Algorithmique (Al Khârezmi)

Un algorithme sert à transmettre de manière intelligible les étapes nécessaires à la réalisation d'un travail.
Il permet d'expliquer clairement les idées de solution à un problème indépendamment d'un langage informatique.
Il permet au développeur de suivre les étapes définies, dans l'ordre défini pour parvenir au résultat escompté.

Déroulé

Introduction : les fondamentaux de l'algorithmique

Variables et constantes

Les types simples

Les types structurés

Actions élémentaires : Affectations, Entrées / Sorties

Structures de contrôles conditionnels : SI SINON…

Structures de contrôles itératives : les boucles

Procédures et Fonctions

Principes de base

Un algorithme est un compromis entre « langage naturel » et « langage informatique », présentant les étapes nécessaires à la résolution d'un problème clairement exposé.

Dans son formalisme, un algorithme présente une en-tête :

- **Nom : titre de l'algorithme (par exemple Tri de données),**
- **Rôle : Définition générale de l'algorithme, but et objectif,**
- **Données : les données connues de l'algorithme,**
- **Résultat : ce que l'algorithme doit produire en résultat,**
- **[Principe] : Principe(s) utilisé(s) dans l'algorithme**

Et… Un corps, l'algorithme lui-même, l'ensemble d'opérations fini nécessaire à l'obtention du résultat énoncé. Le corps est délimité par les mots réservés :

```
Début
…
Fin
```

Exemple : Addition

```
Algorithme : AdditionEntiers
Rôle : Additionner deux entiers et conserver le résultat
Données : deux entiers connus
Résultat : Stockage de la somme des deux valeurs entières
Début
VAR resultat : ENTIER
resultat ← 5 + 3
Fin
```

Note : le signe ← correspond à l'opérateur d'affectation

Dans cet exemple, on utilise deux valeurs définies (5 et 3) pour réaliser l'opération.

Les Variables

Une variable est une « étiquette» destinée à identifier un espace dans la mémoire de la machine. Dans cet espace, on pourra y stocker une information “variable” dans le temps.

On la définit avec :

Le mot réservé : VAR

Un nom (identifiant de la variable).

```
resultatAddition : ENTIER [← 0]
```

Un type : l’ensemble des valeurs acceptées par la variable définie (Entier, Réel, Chaine, Caractère, Booléen, …)

Et, optionnellement, une valeur : une valeur qui sera stockée dans la mémoire du dispositif exécutant le code final

Les Variables : Use Case

On définit une variable lorsque notre algorithme impose de conserver ET modifier une information tout au long des diverses opérations.

L'espace mémoire qui est alloué à cette variable est donc accessible aussi bien :

- **En lecture**
- **En écriture**

```
VAR ma_variable: ENTIER // Définition de l'espace en mémoire

ma_variable <- 5 // Affectation de la valeur 5 à la variable

...

ma_variable <- ma_variable * 2 // Opération sur la variable
```

A l'issue de l'algorithme, l'espace mémoire défini par "ma_variable" vaudra donc 10 (5 x 2)

Les constantes

Une constante est aussi un espace en mémoire, mais à la différence d'une variable, la valeur d'une constante est définie au moment de sa déclaration et ne peut être modifiée en cours de programme.

Elle n'est donc accessible qu'en :

- **En lecture**

```
CONST MA_CONSTANTE: ENTIER <- 3.14116 // PI

Vous ne pouvez plus modifier la valeur de
MA_CONSTANTE.
```

Les constantes

Il existe “nativement” des constantes en algorithmique :

PI qui représente la fameuse valeur utilisée en géométrie

Vous pouvez utiliser la constante PI native dans un algorithme.

On définit donc une constante lorsque nous voulons nous assurer que la valeur de cette constante ne puisse pas être “mutée” durant l’exécution de notre algorithme et ainsi renforcer la cohérence durant l’exécution.

Typage

Lorsqu'on définit une variable ou une constante, on doit impérativement lui associer un TYPE.

Le type est la "famille" des valeurs que peut accepter cette variable.

Par exemple, si vous définissez la variable suivante :

```
ma_variable: Entier
```

L'opération suivante sera illégale :

```
ma_variable <- "James Bond"
```

En effet, le type "James Bond", chaîne de caractère est incompatible avec le type de la variable qui n'accepte de stocker que des Entiers.

