ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS
UNIVERSITÉ HASSAN II DE CASABLANCA

# ALGORITHMIQUE

Années Préparatoires Intégrés ( API )

Département Génie Informatique

Année Universitaire 2024/2025

# CHAPITRE 1

## LE PSEUDO-CODE

# DÉFINITIONS


**Enoncé d'un problème**
- Analyse
- Compréhension
- Résolution

**Algorithme**
- Codification

Pseudo-code

**Programme**
- Interprétation

Langage de programmation (code)

**Exécution par l'ordinateur**
- Résultat

Langage machine

# DÉFINITIONS

## Qu’est-ce qu’un Pseudo-code ?

Le pseudocode est un **langage pour exprimer clairement et formellement un algorithme**. Ce langage est près d'un langage de programmation comme Java, C ou C++, sans être identique à l'un ou à l'autre. Il exprime des **idées formelles** dans une langue près du langage naturel de ses usagers (pour nous, le français) en lui imposant une forme rigoureuse

## Qu’est ce que veut dire « écrire un algorithme »?

- **Analyser et comprendre le problème**

  Etude des données fournies et des résultats attendus.

- **Résoudre le problème**

  C’est trouver les structures de données adaptées ainsi que l’enchaînement des actions à réaliser pour passer des données (entrées) aux résultats(sorties).

# DÉFINITIONS

## Y a-t-il un langage d'algorithme universelle?

Aucune règle « officielle » ou norme n'est définie pour écrire du pseudo-code. Il n'y a donc pas de « mauvaise » syntaxe, ni d'écriture meilleure qu'une autre (vous pouvez constater cela en cherchant sur internet). Mais dans un soucis de compréhension et d'adaptation facile aux future langages qui vont être étudiés, nous allons adopter les notations proposées dans ce cours.

## L'analyse et l'écriture d'un algorithme dépende de quoi?

Les contraintes qui régissent une écriture algorithmique sont :

- ❖ Le niveau d'abstraction du type de langage ciblé.
- ❖ Les opérations élémentaires.
- ❖ La similarité aux langages naturels humain.
- ❖ La recherche d'universalisation.

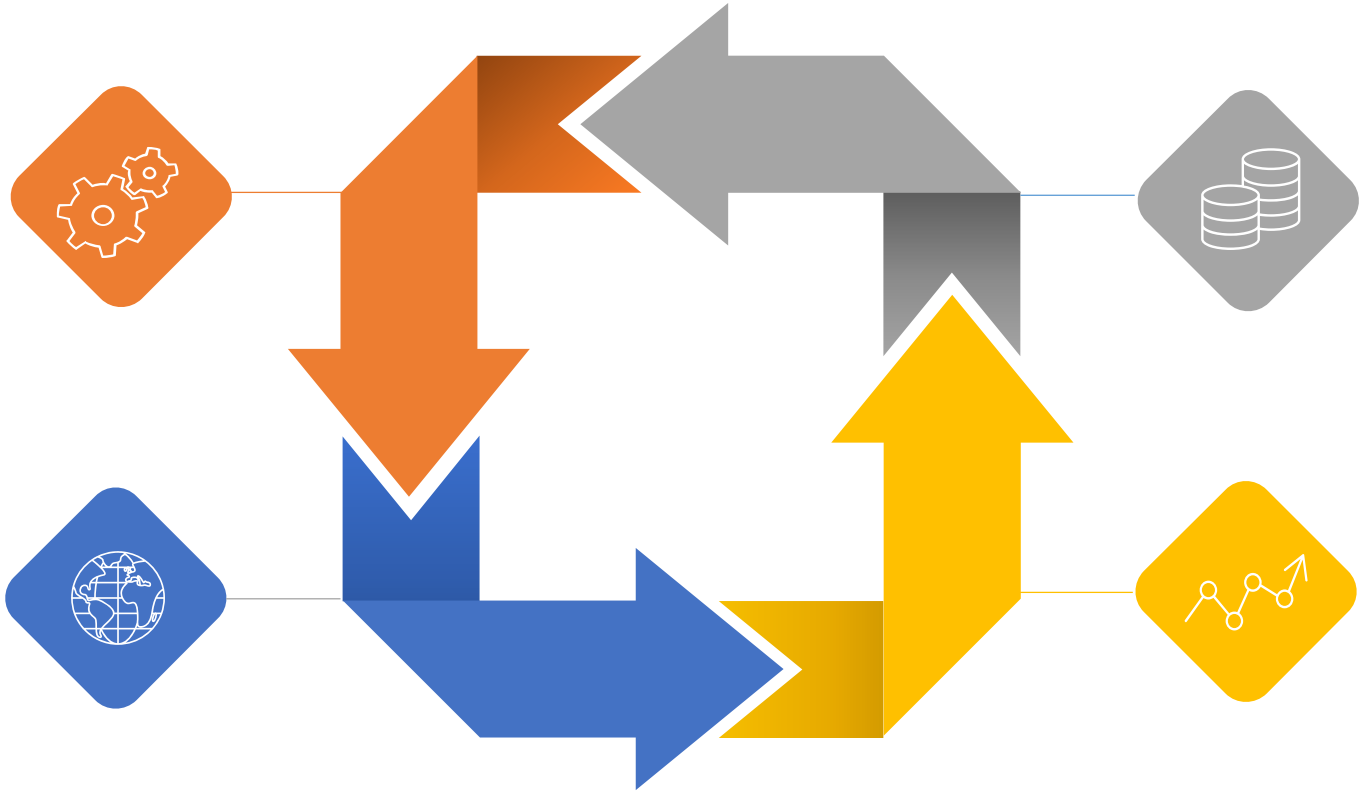
# LES CARACTÉRISTIQUES D'UN ALGORITHME

**La modularité et la réutilisabilité**

une écriture modulaire par décomposition de tâches peut rendre des parties réutilisables dans d'autres contextes.

**La clarté et la compréhensibilité de l'algorithme**

La clarté des instructions et leur enchainement.
L'accompagnement par des commentaire.
Le choix significatif des noms de variables.



**L'organisation des données et utilisation de ressources**

La plupart des algorithmes compromettent une organisation optimisée des données et des ressources impliqué dans les calculs. Cette organisation mène à des structures de données qui sont également des objets d'étude centraux en informatique.

**Nombre de calculs, temps d'exécution, complexité**

Il s'agit de l'évaluation du temps que peut prendre l'exécution d'un algorithme lorsqu'il est traduit en programme.

# MISE EN FORME D'UN ALGORITHME

## Le Modèle à respecter

- Un algorithme **doit porter un nom** (il est désirable qu'il soit significatif et réduit).
- Juste après la déclaration de variables pour stocker les données.
- Les mots clés **doivent être écrits en majuscule**.
- Entre **DEBUT** et **FIN** nous mettons l'ensemble des instructions qui résous le problème.

```
ALGORITHME <nom_algorithme>
   <déclarations_variables>
DEBUT
   <instructions>
FIN
```

# LES VARIABLES 1/3

## Définition des variables

Une variables est définie par :

- Un identificateur : suite quelconque de caractères.
- Un type : Booléen, numérique (entier ou réel), caractère ou chaîne de caractères.
- Une valeur : c'est le contenu de l'objet

## Déclaration

La déclaration des variables :

```
<id1>   : TYPE
```

Si plusieurs variables ont le même type nous pouvons les regrouper :

```
<id2>,<id3>… : TYPE
```

# LES VARIABLES 2/3

## Les types de variables

Types possibles de variables :

- **ENTIER** (ex : 14, -138)
- **REEL** (ex : 3.14, 126.45)
- **BOOLEEN** (Vrai/Faux ou True/False)
- **CARACTERE** (ex : ‘1', 'H')
- **CHAINE** (ex : ''Bonjour'')

Exemple de déclaration de variables

```
ALGORITHME : PrixDuPain
Nom : CHAINE
Nb : ENTIER
Prx, Qt, Tot : REEL
DEBUT
   Instruction1
  Instruction2
FIN
```

# LES VARIABLES 3/3

## Les identificateurs

Un identificateur est un nom donné à une variable pour la différencier de toutes les autres. Et ce nom, c'est au programmeur de le choisir. Cependant, il y a quelques limitations à ce choix :

1. On ne peut utiliser que les 26 lettres de l'alphabet latin, les chiffres et le caractère underscore ( _ ) : pas d'accents, pas de ponctuation ni d'espaces.
2. Un identificateur ne peut pas commencer par un chiffre.
3. Deux variables ne peuvent avoir le même identificateur (le même nom)
4. Certains noms sont réservés pour les compilateurs des langages de programmation et ne doivent pas être donnés à des identificateurs de variables
5. Les identificateurs peuvent être aussi longs que l'on désire, toutefois certains compilateurs ne tiendra compte que des 32 premiers caractères.
6. Le nom d'un variable doit être significatif. On devrait savoir immédiatement, à partir de son nom, à quoi sert la variable ou la constante, et quel sens donner à sa valeur

# AFFECTATION

## Règles d'affectation

L'instruction d'affectation est l'opération qui consiste à attribuer une valeur à une variable pendant l'exécution du programme.

