 1]$ , où *p* est 2 élevé à une puissance entière, par exemple  $b = 2^{31}$ . Les entiers se notent de la manière usuelle : 2 , -45 , +5789 , 5789 , 0

#### Opérations.

Elles se divisent en deux catégories, et sont utilisées en notation infixé (entre les opérandes) :

- opérations arithmétiques à résultat entier : addition (notée +), soustraction (notée -), multiplication (notée \*), division (notée /), modulo (notée %).

- opérations de comparaison à résultat booléen : inférieur (notée  $<$ ), supérieur (notée  $>$ ), inférieur ou égal (notée  $\leq$ ), supérieur ou égal (notée  $\geq$ ), égalité (notée **egal**), différence (notée **non\_egal**).

### Expressions.

$11/3$  (qui vaut 3),  $11\%3$  (qui vaut 2)

$4 * x$  (résultat entier), où  $x$  est une variable de type entier

$v > 4$  (résultat booléen), où  $v$  est une variable de type entier

Nous supposons les priorités habituelles en mathématiques :  $*$  et  $/$  sont prioritaires par rapport à  $-$  et  $+$ , les expressions entre parenthèses sont évaluées en commençant par les plus profondes, enfin on évalue de gauche à droite. Ainsi,  $4 + v * c$  vaut  $4 + (v * c)$  et non pas  $(4 + v) * c$ .

### 1.3.3 Le type réel

#### Domaine.

Une partie des réels, bornée par un intervalle, et en prenant un nombre maximum de chiffres significatifs. Par exemple, les réels inférieurs à  $10^{38}$  en valeur absolue, avec au plus 11 chiffres significatifs.

Les réels se notent avec un point à la place de la virgule (notation anglaise) :  $1.05$ ,  $+1.05$ ,  $-1.005$  avec un exposant :  $1.3E+2$  (pour  $1.3 * 10^2$ )

#### Opérations.

- opérations mathématiques usuelles  $+$   $-$   $*$   $/$ , qui peuvent être utilisées avec des entiers. L'opération  $/$  n'est pas ambiguë, c'est l'opération entière seulement si les deux opérandes sont des entiers.
- opérations de comparaison  $<$   $\leq$   $>$   $\geq$ , **egal**, **non\_egal** (ces deux derniers sont définis à la précision près).

#### Expressions.

$11/3.0$  vaut  $3.666\dots$ ,  $11\%3.0$  n'est pas défini

### 1.3.4 Le type caractère

#### Domaine.

Un ensemble de caractères, dont nous supposons qu'il contient au moins les chiffres, les lettres, les signes de ponctuation, et les opérateurs.

Les caractères se notent entre apostrophes simples (ou "quote" en anglais) : `'A'` `'a'` `'+''`,

On différencie ainsi l'entier 1 du caractère `'1'`.

Nous utiliserons en algorithmique le caractère `'\n'` qui signifie " fin de ligne" et les caractères guillemet ou apostrophe seront précédés du caractère `'\'`.

#### Opérations.

- opérations de comparaison  $<$   $\leq$   $>$   $\geq$ , **egal**, **non\_egal**  
Pour les lettres de l'alphabet, l'ordre coïncide avec l'ordre alphabétique habituel. En algorithmique, nous ne ferons pas d'autres hypothèses.

## Expressions.

'a' ; 'z' (résultat booléen : V)

### 1.3.5 Le type chaîne de caractères

#### Domaine.

Un ensemble de suites finies (ou séquences) de caractères.

Les chaînes de caractères se notent entre guillemets (ou "double quote" en anglais) : "prune", "citron\norange", "" (chaîne vide), "a" (chaîne formée du seul caractère 'a'), "abc\nde" (chaîne formée des caractères a b c " d e).

#### Opérations.

- concaténation de deux chaînes ou d'une chaîne et d'un caractère (notée +)
- opérations de comparaison < ≤ > ≥ , **egal**, **non\_egal**

## Expressions.

"il"+' '+ "pleut" (résultat, la chaîne de caractères "il pleut")

"arbre"; "zoo" (résultat booléen, V)

## 1.4 Instructions simples

Une instruction est une action élémentaire mettant en jeu des variables et/ou des expressions. L'exécution d'un algorithme consiste à exécuter les unes après les autres les instructions qui le composent. Tant que l'on utilise uniquement des instructions simples, le flot de contrôle, c'est-à-dire la suite d'instructions exécutées, progresse de façon linéaire : dans l'ordre où les instructions apparaissent dans l'algorithme.

Nous étudions ici trois instructions simples, l'affectation, la saisie (ou "lecture au clavier") et l'affichage (ou "écriture à l'écran  $i_i$ ).

### 1.4.1 Affectation

Elle permet de donner une valeur à une variable.

#### Syntaxe.

$NV \leftarrow E$

où  $NV$  est un identificateur de variable,  $E$  une expression.

#### Effet.

1. calcul de la valeur de  $E$  (évaluation)

2. *NV* reçoit cette valeur (placée dans la case mémoire correspondante).

L'expression et la variable doivent être de même type, avec certaines tolérances (une variable de type réel peut par exemple recevoir une valeur entière).

#### Algorithme *MontreAffectations*

Objets utilisés	Instructions
entier <i>i</i>	<i>/* i a une valeur indéterminée */</i> <i>i</i> ← 2 <i>/* i vaut 2 */</i> <i>i</i> ← <i>i</i> + 4 <i>/* i vaut 6 */</i>
réel <i>r</i>	<i>r</i> ← <i>i</i> /4 <i>/* r vaut 1.0 (division entière!) */</i> <i>r</i> ← <i>i</i> /4.0 <i>/* r vaut 1.5 */</i>

### 1.4.2 Saisie de valeur

Nous considérons seulement la saisie d'une valeur par l'intermédiaire d'un clavier, cette notion de saisie sera élargie lorsque nous ferons de la programmation.

#### Syntaxe.

```
lire clavier NV
```

où *NV* est un identificateur de variable

#### Effet.

Des caractères doivent être frappés au clavier pour que le programme puisse progresser. Ces caractères sont interprétés comme une valeur du type de *NV* qui est ensuite affectée à *NV*.

Pour une instruction `lire clavier NV`, une frappe des caractères 1 2 7 sera interprétée comme la chaîne "127" si *NV* est de type chaîne de caractères, ou bien comme l'entier 127 si *NV* est de type entier.