Les types alphanumériques

Type	Déclaration	Use Case
Chaine de caractères	VAR chaine: CHAINE	Toutes valeurs alphanumériques, sans limite de longueur
	VAR chaine[25]	Toutes valeurs alphanumériques limitées à 25 caractères
Caractère	VAR char: CARACTERE	La valeur doit être un caractère alphanumérique de longueur 1

Note : certaines fonctions “internes” et opérations sont applicables à ces types algorithmique :
LEN(chaine) : retournera la longueur de la chaîne de caractère
SUCC(char) : renverra le caractère suivant le caractère passé en paramètre
PRED(char): renverra le caractère précédent le caractère passé en paramètre

Les types numériques

Type	Déclaration	Use Case
Nombres entiers	VAR nombre: ENTIER	Toute valeur numérique signée ou non sans décimale
Réels	VAR char: REEL	Toute valeur numérique signée ou non avec décimales

Note : certaines “opérations” sont applicables aux types numériques :

- **Addition (+), soustraction (-), produit (*), division (/),**
- **Division entière : DIV pour récupérer la partie entière d’une division,**
- **Modulo : MOD récupérant le “reste” d’une division entière**
- **Les opérateurs de comparaison : “=”, “>”, “<”, “>=”, “<=”, “<> ou !=”**

Les types logiques

Type	Déclaration	Use Case
Valeurs vrai ou faux	VAR existe: BOOLEEN	Toute valeur pouvant être vraie ou fausse

Note : les valeurs booléennes sont soumises à une algèbre particulière, mais peuvent utiliser les opérateurs de comparaison courants (=, >, <, >=, <=, <> ou !=) ainsi que des opérateurs spécifiques à ce type logique :

- NON,
- OU,
- ET,
- XOU

Les types logiques : tables de vérité

A	B	A ET B	A OU B
FAUX	FAUX	FAUX	FAUX
FAUX	VRAI	FAUX	VRAI
VRAI	FAUX	FAUX	VRAI
VRAI	VRAI	VRAI	VRAI

A	NON A
FAUX	VRAI
VRAI	FAUX

A	B	A XOU B
FAUX	FAUX	FAUX
FAUX	VRAI	VRAI
VRAI	FAUX	VRAI
VRAI	VRAI	FAUX

A vous de jouer…

Vous allez utiliser vos connaissances en définition des variables et constantes en algorithme, pour traiter le problème du rendu de monnaie.
A partir d'un prix donné en entier (par exemple 75 €)
A partir d'un somme versée en entier (par exemple 100 €)

Vous allez écrire l'algorithme qui définit le rendu monnaie en répartissant le rendu en :
Billet de 100,
Billet de 50,
Billet de 20,
Billet de 10,
Billet de 5,
Pièce de 2 €

```
L'attendu pour cet exercice est le suivant :
Billet100 => 0
Billet50 => 0
Billet20 => 1
Billet10 => 0
Billet5 => 1
Piece2 => 0
```

Solution simple

```
# Exercice Rendu Monnaie
### Algorithme : Rendu Monnaie
### Rôle : Ventilation d'un rendu de monnaie en espèce par type de billet et / ou pièce
### Résultat : Obtenir le nombre de billets de 100, de 50, de 20, de 10, de 5 et le
nombre de pièces de 2 euros
DEBUT
CONST PRIX_PRODUIT: ENTIER <- 75
CONST SOMME_VERSEE: ENTIER <- 100
CONST A_RENDRE: ENTIER <- SOMME_VERSEE - PRIX_PRODUIT
VAR billet100, reste100, billet50, reste50, billet20, reste20, billet1
billet5, reste5, piece2, reste2: ENTIER <- 0
```

```
// Traitement pour les billets de 100
billet100 <- A_RENDRE DIV 100 // Dans ce cas 0
reste100 <- A_RENDRE MOD 100 // Dans ce cas 25 / 100 => 0.25 donc 25
// Traitement pour les billets de 50
billet50 <- reste100 DIV 50 // Dans ce cas 0
reste50 <- reste100 MOD 50 // Dans ce cas 25 / 50 => 25
// Traitement pour les billets de 20
billet20 <- reste50 DIV 20 // Dans ce cas 25 / 20 => 1
reste20 <- reste50 MOD 20 // Dans ce cas 5 / 20 => 5
// Traitement pour les billets de 10
billet10 <- reste20 DIV 10 // Dans ce cas 5 / 10 => 0
reste10 <- reste20 MOD 10 // Dans ce cas 5
// Traitement pour les billets de 5
billet5 <- reste10 DIV 5 // Dans ce cas 1
reste5 <- reste10 MOD 5 // Dans ce cas 0
// Traite les pièces de 2
piece2 <- reste5 DIV 2 // Dans ce cas 0
reste2 <- reste5 MOD 2 // Dans ce cas 0
FIN
```