En algorithmique on utilise un opérateur d'affectation.

Nous le notons ← dans le cadre de ce cours.

L'affectation s'effectue de la **droit vers la gauche**

La **valeur** de l'expression **droite** (valeur ou résultat de calcul) sera stocké dans la **variable gauche**.

```
ALGORITHME : Prix_du_pain
Nom : CHAINE
Car : CARACTERE
Nb : ENTIER
Prx, Qt, Tot, Tot1 : REEL
DEBUT
  Nom ← "Mounir"
  Car ← 'g'
  Prx ← 10.25
  Qt ← 6
  Tot ← Prx * Qt
  Tot1 ← Tot
FIN
```

# LECTURE / AFFICHAGE

## Définition

Deux méthodes qui permettent de gérer les entrés et sorties d'un algorithme sont : LIRE et AFFICHER

La méthode LIRE permet de **lire des valeurs à partir du clavier** qui représente l'entrée d'un algorithme. Elle permet de récupérer la valeur taper au clavier et la stocker dans la variable entre les parenthèses.

La méthode AFFICHER permet **d'afficher les messages ou des valeurs de variables à l'écran** qui représente la sortie.

```
ALGORITHME : Prix_du_pain
Nom : CHAINE
Car : CARACTERE
Nb : ENTIER
Prx, Qt, Tot, Tot1 : REEL
DEBUT
  Nom ← "Mounir"
  Car ← 'g'
  Prx ← 10.25
  Qt ← 6
  Tot ← Prx * Qt
  Tot1 ← Tot
FIN
```

# LES COMMENTAIRES

## Définition

Les commentaires servent à donner des explications à une partie de votre pseudo-code.

Un commentaire doit être mis entre **/* et */**, tous ce qui se trouve entre ces deux symboles est considéré comme un commentaire.

Un commentaire n'affecte pas les instructions de votre algorithme.

Le commentaire peut s'écrire sur plusieurs lignes.

```
ALGORITHME : Prix_du_pain
Prx, Qt, Tot : REEL
DEBUT
  AFFICHER("donner la quantité de pains" )
  LIRE(Qt)
  Prx ← 10.25 /* le prix d'unité*/
  Tot ← Prx* Qt /*  ici nous calculons le
                    prix totale*/
  AFFICHER ("le prix total est :",Tot)
FIN
```

# LES OPÉRATIONS 1/5

## Les opérations arithmétiques

Les opérateurs arithmétiques admis sont les suivants :

| Notation en pseudocode | Sens | Remarques |
|---|---|---|
| a+b | Addition | |
| a-b | Soustraction | |
| a*b | Produit | |
| a./b | Division réelle | 5./3 vaut 2.5 |
| a/b | Division entière | 5/3 vaut 2 |
| a%b | Modulo | |

L'opération modulo permet de calculer le reste de la division de a par b, exemple :
17%5=2

| Correct | Incorrect |
|---|---|
| `pi ← 3.14159`<br>`y ← 2 * x`<br>`per ← 2*r*pi` | `π ← 3,14159`<br>`y ← 2(x+y)`<br>`vol ← 4/3π×rayon^3` |

# LES OPÉRATIONS 2/5

## Les opérations relationnelles

- Les opérateurs relationnelles permettent d'exprimer des comparaisons entre deux variables ou expressions.
- Ils sont utilisés **pour évaluer des conditions** .

| Notation en pseudocode | Notation mathématique | Sens |
|---|---|---|
| a<b | a<b | **a** inférieur strictement à **b** |
| a>b | a>b | **a** supérieur strictement à **b** |
| a<=b | a≤b | **a** inférieur ou égale à **b** |
| a>=b | a≥b | **a** supérieur ou égale t à **b** |
| a=b | a=b | **a** égale à **b** |
| a!=b | a≠b | **a** différent de **b** |

# LES OPÉRATIONS 3/5

## Les opérations logiques

- Les opérateurs logiques permettent d ’exprimer des conditions composées de plusieurs opérateurs relationnelles.
- Il est possible de composer plusieurs conditions avec des opérateurs logiques différents :

```
((x>=0)ET(x<=10))OU (x>100)  0<x ET x<10
```

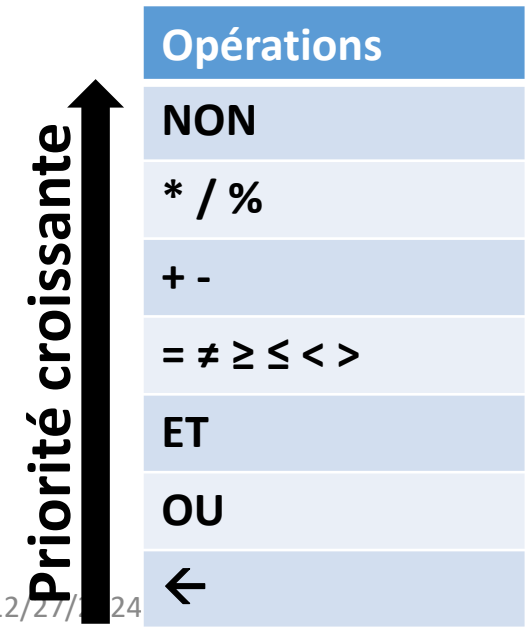
- Les opérateurs logiques admis sont les suivants :

| Notation en pseudocode | Sens | Exemples |
|---|---|---|
| c1 ET c2 | Vrai seulement si c1 et c2 sont tous deux vrais | **(x>=0) ET (x<=10)**<br>**(prix=100) ET (qt>=3) ET (res!=0)** |
| c1 OU c2 | Faux seulement si c1 et c2 sont tous deux faux | **(a<b) OU (c>d)** |
| NON c1 | Vrai seulement si c1 est faux | **NON (a<0)** |

# LES OPÉRATIONS 4/5

## Priorité des opérations

- Il existe une priorité à respecter entre les différents opérateurs que nous venons de voir :




| Opérations |
|---|
| NON |
| * / % |
| + - |
| = ≠ ≥ ≤ < > |
| ET |
| OU |
| ← |

### Exemples

| Expression | Résultat | |
|---|---|---|
| a←5*4+9 | **a←29** | L'operateurs * a plus de priorité que + donc l'opération 5*4 sera évaluée, le résultat 20 sera ensuite additionné a 9, l'affectation est la dernière opération effectuée car sa priorité est la plus faible. |
| 20+x>10 | **Vrai si x>-10** | L'addition + est plus prioritaire que >. |

# LES OPÉRATIONS 5/5

## Usage de parenthèses

- Pourquoi avoir défini une règle de priorité ? C'est pour pouvoir écrire les expressions complexes de manière plus simple, plus lisible. En effet, on peut très bien se passer de ces règles et utiliser systématiquement des parenthèses pour bien rendre compte de l'ordre dans lequel les opérations doivent être effectuées.
- Si nous voulons forcer l'ordinateur à commencer par un opérateur avec une priorité plus faible, nous devons (comme en mathématiques) entourer le terme en question par des parenthèses.

