Algorithmique

Plan :

I. Introduction

La programmation est une démarche qui se déroule en deux phases :

- Phase d'analyse du problème, c.à.d. la recherche d'algorithme ;
- Phase de programmation proprement dite, qui consiste à exprimer le résultat de la première phase dans un langage donné.

Qu'est ce qu'un algorithme ?

D'une façon générale, un algorithme est une suite d'instructions élémentaires, qui une fois exécutée correctement, conduit à un résultat donné.

Exemple : recette de cuisine, mode d'emploi, notice de montage,…

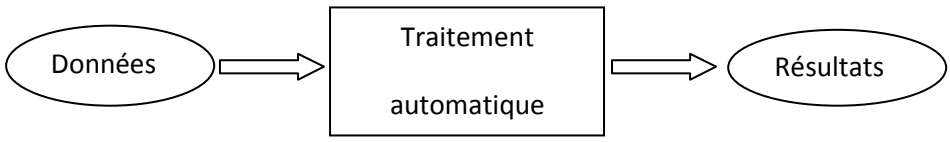
Ainsi, l'ordre d'exécution des instructions est important.

Donc, plus précisément, un algorithme est une description d'un traitement destiné à être réalisé sur un ordinateur pour accomplir une tâche donnée.

Un algorithme est appelé aussi « **pseudo-code** ».

Principe général

Un traitement automatisé consiste à effectuer des opérations sur des informations que l'on peut qualifier de données d'entrée, après le traitement, d'autres informations appelées résultats (ou données de sortie) sont générées.



entrée **sortie**

Exemple :

Considérons le traitement qui consiste à calculer la moyenne d'un groupe d'étudiants pour un module donné. Dans ce cas :

- Pour effectuer le calcul de la moyenne, on a besoin de la note de chaque étudiant dans le module en question ;
- Ensuite, on effectue le calcul comme suit :
    - On calcule la somme des notes,
    - On divise par le nombre d'étudiants,
- Enfin, on affiche le résultat (la moyenne).

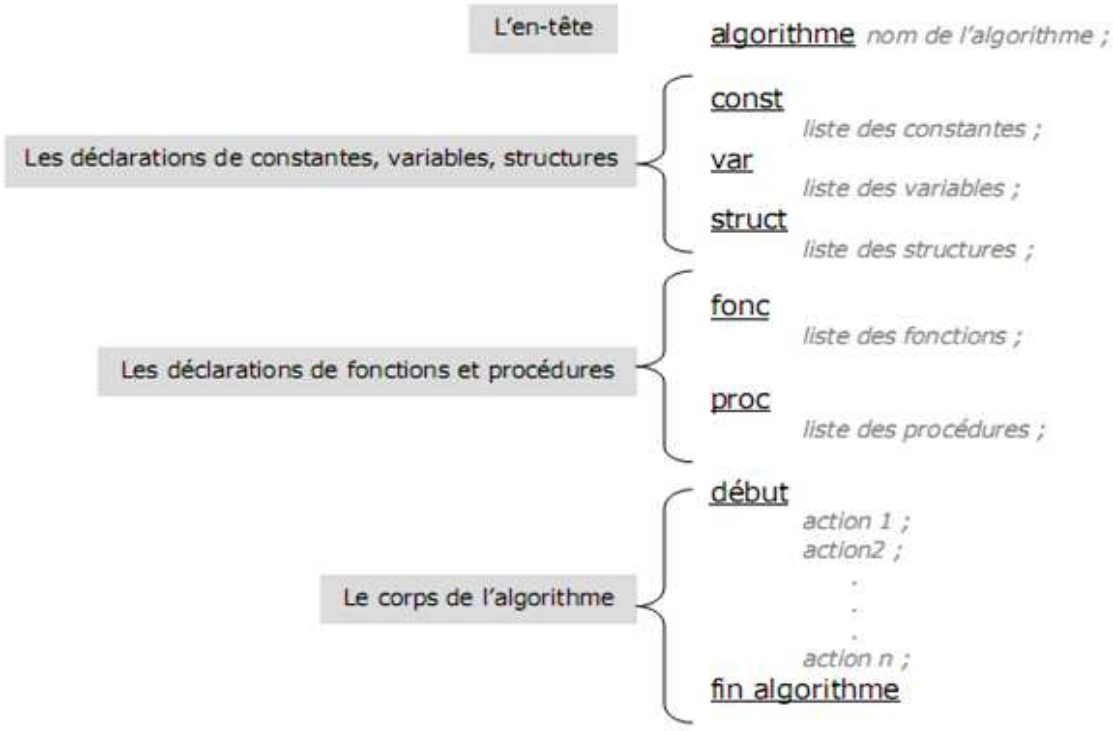
Généralement, lorsqu'on cherche un algorithme permettant d'automatiser un traitement donné, on doit se poser trois questions :