Le code est visible à l'adresse :
https://github.com/dacodemaniak/algortithmique

Types complexes

On arrive très vite à saturation lorsqu'on utilise des types simples (chaine, caractère, booléen, nombre). En effet, si on souhaite par exemple calculer une moyenne, il faudrait déclarer "n" variables de type "réel", puis réaliser une addition de toutes ces variables et finir par diviser par le nombre de notes pour obtenir cette fameuse moyenne.
Le travail serait long et fastidieux, et générateur d'erreurs.
Les types complexes vont apporter une réponse efficace à cette problématique.

Le type Tableau

Un tableau est un ensemble de données de même type (y compris Tableau aussi).

Un tableau peut être un ensemble “fini” dont on a donc défini les limites dès la déclaration ou non fini, dans ce cas, il devient un ensemble dynamique.

```
VAR tableau: TABLEAU[] DE CHAINE
VAR num_tableau: TABLEAU[0..4] DE ENTIER
VAR tablo2tablo: TABLEAU[] DE TABLEAU[] DE REEL
```

La première déclaration défini un tableau non fini qui ne pourra contenir que des chaînes de caractères,
La seconde déclaration définit un tableau de… 5 éléments en effet, l’indice de départ étant à 0, on va dénombrer 0, 1, 2, 3, 4 indices, donc… 5 éléments.
La dernière déclaration définit donc un tableau, qui contiendra un tableau à chaque indice !

Tableau : Ajouter des données

Pour écrire des données dans un tableau, il suffit de faire suivre la variable (ou la constante) tableau de crochets vides.

```
VAR tableau: TABLEAU[] DE CHAINE
// Ecrire les données dans le tableau
tableau[] <- “Jean-Luc”
tableau[] <- “Elsa”
tableau[] <- “Max”
```

Après exécution, le tableau aura donc la forme suivante dans la mémoire de votre machine :

Indice	Valeur
0	Jean-Luc
1	Elsa
2	Max

Tableau : Lire les données

Après avoir alimenté de manière contigüe les données, vous pouvez lire une information en passant entre crochets la valeur de l'indice souhaité dans le tableau

Indice	Valeur
0	Jean-Luc
1	Elsa
2	Max

```
…
VAR secondName: CHAINE <- tableau[1]
```

Tableau : Remplacer une donnée

Pour remplacer une donnée, il suffit juste de redéfinir la valeur à l'indice concerné

```
…
tableau[0]<- "Achraf"
```

Indice	Valeur
0	Achraf
1	Elsa
2	Max

On peut utiliser une pseudo fonction "LEN(tableau)" pour récupérer le nombre d'éléments de ce tableau.

Opérandes, opérateurs et expressions

L'algorithmique met en oeuvre un certain nombre d'opérations (calcul) pour arriver à un résultat.
Les opérandes (variables ou constantes) font l'objet d'opérations (comparaison, addition, soustraction, produit ou division) ainsi que d'opérations plus spécifiques (DIVision entière, MODulo reste de division), ou encore d'opérations logiques (NON, OU, ET, XOU…)

Opérateurs, opérandes et expressions

Un opérateur est un symbole d'opération permettant d'agir sur une variable (+, -, /, *, ET, OU, DIV, MOD, =, <, >, ≤, ≥, <>, …),

Une opérande est une « entité » (variable, constante ou expression) utilisée par un opérateur,

Une expression est une combinaison des deux éléments précédents, évaluée durant l'exécution de l'algorithme et possède une valeur ainsi qu'un type

Opérateurs, opérandes et expressions

```
VAR resultatAddition : Entier ← 0
VAR valeur1 : Entier ← 5
VAR valeur2 : Entier ← 3
resultatAddition ← valeur1 + valeur2
```