***Exemples***

**X ← 2*(A+3)*B+4*C**

l'ordinateur évalue d'abord l'expression entre parenthèses, ensuite les multiplications, ensuite l'addition et enfin l'affectation

| Expression normale | Expression complètement parenthésée |
|---|---|
| 7 + 2 * 3 | (7 + (2 * 3)) |
| 1 - 2 - 3 | ((1 - 2) - 3) |
| 1 + 2 * 3 / 4 | (1 + ((2 * 3) / 4)) |
| 3 * 5 - 2 + 1 - 8 / 2 * 3 | ((((3 * 5) - 2) + 1) - ((8 / 2) * 3)) |

# EXÉCUTION CONDITIONNELLE 1/12

## Qu’est-ce l’exécution conditionnelle ?

L’exécution conditionnelle permet de faire deux choses différentes selon le cas qui se produit. L’instruction ne sera exécutée que sous certaines conditions. Plus précisément, le programme teste une condition, si la condition est satisfaite le programme fait une chose, dans le cas contraire, le programme fait une autre chose. Une instruction conditionnelle peut avoir plusieurs formes.

## Condition si-alors

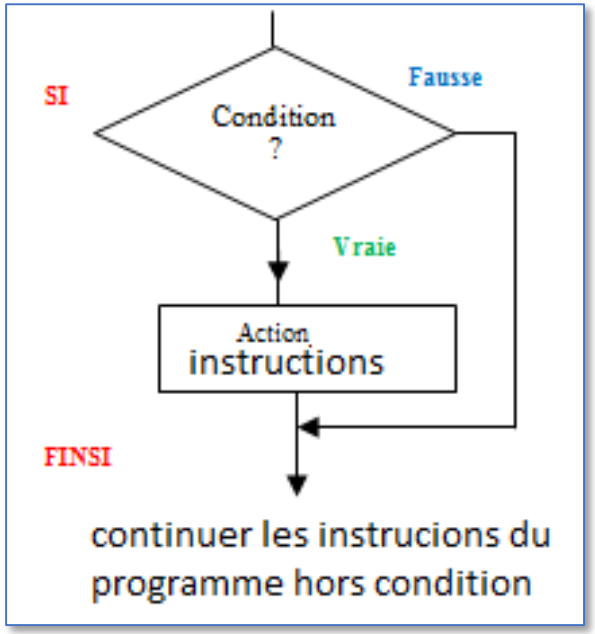
Supposons que l’on ait une condition (par exemple que l’âge du capitaine soit inférieur à 30 ans). Si la condition est vérifiée, on fait quelque chose, dans le cas contraire, on ne fait rien.

En algorithmique, cela s’écrit :

**SI (condition) ALORS**

**Instructions 1**

**FINSI**



# EXÉCUTION CONDITIONNELLE 2/12

## Exemple

L'algorithme ci-contre donne un exemple d'utilisation de SI en fonction d'une condition qui dépende de la valeur de la variable note. Les instructions entre SI et FINSI serons exécutées seulement si la valeurs de note est supérieure ou égale à 12.

les instructions qui sont à l'extérieur de ce bloc serons exécutées indépendamment de cette condition.

```
ALGORITHME note_valide
note :REEL
DEBUT
LIRE (note)
SI (note>=12) ALORS
    AFFICHER("Module valide")
FINSI
AFFICHER("Bon courage!")
FIN
```

Cette instruction ne sera exécutée que si la condition `note>=12` est vrai

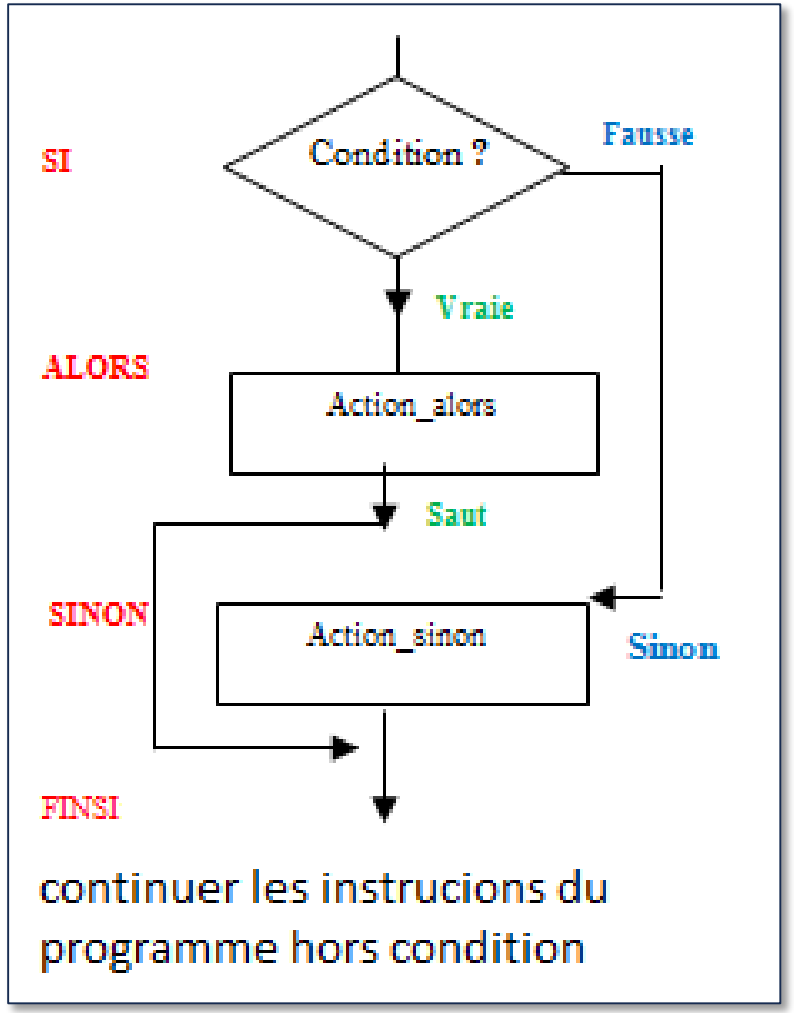
Cette instruction sera exécutée car elle ne dépende pas de la condition. Elle est en d'hors de la structure SI

# EXÉCUTION CONDITIONNELLE 3/12

## Condition alternative : si-alors sinon

Une seconde forme plus intéressant est constituée d'un second bloc d'instructions qui seront exécutées lorsque la condition n'est pas vérifiée. Elle aura pour forme :

```
SI (condition) ALORS
        Instructions 1
SINON
        Instructions 2
FINSI
Instructions 3 /* suite d'instructions du
programme hors condition*/
```



# EXÉCUTION CONDITIONNELLE 4/12

## Exemple

L'algorithme ci-contre donne un exemple d'utilisation de SI en fonction d'une condition qui dépende de la valeur de la variable note. Les instructions entre SI et FINSI serons exécutées que si la valeurs de note est supérieure à 12. les instructions qui sont à l'extérieur de ce bloc serons exécutées indépendamment de cette condition.

```
ALGORITHME note_valide
note :REEL
DEBUT
LIRE (note)
SI (note>=12) ALORS
    AFFICHER("Module valide")
SINON
    AFFICHER("Module NON valide")
FINSI
AFFICHER("Bon courage!")
FIN
```

**Cette instruction ne sera exécutée que si la condition `note>=12` est vrai**

**Cette instruction sera exécutée car elle ne dépende pas de la condition. Elle est en d'hors de la structure SI**

**Cette instruction ne sera exécutée que si la condition `note>=12` est Fausse**

# EXÉCUTION CONDITIONNELLE 5/12

## Conditions imbriquées

Parfois l'expression d'une structure conditionnelle peut contenir plusieurs situations.

