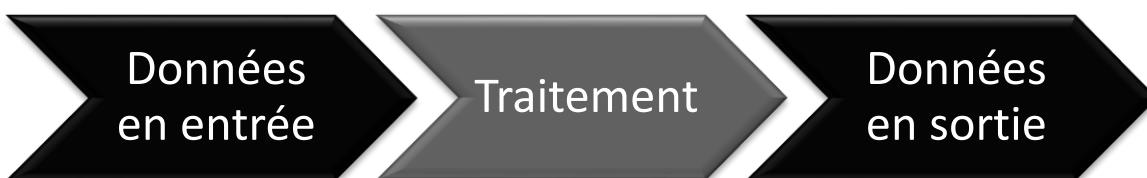


# INTRODUCTION A L'ALGORITHMIQUE

## I. Introduction

Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre une classe de problèmes. Le mot algorithme vient du nom d'un mathématicien perse du 9<sup>ème</sup> siècle, Al-Khwârizmî.

Un algorithme écrit dans un langage naturel ou en pseudo-code qui contient une description des opérations à effectuer pour résoudre le problème compréhensible par un être humain. Ainsi, un algorithme prend des données en entrée, exprime un traitement particulier et fournit des données en sortie.



Généralement, pour un problème donné il existe plusieurs solutions possibles. Le but ce n'est pas de trouvé une solution mais plutôt de trouver la solution optimale. Ainsi, un algorithme est dit bon s'il est :

- **Correct** : pour chaque instance en entrée, l'algorithme se termine en produisant la bonne sortie.
- **Efficace** : mesure de la durée que met un algorithme pour produire un résultat (temps d'exécution et espace mémoire utilisé)
- **Se termine** : se termine en un temps fini.

Un algorithme n'est pas compréhensible par un ordinateur, il faut le transformer ou le traduire en un programme écrit avec un langage de programmation. Un programme est la version de l'algorithme qui peut s'exécuter sur un ordinateur en séquence ou en parallèle.

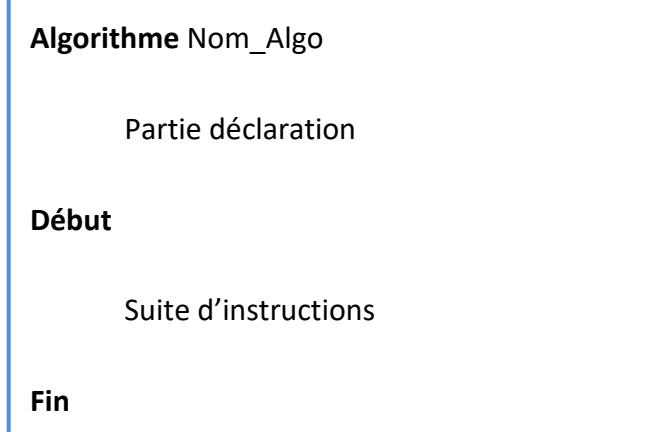
Les étapes de conception d'un programme informatique est comme suit :

1. Identifier le problème : quelle(s) donnée(s), quel(s) résultat(s) ?
2. Organiser les actions : écrire l'algorithme (pseudo-code, organigramme)
  - Réfléchir aux informations à manipuler
  - Analyser le problème et le décomposer éventuellement en sous-problèmes
  - Rendre l'algorithme compréhensible et efficace
  - Penser à l'utilisateur
3. Traduire cet algorithme en langage de programmation
4. Compiler le programme pour qu'il puisse être exécutable

Un langage de programmation permet à un humain d'écrire un code pouvant être analysé par une machine puis transformé en un programme informatique. Il existe beaucoup de langage de programmation. Généralement, le choix du langage dépend du problème à résoudre.

## II. Structure d'un algorithme

Il n'y a vraiment de règle pour l'écriture d'un algorithme. Tout simplement, il doit être écrit en langage naturel. Mais pour être sur la même longueur d'onde, nous allons suivre une certaine norme pour l'écriture d'un algorithme.



**Partie déclaration :** description de tous les objets utilisés dans l'algorithme, définition de variables, de constantes et de types.

**Instructions :** opérations sur les données (algorithme proprement dit), délimité par les mots début et fin. Les opérations sur les données utilisent des « instructions » : affectations, opérateurs arithmétiques ou de comparaison, structures composées (alternatives, répétitives). L'ensemble des instructions constituent ce qu'on appelle le « code ».

# ELEMENTS DE BASE DE L'ALGORITHMIQUE

## I. Structures de données

### 1. Variable et constante

Chaque programme reçoit en entrée des données qui va les manipuler et le traiter afin de produire des données en sortie. Ces données sont représentées dans un ordinateur sous forme d'objet. Un objet a besoin de trois qualificatifs pour sa définition :

- **Identificateur** : nom de l'objet qui doit être unique dans le programme, ne contenant pas d'espace et ne commence pas par un nombre.
- **Type** : qui indique la nature de l'objet (nombre, texte, booléen,...)
- **Valeur** : c'est la donnée à manipuler.

Un objet est en réalité un emplacement mémoire qui sert à stocker la donnée et qui peut être soit une variable ou soit une constante :

- **Une variable** : il est possible de changer son contenu le long du programme.
- **Une constante** : il a une valeur initiale fixée lors de sa déclaration et il n'est pas possible de changer son contenu le long du programme.

Tout objet que ce soit une variable ou une constante doit être déclaré généralement au début du programme. En algorithmique, pour déclarer un objet il faut :

Déclaration d'une variable

Nom\_variable : type\_variable

Exemple :

x : entier

y, z : réel

Déclaration d'une constante :

Const Nom\_constante= valeur

Exemple :

Const PI=3.14

Const MAX=100

## 2. Type de donnée

Un type de données définit le genre de contenu d'une donnée et les opérations pouvant être effectuées sur la variable correspondante. Chaque langage de programmation définit ses propres opérations mais généralement il y a des opérations standard en commun à tous les langages.

### a. Type entier

Une variable de type entier prend ses valeurs l'ensemble des nombres entiers. Les opérateurs qui, appliqués à des entiers, donnent un résultat entier tel que : addition (+), soustraction (-), multiplication (\*), division entière (/), reste de la division entière (MOD).