#### Forme simplifiée.

la séquence d'instructions `lire clavier V1; lire clavier V2; ... lire clavier VN` s'écrit plus simplement `lire clavier V1,V2, ... VN`;

### 1.4.3 Affichage de valeur

Symétriquement, nous traitons l'affichage à l'écran, que nous étendrons à l'écriture sur d'autres périphériques.

**Syntaxe.**

<p>écrire écran E</p> <p>où E est une expression</p>
--

**Effet.**

La valeur de l'expression E est calculée, puis affichée à l'écran.

**Forme simplifiée.**

la séquence d'instructions écrire écran E1; écrire écran E2;... écrire écran EN s'écrit plus simplement écrire écran E1,E2, ... EN;

**1.4.4 Exemple d'algorithme**

Nous mettons en œuvre ce que nous venons de voir dans un algorithme simple :

**Algorithme** *SommeEtProduit*

**Données :** *deux réels lus au clavier*

**Résultats :** *affiche la somme et le produit des deux réels*

Objets utilisés	Instructions
réels $x, y$ /* recoivent les valeurs lues */	écrire écran "entrez deux réels" lire clavier $x$ lire clavier $y$ écrire écran "somme = ", $x + y$ , "produit = ", $x * y$

## Cours 2

# Instructions de contrôle

Les instructions présentées ci-dessous ont pour principale conséquence de permettre au flot de contrôle d'avoir une progression non-linéaire : certaines instructions ne seront pas exécutées (suivant le résultat d'une expression booléenne lors de l'exécution), tandis que d'autres pourront être exécutées plusieurs fois.

Avec les instructions de contrôle apparaît aussi la notion de *bloc d'instructions* : il s'agit d'instructions se suivant dans l'algorithme et que l'on regroupe car elles constituent un ensemble d'instructions exécutées conjointement, suite à une instruction de contrôle (le sens de cette notion se précisera au fur et à mesure que les instructions de contrôle seront présentées).

Les blocs d'instructions sont indiqués par une *indentation* spécifique : chaque ligne du bloc commence avec un retrait à droite plus important que les lignes le précédant et le suivant.

## 2.1 Instructions conditionnelles

Les instructions conditionnelles permettent d'exprimer des choix dans un algorithme, sur la base de la valeur de vérité d'une condition (expression de type booléen).

### 2.1.1 Manipuler les conditions

Nous faisons quelques rappels sur les opérations logiques, qui seront utiles en particulier pour exprimer, simplifier ou reconnaître l'équivalence des conditions.

Les opérations logiques ont les propriétés suivantes, pour  $a, b, c$  des expressions de type booléen.

#### **Commutativité**

$a \text{ et } b = b \text{ et } a$

$a \text{ ou } b = b \text{ ou } a$

#### **Associativité**

$(a \text{ et } b) \text{ et } c = a \text{ et } (b \text{ et } c)$

$(a \text{ ou } b) \text{ ou } c = a \text{ ou } (b \text{ ou } c)$

#### **Simplifications**

$a \text{ ou } a = a, a \text{ et } a = a, \text{non}(\text{non } a) = a$

$a \text{ ou non } a = V, a \text{ et non } a = F$

$a \text{ ou } V = V, a \text{ et } V = a$

$a \text{ ou } F = a, a \text{ et } F = F$

**Distributivité**

$$a \text{ ou } (b \text{ et } c) = (a \text{ ou } b) \text{ et } (a \text{ ou } c)$$

$$a \text{ et } (b \text{ ou } c) = (a \text{ et } b) \text{ ou } (a \text{ et } c)$$

**Lois de Morgan**

$$\text{non}(a \text{ ou } b) = \text{non}a \text{ et } \text{non}b$$

$$\text{non}(a \text{ et } b) = \text{non}a \text{ ou } \text{non}b$$

**Exercice.** Pour  $a, b, c$  variables de type entier,

1. Simplifiez la condition  $(a \text{ non\_egal } 0)$  ou  $((a \text{ egal } 0) \text{ et } (b \neq c))$
2. Pour trois variables de type booléen  $a, b, c$ , construire la table de vérité de l'expression  $a \text{ ou } (\text{non}a \text{ et } b)$ .

## 2.1.2 Conditionnelle à une possibilité

### Syntaxe.

```
si C alors
  A1; A2; ...; An
fsi
```

où  $C$  est une condition,  $A_1; A_2; \dots A_n$  sont des instructions.

Pour séparer plusieurs instructions on utilisera conventionnellement le ";" (comme dans de nombreux langages de programmation).

### Effet.

$C$  est évaluée. Si  $C$  vaut V, les instructions  $A_1; A_2; \dots A_n$  sont exécutées puis l'instruction termine. Si  $C$  vaut F, l'instruction termine. La suite d'instructions  $A_1; A_2; \dots A_n$  constitue un bloc d'instructions. Elles ont un destin commun : en fonction du résultat de la condition, elles sont soit toutes considérées, soit toutes ignorées.

### Algorithme *EcrireOrdre1*

**Données :** deux entiers lus au clavier

**Résultats :** affichage à l'écran du plus petit puis du plus grand de ces nombres

**Principe :** Les deux nombres sont placés dans deux variables  $a$  et  $b$ . Si nécessaire les contenus de  $a$  et  $b$  sont échangés de sorte que  $a$  contienne le plus petit, et  $b$  le plus grand.

Objets utilisés	Instructions
entiers $a, b$ les entiers lus	écrire écran "entrez deux entiers" lire clavier $a, b$ si $a > b$ alors /* échange des valeurs */
entier $vi$ sert pour l'échange	$vi \leftarrow a$ $a \leftarrow b$ $b \leftarrow vi$ fsi écrire écran $a, ' ', b, '\n'$

### 2.1.3 Conditionnelle à deux possibilités (alternative)

**Syntaxe.**

```
si  $C$  alors
   $A_1; A_2; \dots; A_n$ 
sinon
   $B_1; B_2; \dots; B_n$ 
fsi
```

où  $C$  est une condition,  
 $A_1; A_2; \dots; A_n$  ,  $B_1; B_2; \dots; B_n$  sont des instructions

**Effet.**

$C$  est évaluée. Si  $C$  vaut V, les instructions  $A_1; A_2; \dots; A_n$  sont exécutées puis l'instruction termine.

Si  $C$  vaut F, les instructions  $B_1; B_2; \dots; B_n$  sont exécutées puis l'instruction termine.

Les instructions  $A_1; A_2; \dots; A_n$  et  $B_1; B_2; \dots; B_n$  constituent deux blocs d'instructions.