- Qu'est ce qu'on doit obtenir comme résultat ?
- Quelles sont les données dont on a besoin ?
- Comment faire (traitement proprement dit) ?

Comment écrire un algorithme ?

Un algorithme exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage.

Il a généralement la structure suivante :



II. Notions de base

Quatre notions de base en algorithmique et en programmation vont être étudiées, à savoir : la notion de variables et de constantes, la notion d'affectation et les deux notions d'écriture et de lecture.

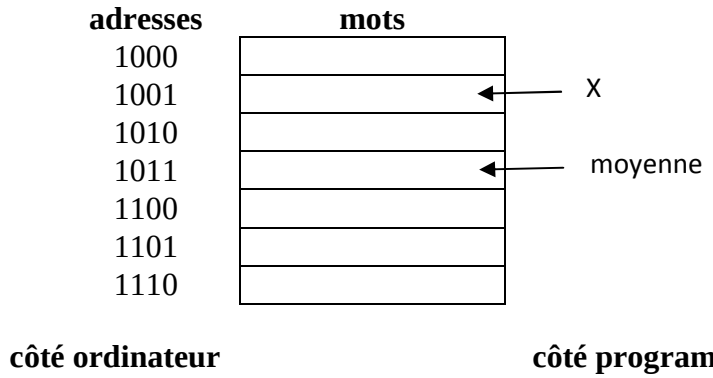
1. Les variables et les constantes

Avant d'entamer ces deux notions, il serait nécessaire de comprendre le fonctionnement de la mémoire d'ordinateur.

Physiquement, la mémoire vive de l'ordinateur est formée d'éléments dont chacun ne peut prendre que deux états distincts. Ce sont les bits d'information. Ces bits sont manipulés par groupes de 8 bits (octets), ou plus (mots de 16, 32, 64,… bits selon les ordinateurs).

La mémoire est donc formée de **mots** (cases mémoire) et pour que l'unité centrale puisse y placer une information et la retrouver, chaque mot est repéré par une **adresse**.

Dans un langage de programmation, les adresses mémoire sont représentées par des noms. Le programmeur ne connait pas donc l'adresse d'une case mais plutôt son nom. Il y a donc deux façons de voir la mémoire centrale de l'ordinateur : côté programmeur et côté ordinateur.



1.1. Variable

Elle peut stocker des nombres, des caractères, des chaînes de caractères,… dont la valeur peut être modifiée au cours de l'exécution de l'algorithme. Mot clé : **var**

1.2. Constante

Elle représente des nombres, des caractères, des chaînes de caractères,… dont la valeur ne peut pas être modifiée au cours de l'exécution de l'algorithme. Mot clé : **const**

Les variables et les constantes sont définies dans la partie déclarative par deux caractéristiques essentielles :

- **L'identificateur** : c'est le nom de la variable ou de la constante, il est composé de lettres et de chiffres sans espaces.
- **Le type** : il définit la nature de la variable ou de la constante (entier, réel, caractère,...)

1.3. Types de base

Les variables et les constantes peuvent avoir cinq types de base :

a) **Type entier :** un type numérique qui représente l'ensemble des entiers naturels et relatifs, tels que : 0, 45, -10,…
   Mot clé : **entier**
b) **Type réel :** un autre type numérique qui représente les nombres réels, tels que : 0.5, -3.67, $1.5e^{+5}$,…
   Mot clé : **réel**
c) **Type caractère** : représente tous les caractères alphanumériques tels que : ′a′, ′B′, ′*′, ′9′, ′@′, ′ ′,…
   Mot clé : **car**
d) **Type chaînes de caractères** : concerne des chaines de caractères tels que des mots ou des phrases : ″informatique″, ″la section B″,…
   Mot clé : **chaîne**
e) **Type booléen :** ce type ne peut prendre que deux états : vrai ou faux
   Mot clé **: booléen**

Exemple de déclaration :

```
var   A : entier
      moyenne, note1, note2 : réel
      nom : chaîne
      lettre : car
const  n = 5
       arobase= ′@′
       e = ″425″
```

2. L'affectation

C'est une instruction simple qui permet d'affecter une valeur à une variable

Syntaxe : variable ← expression ;

Une instruction d'affectation se fait toujours en deux temps : évaluation de l'expression située à droite, et affectation du résultat à la variable située à gauche.