### b. Type réel

Une variable de type réel fait partie des nombres réels. Le résultat des opérateurs arithmétiques est un réel, si au moins l'un des opérandes est un réel (l'autre pouvant être entier).

### c. Type booléen

Une variable booléenne ou de type logique doit prendre pour valeur la constante TRUE (pour VRAI) ou la constante FALSE (pour FAUX). Ces valeurs pouvant être obtenues par l'évaluation d'une expression logique. Les booléens se représentent sur un seul bit, en général avec le codage « 0 » pour FAUX et « 1 » pour VRAI.

Les opérateurs booléens (ou logiques) sont :

- NOT (Non : Négation)
- AND (ET logique)
- OR (OU logique)

Les opérateurs relationnels ou de comparaison sont :

| Symbol | Description        |
|--------|--------------------|
| =      | Egale              |
| <>     | Different          |
| >      | Supérieur          |
| <      | Inférieur          |
| >=     | Supérieur ou égale |
| <=     | Inférieur ou égale |

#### d. Type caractère

Le type caractère est l'ensemble fini et ordonné de tous les signes qui peuvent être représentés par la machine :

- Lettres minuscules et lettres majuscules : a..z et A..Z
- Les chiffres : 0..9
- Les caractères spéciaux : ?, \*, /, ), ....

Un caractère est représenté sur 8 bits, ce qui donne 256 combinaisons possibles. Le code, couramment utilisé, est le code ASCII.

#### e. Type chaîne de caractères

Le type chaîne de caractères est l'ensemble des chaînes de caractères que l'on peut former. Une chaîne vide est une chaîne qui ne contient aucun caractère. Une chaîne peut contenir au maximum 256 caractères.

Chaque langage de programmation défini un ensemble de fonctions pour la gestion des chaînes de caractères. Par exemple :

- La fonction Longueur : donne le nombre de caractère dans une chaîne.
- La fonction Compare : qui vérifie si deux chaînes sont identiques ou non.
- La fonction Concaténer : qui fait la concaténation de deux chaînes.

## II. Opération sur les données

### 1. Affectation

L'affectation consiste à attribuer une valeur à une variable. En algorithmique, l'affectation est notée par le signe  $\leftarrow$ .

**Variable  $\leftarrow$  expression**

Chaque langage de programmation définit son opérateur d'affectation ( $=$ ,  $:=$ , ...).

On attribue la valeur ou le résultat de l'expression à la variable. Une expression peut être une valeur numérique, une autre variable, une opération arithmétique ou logique ou une fonction.

Exemple :

$X \leftarrow 10$

$Y \leftarrow X + 20$

L'affectation exige que la variable et la valeur de l'expression soit compatible. Également, une instruction d'affectation ne modifie que ce qui est situé à gauche de la flèche.

Lors d'une affectation, l'expression de droite est évaluée et la valeur trouvée est affectée à la variable de gauche. Ainsi,  $A \leftarrow B$  est différente de  $B \leftarrow A$ .

### 2. Saisir des données

Cette action permet d'affecter, à une variable, une valeur saisie au clavier. Cette action est validée par la frappe de la touche entrée.

**Lire(v)**

Saisit une valeur au clavier et la mémorise dans la variable v.

### 3. Afficher des données

Il est possible de faire afficher une information à l'écran. Cette information peut être une constante, la valeur d'une variable, une suite de valeurs, la combinaison de valeurs et de messages.

Exemple :

Ecrire (i)

Ecrire ("Calcul de la somme des n premiers entiers")

Ecrire ("Résultat du calcul = ", S)

## III. Exemples d'algorithmes

### Algorithme Exemple1

A, B, C : Entier

#### Début

A ← 3  
B ← 10  
C ← A + B  
B ← A + B  
A ← C

#### Fin

### Algorithme Exemple2

x, y : Entier

#### Début

x ← 8  
y ← 3  
x ← 2 \* y + x  
y ← x - y + 2 \* (x - 1)

#### Fin

### Algorithme Surface1

Const Longueur = 4,32

Const Largeur = 3,77

Surface : réel

#### Début

Surface ← Longueur \* Largeur  
Ecrire ("La surface du séjour est :", Surface)

#### Fin

# STRUCTURES CONDITIONNELLES

## I. Introduction

Souvent les problèmes nécessitent l'étude de plusieurs situations qui ne peuvent pas être traitées par les séquences d'actions simples. Puisqu'on a plusieurs situations, et qu'avant l'exécution, on ne sait pas à quel cas de figure on aura à exécuter, dans l'algorithme on doit prévoir tous les cas possibles. Ce sont les structures conditionnelles qui le permettent, en se basant sur ce qu'on appelle prédicat ou condition booléenne.

Un prédicat est un énoncé ou proposition qui peut être vrai ou faux selon ce qu'on est en train de parler. En mathématiques, c'est une expression contenant une ou plusieurs variables et qui est susceptible de devenir une proposition vraie ou fausse selon les valeurs attribuées à ces variables.

### Exemple

- $(10 < 15)$  est un prédicat vrai
- $(10 < 3)$  est un prédicat faux

Une condition est une expression logique qui peut contenir des opérateurs de comparaison ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$ ,  $=$ ) et des opérateurs logiques (NON, ET, OU).

En algorithmique, il y a deux formes de structure conditionnelle :

- La structure conditionnelle simple : « Si »
- La structure conditionnelle à choix multiple : « Selon »

## II. Structure conditionnelle « Si »

### 1. Forme complète

La structure conditionnelle simple « Si » permet selon le résultat d'une condition d'exécuter une suite d'instructions spécifiques. Elle est commune à de nombreux langages de