**Algorithme** *EcrireOrdre2*

**Données :** *deux entiers lus au clavier*

**Résultats :** *affichage à l'écran du plus petit puis du plus grand de ces nombres*

**Principe :** Les deux nombres sont placés dans deux variables  $a$  et  $b$ . L'ordre entre  $a$  et  $b$  détermine l'ordre d'affichage.

Objets utilisés	Instructions
entier $a, b$ les entiers lus	écrire écran "entrez deux entiers" lire clavier $a, b$ <b>si</b> $a > b$ <b>alors</b> écrire écran $b, ',a, '\n'$ <b>sinon</b> écrire écran $a, ',b, '\n'$ <b>fsi</b>

### 2.1.4 Conditionnelle à possibilités multiples

Nous présentons une forme affaiblie de cette instruction qui consiste en toute généralité à proposer un nombre quelconque de couples (condition, séquence d'instructions).

### Syntaxe.

```
cas où  $E$  vaut
   $v_1 : A_{11}; A_{12}; \dots; A_{1n_1}$ 
   $v_2 : A_{21}; A_{22}; \dots; A_{2n_2}$ 
  .....
   $v_m : A_{m1}; A_{m2}; \dots; A_{mn_m}$ 
  autrement :  $B_1; B_2; \dots; B_n$ 
fcas
```

où  $E$  est une variable entière ou caractère,  
 $v_1; \dots; v_m$  sont des constantes distinctes du même type que  $E$   
 $A_{11}; \dots; A_{mn_m}; B_1; \dots; B_n$  sont des instructions

### Effet.

Nous pouvons décrire le fonctionnement en utilisant les instructions conditionnelles vues précédemment.

```
si  $E$  egal  $v_1$  alors
   $A_{11}; A_{12}; \dots; A_{1n_1}$ 
sinon
  si  $E$  egal  $v_2$  alors
     $A_{21}; A_{22}; \dots; A_{2n_2}$ 
  sinon
    .....
    si  $E$  egal  $v_m$  alors
       $A_{m1}; A_{m2}; \dots; A_{mn_m}$ 
    sinon
       $B_1; B_2; \dots; B_n$ 
  fsi
fsi
fsi
```

Quelques remarques sur cette instruction.

La partie suivante est optionnelle :

**autrement** :  $B_1; B_2; \dots; B_n$

Dans le cas où elle n'est pas présente, l'alternative équivalente est réduite de :

**sinon**

$B_1; B_2; \dots; B_n$

L'ordre d'écriture des cas n'est pas significatif, ce qui est logique puisqu'ils sont mutuellement exclusifs et que l'instruction termine à l'issue de la première séquence d'instructions exécutée.

Lorsque l'on doit exécuter la même suite d'instructions  $A_{j1}; A_{j2}; \dots; A_{jn_j}$  pour plusieurs valeurs  $v_{i_1}, v_{i_2}, \dots, v_{i_n}$ , on peut utiliser la notation abrégée suivante :

$v_{i_1}, v_{i_2}, \dots, v_{i_n} : A_{j1}; A_{j2}; \dots; A_{jn_j}$

### Algorithme Reconnaît Voyelles

**Données :** un caractère lu au clavier

**Résultats :** affiche un message différenciant voyelles minuscules et voyelles majuscules des autres caractères

Objets utilisés	Instructions
caractère <i>c</i> lu au clavier	écrire écran "Entrez un caractère" lire clavier <i>c</i> cas où <i>c</i> vaut 'a', 'e', 'i', 'o', 'u', 'y' : écrire écran "voyelle minuscule" 'A', 'E', 'I', 'O', 'U', 'Y' : écrire écran "voyelle majuscule" autrement : écrire écran "ce n'est pas une voyelle!" f cas

## 2.2 Instructions répétitives

Ces instructions, qui sont aussi appelées "itérations" ou plus familièrement "boucles" permettent de répéter un bloc d'instructions jusqu'à ce qu'une certaine condition devienne fausse. Bien évidemment, si les instructions effectuées au sein de la boucle ne permettent pas à cette condition de devenir fausse, l'algorithme ne peut terminer.

### 2.2.1 Répétition contrôlée par un compteur

**Syntaxe.**

```
pour (vc ← E; C; I) faire  
    A1; A2; ...; An  
fpour
```

où *vc* est une variable entière, appelée variable de contrôle ou compteur,  
*E* est une expression entière qui donne la valeur initiale de *vc*,  
*C* est une condition portant sur la valeur de *vc*,  
*I* est une instruction "de pas" dont le rôle est d'incrémenter ou de décrémenter le compteur,  
*A*<sub>1</sub>; *A*<sub>2</sub>; ...; *A*<sub>*n*</sub> sont des instructions.

**Effet.**

La valeur *E* est affectée à *vc*.

(1) si *C* vaut V alors *A*<sub>1</sub>; *A*<sub>2</sub>; ...; *A*<sub>*n*</sub> sont exécutées, suivies de *I*, puis le traitement recommence en (1).

sinon (*C* vaut F), l'instruction termine (sortie de la boucle).

**Exercice.** Effectuez la trace de cet algorithme pour  $x = 4$ ,  $y = 3$ .

### Algorithme *PuissanceEntière1*

**Données :** deux entiers  $x > 0, y \geq 0$ , lus au clavier

**Résultats :** affiche à l'écran  $x^y$

**Principe :**  $x^0$  vaut 1,  $x^y$  vaut  $1 * x * x * x * \dots * x$  (y occurrences de x)

Objets utilisés	Instructions
entiers $x, y$ lus au clavier	écrire écran "Entrez un entier $x > 0$ " lire clavier $x$
entier $res$ vaut $x^v$ après l'étape $v$	écrire écran "Entrez un entier $y \geq 0$ " lire clavier $y$ $res \leftarrow 1$
entier $v$ var. de contrôle	<b>pour</b> ( $v \leftarrow 1; v \leq y; v \leftarrow v + 1$ ) <b>faire</b> $res \leftarrow res * x$ <b>fpour</b> écrire écran " $x^y$ vaut ", $res$

Pour tracer l'exécution d'un algorithme on peut avoir recours à un *tableau de variables*. Chaque colonne du tableau correspond à une variable différente. Les lignes correspondent aux instructions dans l'ordre où elles sont exécutées (ainsi une instruction exécutée plusieurs fois fera l'objet de plusieurs lignes). Chaque case du tableau indique la valeur de la variable de sa colonne quand l'instruction de sa ligne a été exécutée.