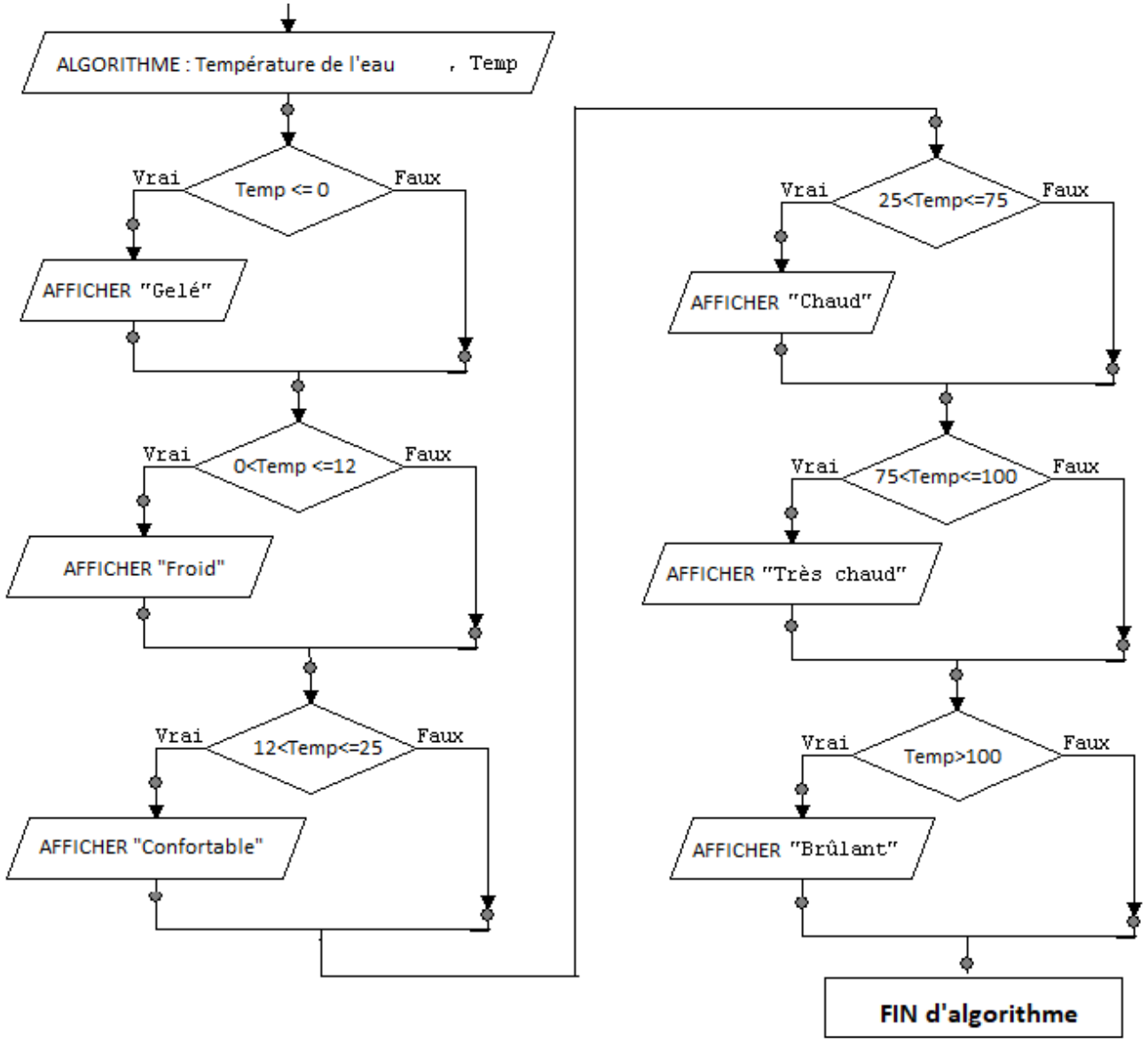
Prenons l'exemple d'un algorithme qui donne l'état de l'eau en fonction de sa température. Il prend en entrée une variable Temp qui représente la température de l'eau et puis il affiche des messages:

- Si la température est inférieure a 0 degrés alors il affiche que l'eau est Gelé.
- Si Temp est entre 0 et 12 degrés alors il affiche que l'eau est Froid.
- Si Temp est entre 12 et 25 degrés alors il affiche que l'eau est Confortable.
- Si Temp est entre 25 et 75 degrés alors il affiche que l'eau est Chaud.
- Si Temp est entre 75 et 100 degrés alors il affiche que l'eau est Très chaud.
- Si Temp est supérieure à 100 degrés alors il affiche que l'eau est Brûlant.

# EXÉCUTION CONDITIONNELLE 6/12

## Conditions imbriquées

**Organigramme de l'algorithme**



# EXÉCUTION CONDITIONNELLE 7/12

## Conditions imbriquées

**L'algorithme de la structure conditionnelle non-imbriquées**

```
ALGORITHME Temp_Eau
Temp :REEL
DEBUT
AFFICHER("Donner la température")
LIRE(Temp)
 SI (Temp <= 0) ALORS
   AFFICHER("C'est gelé")
 FINSI
 SI ((Temp > 0) ET (Temp <= 12)) ALORS
   AFFICHER("C'est froid")
 FINSI
 SI  ((Temp > 12) ET (Temp <= 25)) ALORS
   AFFICHER("C'est confortable")
 FINSI
 SI  ((Temp > 25) ET (Temp <= 75)) ALORS
   AFFICHER("C'est chaud")
 FINSI
```

```
/*suite de l'algorithme*/
 SI ((Temp > 75) ET (Temp <= 100)) ALORS
   AFFICHER("C'est très chaud")
 FINSI
 SI (Temp > 100) ALORS
   AFFICHER("C'est brulant")
 FINSI
FIN
```

# EXÉCUTION CONDITIONNELLE 8/12

## Conditions imbriquées

L'imbrication des conditions consistes à utiliser la structure **SI-ALORS SINON** l'une à l'intérieure d'une autre.

Elle permet de mieux structurer l'algorithme afin que cela soit plus clair, mais surtout à réduire le nombre d'opérations de comparaisons effectuées. Cela permet de réduire la complexité de l'algorithme.

**Question? Combien d'opérations de comparaison sont effectuées par l'algorithme précèdent?**

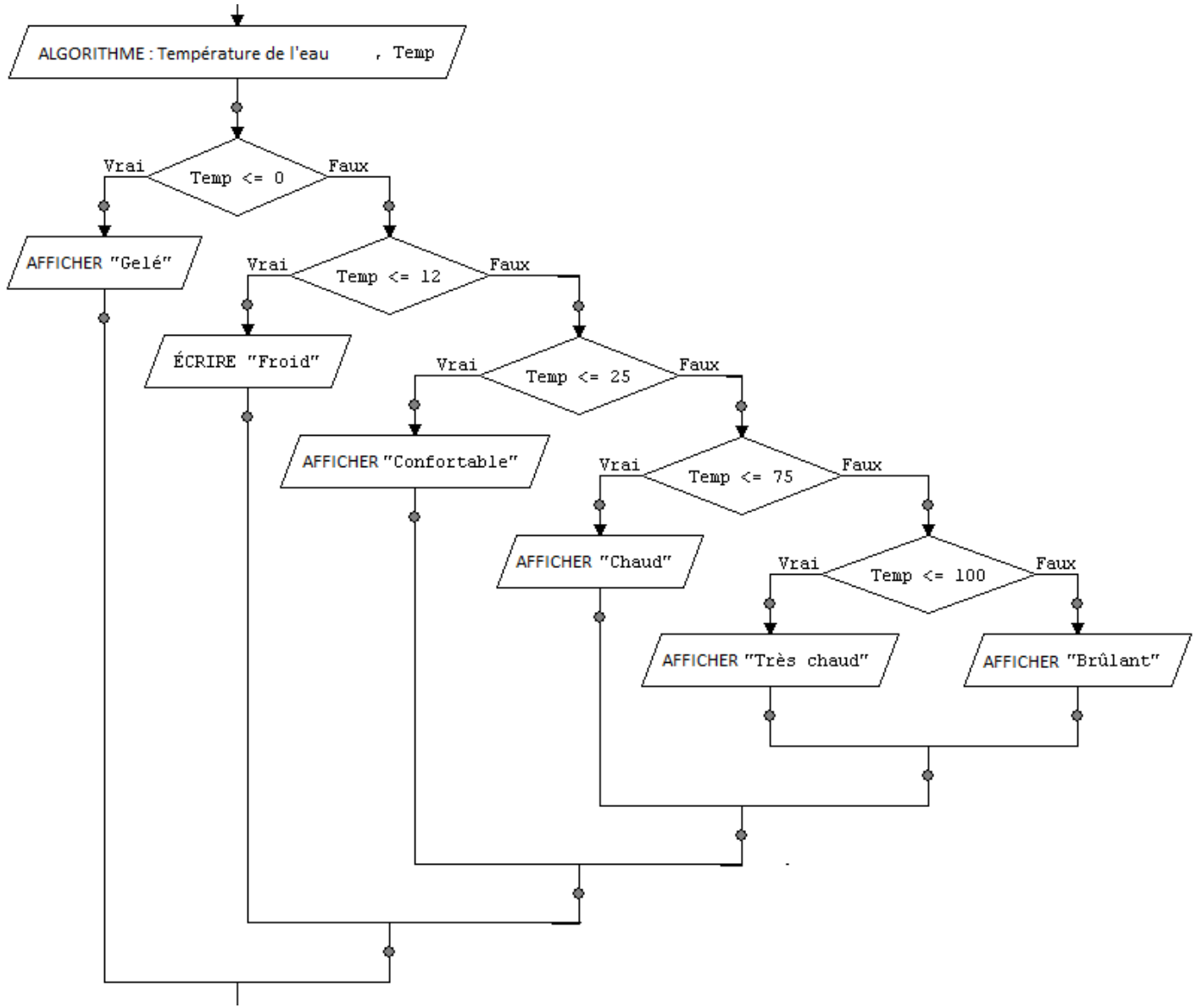
L'algorithme précèdent peut être conçus de manière à utiliser la structure de conditions imbriquées comme nous pouvons voir dans l'organigramme de la page suivante.

```
SI (condition 1) ALORS
        Instructions1
SINON
    SI(condition 2) ALORS
            Instructions2
    SINON
        SI(condition 3) ALORS
                Instructions3
        SINON
                ...
        FINSI
    FINSI
FINSI
```