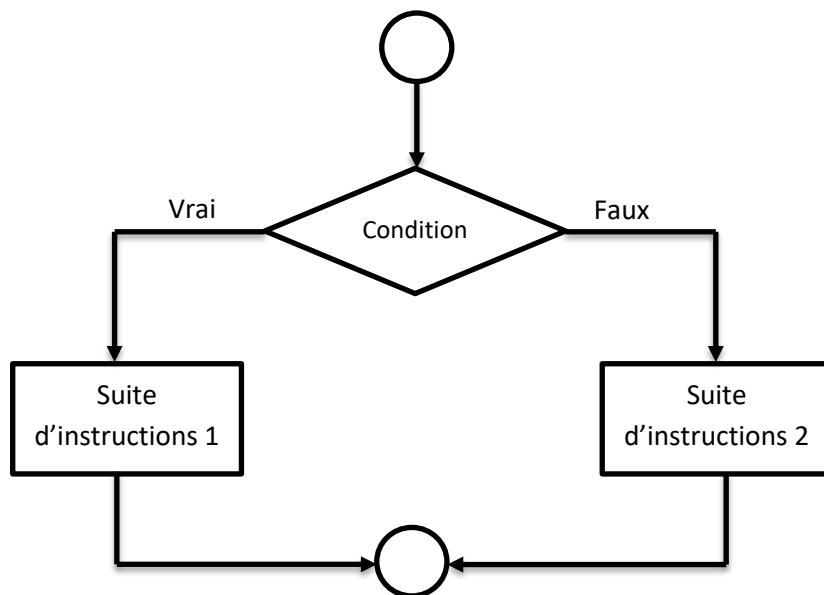
programmation. Bien que la syntaxe varie quelque peu selon le langage, la syntaxe générale en pseudo code est la suivante :

```
Si Condition alors
    Suite d'instructions 1
Sinon
    Suite d'instructions 2
Fin si
```

- Condition : est une expression logique ayant pour valeur Vrai ou Faux.
- Si la condition est vraie alors la suite d'instructions 1 est exécutée.
- Si la condition est fausse alors la suite d'instruction 2 est exécutée.

La suite d'instructions peut être une seule ligne de code comme elle peut être un ensemble de ligne de code.

On peut aussi noter la structure conditionnelle « Si » en utilisant un diagramme de flux :



## 2. Forme réduite

La structure conditionnelle « Si » peut être réduite en éliminant la partie « Sinon » qui est optionnelle. Ainsi, si la condition est vraie alors la suite d'instructions 1 est exécutée sinon on ne fait rien. La syntaxe réduite est la suivante :

```
Si Condition alors
    Suite d'instructions 1
Fin si
```

### Exemple 1

```
Algorithme ValeurAbsolue
    x : réel
Début
    Ecrire ("Entrez un réel : ")
    Lire (x)
    Si x < 0 alors
        Ecrire ("La valeur absolue de ",x,"est:",-x)
    Sinon
        Ecrire ("La valeur absolue de ",x,"est:",x)
    Fin si
Fin
```

### Exemple 2

```
Algorithme Inverse
    x : réel
Début
    Ecrire ("Entrez un réel : ")
    Lire (x)
    Si x <> 0 alors
        Ecrire ("L'inverse de ",x,"est:",1/x)
    Fin si
Fin
```

### 3. Forme imbriquée

Les suites d'instructions dans une structure conditionnelle « Si » peuvent être eux-mêmes d'autres structures conditionnelles « Si ». Ainsi, on parle de structures conditionnelles imbriquées. Il est préférable de faire des décalages (indentations) pour la lisibilité de ces structures et pour ne pas commettre des imbrications interdites.

```
Si Condition1 alors
    Si Condition2 alors
        Suite d'instructions 1
    Sinon
        Suite d'instructions 2
    Fin si
Sinon
    Si Condition3 alors
        Suite d'instructions 3
    Fin si
Fin si
```

#### Exemple 3

```
Algorithme Signe
    n : 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")
        Fin si
    Fin si
Fin
```

**Exemple 4**

On veut réaliser un algorithme nommé JourTexte qui selon un nombre donné entre 1 et 7 affiche le jour de la semaine correspondant.

```
Algorithme JourTexte
    n : entier
Début
    Ecrire ("Entrez un nombre entre 1 et 7 : ")
    Lire (n)
    Si n = 1 alors
        Ecrire ("Lundi")
    Sinon
        Si n = 2 alors
            Ecrire ("Mardi")
        Sinon
            Si n = 3 alors
                Ecrire ("Mercredi")
            Sinon
                Si n = 4 alors
                    Ecrire ("Jeudi")
                Sinon
                    Si n = 5 alors
                        Ecrire ("Vendredi")
                    Sinon
                        Si n = 6 alors
                            Ecrire ("Samedi")
                        Sinon
                            Si n = 7 alors
                                Ecrire ("Dimanche")
                            Sinon
                                Ecrire ("Erreur !")

                                Fin si
                            Fin si
                        Fin si
                    Fin si
                Fin si
            Fin si
        Fin si
    Fin
```

Comme on remarque, cette une écriture qui est très lourde qui nous oblige à répéter beaucoup de fois la même chose ce qui augmente les chances de se tromper dans l'algorithme ou le programme. Pour résoudre ce genre de problème, il est préférable d'utiliser une autre structure conditionnelle appelée à choix multiple.

### III. Structure conditionnelle à choix multiple « Selon »

La structure conditionnelle « Selon » est une forme qui remplace la structure conditionnelle simple lorsque nous avons des choix multiples. Sa syntaxe générale est la suivante :

```
Selon Variable faire
    Valeur 1 : suite d'instructions 1
    Valeur 2 : suite d'instructions 2
    .
    .
    .
    Valeur n : suite d'instructions n
    Autre :     suite d'instructions 0

Fin Selon
```

#### Exemple 5

```
Algorithme JourTexte
    n : entier
Début
    Ecrire ("Entrez un nombre entre 1 et 7 : ")
    Lire (n)
    Selon n faire
        1 : Ecrire ("Lundi")
        3 : Ecrire ("Mercredi")
        4 : Ecrire ("Jeudi")
        5 : Ecrire ("Vendredi")
        6 : Ecrire ("Samedi")
        7 : Ecrire ("Dimanche")
        Autre : Ecrire ("Erreur !")
    Fin Selon
Fin
```

#### Exercice

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leurs 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.