## 2.2.2 Répétition contrôlée par une condition (forme 1)

**Syntaxe.**

```

tant que  $C$  faire
   $A_1; A_2; \dots; A_n$ 
ftant

où  $C$  est une condition,
 $A_1; A_2; \dots; A_n$  sont des instructions
    
```

**Effet.**

(1) si  $C$  vaut V alors  $A_1; A_2; \dots; A_n$  sont exécutées, puis le traitement reprend en (1).  
 sinon ( $C$  vaut F), l'instruction termine.

**Algorithme** *PuissanceEntière2*

**Données :** deux entiers  $x > 0, y \geq 0$ , lus au clavier

**Résultats :** affiche à l'écran  $x^y$

**Principe :**  $x^0$  vaut 1,  $x^y$  vaut  $1 * x * x * x * \dots * x$  (y occurrences de x)

Objets utilisés	Instructions	Actions non-élémentaires
entier x,y lus au clavier	écrire écran "Entrez un entier $x > 0$ " lire clavier x écrire écran "Entrez un entier $y \geq 0$ " lire clavier y	
entier $res$ vaut $x^i$ à l'étape i	$res \leftarrow 1$ $v \leftarrow 1$ <b>tant que</b> ( $v \leq y$ ) <b>faire</b> $res \leftarrow res * x$	
entier $v$ var. de contrôle	$v \leftarrow v + 1$ <b>ftant</b> écrire écran " $x^y$ vaut ", res	

### 2.2.3 Répétition contrôlée par une condition (forme 2)

Cette forme permet d'effectuer au moins une fois la séquence d'instructions avant de tester la condition.

#### Syntaxe.

<pre>faire   A<sub>1</sub>; A<sub>2</sub>; ...; A<sub>n</sub> tant que C  où C est une condition, A<sub>1</sub>; A<sub>2</sub>; ...; A<sub>n</sub> sont des instructions</pre>
--

#### Effet.

(1)  $A_1; A_2; \dots; A_n$  sont exécutées  
si  $C$  vaut V alors le traitement reprend en (1).  
sinon ( $C$  vaut F), l'instruction termine.

#### Exemple.

Dans l'algorithme précédent, si nous voulons être certains d'avoir saisi un entier strictement positif pour initialiser  $x$ , nous pouvons remplacer les instructions :

écrire écran "Entrez un entier  $x > 0$ "

lire clavier  $x$

par les instructions suivantes :

entier $x$	faire écrire écran "Entrez un entier $x > 0$ " lire clavier $x$ tant que ( $x \leq 0$ )	
------------	--	--

## Cours 3

# Fonctions

### 3.1 Motivation

Lorsqu'on écrit un algorithme (programme), il arrive que l'on ait besoin de réaliser plusieurs fois la même fonction ou le même traitement sur des valeurs ou des variables différentes, comme par exemple, calculer le numéro de la semaine (entre 1 et 52) en fonction de la date d'une journée, calculer la moyenne des notes d'un élève, ....

Comme les différentes fois où l'on a besoin d'effectuer ce même traitement peuvent ne pas être consécutives et/ou peuvent concerner des valeurs ou des variables différentes, on ne peut pas intégrer ce traitement répétitif dans une boucle (tant que, pour, etc).

Dans cette situation, la solution consiste à écrire une **fonction** : bloc d'instructions nommé acceptant en entrée des **paramètres** (variables particulières) et donnant un résultat.

Le nom de *fonction* vient de l'analogie avec les mathématiques, par exemple la fonction  $abs(x)$  qui renvoie la valeur absolue d'une valeur  $x$ .

Une fonction représente un bloc d'instructions *secondaire* qui peut être utilisé à différentes reprises par le bloc *principale* de l'algorithme, et ce, sur différentes expressions ou variables.

### 3.2 Déclaration d'une fonction

La déclaration d'une fonction comporte 4 parties, agencées ici de la façon suivante :

entête de la fonction		
déclaration de variables	instructions	actions non-élémentaires

Vous remarquerez que nous changeons légèrement l'intitulé de la colonne de gauche, les "objets utilisés" étant plus précisément, dans le cas des fonctions, des variables locales.

### 3.2.1 Entête

**type fonction** *nom\_de\_la\_fonction* (liste des paramètres et de leurs types)

**But** : spécification du problème à résoudre

**Paramètres** : liste des paramètres de la fonction, classés par genre (E,E/S,S)

**Autres données** : données extérieures utilisées par l'algorithme

**valeur renvoyée** : résultat produit par l'algorithme vers l'extérieur. Il s'agit d'une seule valeur

**Autres résultats** : essentiellement les sorties écrans

L'entête précise le nom de la fonction, le type de valeur qu'elle retourne (entier, réel, etc), les valeurs ou variables à lui fournir, les autres données éventuelles, le résultat renvoyé par la fonction et les autres résultats éventuels.

La fonction désigne par des noms génériques, qu'on appelle **paramètres formels**, les valeurs et variables qui lui sont fournies lors de son appel. Ces noms génériques permettent de décrire les traitements effectués par la fonction quel que soit le nom des variables ou les valeurs avec lesquelles elle pourra être réellement appelée lors de l'exécution.

Les paramètres peuvent être de trois catégories :

- **Entrée** : leur valeur est consultée par la fonction, mais ne sera pas modifiée par son exécution ;
- **Sortie** : leur valeur ne sera pas lue par la fonction mais sera modifiée après son exécution ;
- **Entrée/Sortie** : leur valeur peut être consultée *et* modifiée par la fonction.

Pour chaque catégorie de paramètres (E,S,E/S), on donne la liste des paramètres en précisant pour chacun le type et le nom générique adopté (complétés d'une courte explication du sens de cette variable).

Enfin, l'entête détaille le type et le sens de la valeur calculée par la fonction.

Exemple :

<pre>réel fonction Moyenne (entier <i>n</i>)   paramètre E :      <i>n</i>, nombre de notes dont on veut calculer la moyenne   valeur retournée : réel (la moyenne de <i>n</i> nombres lus au clavier)</pre>
--

### 3.2.2 Partie Déclarations de variables d'une fonction

Outre les variables correspondant aux paramètres, la fonction peut utiliser des **variables locales**, dont on précisera le type, le nom et le sens dans la colonne de gauche.

Exemple : entier *i* un compteur.

### 3.2.3 Partie Instructions d'une fonction

La colonne centrale abrite les instructions exécutées par la fonction (appelées parfois le **corps** de la fonction).