L'expression est une suite d'opérations sur des constantes et des variables déjà déclarées.

Exemples : a ← 15 ; b ← a

somme ← a + b

2.1. Opérateurs

Un opérateur est un signe qui peut relier deux valeurs pour produire un résultat. Un opérateur dépend du type des valeurs qu'il relie.

a- Opérateurs arithmétiques (numériques) :

+	addition
-	soustraction
*	multiplication
/	division
mod	modulo
^	puissance

b- Opérateurs de comparaison :

>	supérieur
<	inférieur
>=	supérieur ou égal
=<	inférieur ou égal
=	égal
≠	différent

c- Opérateurs logiques (booléens):

et	fonction et
ou	fonction ou
non	fonction non
non et	fonction non et
non ou	fonction non ou

d- Opérateur de concaténation : &

Cet opérateur permet de « concaténer » ou relier deux chaines de caractères.

Exp. ″bon″ & ″jour″ = ″bonjour″

Remarque :

En algorithmique, le signe de l'affectation est le signe ←. Mais en pratique, la plupart des langages de programmation emploient le signe égal. Et là, la confusion avec les maths est également facile. En maths, A = B et B = A sont deux propositions strictement équivalentes. En informatique, absolument pas, puisque cela revient à écrire A ← B et B ← A, deux choses bien différentes.

3. Les notions de lecture et d'écriture

Supposons qu'on veut calculer le carré d'un nombre (5 par exemple), l'algorithme le plus simple serait de ce genre :

Algorithme carré

Var a : entier ;

Début

a ← 5 ^ 2

Fin

Mais, si on veut calculer le carré d'un autre nombre que 5, est-ce que ça serait intéressant de réécrire l'algorithme ? !

D'autre part, le carré est calculé et affecté à la variable « a », mais l'utilisateur ne peut jamais connaitre ce résultat ! (puisque tout se fait dans la mémoire).

Pour cela, il existe des instructions permettant à la machine de communiquer avec son utilisateur.

Dans un sens, elles permettent à l'utilisateur de fournir des données par le clavier pour qu'elles soient utilisées par le programme. C'est l'instruction de **lecture**.

Dans l'autre sens, elles permettent au programme de communiquer des valeurs à l'utilisateur en les affichant à l'écran. C'est l'instruction d'**écriture**.

3.1. L'instruction de lecture

Permet de recevoir une valeur (par le clavier, par exemple) et l'attribue à une variable.

Mot clé : **Lire**

3.2. L'instruction d'écriture

Permet d'afficher une valeur d'une variable sur l'écran.

Mot clé : **Ecrire**

Exemple de lecture et d'écriture :

```
Algorithme produit
Var x,y,z : entier
Début
   Ecrire ″Donnez valeur de x :″
   Lire x
   Ecrire ″Donnez valeur de y :″
   Lire y
   z ← x * y
   Ecrire z
Fin
```

Exercice :

Ecrire un algorithme qui permet d'échanger les valeurs de deux variables a et b.

Solution :

Algorithme permutation

Var a,b,c : réel

Début

Ecrire ″Donnez valeur de a :″
Lire a
Ecrire ″Donnez valeur de b :″
Lire b
c ← a
a ← b
b ← c
Ecrire c

Fin

La variable c est une variable temporaire qu'on est obligé de l'utiliser, elle doit être déclarée de même type que a et b.

Les trois variables peuvent être déclarées d'un autre type (entier, booléen,…)

III. Les tests

Un algorithme (ou programme) comporte deux types d'instructions :

- Les instructions de base : qui permettent la manipulation de variables telles que l'affectation, la lecture et l'écriture.
- Les instructions de structuration d'un programme : qui précisent l'enchainement chronologique des instructions de base.