# **STRUCTURES CONDITIONNELLES**

## EXEMPLE INTRODUCTIF

On veut écrire un algorithme qui calcul l'inverse d'un nombre réel donné :

```
Algorithm Inverse
    x, y : réel
Début
    Ecrire ("Donner un nombre :")
    Lire (x)
    y ← 1/x
    Ecrire ("Le résultat est:", y)
Fin
```

## EXEMPLE INTRODUCTIF

Un algorithme est correcte si :

- Il est optimal
- Il se termine correctement
- Il tient compte de tous les cas possibles

### **Algorithme** Inverse

$x, y$  : réel

#### Début

Ecrire ("Donner un nombre :")

Lire ( $x$ )

$y \leftarrow 1/x$

Ecrire ("Le résultat est:",  $y$ )

#### Fin

### Exécution manuelle

| X  | Y      |
|----|--------|
| -2 | -0.5   |
| 2  | 0.5    |
| 0  | Erreur |

→ L'algorithme fonctionne bien lorsque  $x \neq 0$

→ L'algorithme génère une erreur lorsque  $x = 0$

→ Donc on a deux différents cas que l'algorithme doit tenir en compte

## STRUCTURE CONDITIONNELLE SI

Pour corriger l'algorithme précédent, il faut utiliser **une structure conditionnelle simple qui est la structure conditionnelle « Si ».**

La syntaxe générale est la suivante:

```
Si Condition alors
    suite d'instructions 1
Sinon
    suite d'instructions 2
Fin Si
```

- **Condition** : est une expression logique ayant pour valeur Vrai ou Faux.
- **Si la condition est vraie** alors la suite d'instructions 1 est exécutée.
- **Si la condition est fausse** alors la suite d'instructions 2 est exécutée.

## STRUCTURE CONDITIONNELLE SI

On corrige l'algorithme Inverse en utilisant la structure conditionnelle « SI »

**Algorithme** Inverse

    x, y : réel

**Début**

    Ecrire("Donner un nombre :")

    Lire(x)

**Si** x<>0 **alors**

        y ← 1/x

        Ecrire("Le résultat est:", y)

**Sinon**

        Ecrire("La valeur x doit être différente de 0")

**Fin si**

**Fin**

## FORME RÉDUITE DE SI

La structure conditionnelle « Si » peut être réduite en éliminant la partie « Sinon » qui est optionnelle.

Ainsi, si la condition est vraie alors la suite d'instructions 1 est exécutée sinon on ne fait rien.

La syntaxe réduite est la suivante :

```
Si Condition alors
    suite d'instructions 1
Fin Si
```

## FORME RÉDUITE DE SI

On peut écrire l'algorithme Inverse en utilisant la forme réduite de la structure conditionnelle « Si »

**Algorithme** Inverse

    x, y : réel

**Début**

    Ecrire("Donner un nombre :")

    Lire(x)

**Si** x<>0 **alors**

        y ← 1/x

        Ecrire("Le résultat est:", y)

**Fin si**

**Fin**



L'algorithme est correcte mais il n'est pas complet

**EXEMPLE 1**

On veut écrire un algorithme qui compare deux nombres entiers :

**Algorithme** Compare

a, b : entier

**Début**

Ecrire ("Donner le premier nombre :")

Lire (a)

Ecrire ("Donner le deuxième nombre :")

Lire (b)

**Si** a < b **alors**

Ecrire ("a est inférieur à b")

**Sinon**

Ecrire ("a est supérieur à b")

**Fin si**

**Fin**

**Exécution manuelle**

| a | b | Ecran                      |
|---|---|----------------------------|
| 0 | 1 | a est inférieur à b        |
| 2 | 0 | a est supérieur à b        |
| 2 | 2 | <b>a est supérieur à b</b> |



On remarque que le programme ignore le cas où il y a une égalité entre a et b



On a trois cas : il faut ajouter une autre structure conditionnelle « Si »

## FORME IMBRIQUÉE DE SI

Les suites d'instructions dans une structure conditionnelle « Si » peuvent être eux-mêmes d'autres structures conditionnelles « Si ».

La syntaxe est la suivante :

```
Si Condition1 alors
    Si Condition2 alors
        Suite d'instructions 1
    Sinon
        Suite d'instructions 2
    Fin si
Sinon
    Si Condition3 alors
        Suite d'instructions 3
    Fin si
Fin si
```

## EXEMPLE 2

On ajoute une autre structure conditionnelle imbriquée :

```
Algorithm Compare
    a, b : entier
Début
    Ecrire("Donner le premier nombre :")
    Lire(a)
    Ecrire("Donner le deuxième nombre :")
    Lire(b)
    Si a<b alors
        Ecrire("a est inférieur à b")
    Sinon
        Si a> b alors
            Ecrire("a est supérieur à b")
        Sinon
            Ecrire("a est égale à b")
        Fin si
    Fin si
Fin
```

## EXEMPLE 3

On veut réaliser un algorithme nommé JourTexte qui selon un nombre donné entre 1 et 7 affiche le jour de la semaine correspondant.

### Algorithme JourTexte

n : entier

#### Début

Ecrire ("Entrez un nombre entre 1 et 7 : ")

Lire (n)

**Si** n = 1 **alors**

Ecrire ("Lundi")

**Sinon**

**Si** n = 2 **alors**

Ecrire ("Mardi")

**Sinon**

**Si** n = 3 **alors**

Ecrire ("Mercredi")

**Sinon**

**Si** n = 4 **alors**

Ecrire ("Jeudi")

**Sinon**

## EXEMPLE 3

On veut réaliser un algorithme nommé JourTexte qui selon un nombre donné entre 1 et 7 affiche le jour de la semaine correspondant.

```
Si n = 5 alors
    Ecrire ("Vendredi")
Sinon
    Si n = 6 alors
        Ecrire ("Samedi")
    Sinon
        Si n = 7 alors
            Ecrire ("Dimanche")
        Sinon
            Ecrire ("Erreur !")
        Fin si
    Fin si
Fin si
Fin si
Fin si
Fin si
Fin si
Fin si
Fin si
Fin
```

## STRUCTURE CONDITIONNELLE SELON

La structure conditionnelle « **Selon** » est une forme qui remplace la structure conditionnelle simple lorsque nous avons des choix multiples.

Sa syntaxe générale est la suivante :

```
Selon Variable faire
    Valeur 1 : suite d'instructions 1
    Valeur 2 : suite d'instructions 2
    .
    .
    .
    Valeur n : suite d'instructions n
    Autre      : suite d'instructions 0
Fin Selon
```

## EXEMPLE 4

On veut réaliser un algorithme nommé JourTexte qui selon un nombre donné entre 1 et 7 affiche le jour de la semaine correspondant.