Expression : Entier

Opérande

Opérateur

Un opérateur est « unaire » s'il n'accepte qu'un seul opérande :
resultatTest ← NON isAuthenticate

Un opérateur est « binaire » s'il accepte deux opérandes

Opérateurs, opérandes et expressions

Expression Unaire : n'acceptant qu'un seul opérande

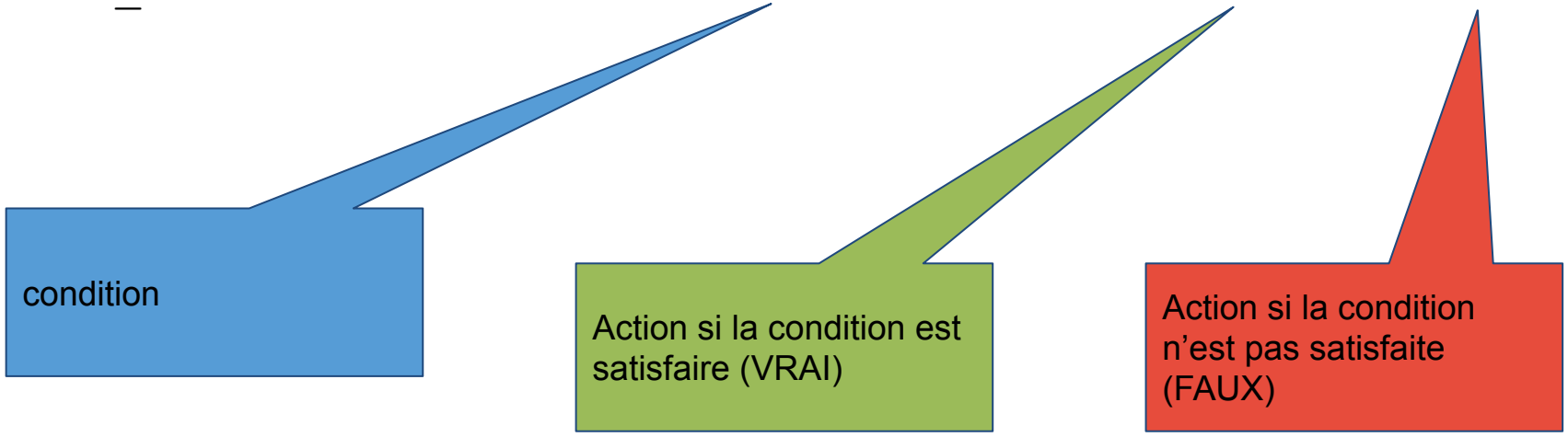
```
VAR authentification: BOOLEEN <- VRAI
authentification = NON authentification
```

Dans cet exemple, on parle d'expression "unaire" car elle ne met en oeuvre qu'un seul opérande

Opérateurs, opérandes et expressions

Expression “ternaire” avec trois opérandes

```
VAR authentification: BOOLEEN <- VRAI
VAR ouvert: CHAINE <- “Vous pouvez entrer”
VAR ferme: CHAINE <- “Désolé nous sommes fermés”
VAR acces_autorise: CHAINE <- authentification ? ouvert : ferme
```



Opérateurs : Mises en garde

Un opérateur ne peut s'appliquer sur des types différents :

```
VAR test : Booleen ← VRAI
VAR resultat ← « Bonjour » + test
```

Cependant, dans certains cas, cette règle peut être dérogée :

```
VAR prenom : Chaine ← « Marie »
VAR resultat ← « Bonjour » + chaine
```

Dans ce cas, l'opérateur « + » joue le rôle d'un opérateur de <u>concaténation</u>.