L'instruction **Retourner** permet d'indiquer le résultat qui est renvoyé par la fonction à l'algorithme (ou la fonction) qui l'utilise.

### 3.2.4 Partie Appels de fonctions

La colonne de droite indique quelles sont les fonctions utilisées (on dit aussi "appelées") par la fonction décrite (nous verrons bientôt des exemples où cela se produit).

### 3.2.5 Exemple complet de déclaration de fonction

réel fonction Moyenne (entier $n$ ) paramètre E :            le nombre $n$ de notes dont on veut calculer la moyenne autres données : $n$ réels lus au clavier valeur retournée :      réel (la moyenne de $n$ nombres lus au clavier)		
réel $s$ la somme des réels lus entier $i$ un compteur réel $r$ les nbs lus au clavier	$s \leftarrow 0$ <b>pour</b> ( $i \leftarrow 1; i \leq n; i \leftarrow i+1$ ) <b>faire</b> lire clavier $r$ $s \leftarrow s + r$ <b>fpour</b> $r \leftarrow s/n$ retourner $r$	

### 3.2.6 Appel de la fonction

Après la déclaration d'une fonction, tout est maintenant comme si pour écrire l'algorithme l'on disposait d'une nouvelle instruction ou expression, correspondant au nom de la fonction et prenant comme arguments les variables ou valeurs sur lesquelles on veut qu'elle travaille. Son résultat peut être récupéré dans une variable par une affectation simple (l'appel sera alors considéré comme une expression) :

<b>Algorithme Utilisation</b> But :afficher la moyenne de plusieurs nombres lus au clavier.		
entier $n\text{notes}$ nombre de notes à moyenner réel $result$ la moyenne calculée	écrire écran "Donnez le nombre de notes" lire clavier $n\text{notes}$ $result \leftarrow moyenne(n\text{notes})$ écrire écran $result$ .	moyenne( $n\text{notes}$ ) consiste à calculer une moyenne d'autant de notes qu'indiqué.

Une fonction peut ne rien renvoyer (par exemple, la fonction  $Permute(a, b)$ <sup>1</sup>, dont le seul but est de modifier la valeur de ses deux paramètres), auquel cas tout appel à cette fonction est considéré comme une instruction. Dans les autres cas on considère un appel de fonction comme une expression.

## 3.3 Exécution d'une fonction

Lorsque le flot de contrôle (parfois appelé **pointeur d'exécution**) rencontre un appel de fonction, schématiquement les étapes suivantes sont suivies :

1. le pointeur passe dans la partie du code qui correspond à cette fonction. La fonction constitue un bloc secondaire à exécuter.
2. autant de cases mémoires que nécessaires pour les paramètres formels sont réservées; ces cases portent les noms des variables génériques indiqués dans la déclaration de la fonction et

<sup>1</sup>utilisée p31

sont remplies par les valeurs respectives des variables indiquées dans l'appel de la fonction, appelées **paramètres réels** ou **effectifs**.

3. les cases mémoires nécessaires aux variables locales sont aussi réservées, mais non initialisées (idem dans le bloc principal).
4. la 1ère instruction de la fonction est exécutée et le pointeur progresse au fur et à mesure des instructions de la fonction jusqu'à atteindre la dernière instruction de la fonction ou rencontrer l'instruction **retourner** (qui devient la dernière instruction par son sens).
5. Si la dernière instruction à exécuter est **retourner**, la valeur indiquée dans cette instruction, est renvoyée comme résultat de la fonction. Par exemple, **retourner 2**, ou **retourner res**, signifiant "retourner la valeur de la variable *res*".
6. Dans tous les cas, une fois exécutée la dernière instruction, la valeur des paramètres réels de type E/S et S est remplacée par celle des paramètres formels, puis toutes les variables de la fonction sont alors "oubliées" et l'exécution du bloc appelant reprend à l'instruction qui suivait l'appel de la fonction.

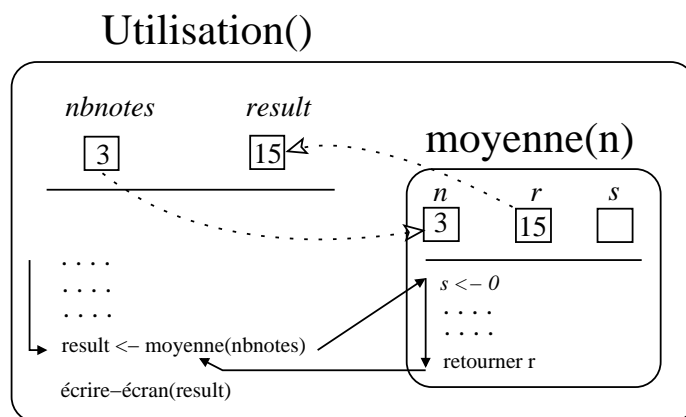


FIG. 3.1 – Exemple d'exécution de la fonction *moyenne(n)*.

On représente schématiquement l'exécution d'un bloc (principal ou secondaire) par une *boîte* contenant les instructions du bloc ainsi que les cases mémoires de ses variables. Dans la figure 3.1, la boîte de la fonction *moyenne(n)* est contenue dans celle de l'algorithme pour indiquer qu'elle est appelée par celui-ci.

Le **contexte** d'un bloc désigne l'ensemble des variables qu'il connaît : celles qu'il déclare (dites variables *locales*) et ses paramètres éventuels. Malheureusement, dans certains langages de programmation, le contexte d'un bloc inclut aussi les variables déclarées par le bloc qui l'appelle (dites variables *globales*). Toutefois, utiliser de telles variables dans une fonction est source de nombreuses erreurs et nuit à la clarté des programmes, aussi nous éviterons ici cette pratique et utiliserons les paramètres pour transmettre toutes les variables nécessaires à une fonction.



*En raison de la définition du contexte, on portera une attention particulière au choix du nom des variables utilisées (bloc principal/fonctions/paramètres) de façon à éviter toute confusion possible. Donc si possible ne jamais utiliser le même nom de variables dans deux déclarations différentes.*

**Exercice** : comment réaliser la fonction `Permute(A,B)` qui échange la valeur de deux variables de type `entier` ?

**Exercice** : prendre un algorithme et le transformer en fonction.

### 3.4 Exemple : c'est les J.O.!!!!

*Problème à résoudre* : on veut calculer l'écart de temps entre deux sportifs à l'arrivée d'une épreuve d'après le temps qu'ils ont mis chacun pour l'épreuve. Par exemple si le 1er a terminé en `2h18mn20s` et le 2ème en `2h28mn30s` l'écart est de `0h10mn10s`.