**Algorithme** JourTexte

n : entier

**Début**

Ecrire ("Entrez un nombre entre 1 et 7 : ")

Lire (n)

**Selon** n **faire**

1 : Ecrire ("Lundi")

3 : Ecrire ("Mercredi")

4 : Ecrire ("Jeudi")

5 : Ecrire ("Vendredi")

6 : Ecrire ("Samedi")

7 : Ecrire ("Dimanche")

Autre : Ecrire ("Erreur !")

**Fin Selon**

**Fin**

## EXERCICE ALGORITHME SIGNE

Ecrire un algorithme nommé **Signe** qui demande deux nombres réel à l'utilisateur et l'informe ensuite si leurs produit est négatif ou positif (on ne considère pas le cas où le produit est nul). Attention toutefois, on ne doit pas calculer le produit des deux nombres.

### Travail à rendre:

Déposer l'algorithme Signe dans le « **Dépôt de l'exercice algorithme Signe** » qui figure dans la plateforme. La date limite est fixée pour le **15/11/2020 à 21h00**. Le travail est noté.

# STRUCTURES ITERATIVES

## I. Introduction

En programmation, souvent on a besoin de répéter une instruction ou un ensemble d'instructions un nombre connu ou inconnu de fois. Pour résoudre ce problème, on fait appel à des structures itératives appelées également boucles.

Il existe différentes structures itératives dont la principale différence réside selon le nombre de répétition qui peut être connu ou inconnu. La chose la plus importante à savoir lors de l'utilisation des boucles c'est qu'il faut toujours sortir de la boucle, sinon le programme risque de se planter en entrant donne une boucle infinie.

En algorithmique, on distingue trois formes de boucles : la boucle « Pour », la boucle « Répéter - Jusqu'à », et la boucle « Tant que ». La plupart des langages de programmation intègre ces trois boucles mais avec des syntaxes différentes. Le choix de l'utilisation de la forme de boucle dépend généralement du problème à résoudre.

## II. Structure itérative « Pour »

La boucle « Pour » est une structure itérative à nombre de répétition connu à l'avance. Elle utilise un compteur qui évolue (incrémentation ou décrémentation) entre une valeur initiale et une valeur finale. La syntaxe générale en pseudo code de la boucle « Pour » est la suivante :

```
Pour Compteur de Valeur_Début à Valeur_Fin [par Pas Valeur_Pas]
    Suite d'instructions
Fin Pour
```

- **Compteur** : est une variable de type entier.
- **Valeur\_Début** et **Valeur\_Fin** : peuvent être des valeurs ou des variables entières.

- **Valeur\_Pas** : cette partie est optionnelle. Si elle est omis alors la valeur du pas est par défaut est 1. Si elle est fixée alors à chaque itération la valeur du pas est ajoutée à la variable compteur.

Au début, le compteur est initialisé par Valeur\_Début. Puis à chaque exécution de la boucle, la valeur actuelle du compteur est comparée à Valeur\_Fin :

- Si le compteur est  $\leq$  à Valeur\_Fin dans le cas d'un pas positif (ou si le compteur est  $\geq$  à Valeur\_Fin pour un pas négatif), alors la suite instructions est exécutée.
- Si le compteur est  $>$  à Valeur\_Fin dans le cas d'un pas positif (ou si le compteur est  $<$  à Valeur\_Fin pour un pas négatif), alors on sort de la boucle et on continue avec les instructions qui suit Fin Pour.

### Exemple 1

On veut écrire un algorithme qui affiche le mot « Bonjour » 100 fois sur l'écran.

```
Algorithme Affichage1
    i : entier
Début
    Pour i de 1 à 100
        Ecrire ("Bonjour")
    Fin Pour
Fin
```

### Exemple 2

On veut écrire un algorithme qui affiche tous les nombres pairs entre 20 et 50 sur l'écran.

```
Algorithme Affichage2
    i : entier
Début
    Pour i de 20 à 50 par Pas 2
        Ecrire (i)
    Fin Pour
Fin
```

### Exemple 3

On veut écrire un algorithme qui calcul la somme des nombres entre 1 et 1000 puis affiche le résultat sur l'écran.

```
Algorithme Somme
    i, s : entier
Début
    s←0
    Pour i de 1 à 1000
        s←s+i
    Fin Pour
    Ecrire ("La somme des nombres entre 1 et 1000 est :",s)
Fin
```

## III. Structure itérative « Tant que »

La boucle « Tant que » est une structure itérative à nombre de répétition inconnu. L'arrêt ou la continuité de la boucle dépend d'une condition logique. La syntaxe générale en pseudo code de la boucle « Tant que » est la suivante :

```
Tant que Condition
    Suite d'instructions
Fin Tant que
```

La condition est évaluée avant chaque itération :

- Si la condition est vraie alors on exécute la suite des instructions puis on retourne tester la condition et ainsi de suite.
- Si la condition est fausse alors la boucle est terminée puis les instructions qui suivent « Fin tant que » seront exécutées.

Puisque la vérification de l'arrêt de la boucle se fait au début, alors la boucle « Tant que » peut être exécutée 0 à n fois.

Egalement, il faut faire attention lors de l'écriture de la condition par ce qu'il a un risque de faire une boucle infinie.

#### Exemple 4

Ecrire un algorithme qui fait la somme des nombres saisis par l'utilisateur tant que la somme est inférieure ou égale à 1000.

```
Algorithm SommeNombre1
    n, s : entier
Début
    s ← 0
    Tant Que s<=1000
        Ecrire ("Donner un nombre :")
        Lire (n)
        s←s+n
    Fin Tant Que
    Ecrire ("La somme est :",s)
Fin
```

#### Exemple 5

Ecrire un algorithme qui fait la somme des nombres entre 100 et un nombre saisi par l'utilisateur.

```
Algorithm SommeNombre2
    n, s, i : entier
Début
    s←0
    i←100
    Ecrire ("Donner un nombre :")
    Lire (n)
    Tant Que i<=n
        s←s+i
        i←i+1
    Fin Tant Que
    Ecrire ("La somme est :",s)
Fin
```

Si l'utilisateur saisie un nombre supérieur ou égale à 100, on peut entrer dans la boucle « Tant que » sinon la boucle ne sera jamais exécutée.