C'est le cas aussi si vous concaténez une chaîne et un nombre :

```
VAR resultat: CHAINE <- prenom + " " + 3.25
```

resultat vaudra : "Marie 3.25"

Entrées / Sorties

En général, un algorithme tient compte de données « entrantes » (saisies utilisateurs, lecture de données, etc.) et permet « d'écrire » vers le périphérique de sortie :

```
prenom : Chaine
prenom ← lire()
ecrire(« Bonjour »  + prenom)
```

Instructions conditionnelles

On doit souvent opter pour une série de traitements, en fonction de l'évaluation d'une expression :

Si l'orage gronde, alors, ne pas oublier de prendre son parapluie

L'algorithmique permet de traiter des conditions :

```
Si expression booléenne Alors
      ….
      ….
Sinon
      ….
      ….
FinSi
```

L'instruction SINON est optionnelle.

```
VAR data: Reel ← lire()
Si data < 0 Alors
     data ← data * -1
FinSi
ecrire(data)
```

Traitement de cas

Il est parfois nécessaire de comparer une valeur à plusieurs autres : Lors du paiement d'un « panier » de commande, on propose à l'utilisateur plusieurs modes de paiement => carte, virement, chèque, mandat, …

Ce problème peut s'exprimer facilement par l'instruction « Cas » :

```
Cas où modePaiement Valant
   'carte' : instruction
   'virement' : instruction
    'cheque' : instruction
    'mandat' : instruction
FinCas
```

OPÉRATEUR TERNAIRE

VAR a_var: CHAINE = (condition) ? ‘vrai’ : ‘faux’

Itérations ou boucles

Il arrive souvent qu'il soit nécessaire de traiter une série d'instructions plusieurs fois d'affilée, sur un ensemble fini :

Pour l'ensemble des 10 verres sur la table, les remplir

Ou bien sur un ensemble dont on ne connaît pas la limite :

Tant qu'il y a du soleil, je profite de la plage

Dans le premier cas, on parlera de répétitions inconditionnelles, et de répétitions conditionnelles dans l'autre.

Répétition Inconditionnelle

La boucle Pour permet de répéter une série d'instructions un nombre « connu » de fois :

```
Pour var: Entier Allant de valeurDepart à valeurFin Pas de inc
    ….
    ….
FinPour
```

Valeur d'itération,
Indice d'itération
ou Compteur

L'incrémentation de 1
est implicite.
Cet élément n'est pas
nécessaire

Boucles Conditionnelles

Une boucle conditionnelle exécutera les instructions tant qu'une condition est satisfaite, ou jusqu'à ce qu'une condition soit satisfaite.
Dans le premier cas « tant que », les instructions peuvent ne pas être exécutée, si la condition en amont n'est pas satisfaite.

Dans le second cas, les instructions seront exécutées au moins une fois.

Boucles Conditionnelles : Tant que

La boucle « Tant que » permet d'exécuter une instruction tant qu'une condition est satisfaite. Si vous devez rendre la monnaie sur un montant donné et une somme fournie :

```
montant : Entier ← 15
somme : Entier ← lire()
aRendre : Entier ← somme - montant
Tant que aRendre > 0
   aRendre ← aRendre – 1
FinTantQue
```

Une boucle Pour

```
tablo : Tableau ← [125, 201, 302, 5, 15, 12]
POUR indice : Entier de 0 à 5 inc 1
    afficher(tablo[indice] * 2)
FIN POUR
```

indice varie de 0 à 5 car
il y a 6 éléments dans
le tableau, mais
le premier indice vaut 0 !

Cette boucle va parcourir un à un chacun des éléments du tableau et afficher la valeur multipliée par deux à chaque occurrence :

250
402,
604,
10,
30,
24

Une boucle Tant Que

```
tablo : Tableau ← [125, 201, 302, 5, 15, 12]
indice : Entier ← 0
TANT QUE indice < 5
    afficher(tablo[indice] * 2)
     indice ← indice + 1
FIN TANT QUE
```

Le résultat de cet algorithme est le même que précédemment. A noter cependant quand dans ce cas :
- Si vous oubliez d'initialiser l'indice à une valeur inférieure au nombre d'éléments du tableau, la boucle peut ne pas être exécutée,
- Que vous devez incrémenter vous même la valeur de l'indice

Une boucle Répète… Jusqu'à…

```
tablo : Tableau ← [125, 201, 302, 5, 15, 12]
indice : Entier ← 0
REPETE
    afficher(tablo[indice] * 2)
     indice ← indice + 1
JUSQU'A indice = 6
```

Une nouvelle fois, le résultat sera identique, pourtant :
- La boucle sera exécutée systématiquement au moins une fois,
- La condition de sortie est évaluée en fin de boucle, ce qui explique la valeur 6 au lieu de 5 en fin d'algorithme