# EXÉCUTION CONDITIONNELLE 9/12

## Conditions imbriquées

**Organigramme de l'algorithme**



# EXÉCUTION CONDITIONNELLE 10/12

## Conditions imbriquées

```
ALGORITHME Temp_Eau
Temp :REEL
DEBUT
AFFICHER("Donner la température")
LIRE(Temp)
 SI (Temp <= 0) ALORS
   AFFICHER("C'est gelé")
 SINON
   SI (Temp <= 12) ALORS
     AFFICHER("C'est froid")
   SINON
     SI (Temp <= 25) ALORS
       AFFICHER("C'est confortable")
     SINON
       SI (Temp <= 75) ALORS
          AFFICHER("C'est chaud")
       SINON
```

```
  /*suite de l'algorithme*/
         SI (Temp <= 100) ALORS
           AFFICHER("C'est très chaud")
         SINON
           AFFICHER("C'est brûlant")
         FINSI
       FINSI
     FINSI
   FINSI
 FINSI
FIN
```

Étudier combien d'opérations de comparaison sont effectuées ?

# EXÉCUTION CONDITIONNELLE 11/12

## L'instruction SELON - CAS

Lorsque nous somme dans une situation ou l'algorithme doit régler un problème ou plusieurs conditions sont liées à un ensemble de valeurs finis et bien déterminées, est le problème consiste plutôt à une liste de choix.

Il est possible d'utiliser **SI-ALORS** plusieurs fois, mais il existe une structure qui permet de résoudre ce problème plus efficacement.

Voir ci-contre la structure **SELON-CAS**

```
SELON variable

CAS valeur 1 :
Instructions 1

CAS valeur 2 :
Instructions 2

CAS valeur 3 :
Instructions 3
…

SINON :
Instructions
FINSELON
```

# EXÉCUTION CONDITIONNELLE 12/12

## L'instruction SELON - CAS

L'exemple de l'algorithme suivant présente l'utilisation de SELON-CAS pour présenter à l'utilisateur une liste choix d'opération.

Il y'a 3 valeurs (1,2,3) que l'utilisateur doit tapé chacun dirige l'exécution vers le cas sélectionné. Noter que dans le cas ou l'utilisateur introduit une valeur outre que les cas traité l'exécution se dirige directement au bloc SINON

```
ALGORITHME Operations
Choix : ENTIER
a,b,C : REEL
DEBUT
AFFICHER("donner Deux Valeurs a et b")
LIRE(a,b)
AFFICHER("Taper 1 pour Additionner, 2 pour multiplier 3 pour le
modulo de a et b")
LIRE(Choix)
SELON Choix
  CAS 1 :
  C ← a+b
  AFFICHER("le resultat de l'operation choisie =",C)
  CAS 2 :
  C ← a*b
  AFFICHER("le resultat de l'operation choisie =",C)
  CAS 3 :
  C ← a%b
  AFFICHER("le resultat de l'operation choisie =",C)
  SINON :
  AFFICHER("Vous avez tapé un mauvais choix")
FINSELON
FIN
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle POUR

Si nous désirons que l'algorithme répète un bloc d'instructions un certain nombre de fois (**bien déterminé**). Nous utilisons la boucle **POUR.** En pseudo-code cela s'exprime comme suivant :

.

```
instructions 1 /* début du programme */
POUR (ide initial, condition sur ide, changement de ide)
   instructions 2 /* instructions répétées */
FINPOUR
instructions 3 /* suite du programme */
```

Lorsque le nombre de fois où un bloc d'instructions doit être exécuté est connu à l'avance, la boucle **POUR** est préférable. L'usage principal de la boucle **POUR** est de faire la gestion d'un compteur (**ide** de type entier) qui évolue d'une valeur à une autre. La variable **ide** est initialisé par une valeur, ensuite nous mettons une condition sur ide, le changement de ide consiste la plupart du temps à incrémenter ou décrémenter  la valeur de **ide** pour que la condition change au fur et à mesure de l'avancement des calculs.

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle POUR

L'exemple ci-contre présente l'algorithme qui permet de calcul le factoriel de n.

l'idée est d'effectuer des multiplications successives jusqu'à ce que la variable i arrive à la valeur de n la variable i change progressivement avec l'instruction i ← i+1.

```
ALGORITHME factoriel
n,i,F :ENTIER
DEBUT
AFFICHER("Donner la valeur de n")
LIRE(n)
F ← 1     /* initialisation obligatoire */
POUR (i ← 1, 1 i<=n , 3 i ← i+1)
    2 F ← F*i
FINPOUR
AFFICHER("le factoriel =", F)
FIN
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle POUR : Fonctionnement

Lorsque l'ordinateur rencontre cette structure, il procède systématiquement de la manière suivante :

- La variable i, jouant le rôle de compteur, est initialisée par 1 (cet initialisation s'effectue une et une seul fois).
- L'ordinateur teste si la valeur de i est inférieure ou égale à la valeur de n :
  - Si c'est le cas, l'instruction ou le bloc d'instruction est effectué, la variable i jouant le rôle de compteur est augmentée de 1, et répète POUR sans initialiser la valeur de la variable i.
  - Si ce n'est pas le cas, l'instruction ou le bloc d'instruction n'est pas effectuée, et l'ordinateur passe aux instructions suivantes.

```
ALGORITHME factoriel
n,i,F :ENTIER
DEBUT
AFFICHER("Donner la valeur de n")
LIRE(n)
F ← 1     /* initialisation obligatoire
*/
POUR (i ← 1, i<=n ,i ← i+1)
   F ← F*i
FINPOUR
AFFICHER("le factoriel =", F)
FIN
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle TANT QUE

Si nous voulons répété un bloc d'instructions cette fois sans savoir explicitement le nombre de fois que cela va se répéter.

Mais plutôt nous avons une condition lorsqu'elle vrai, on répète les instructions de ce bloc.

Une autre forme de structure de contrôle itérative est proposée : C'est La boucle **TANT QUE**

```
instructions 1 /* début du programme */
TANTQUE (condition) /* Etape 1 test de condition*/
instructions 2 /* instructions répétées */
FINTQ
instructions 3 /* suite du programme */
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle TANT QUE

Lorsque l'ordinateur rencontre cette structure, il procède systématiquement de la manière suivante :

- La condition est testée (on dit aussi évaluée).
- Si la condition est vraie, l'instruction ou les instructions 2 du bloc sont exécutées, et on recommence à l'étape 1) : test de la condition.
- Si la condition est fausse, l'instruction ou les instructions du bloc ne sont pas exécutées et on passe aux instructions suivantes (après la structure de contrôle).

```
instructions 1 /* début du programme */
TANTQUE (condition) /* Etape 1 test de condition*/
instructions 2 /* instructions répétées */
FINTQ
instructions 3 /* suite du programme */
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle TANT QUE

Prenons un exemple. Supposons que l'on veuille faire une fonction qui calcule et affiche ***x^k***, où x est un réel et k est un exposant entier saisi au clavier.

Le résultat doit être un réel.