#### IV. Structure itérative « Répéter – jusqu'à »

La boucle « Répéter – jusqu'à » est une structure itérative à nombre répétition inconnu. L'arrêt ou la continuité de la boucle dépend d'une condition logique mais contrairement à la boucle « Tant que », la vérification se fait à la fin de la boucle ainsi que la condition arrêt. La syntaxe générale en pseudo code de la boucle « Répéter – jusqu'à » est la suivante :

**Répéter**

    Suite d'instructions

**Jusqu'à** Condition

La condition est évaluée après chaque itération :

- Si la condition est vraie alors la boucle est terminée.
- Si la condition est fausse alors on répète la boucle.

Puisque, la vérification de l'arrêt de la boucle se fait à la fin, alors la boucle peut être exécutée 1 à n fois.

Il existe une autre forme de cette boucle dont la condition d'arrêt est inversée par rapport à la boucle « Répéter – jusqu'à ». C'est-à-dire, si la condition est vraie alors on continu l'itération sinon on arrête. C'est la boucle « Répéter – Tant que »

**Répéter**

    Suite d'instructions

**Tant que** Condition

Généralement, les boucles « Répéter – jusqu'à » et « Répéter – Tant que » sont utilisées pour la saisie conditionnée.

### Exemple 6

Ecrire un algorithme qui permet de saisir un nombre entier compris entre 1 et 100.

#### En utilisant la boucle « Répéter – Jusqu'à »

```
Algorithm Saisiel
    n : entier
Début
    Répéter
        Ecrire ("Donner un nombre entre 1 et 100:")
        Lire (n)
    Jusqu'à n>=1 et n<=100
        Ecrire ("Le nombre saisis est :",n)
Fin
```

#### En utilisant la boucle « Répéter – Tant que »

```
Algorithm Saisie2
    n : entier
Début
    Répéter
        Ecrire ("Donner un nombre entre 1 et 100:")
        Lire (n)
    Tant que n<1 ou n>100
        Ecrire ("Le nombre saisis est :",n)
Fin
```

## V. Structure itérative « Répéter – Fin répéter »

La boucle « Répéter – Fin Répéter » est la forme générale des structures itératives. Elle est considérée comme structure à nombre de répétition inconnu. L'arrêt de la boucle dépend d'une structure conditionnelle « Si » qui est ajoutée comme instruction dans la boucle. La syntaxe générale en pseudo code de la boucle « Répéter – Fin répéter » est la suivante :

**Répéter**

Suite d'instructions 1

**Si** Condition **alors**        **Sortir**    **Fin si**

Suite d'instructions 1

**Fin répéter****Exemple :**

Algorithme Ex5

*i,j* : entier

Début

*i*←1    *j*←0

Répéter

*i*←*i*+2        **Si** *i*>10 **alors**            **Sortir**        **Fin si**        *j*←*j*\**i*    **Fin répéter**

Fin

# LES TABLEAUX

## I. Introduction

Souvent en programmation, on a besoin d'utiliser un grand nombre de variables de même type ce qui rend impossible leurs déclaration comme simple variables. Il existe une autre structure de données mieux adaptée pour résoudre de ce genre de problème. Les tableaux sont des structures de données complexes qui contiennent un ensemble d'éléments de même type. Un tableau est une variable qui regroupe un nombre connu d'élément de même type qui doivent être fixés lors de sa déclaration.

## II. Tableau à une dimension

### 1. Définition

Un tableau à une dimension, appelé aussi vecteur, est une structure de données formée de données de même type pouvant être accédé avec un indice. Il nous permet de manipuler plusieurs valeurs en utilisant un seul nom de variable.

Pour définir un tableau, il faut préciser :

- L'identificateur du tableau (son nom)
- Le type des indices (entier)
- Le type des éléments du tableau (composants)

### 2. Notations

Dans tous les langages, le traducteur (compilateur ou interpréteur) doit nécessairement connaître le nombre d'éléments d'un tableau (sa taille). Cette information lui permet de réserver l'emplacement mémoire correspondant. Généralement, il n'est pas possible de changer la taille d'un tableau après la déclaration (dépend du langage de programmation). De même, comme pour une variable, il faut préciser quel est le type d'un tableau et donc le type de tous ses éléments.

Lors de la déclaration, le tableau est créé, mais les composants n'ont encore aucune valeur. On dit qu'ils ne sont pas initialisés. Un tableau peut être déclaré en précisant le nombre d'élément (taille), ou la valeur du premier indice (indice\_début) et la valeur du dernier indice (indice\_fin) de la manière suivante :

<Nom\_tableau> : Tableau[<taille>] de <type>

<Nom\_tableau> : Tableau[<indice\_début..indice\_fin>] de <type>

### **Remarque**

La manière de déclarer un tableau et l'indice du premier élément dépendent du langage de programmation utilisé. Par exemple, le langage C exige le commencement des indices par zéro donc le dernier élément possède un indice égal à taille-1.

### **Exemple**

- Notes : Tableau[20] de réels ➔ déclare un tableau de 20 réels nommé « Notes »
- Tab : Tableau[1..5] d'entiers ➔ déclare un tableau de 5 entiers

Dans un tableau, un élément est identifié par sa position ce qui traduit l'accès directe. Un élément du tableau peut être considéré comme une variable qu'on peut utiliser dans des expressions, changer sa valeur par des affectations, ou l'utiliser dans des opérations de lecture écriture. Pour accéder à un élément d'un tableau, il faut préciser l'indice de l'élément dans le tableau :

<Nom\_tableau> [<indice>]

Il faut que l'indice soit dans la plage des éléments du tableau. Il peut être un entier, une variable entière, ou une expression.

### **Exemple**

- Notes[5] ← 15.5      mettre la valeur « 15.5 » dans l'élément 5
- Notes[i] ← 11          mettre la valeur « 11 » dans l'élément ayant la valeur de i

### 3. Opérations sur les tableaux

#### a. Affectation

L'affectation se traduit par la recopie d'un tableau dans un autre. Les deux tableaux doivent évidemment avoir le même type de composants et d'indice ainsi que la même taille. En algorithmique, il est possible d'affecter directement un tableau dans un autre mais cette opération peut être interdite dans certain langage de programmation (comme le langage C). Ainsi, il faut faire une affectation des éléments un à un.