Les deux temps  $T_1$  et  $T_2$  sont composés de trois variables pour les heures, minutes et secondes  $(h_1, m_1, s_1, h_2, m_2, s_2)$ . On supposera pour l'instant que  $0 < T_1 < T_2 < 24h$  (c'est-à-dire qu'on peut se concentrer sur la résolution du problème sans avoir à vérifier que les valeurs des variables sont cohérentes, ce qui est quand même un avantage certain).

L'écart de temps est donné par la formule  $T_2 - T_1$ , mais la difficulté du problème vient du fait qu'il nous faut gérer les différentes unités de temps séparément. On peut procéder en soustrayant les composantes respectives de chaque temps, par exemple :

$$\begin{array}{r} 02 \ 28 \ 30 \\ - \ 02 \ 18 \ 20 \\ \hline = \ 00 \ 10 \ 10 \end{array}$$

Bien sûr, il est des cas plus difficiles ou il faut gérer une/des retenue(s) et convertir les valeurs obtenues :

$$\begin{array}{r} 04 \ 11 \ 10 \\ - \ 03^{+1} \ 11^{+1} \ 30 \\ \hline = \ 00 \ 59 \ 40 \end{array}$$

(ici  $10 - 30 = -20$ , ce qui donne 40s quand on complète à 60 secondes, et une retenue pour les minutes).

<pre> vide fonction EcartTemps(entiers <math>h_1, m_1, s_1, h_2, m_2, s_2, h_r, m_r, s_r</math>)   paramètres E : entiers <math>h_1, m_1, s_1, h_2, m_2, s_2</math>   paramètres S : entiers <math>h_r, m_r, s_r</math>   valeur renvoyée : aucune </pre>	
<pre> entier <math>r</math> : une retenue  entier <math>diff</math> : différence entre les temps 1 et 2 sur une unité (h,m ou s). </pre>	<pre> <math>diff \leftarrow s_2 - s_1</math> si <math>diff \geq 0</math> alors   <math>s_r \leftarrow diff</math>   <math>r \leftarrow 0</math> sinon   <math>s_r \leftarrow 60 - diff</math>   <math>r \leftarrow 1</math> fsi  <math>diff \leftarrow m_2 - (m_1 + r)</math> si <math>diff \geq 0</math> alors   <math>m_r \leftarrow diff</math>   <math>r \leftarrow 0</math> sinon   <math>m_r \leftarrow 60 - diff</math>   <math>r \leftarrow 1</math> fsi  <math>h_r \leftarrow h_2 - (h_1 + r)</math> </pre>

**Exercice :** Compléter la fonction ci-dessus pour qu'elle puisse prendre en compte tous les cas (i.e., on enlève l'hypothèse que  $0 < T_1 < T_2 < 24h$ ), et délivre des messages d'erreurs appropriés si besoin est.

### 3.5 Modularité

Les fonctions sont un outil indispensable dès que l'on veut écrire un programme pour résoudre des problèmes relativement compliqués : lors d'une phase d'analyse et de formalisation, le problème est décomposé en sous-problèmes, qui peuvent eux-même selon leur complexité être décomposés en sous-problèmes, etc. Chaque sous-problème final correspondra dans le programme à une *fonction* spécifique qui peut être écrite et testée de façon indépendante du reste du programme. Les différentes fonctions, une fois correctement spécifiées, peuvent même être écrites par des personnes différentes.

## Cours 4

# Tableaux - Récursivité

### 4.1 Motivation

Il arrive que durant un traitement on ait besoin de conserver en mémoire un nombre "important" de valeurs de même type et jouant un rôle similaire : les dix notes d'un étudiant, la température moyenne de chaque jour de l'année, etc.

Utiliser une succession de variables simples, en les nommant de façon similaire, par exemple  $n_1, n_2, n_3, n_4, \dots, n_{10}$ , est une solution vraiment inconmode :

- on effectue souvent les mêmes instructions sur ces variables et il faudra donc recopier  $x$  fois les mêmes instructions en changeant seulement le nom de la variable d'une fois sur l'autre ( $n_1$ , puis  $n_2$ , etc).
- si après avoir écrit l'algorithme on se rend compte qu'il nous faut une variable de plus dans cette série, il faut modifier le code à tous les endroits où cette série de variables intervient.
- si le nombre de ces variables n'est pas connu à l'avance, c'est encore plus problématique.

### 4.2 Définition et syntaxe

Pour pallier les inconvénients énumérés ci-dessus, la notion de **tableau** a été introduite. Un tableau est un ensemble de variables de même type qui sont regroupées sous un même nom (celui du tableau). La déclaration d'un tableau précise :

- le type des variables
- le nom du tableau
- le nombre de variables qu'il regroupe

**Syntaxe.**

```
Tableau de 10 entiers    Table
```

**Effet.**

un ensemble de cases mémoires contiguës sont réservées en mémoire et désignées sous le nom de **Table**. La numérotation des cases commence à 0.

Table									
?	?	?	?	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

FIG. 4.1 – Cases mémoires de la variable **Table**.



*Comme dans le cas de variables de types simples, les cases mémoires obtenues contiennent une valeur quelconque au départ. Il faut les initialiser avant de s'en servir, sinon le résultat n'est pas garanti.*

On peut accéder à chaque case (cellule) du tableau indépendamment, par l'intermédiaire de son indice, que ce soit pour y mettre une valeur ou pour connaître la valeur stockée.

Dans certains langages, on pourra se permettre une déclaration de tableau avec une variable  $n$  représentant le nombre de cases. Dans ces langages, la variable doit être initialisée avant la déclaration du tableau.

Tableau de  $n$  entiers    **Table**

**Syntaxe.**

Table[2] ← 0  
ou  
A ← Table[2]

**Effet.**

Dans le 1er cas, la valeur 0 est mise dans la case d'indice 2 de **Table**. Dans le 2ème cas, la variable **A** reçoit la valeur contenue dans la case d'indice 2 de **Table**.

Table									
?	?	0	?	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

FIG. 4.2 – `Table[2] ← 0` : écriture d'une valeur dans une case mémoire du tableau.

L'index désignant la case qui nous intéresse peut lui-même être contenu dans une variable :

```
Entier i
Tableau de 100 entiers Table
i ← 2
Table[i] ← 0
ce code est équivalent au précédent.
```

### 4.2.1 Fonctions classiques sur les tableaux

La fonction `taille(...)` : renvoie le nombre de cases contenues dans le tableau indiqué en paramètre.