```
ALGORITHME Puissance
x,P :REEL
k,i :ENTIER
DEBUT
AFFICHER("Donner la valeur de x est k")
LIRE(x,k)
i ← 0     /* initialisation obligatoire */
P ← 1   /*initialisation à 1 calcul du produit*/
TANTQUE (i < k)
   P ← P*x
   i ← i+1   /* progression obligatoire */
FINTQ
AFFICHER("x a la puissance k =", P)
FIN
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle TANT QUE

- Au départ, la variable i vaut 0 et P vaut 0 (initialisation).
- Si k est strictement positif, au départ, la condition d'arrêt i<k est vraie, et on rentre dans TANTQUE.
- À chaque exécution des instructions du TANTQUE, l'instruction i=i+1 est exécutée, ce qui augmente la valeur de la variable i (on parle d'incrémentation lorsqu'on augmente de 1 la valeur d'une variable).
- La condition d'arrêt est alors évaluée avant d'effectuer l'itération suivante. Comme la variable i augmente à chaque itération, au bout d'un moment la condition d'arrêt i<k devient fausse, et la boucle se termine ; on passe à la suite du programme

```
ALGORITHME Puissance
x,P :REEL
k,i :ENTIER
DEBUT
AFFICHER("Donner la valeur de x est k")
LIRE(x,k)
i ← 0     /* initialisation obligatoire */
P ← 1  /*initialisation à 1 calcul du produit*/
TANTQUE (i < k)
   P ← P*k
   i ← i+1  /* progression obligatoire */
FINTQ
AFFICHER("x a la puissance k =", P)
FIN
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle FAIRE - TANT QUE

- La structure TANTQUE que nous venons de voir, commence par vérifier la condition au départ pour ensuite exécuter le bloc d'instructions inclus.

- Parfois il est préférable de laisser la vérification de la condition de répétition à la fin des instructions à répéter pour cela nous avons une autre forme :

  La Boucle **FAIRE TANT QUE**

```
instructions 1 /* début du programme */
FAIRE
instructions 2 /* instructions répétées */
TANTQUE (condition) /* vérification de la
condition de répétition après l'exécution des
instructions2 */

instructions 3 /* suite du programme */
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle FAIRE - TANT QUE

L'algorithme ci-contre calcul le PGCD de 2 nombre es introduit par l'utilisateur en utilisant la boucle FAIRE - TANT QUE

```
ALGORITHME PGCD
a,b,r :ENTIER
DEBUT
AFFICHER("Donner deux nombres m et n ")
LIRE(a,b)
FAIRE
    r ← a%b
    a ← b
    b ← r
TANTQUE (b != 0)
AFFICHER("le PGCD =", a)
FIN
```

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle FAIRE - TANT QUE

Pour comprendre ce que fait cet algorithme, il est préférable de faire des exemples d'exécution d'algorithme et de voir le changement du contenu des variables étape par étape.

Nous appellerons cette simulation d'algorithme dans ce cours : une preuve d'algorithme. Vous aurez souvent à la faire lorsque un algorithme inconnu se présente à vous dans les Tds.

| r | a | b | États de la condition |
|---|---|---|---|
| --- | 260 | 170 | **Introduction de valeur** |
| 90 | 170 | 90 | b!=0 est vrai |
| 80 | 90 | 80 | b!=0 est vrai |
| 10 | 80 | 10 | b!=0 est vrai |
| 0 | 10 | 0 | b!=0 est **faux** |
| | | | FINTQ affichage de a<br>Le PGCD = 10 |

# EXÉCUTION REPETITIVE - ITÉRATION

## Boucle FAIRE - TANT QUE

Pour comprendre ce que fait cet algorithme, il est préférable de faire des exemples d'exécution d'algorithme et de voir le changement du contenu des variables étape par étape.

Nous appellerons cette simulation d'algorithme dans ce cours : une preuve d'algorithme. Vous aurez souvent à la faire lorsque un algorithme inconnu se présente à vous dans les Tds.

| r | a | b | États de la condition |
|---|---|---|---|
| --- | 260 | 170 | **Introduction de valeur** |
| 90 | 170 | 90 | b!=0 est vrai |
| 80 | 90 | 80 | b!=0 est vrai |
| 10 | 80 | 10 | b!=0 est vrai |
| 0 | 10 | 0 | b!=0 est **faux** |
| | | | FINTQ affichage de a<br>Le PGCD = 10 |

# Les tableaux

## Problème

- Manipulation de nombreuses variables représentant des valeurs distinctes mais de même nature.

**Exemple**, si nous avons besoin de traiter simultanément de 10 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer dix variables, appelées par exemple Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc.

```
ALGORITHME Moy_note_1
        N1,N2,N3,N4,N5,N6,N7,N8,N9,N10,Moy: REEL
DEBUT
        AFFICHER("donner Les 10 notes")
        AFFICHER("note1")
        LIRE(N1)
        AFFICHER("note2")
        LIRE(N2)
        AFFICHER("note3")
        LIRE(N3)
        AFFICHER("note4")
        LIRE(N4)
        AFFICHER("note5")
        LIRE(N5)
        AFFICHER("note6")
        LIRE(N6)
        AFFICHER("note7")
        LIRE(N7)
        AFFICHER("note8")
        LIRE(N8)
        AFFICHER("note9")
        LIRE(N9)
        AFFICHER("note10")
        LIRE(N10)

        Moy ← (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10)/10

        AFFICHER("la moyenne est =",Moy)
FIN
```

# Les tableaux

## Solution

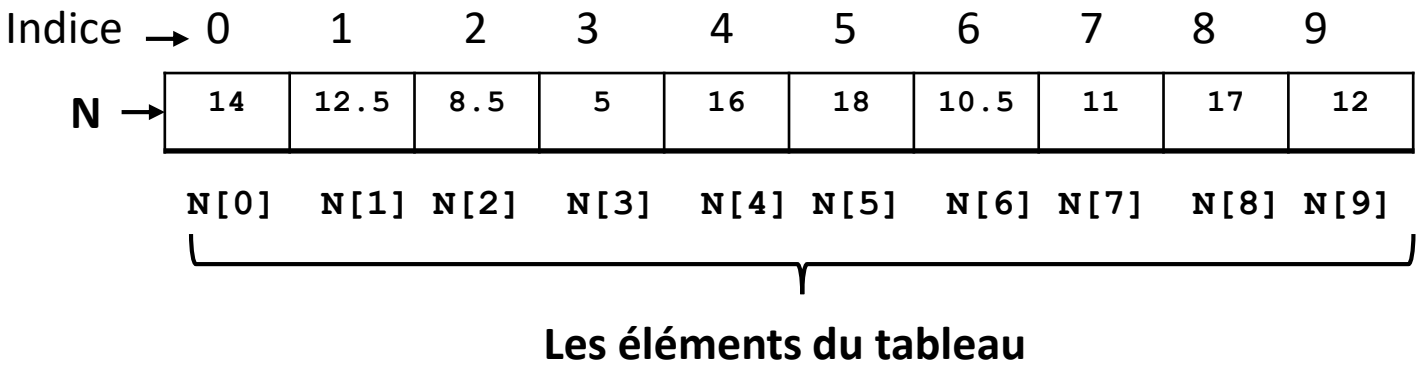
- Les tableaux permettent de rassembler toutes ces variables en une seule, au sein de laquelle chaque valeur sera désignée par un numéro.
- Un tableau est un ensemble de valeurs portant le même nom de variable et repérées par un nombre (indice).
- La déclaration d'un tableau de variables s'effectue par **N[10] : REEL** (10 le nombre d'éléments est **obligatoire** dans la déclaration)
- L'accès au contenus des éléments du tableau s'effectue par in indice entre les crochets **[ ]**.

```
ALGORITHME Moy_note_2
    i,n : ENTIER
    N[40] : REEL /*déclaration d'un
        tableaux de 40 valeurs réels*/
    Moy : REEL
DEBUT
    AFFICHER("Donner le nombre de notes <40")
    LIRE(n)
    AFFICHER("donner Les notes")
    POUR(i ← 0, i<n, i ← i+1)
      AFFICHER("note",i+1)
      LIRE(N[i])
    FINPOUR
    Moy ← 0
    POUR(i ← 0,i<n,i ← i+1)
      Moy ← Moy+N[i]
    FINPOUR
    Moy ← Moy/n
    AFFICHER("la moyenne est =",Moy)
FIN
```