#### b. Saisie des éléments d'un tableau

Soit l'algorithme suivant qui permet de remplir un tableau de 5 entiers. On considère que l'indice du premier élément est zéro pour faciliter la traduction en langage C.

```
Algorithme SaisieTab
    Tab : Tableau[5] d'entiers
    i: entier
Début
    Pour i de 0 à 4
        Ecrire("Donner l'élément n°",i," :")
        Lire(Tab[i])
    Fin pour
Fin
```

#### c. Affichage des éléments d'un tableau

```
Algorithme AffichageTab
    Tab : Tableau[3] d'entiers
    i: entier
Début
    Tab[0]←3
    Tab[1]←2
    Tab[2]←10
    Pour i de 0 à 2
        Ecrire("l'élément n°",i," =",Tab[i])
    Fin pour
Fin
```

#### Exercice n°1

Ecrire un algorithme qui saisit un tableau de 10 entiers puis affiche la somme de ses éléments.

```

Algorithme SommeTab
    Tab : Tableau[10] d'entiers
    i, som: entier
Début
    Pour i de 0 à 9
        Ecrire("Donner l'élément n°",i," :")
        Lire(Tab[i])
    Fin pour
    som←0
    Pour i de 0 à 9
        som←som+Tab[i]
    Fin pour
    Ecrire("La somme est :",som)
Fin

```

### Exercice n°2

Ecrire un algorithme qui saisit un tableau de 10 entiers puis affiche le nombre d'éléments ayant une valeur nulle :

```

Algorithme NbElemTab
    Tab : Tableau[10] d'entiers
    i, nb: entier
Début
    Pour i de 0 à 9
        Ecrire("Donner l'élément n°",i," :")
        Lire(Tab[i])
    Fin pour
    nb←0
    Pour i de 0 à 9
        Si Tab[i]=0 alors
            nb←nb+1
        Fin si
    Fin pour
    Ecrire("Le nombre d'éléments nuls est :",nb)
Fin

```

### Exercice n°3

Ecrire un algorithme qui saisit un tableau de 5 entiers puis affiche la valeur la plus petite :

```

Algorithme PetitTab
    Tab : Tableau[5] d'entiers
    i, p: entier
Début
    Pour i de 0 à 4
        Ecrire("Donner l'élément n°",i," :")
        Lire(Tab[i])
    Fin pour
    p←Tab[0]
    Pour i de 1 à 4
        Si Tab[i]<p alors
            p← Tab[i]
        Fin si
    Fin pour
    Ecrire("La valeur la plus petite est :",p)
Fin

```

### III. Tableau à deux dimensions

Un tableau à deux dimensions est un tableau composé d'éléments de même type et ayant deux indices qui permettent d'identifier un élément. On peut le simuler à une matrice qui possède des lignes et des colonnes. Donc pour identifier un élément, il faut donner l'indice de la ligne et l'indice de la colonne.

Un tableau à deux dimensions est déclaré de la manière suivante :

<Nom\_tableau> : Tableau[<taille1>] [<taille2>] de <type>

#### Exemple

Mat : Tableau[10][5] de réels → une matrice de 10 lignes et 5 colonnes de réels

Pour accéder à un élément d'un tableau à deux dimensions, il faut mentionner les indices de l'élément à utiliser :

<Nom\_tableau>[<indice1>][<indice2>]

#### Exemple

- Mat[1][2]←10
- Mat[1][2]← Mat[1][2]+5

**Exercice n°4**

On veut écrire un algorithme qui saisit une matrice d'entiers composée de 3 lignes et 4 colonnes :

```
Algorithme SaisieMat
    Mat : Tableau[3][4] d'entiers
    i, j: entier
Début
    Pour i de 0 à 2
        Pour j de 0 à 3
            Ecrire("Mat[",i,",",",j,"]:")
            Lire(Mat[i][j])
        Fin pour
    Fin pour
Fin
```

**Exercice n°5**

On veut écrire un algorithme qui initialise à zéro une matrice d'entiers composée de 2 lignes et 3 colonnes :

```
Algorithme InitMat
    Mat : Tableau[2][3] d'entiers
    i, j: entier
Début
    Pour i de 0 à 1
        Pour j de 0 à 2
            Mat[i][j]←0
        Fin pour
    Fin pour
Fin
```

# **LES TABLEAUX**

## EXEMPLE INTRODUCTIF

On veut écrire un algorithme qui calcul la moyenne arithmétique de cinq notes avec leurs coefficients (les notes dans [0..20] et les coefficients dans [1..3])

```
Algorithme Ex7
    n,c,som,sco,moy : réel
    i : entier
Début
    som←0
    sco←0
    Pour i de 1 à 5
        Répéter
            Ecrire("Donner la note",i," :")
            Lire(n)
            Jusqu'à n>=0 et n<=20
        Répéter
            Ecrire("Donner le coef",i," :")
            Lire(c)
            Jusqu'à c>=1 et c<=3
            som←som+(n*c)
            sco←sco+c
        Fin pour
        moy=som/sco
        Ecrire("La moyenne est :",moy)
Fin
```



**L'algorithme ne permet pas de sauvegarder toutes les notes et les coefficients**



**Pour résoudre ce problème, il faut utiliser les tableaux**

## INTRODUCTION

- Souvent en programmation, on a besoin d'utiliser un **grand nombre de variables** de **même type** ce qui rend **impossible leurs déclaration** comme simple variables.
- Les **tableaux** sont des **structures de données complexes** qui contiennent un **ensemble d'éléments de même type**.
- Un tableau est **une variable** qui regroupe un **nombre connu d'élément** de même type qui doivent être **fixés lors de sa déclaration**.
- Généralement le **nombre d'éléments** d'un tableau est **fixé lors de la déclaration** et **ne change** pas lors du programme.
- Un tableau est un ensemble d'élément alloués dans la mémoire d'une **façon contigüe**.