A l'exécution :
indice : 0 => affichage 250, indice : 1,
indice : 1 => affichage 402, indice : 2
indice : 2 => affichage 604, indice : 3,
indice : 3 => affichage 10, indice : 4,
indice : 4 => affichage 30, indice : 5,
indice : 5 => affichage 24, indice : 6 => Sortie de boucle

Boucles : avertissement

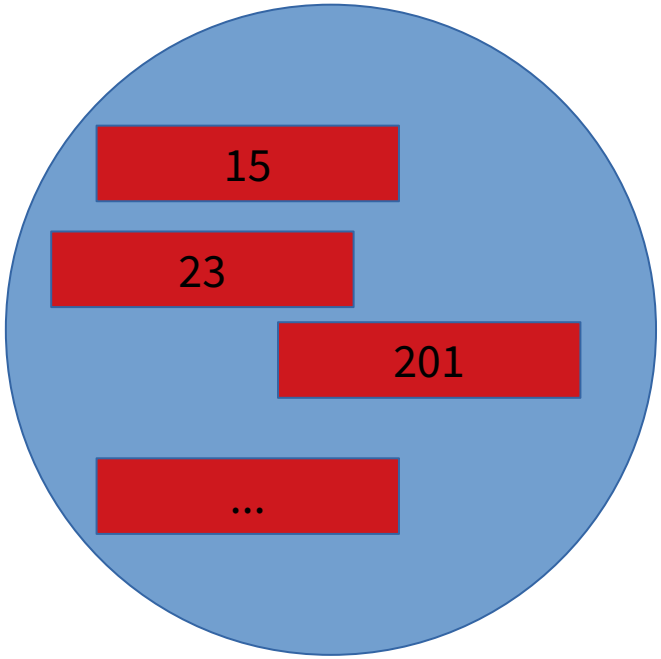
Des boucles mal formées peuvent conduire à des « dead loops », c'est à dire des boucles qui ne s'arrêtent jamais, conduisant à un plantage des applications.

```
POUR i : Entier VALANT 0 A 10 INC 1
        afficher('indice : ' + indice)
        indice ← indice – 1
FIN POUR
```

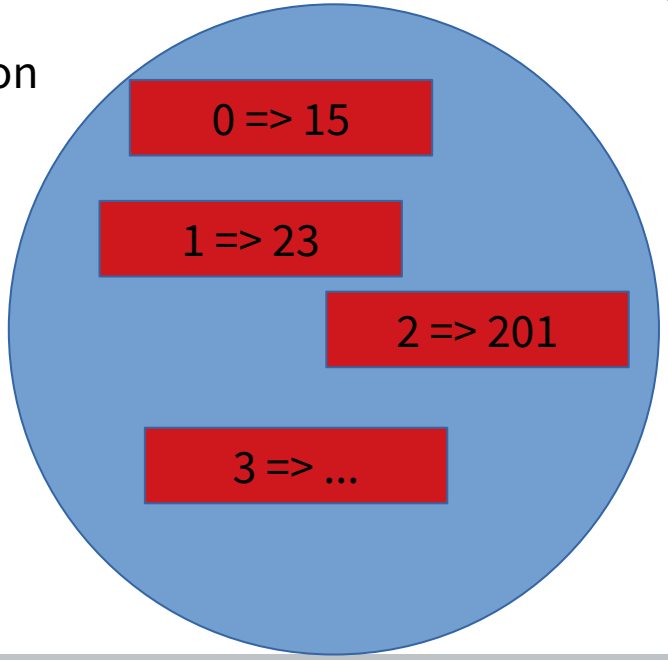
```
indice : Entier ← 0
TANT QUE indice < 5
        saisie : Chaine ← lire()
FIN TANT QUE
```

Les Tableaux

Les tableaux peuvent se concevoir comme des collections, ensembles d'informations.



Dans un tableau, les informations sont « indicées », en d'autres termes, chaque valeur de la collection est associée à une « étiquette » numérique qui donne sa position dans le tableau.