# Les tableaux

## Consignes

- La déclaration d'un tableau de variables s'effectue par **N[10] : REEL** (10 le nombre d'éléments est obligatoire dans la déclaration).
- **L'accès** au contenus des éléments du tableau **s'effectue par l'indice** (un nombre entier) entre les crochets **[ ]**.
- Les **indices** des éléments d'un tableau **commence toujours par 0** et le **dernier élément** dans cet algorithme est **N[9]** (puisque nous avons déclaré exactement 10 éléments).
- Il **ne** faut pas **confondre** la **valeur** notée **entre crochets** lors de la **déclaration** du **tableau** (la taille maximale) et la **valeur** notée **entre crochets lors** des **instructions** (l'indice).



# Les tableaux

## Relation entre tableaux et boucles

Les boucles sont extrêmement utiles pour les algorithmes associés aux tableaux. En effet, de nombreux algorithmes relatifs au tableau nécessitent de parcourir les éléments du tableau dans un certain ordre, le plus souvent dans le sens des indices croissant. Le traitement de chacun des éléments étant souvent le même, seule la valeur de l'indice est amenée à changer. Une boucle est donc parfaitement adaptée à ce genre de traitements.

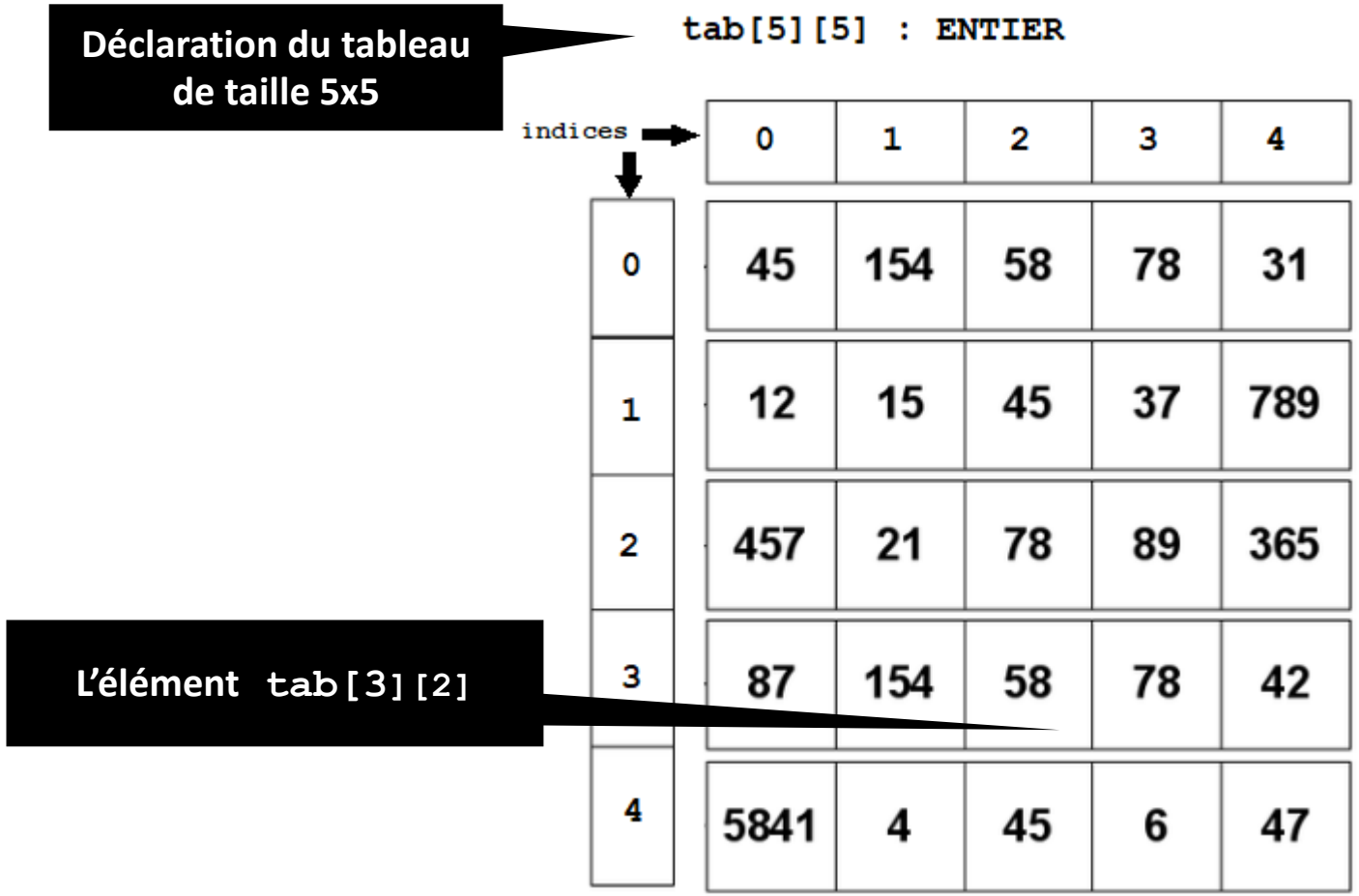
```
ALGORITHME recherche_max
   maxi :ENTIER /* stocke la valeur du maximum*/
   tablval[50] :ENTIER /* un tableau stockant des valeurs*/
   Nb,i,j: ENTIER /* la taille utile du tableau et les indices de
                  parcourt*/
DEBUT
  AFFICHER("entrez le nombre d'elements du tableau ( taille max 50) ")
  LIRE(Nb)
  POUR (i ← 0 , i< Nb , i ← i+1) FAIRE
     AFFICHER("entrez une valeur dans le tableau : ")
     LIRE(tablval[i])
  FINPOUR
    maxi ← tablval[0] /* pour l'instant, le plus grand est dans la case 0
  cherchons case par case (de l'indice 1 à Nb)*/
  POUR (j ← 1 , j< Nb , j ← j+1) FAIRE
    SI (tablval[j] > maxi) ALORS
      maxi ← tablval[j] /* la valeur est mémorisée dans maxi*/
    FINSI
  FINPOUR
  AFFICHER("la valeur maximal de ce tableau: ", maxi)
FIN
```

# Les tableaux

## Multi-dimension

Il est possible de définir des tableaux **à plusieurs dimensions** en les indiquant dans des crochets successifs lors de la définition du tableau. Pour des propos d'illustration l'exemple se limitera à deux dimensions, la généralisation à N dimensions est immédiate. Certains problèmes (notamment le calcul matriciel) ont une représentation naturelle en deux dimensions avec un repérage en lignes/colonnes ou abscisse/ordonnée.

### Tableau à deux dimensions



# Les tableaux

## Multi-dimension

L'exemple ci-contre, montre comment manipuler les élément d'in tableau dans le but de remplir un tableau de 2 dimensions, puis de l'afficher et ensuite calculer la somme de ses éléments.