## DÉFINITION

- **Un tableau à une dimension**, appelé aussi vecteur, est une structure de données formée de données **de même type** pouvant être accédé avec un indice.
- Il nous permet de manipuler **plusieurs valeurs** en utilisant un **seul nom de variable**.
- Pour définir un tableau, il faut préciser :
  - L'identificateur du tableau (son nom)
  - Le type des indices (entier)
  - Le type des éléments du tableau (composants)

| Notes | 15.5 | 10 | 9 | 12.5 | 8.5 |
|-------|------|----|---|------|-----|
|       | 0    | 1  | 2 | 3    | 4   |

## NOTATION

Un tableau peut être déclaré en précisant le nombre d'élément (**taille**), ou la valeur du premier indice (**indice\_début**) et la valeur du dernier indice (**indice\_fin**) de la manière suivante :

**<Nom\_tableau> : Tableau[<taille>] de <type>**

**<Nom\_tableau> : Tableau[<indice\_début..indice\_fin>] de <type>**

### Exemple

Notes : Tableau[20] de réels → déclare un tableau de 20 réels nommé « Notes »

Tab : Tableau[1..5] d'entiers → déclare un tableau de 5 entiers

## NOTATION

- Un élément est identifié par **sa position** ce qui traduit **l'accès directe**.
- Un **élément du tableau** peut être considéré comme **une variable**
- Pour accéder à un élément d'un tableau, il faut préciser l'indice de l'élément dans le tableau :

**<Nom\_tableau> [<indice>]**

- Il faut que l'indice soit dans **la plage des éléments** du tableau.
- Il peut être un entier, une variable entière, ou une expression.

### Exemple

Notes[5]←15.5      mettre la valeur « 15.5 » dans l'élément d'indice 5

Notes[i]←11      mettre la valeur « 11 » dans l'élément ayant un indice la valeur de i

## OPÉRATIONS SUR LES TABLEAUX

### Saisie des éléments d'un tableau

Soit l'algorithme suivant qui permet de remplir un tableau de 5 entiers. On considère que **l'indice du premier élément est zéro** pour faciliter la traduction en langage C.

```
Algorithme SaisieTab
    Tab : Tableau[5] d'entiers
    i: entier
Début
    Pour i de 0 à 4
        Ecrire("Donner l'élément n°",i," :")
        Lire(Tab[i])
    Fin pour
Fin
```

## OPÉRATIONS SUR LES TABLEAUX

### Affichage des éléments d'un tableau

```
Algorithme AffichageTab
    Tab : Tableau[3] d'entiers
    i: entier
Début
    Tab[0] ← 3
    Tab[1] ← 2
    Tab[2] ← 10
    Pour i de 0 à 2
        Ecrire("l'élément n°", i, " = ", Tab[i])
    Fin pour
Fin
```

## OPÉRATIONS SUR LES TABLEAUX

### Exercice n°1

Ecrire un algorithme qui saisit un tableau de 10 entiers puis affiche la somme de ses éléments.

```
Algorithme SommeTab
    Tab : Tableau[10] d'entiers
    i, som: entier
Début
    Pour i de 0 à 9
        Ecrire("Donner l'élément n°", i, " :")
        Lire(Tab[i])
    Fin pour
    som←0
    Pour i de 0 à 9
        som←som+Tab[i]
    Fin pour
    Ecrire("La somme est :", som)
Fin
```

## OPÉRATIONS SUR LES TABLEAUX

### Exercice n°2

Ecrire un algorithme qui saisit un tableau de 10 entiers puis affiche le nombre d'éléments ayant une valeur nulle :

```
Algorithme NbElemTab
    Tab : Tableau[10] d'entiers
    i, nb: entier
Début
    Pour i de 0 à 9
        Ecrire("Donner l'élément n°",i," :")
        Lire(Tab[i])
    Fin pour
    nb←0
    Pour i de 0 à 9
        Si Tab[i]=0 alors
            nb←nb+1
        Fin si
    Fin pour
    Ecrire("Le nombre d'éléments nuls est :",nb)
Fin
```

## OPÉRATIONS SUR LES TABLEAUX

### Exercice n°3

Ecrire un algorithme qui saisit un tableau de 5 entiers puis affiche la valeur la plus petite :

```
Algorithme PetitTab
    Tab : Tableau[5] d'entiers
    i, p: entier
Début
    Pour i de 0 à 4
        Ecrire("Donner l'élément n°",i," :")
        Lire(Tab[i])
    Fin pour
    p←Tab[0]
    Pour i de 1 à 4
        Si Tab[i]<p alors
            p← Tab[i]
        Fin si
    Fin pour
    Ecrire("La valeur la plus petite est :",p)
Fin
```

## DÉCLARATION

- ❑ Un **tableau à deux dimensions** est un tableau composé d'éléments de même type et ayant **deux indices** qui permettent d'identifier un élément.
- ❑ On peut le simuler à **une matrice** qui possède des lignes et des colonnes.
- ❑ Donc pour identifier un élément, il faut donner **l'indice de la ligne et l'indice de la colonne**.
- ❑ Un tableau à deux dimensions est déclaré de la manière suivante :

**<Nom\_tableau> : Tableau[<taille1>] [<taille2>] de <type>**

### Exemple

Mat : Tableau[10][5] de réels → une matrice de 10 lignes et 5 colonnes de réels

## UTILISATION

Pour accéder à un élément d'un tableau à deux dimensions, il faut mentionner les indices de l'élément à utiliser :

**<Nom\_tableau>[<indice1>][<indice2>]**

### Exemple

Mat[1][2] ← 10

Mat[1][2] ← Mat[1][2] + 5

## EXEMPLES

### Exercice n°4

On veut écrire un algorithme qui saisit une matrice d'entiers composée de 3 lignes et 4 colonnes :

```
Algorithme SaisieMat
    Mat : Tableau[3][4] d'entiers
    i, j : entier
Début
    Pour i de 0 à 2
        Pour j de 0 à 3
            Ecrire("Mat[", i, ", ", j, "] : ")
            Lire(Mat[i][j])
        Fin pour
    Fin pour
Fin
```

## EXEMPLES

### Exercice n°5

On veut écrire un algorithme qui initialise à zéro une matrice d'entiers composée de 2 lignes et 3 colonnes :

```
Algorithme InitMat
    Mat : Tableau[2][3] d'entiers
    i, j: entier
Début
    Pour i de 0 à 1
        Pour j de 0 à 2
            Mat[i][j] ← 0
        Fin pour
    Fin pour
Fin
```