Un tableau se déclare en algorithmique de la manière suivante :
tablo : Tableau ← [15, 23, 201]

Les tableaux

L'accès aux éléments d'un tableau se fait par référence aux indices, l'indice de la valeur concernée est passé entre crochets :
uneValeur : Entier ← tablo[1]

0	1	2	3	4	5
15	3	20	18	1	6

Un tableau est considéré comme faisant partie de la famille des « itérables » . Un « itérable » est une variable sur laquelle il est possible de boucler :

```
unTablo : Tableau ← [15, 3, 20, 18, 1, 6]
POUR indice : Entier VALANT 0 A 5 INC 1
        afficher(unTablo[indice])
FIN POUR
```

Des fonctions « algorithmiques » aident à traiter les boucles :
longueur(unTablo)
retourne le nombre d'éléments de la variable « unTablo »

Les tableaux

A l'aide de la fonction « longueur(tableau) » il est possible de reprendre l'exemple précédent sous une autre forme :

```
unTablo : Tableau ← [15, 3, 20, 18, 1, 6]
POUR indice : Entier VALANT 0 A longueur(unTablo) - 1 INC 1
        afficher(unTablo[indice])
FIN POUR
```

Notez dans la boucle l'utilisation de :
longueur(unTablo) – 1
en effet, il y a 6 éléments dans le tableau
mais l'indice varie de 0 à 5 !

Les tableaux à plusieurs dimensions

Un tableau peut avoir plusieurs « dimensions », en d'autres termes, chaque élément d'un tableau peut lui-même être un tableau.

0	1	2	3	4	5
[1,2]	[3,4]	[5,6]	[7,8]	[9,10]	[11,12]

L'accès à un élément d'un tableau à plusieurs dimensions se fait de la manière suivante :
unTablo[1][0]
permettra d'accéder à la valeur « 4 » (premier indice du tableau présent à l'indice 1)

Pour pouvoir afficher le contenu de l'ensemble d'un tableau à « n » dimensions, il est donc nécessaire d'imbriquer « n » boucles :

```
unTablo : Tableau ← [[1,2],[3,4],[5,6], [7,8],[9,10], [11,12]]
POUR indice : Entier VALANT 0 A longueur(unTablo) - 1 INC 1
        POUR subIndice : Entier VALANT 0 A longueur(unTablo[indice]) – 1 INC 1
                afficher(unTablo[indice][subIndice])
        FIN POUR
FIN POUR
```

Les Fonctions

En algorithmique, on peut utiliser des fonctions « virtuelles » :
- **lire()** est une fonction capable de récupérer des informations venant du « monde extérieur » (saisie clavier, résultat d'une requête SQL, informations provenant d'une autre partie de code…),
- **ecrire(message : Chaine)** est une fonction capable de produire une « sortie » vers le « monde extérieur » (écran, imprimante, …) d'une information,
- **longueur(tableau : Tableau)** retourne le nombre d'éléments d'un tableau

Nous n'avons pas à savoir comment ces instructions fonctionnent mais juste à savoir ce qu'elles font. On parle souvent de « boîtes noires »

Ces fonctions n'ont aucune existence réelle. Elles sont juste là pour permettre de faciliter l'écriture des algorithmes.

Les Fonctions

Une fonction peut être considérée comme un « programme » autonome, dont le but est de traiter une série d'instructions et de retourner un résultat.

Imaginons que vous deviez réaliser un algorithme qui permet d'additionner les valeurs présentes dans deux tableaux distincts (de même dimension), et de ranger le résultat de l'addition dans un troisième tableau.

Une des solutions possibles serait la suivante :

```
firstValues : Tableau ← [5, 10, 15, 20]
secondValues : Tableau ← [3, 5, 7, 9]
results : Tableau
POUR indice : Entier VALANT 0 A longueur(firstValues) – 1 INC 1
        results[] ← firstValues[indice] + secondValues[indice]
FIN POUR
```

Après réflexion, il s'avère que vous devez non plus additionner mais multiplier, vous devriez revoir votre boucle pour changer l'opérateur… avec le risque de devoir à nouveau changer pour revenir à l'état précédent.

Pour éviter les risques

Les Fonctions

On peut donc écrire deux fonctions :
- L'une qui se chargera d'additionner deux valeurs et de retourner le résultat,
- L'autre qui se chargera de multiplier ces deux valeurs et de retourner le résultat.