- Note que Pour parcourir les éléments du tableau tab nous avons besoin de deux boucles imbriquées, chacune a son propre indice de parcourt (i et j), il faut faire attention au sens des indices pour ils ne soient pas mélangés dans les boucles imbriquées.
- Noter aussi que nous pourront réutiliser les indices dans les autres Double boucles, car ils sont indépendantes cette fois-ci.

```
ALGORITHME Moy_note_2
tab[10][10],S : REEL /*déclaration d'un tableaux 2 dimensions de 10x10 valeurs réels*/
i,j,N,M : ENTIER
DEBUT
AFFICHER("entrez le nombre de lignes et le nombre de colonnes ")
LIRE(N,M)
POUR(i ← 0, i<N, i ← i+1)     /* Double boucle 1 :parcourt des éléments pour
  POUR(j ← 0, j<M, j ← j+1)   remplir le tableau tab*/
    LIRE(tab[i][j])
  FINPOUR
FINPOUR

POUR(i ← 0, i<N, i ← i+1) /* Double boucle 2 : parcourt des éléments pour
 POUR(j ← 0, j<M, j ← j+1) afficher le tableau tab*/
  AFFICHER(tab[i][j])
 FINPOUR
FINPOUR
S ← 0
POUR(i ← 0, i<N, i ← i+1) /* Double boucle 3 : parcourt des éléments pour
 POUR(j ← 0, j<M, j ← j+1) calculer la somme des éléments du tableau tab*/
   S ← S+tab[i][j])
 FINPOUR
FINPOUR
AFFICHER("la somme des elements = ",S)
FIN
```

# Les tableaux

## Multi-dimension

L'exemple ci-contre, montre comment manipuler les élément d'in tableau dans le but de remplir un tableau de 2 dimensions, puis de l'afficher et ensuite calculer la somme de ses éléments.

- Note que Pour parcourir les éléments du tableau tab nous avons besoin de deux boucles imbriquées, chacune a son propre indice de parcourt (i et j), il faut faire attention au sens des indices pour ils ne soient pas mélangés dans les boucles imbriquées.
- Noter aussi que nous pourront réutiliser les indices dans les autres Double boucles, car ils sont indépendantes cette fois-ci.

```
ALGORITHME Moy_note_2
tab[10][10],S : REEL /*déclaration d'un tableaux 2 dimensions de 10x10 valeurs réels*/
i,j,N,M : ENTIER
DEBUT
AFFICHER("entrez le nombre de lignes et le nombre de colonnes ")
LIRE(N,M)
POUR(i ← 0, i<N, i ← i+1)     /* Double boucle 1 :parcourt des éléments pour
  POUR(j ← 0, j<M, j ← j+1)   remplir le tableau tab*/
    LIRE(tab[i][j])
  FINPOUR
FINPOUR

POUR(i ← 0, i<N, i ← i+1) /* Double boucle 2 : parcourt des éléments pour
 POUR(j ← 0, j<M, j ← j+1) afficher le tableau tab*/
  AFFICHER(tab[i][j])
 FINPOUR
FINPOUR
S ← 0
POUR(i ← 0, i<N, i ← i+1) /* Double boucle 3 : parcourt des éléments pour
 POUR(j ← 0, j<M, j ← j+1) calculer la somme des éléments du tableau tab*/
   S ← S+tab[i][j])
 FINPOUR
FINPOUR
AFFICHER("la somme des elements = ",S)
FIN
```

# LES SOUS-PROGRAMMES OU FONCTIONS

## Définition

Lorsque l'on progresse dans la conception d'un algorithme, ce dernier peut prendre une taille et une complexité croissante. De même des séquences d'instructions peuvent se répéter à plusieurs endroits.

Un algorithme écrit d'un seul tenant devient difficile à comprendre et à gérer dès qu'il dépasse deux pages. La solution consiste alors à découper l'algorithme en plusieurs parties plus petites. Ces parties sont appelées des sous-algorithmes.

**Le sous-algorithme est écrit séparément du corps de l'algorithme principal et sera appelé par celui-ci quand ceci sera nécessaire.**
**Le sous-algorithme est une partie de l'algorithme presque indépendante qui a un nom et peut être appelée d'un autre sous-algorithme ou de l'algorithme principal.**

# LES SOUS-PROGRAMMES OU FONCTIONS

## Intérêt des sous-algorithmes

### Factorisation du code

Les sous-algorithmes permettent de réutiliser des parties d'un algorithme. Par exemple, pour calculer le factoriel de plusieurs nombres, on peut écrire une fonction pour ce calcul. Cela rend le code plus simple et évite la répétition.

### Mise au point

Une fois qu'un sous-algorithme est écrit, il doit être testé. Cela permet de vérifier chaque partie séparément, facilitant l'identification des erreurs et de leur origine, contrairement à un test de l'algorithme entier d'un seul coup.

### Amélioration de la maintenance

Comme la compréhension d'un algorithme, la maintenance est automatiquement améliorée, car il sera plus facile d'identifier les parties de l'algorithme à modifier et d'en évaluer l'impact. L'idéal est bien entendu que la modification puisse être limitée à un petit nombre de sous-algorithmes.

# LES SOUS-PROGRAMMES OU FONCTIONS

## Définir une fonction

Une fonction doit être définie avant d'être utilisée, c'est-à-dire que l'on doit indiquer quelles sont les instructions qui la composent : il s'agit de la définition de la fonction, où l'on associe les instructions à l'identification de la fonction.

La fonction doit être définie et comporter : un en-tête, pour l'identifier et un corps contenant ses instructions, pour la définir.

```
FONCTION nom_de_fonction(«liste entrées avec leurs types»):type de la sortie
« Déclarations des variables »
DEBUT
« instructions »
RETOURNER (« la valeurs ou la variable à retourner »)
FINFCT
```

# LES SOUS-PROGRAMMES OU FONCTIONS

## Définir une fonction

Une fonction doit être définie avant d'être utilisée, c'est-à-dire que l'on doit indiquer quelles sont les instructions qui la composent : il s'agit de la définition de la fonction, où l'on associe les instructions à l'identification de la fonction.

La fonction doit être définie et comporter : un en-tête, pour l'identifier et un corps contenant ses instructions, pour la définir.

```
FONCTION nom_de_fonction(«liste entrées avec leurs types»):type de la sortie
« Déclarations des variables »
DEBUT
« instructions »
RETOURNER (« la valeurs ou la variable à retourner »)
FINFCT
```

# LES SOUS-PROGRAMMES OU FONCTIONS

## L'entête d'une fonction et la valeur retourner

- L'entête de la fonction : comporte **le nom de la fonction** qui doit être significatif en vérifiant les mêmes règles de nomination des variables.
- Entre les parenthèses : nous mettons **la liste des variables** requises pour que la fonction réalise les tâches désirées. Pour chaque variable de cette liste on doit écrire sont type. (NB : une fonction peut avoir plusieurs variables dans ses arguments, il se peut aussi que la fonction n'ai aucun paramètre, dans se cas on écrit VIDE pour signaler qu'elle ne récupère aucune valeur de l'extérieur. )
- Après les parenthèses : on écrit le **type de la valeur que la fonction doit retourner** avec l'instruction RETOURNER() (NB : Une fonction ne peut retourner qu'une **seule valeur,** ou dans certain cas elle ne retourne rien, dans ce cas on ne met pas le type de retour et on ne met pas l'instruction RETOURNER. )

```
FONCTION nom_de_fonction («liste entrées avec leurs types») : type de la sortie
« Déclarations des variables »
DEBUT
« instructions »
RETOURNER (« la valeurs ou la variable à retourner »)
FINFCT
```

# LES SOUS-PROGRAMMES OU FONCTIONS

## L'appel de la fonction

une fonction nommée « factoriel » qui prend un entier n en paramètre et retourne son factoriel

## La fonction

```
FONCTION factoriel(n : ENTIER) :ENTIER
i,F : ENTIER
DEBUT
F ← 1
SI(n!=0) ALORS
 POUR(i ← 1, i<=n, i ←i+1)FAIRE
    F ← F*i
 FINPOUR
FINSI
RETOURNER (F)
FINFCT
```

## L'appel de la fonction

```
ALGORITHME combinaison /*L'algorithme principale*/
n,f : ENTIER
DEBUT
 AFFICHER("donner un entier : ")
 LIRE(n)
 f ← factoriel(n)
 AFFICHER("le focatoriel de votre entier est :",f)
FIN
```