Dans la plupart des langages, les opérateurs arithmétiques ne sont pas définis sur les tableaux (ils doivent être appliqués case par case).

## 4.3 Exemples d'utilisation

### 4.3.1 Initialisation d'un tableau

L'*initialisation* désigne l'ensemble d'instructions qui permettent d'affecter une valeur initiale à un tableau (ou à une variable simple). L'algorithme suivant montre comment initialiser à zéro les cases d'un tableau. Ne pas oublier que ces cases contiennent initialement une valeur quelconque.

Algorithme ExempleInitialisation But : exemple de mise à zéro des cases d'un tableau		
Tableau de 10 entiers $T$ entier $i$ : un indice pour désigner successivement les cases de $T$	pour ( $i \leftarrow 0; i < 10; i \leftarrow i + 1$ ) faire $T[i] \leftarrow 0$ fpour	

### 4.3.2 Inversion d'un tableau

*Problème à résoudre* : on dispose d'un tableau dans lequel on veut inverser totalement l'ordre des valeurs contenues dans le tableau, c'est-à-dire, mettre la première valeur dans la dernière case, la deuxième dans l'avant-dernière case, ..., la dernière valeur dans la 1ère case.

Une méthode consiste à permuter les valeurs de la 1ère et de la dernière case, de la 2ème et de l'avant-dernière case, et ainsi de suite jusqu'aux deux cases centrales. On peut donc utiliser la fonction `Permute(A,B)` vue précédemment.

Algorithme InversionTableau But : Inverser l'ordre des valeurs d'un tableau		
Tableau de 6 entiers $T$ entier $l$ : longueur du tableau. entier $g$ : indice de la première case à permuter. entier $d$ : indice de la dernière case à permuter. entier $i$ : compteur avançant tant qu'on a des cases à permuter.	$l \leftarrow \text{Taille}(T)$ $g \leftarrow 0$ $d \leftarrow l - 1$ pour ( $i \leftarrow 1; i \leq l/2; i \leftarrow i + 1$ ) faire Permute( $T[g], T[d]$ ) $g \leftarrow g + 1$ $d \leftarrow d - 1$ fpour	<b>Permute(A,B)</b> : échange entre elles les valeurs de A et B.

**Note** : le passage en paramètres de deux cases (!) du tableau illustre le fait que chaque case est une variable et peut aussi "mener sa propre vie" indépendamment du reste du tableau.

En **traçant le programme**, on peut vérifier que le code est exact et fonctionne même si la taille du tableau est impaire (cas où la case centrale est la seule à ne pas changer).

### 4.3.3 Tri d'un tableau

*Problème à résoudre* : on dispose d'un tableau  $T$  de valeurs réelles que l'on désire trier par ordre croissant. Comme il s'agit d'une manipulation dont on a souvent besoin, on optera pour la réalisation d'une fonction nommée par exemple  $\text{Tri}(T)$ .

On peut envisager plusieurs méthodes pour réaliser cette fonction. Celle que nous choisissons de détailler ici considère le tableau comme scindé en deux parties : *une partie gauche* contenant les cases déjà triées (c'est-à-dire possédant déjà leur valeur finale) et une *partie droite* (le reste du tableau) dont les cases contiennent les valeurs non encore triées. Initialement, la partie triée du tableau finit avant la première case (contient 0 cases) et la partie non-triée commence à la première case (contient toutes les cases).

La méthode consiste à étendre de façon répétée la partie triée et à diminuer conjointement la partie non triée. Pour ceci on utilise la manipulation suivante : on recherche la plus petite valeur contenue dans une case de la partie non triée du tableau, soit  $c$  l'indice de cette case. On permute la valeur en  $T[c]$  avec celle de la 1ère case de la partie non triée du tableau, ce qui permet d'étendre la partie triée d'une case tout en conservant la valeur de cette case pour la suite du tri. Cette manipulation est répétée jusqu'à ce que la partie triée corresponde à l'ensemble du tableau.

Pour l'écriture de la fonction on procédera par étapes :

1. Essayer la méthode sur un exemple pour se convaincre qu'elle marche bien et s'assurer qu'on en comprend correctement le principe.
2. Ecrire l'entête de la fonction.
3. Essayer d'énumérer a priori les variables dont on va avoir besoin pour écrire la méthode.
4. Ecrire progressivement les instructions de la fonction, en ne s'attachant pas forcément à produire les instructions dans leur ordre d'apparition final<sup>1</sup>. Ne pas oublier d'ajouter dans la colonne de gauche la déclaration des nouvelles variables dont le besoin peut apparaître au fur et à mesure que vous écrivez les instructions.
5. Tracer votre programme sur l'exemple initial pour vérifier qu'il fonctionne comme il faut, et pour bien faire sur un autre exemple, si possible très différent du premier. Rectifier le programme en cas d'erreur à l'exécution de ces traces.
6. Admirer le résultat!

**Note** : dans le cas du tri, on a régulièrement besoin de trouver la case contenant la plus petite valeur dans une partie du tableau. On utilisera pour ce faire une fonction  $\text{ChercheMin}(T, g, d)$  qui recherche cette valeur entre les indices  $g$  et  $d$  du tableau  $T$ . Cette fonction sera vue en TD.

<pre> vide fonction Tri(Tableau d'entiers T)   paramètre E/S :   T, un tableau de valeurs que l'on ordonne par ordre croissant.   valeur retournée : aucune         </pre>		
<pre> entier l : longueur du tableau T. entier f : indice de la 1ère case de la partie non-triée (frontière). entier c : indice de la case à permuter avec la première case de la partie non triée         </pre>	<pre> l ← Taille(T) pour (f ← 0; f &lt; l - 1; f ← f + 1) faire   c ← ChercheMin(T, f, l - 1)   Permute(T[f], T[c]) fpour         </pre>	<pre> ChercheMin(T, g, d) : renvoie l'indice de la (d'une) case de T entre les indices g et d (in- clus) qui contient la plus petite valeur. Permute(A, B) : ...         </pre>

Notes sur l'écriture de la fonction :

<sup>1</sup>l'écriture d'un algorithme ou d'une fonction est rarement un processus linéaire.