FONCTION **additionner**(value1: Entier, value2: Entier) : Entier
    RETOURNE value1 + value2
FIN FONCTION

FONCTION muliplier(value1: Entier, value2: Entier) : Entier
    RETOURNE value1 * value2
FIN FONCTION

Ces deux fonctions « acceptent » en entrée deux « paramètres » (value1 et value2) qui seront à l'appel de la fonction, alimentés par les valeurs fournies.

Au final, il est donc possible de « refactorer » l'algorithme de la manière suivante :

```
firstValues : Tableau ← [5, 10, 15, 20]
secondValues : Tableau ← [3, 5, 7, 9]
results : Tableau
POUR indice : Entier VALANT 0 A longueur(firstValues) – 1 INC 1
    results[] ← additionner(firstValues[indice], secondValues[indice])
FIN POUR
```

```
firstValues : Tableau ← [5, 10, 15, 20]
secondValues : Tableau ← [3, 5, 7, 9]
results : Tableau
POUR indice : Entier VALANT 0 A longueur(firstValues) – 1 INC 1
    results[] ← multiplier(firstValues[indice], secondValues[indice])
FIN POUR
```

Les Fonctions

Que se passe-t-il dans cet algorithme ?

```
FONCTION additionner(value1, value2) : Entier
        RETOURNE value1 + value2
FIN FONCTION
```

```
firstValues : Tableau ← [5, 10, 15, 20]
secondValues : Tableau ← [3, 5, 7, 9]
results : Tableau
POUR indice : Entier VALANT 0 A longueur(firstValues) – 1 INC 1
        results[] ← additionner(firstValues[indice], secondValues[indice])
FIN POUR
```

Itération	Appel	Fonction
1 (indice ← 0)	additionner(5,3)	Value1 ← 5 value2 ← 3 Retourne 5 + 3
2 (indice ← 1)	additionner(10,5)	value1 ← 10 value2 ← 5 Retourne 10 + 5
…	…	…

Les Fonctions

Une fonction peut appeler une autre fonction.

```
FONCTION choixOperation(value1 : Entier, value2 :
Entier, isAddition : Booleen) : Entier
        SI isAddition = VRAI ALORS
                RETOURNE additionner(value1, value2)
        SINON
                RETOURNE multiplier(value1, value2)
        FIN SI
FIN FONCTION
```

```
FONCTION additionner(value1, value2) : Entier
        RETOURNE value1 + value2
FIN FONCTION

FONCTION muliplier(value1, value2) : Entier
        RETOURNE value1 * value2
FIN FONCTION
```

Le code peut donc à nouveau être refactoré :

```
firstValues : Tableau ← [5, 10, 15, 20]
secondValues : Tableau ← [3, 5, 7, 9]
results : Tableau
POUR indice : Entier VALANT 0 A longueur(firstValues) – 1 INC 1
        results[] ← choixOperation(firstValues[indice],
secondValues[indice], VRAI)
FIN POUR
```

```
firstValues : Tableau ← [5, 10, 15, 20]
secondValues : Tableau ← [3, 5, 7, 9]
results : Tableau
POUR indice : Entier VALANT 0 A longueur(firstValues) – 1 INC 1
        results[] ← choixOperation(firstValues[indice],
secondValues[indice], FAUX)
FIN POUR
```

Les Fonctions

Une fonction peut aussi s'appeler elle-même.
Une telle fonction est dite « récursive ».
Les fonctions « récursives » peuvent être risquées si les conditions de sortie sont mal évaluées, mais excessivement pratiques.
En reprenant le tableau multidimensionnel, on peut très bien « afficher » tout le contenu en une seule boucle.

```
FONCTION dump(array : Tableau)
        POUR indice : Entier VALANT 1 A longueur(array) – 1 INC 1
                SI array[indice] IS Tableau ALORS
                        dump(array[indice])
                SINON
                        afficher(array[indice])
                FIN SI
        FIN POUR
FIN FONCTION
```

0	1	2	3	4	5
[1,2]	[3,4]	[5,6]	[7,8]	[9,10]	[11,12]

unTablo : Tableau ← [[1,2],[3,4],[5,6], [7,8],[9,10], [11,12]]
dump(unTablo)

Le résultat de l'appel à la fonction « dump(unTablo) » donnera :
1
2
3
4
…
11
12

IS est utilisé ici pour déterminer le « type » de la donnée array[indice]