- on voit comment le découpage des instructions en fonctions permet une compréhension rapide du code, où le principe de tri défini plus haut se retrouve traduit presque directement.
- quand un paramètre correspond à un tableau, la plupart des langages de programmation n'exigent pas qu'on indique une taille précise, ce qui permet d'utiliser la fonction sur des tableaux de différentes tailles.
- il peut arriver que la case  $c$  soit la première de la partie non triée du tableau et que l'on demande donc la permutation d'une variable avec elle-même ("ça a-t-il un sens?"). Il est donc nécessaire de s'assurer du comportement de la fonction `Permute(A,B)` dans un tel cas et de rajouter une instruction de test (`si...fsi`) pour éviter cette situation. *La plupart des erreurs de programmation concernant les tableaux sont dues à l'utilisation d'indices dépassant les bornes du tableau référencé.* Pour la même raison, attention à la condition d'arrêt de la boucle : exécuter la fonction `ChercheMin(T,g,d)` une fois de trop pourrait conduire à un résultat hasardeux (car on aurait  $g > d$ ).

## 4.4 Tableaux à plusieurs dimensions

Nous avons utilisé précédemment des tableaux à une seule "dimension", c'est-à-dire que leurs cases étaient repérées par un seul indice. Il est parfois utile de disposer de tableaux à plus d'une dimension (fournis dans la plupart des langages de programmation), pour représenter des concepts plus compliqués qu'une suite de valeurs, par exemple une matrice, un labyrinthe, etc.

On a toujours la contrainte selon laquelle toutes les cases doivent être de même type. Toutefois, on peut avoir un nombre de colonnes différent du nombre de lignes.

### 4.4.1 Déclaration

Exemple :

#### Syntaxe.

Tableau de $10 \times 5$ entiers M
------------------------------------

#### Effet.

Un ensemble de cases mémoires est réservé, et référencé par le nom M. Les cases sont individuellement accessibles au moyen de deux indices. La numérotation commence à l'indice 0.

**Remarque** : il faut prendre une convention pour les dimensions (ex : la première indique les lignes, la 2ème les colonnes) et s'y tenir.

Schématiquement on représente la déclaration d'un tel tableau en mémoire par une table d'autant de lignes / colonnes que demandées :

**Accès à une case** :  $M[1, 5] \leftarrow 10$ .

On utilise bien sûr la même convention que lors de la déclaration pour savoir quel indice désigne les lignes et lequel désigne les colonnes.

	0	1	2	3	4	5	6	7	8	9
0	?	?	?	?	?	?	?	?	?	?
1	?	?	?	?	?	?	?	?	?	?
2	?	?	?	?	?	?	?	?	?	?
3	?	?	?	?	?	?	?	?	?	?
4	?	?	?	?	?	?	?	?	?	?

FIG. 4.3 – Cases mémoires associées au tableau M.

#### 4.4.2 Exemple 1 : initialisation à zéro d'un tableau à deux dimensions

Algorithme ExempleInitialisation2 But : exemple de mise à zéro des cases d'un tableau à plusieurs dimensions	
Tableau de $10 \times 5$ entiers $M$ entier $i$ : indice de la ligne. entier $j$ : indice de la colonne.	<pre> pour (<math>i \leftarrow 0; i &lt; 10; i \leftarrow i + 1</math>) faire   pour (<math>j \leftarrow 0; j &lt; 5; j \leftarrow j + 1</math>) faire     <math>M[i, j] \leftarrow 0</math>   fpour fpour </pre>

#### 4.4.3 Exemple 2 : carré "faiblement magique"

Ecrire une fonction qui prend comme donnée un tableau carré de  $n \times n$  cases et renvoie **vrai** si la somme des cases de chaque colonne et chaque ligne est identique (à un certain nombre  $x$ ), et renvoie **faux** sinon.

**Exercice** : carré (complètement) magique : compléter la fonction précédente pour que le carré soit déclaré magique si la somme des cases de chacune des deux diagonales principales donne aussi  $x$ .

**Exercice** : réécrire la fonction `EcartTemps(...)` si les 3 composantes d'un temps sont placées dans un tableau `Temps`.

### 4.5 Fonctions récursives

#### 4.5.1 Définition

Une **fonction récursive** est une fonction s'appelant elle-même. A l'exécution, ça se passe de la même façon que si on appelait n'importe quelle autre fonction : imbrication de boîtes, passage des paramètres, etc (cf paragraphe 3.3).

#### 4.5.2 Exemple : la fonction *factorielle*( $n$ )

Rappel :

$$factorielle(n) = 1 \times 2 \times \dots \times n.$$

Suivant l'idée

$$fact(n) = n \times fact(n - 1),$$

on peut écrire cette fonction ainsi :

entier fonction Factorielle (entier $n$ )		
paramètre E : $n$ (un nombre)		
valeur retournée : entier (factorielle du nombre $n$ )		
entier $r$	si ( $n \leq 1$ ) alors retourner 1 sinon $r \leftarrow n \times fact(n - 1)$ retourner $r$ fsi	$fact(n)$ la même fonction!!!

On peut tracer l'appel  $fact(4)$  pour se convaincre que cela fonctionne. A l'exécution la boîte de  $fact(4)$  contient celle de  $fact(3)$  qui contient celle de  $fact(2)$  qui contient celle de  $fact(1)$ !!! C'est comme des poupées russes!

**Notes :**

- Une fonction récursive est principalement composée de **deux parties importantes** : la condition d'arrêt de la récursion, permettant de stopper le processus des appels récursifs, et la description du calcul réalisé par l'appel courant dépendamment ou pas du résultat de l'appel récursif suivant.
- **L'erreur la plus fréquente** concernant une fonction récursive consiste à faire s'enchaîner indéfiniment les appels lors de l'exécution, le **programme ne terminant alors jamais**. Une telle erreur vient très souvent du fait qu'on oublie de faire progresser la valeur de la variable passée en paramètre, ou du fait que le test d'arrêt est mal conçu.
- La plupart des fonctions récursives simples peuvent aussi s'écrire de façon non-récursive en remplaçant un appel récursif par une boucle et en ajoutant une ou plusieurs variables. L'exemple est donné ci-dessous pour la fonction  $fact(n)$ . L'inverse est aussi vrai.

entier fonction Factorielle (entier $n$ )		
paramètre E : $n$ (un nombre)		
valeur retournée : entier (factorielle du nombre $n$ )		
entier $r$ entier $i$	$r \leftarrow 1$ pour ( $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ ) faire $r \leftarrow r \times i$ fpour retourner $r$	

**Exercice :** inversement, écrire de façon récursive la fonction  $Tri(T)$  vue précédemment.