Dans les notions précédentes, on a vu des algorithmes dans lesquels les instructions s'exécutent dans l'ordre de leur écriture (exécution séquentielle). Mais, la puissance d'un programme (algorithme) provient du fait qu'il peut effectuer des choix dans la façon d'exécuter les instructions.

Par exemple, dans un programme de calcul d'impôts, le montant à payer diffère selon le revenu déclaré. Le programme doit donc être capable d'appliquer à chaque tranche de revenu, un taux différent. Il effectuera par conséquent des choix sur les opérations à exécuter en fonction du revenu.

Donc, il s'agit du deuxième type d'instructions appelées structures conditionnelles, ou structures alternatives ou tout simplement tests.

1. Structure d'un test

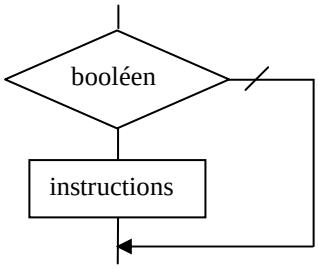
Il existe deux formes possibles pour un test : la forme réduite et la forme complète.

a- Forme réduite :

Dans cette structure, seule la situation correspondant à la validation de la condition entraine l'exécution des instructions, l'autre situation conduisant systématiquement à la sortie de la structure.

Syntaxe :

Si booléen **Alors**
  instructions
Finsi

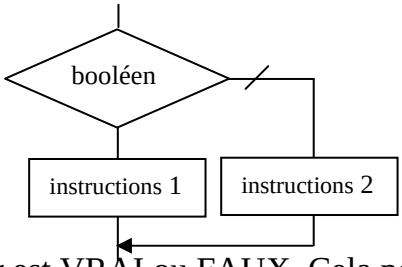


b- Forme complète :

Dans cette structure, l'exécution d'un des deux traitements distincts ne dépend que du résultat du test effectué sur la condition. Si cette dernière est vérifiée, seul le premier traitement est exécuté ; si elle n'est pas vérifiée, seul le deuxième traitement est exécuté.

Syntaxe :

Si booléen **Alors**
  instructions 1
Sinon
  instructions 2
Finsi



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

- une variable de type booléen

- une condition

2. Qu'est ce qu'une condition ?

Une condition est tout simplement une comparaison. Cela signifie qu'elle est composée de trois éléments :

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

A noter que les deux valeurs d'une comparaison doivent être du même type.
L'ensemble des trois éléments de la condition constitue donc, une assertion, qui à un moment donné est VRAIE ou FAUSSE.

Exemples :

5 < 10	Vrai
10 > 10	Faux
'a' < 'd'	Vrai
'a' < 'D'	Faux

Remarque
En formulant une condition dans un algorithme, il faut faire attention à certaines notations qui sont valides en mathématiques, mais qui mènent à des non-sens informatiques. Prenons par exemple la phrase « 7 est compris entre 5 et 8 ». On peut la traduit par : 5 < 7 < 8
Or, une telle expression, qui a du sens en mathématiques, ne veut rien dire en programmation. En effet, elle comprend deux opérateurs de comparaison et trois valeurs. Pour cela, on utilise ce qu'on appelle les **conditions composées** pour traduire convenablement une telle condition.

3. Conditions composées

Une condition est dite composée (ou complexe) lorsqu'elle est formée de plusieurs conditions simples reliées par des opérateurs logiques. Reprenons l'exemple : « 7 est compris entre 5 et 8 ». En fait, cette expression contient deux conditions et non pas une, et on peut dire que « 7 est supérieur à 5 et 7 est inferieur à 8 ». Donc, il y a bien deux conditions simples reliées par l'opérateur logique ET.

Exemple :

si X>0 **et** Y<10 **alors** …
si X>0 **ou** Y>0 **alors**…

Dans le premier cas, la condition est vraie si les deux conditions simples sont vraies. Il suffit que l'une des deux soit fausse pour que la condition soit fausse.

Dans le deuxième cas, la condition est vraie si l'une au moins des deux conditions simples est vraie.

4. Tests imbriqués

Un test avec **Si** ouvre donc deux voies, correspondant à deux traitements différents. Mais dans certains cas, ces deux voies ne suffisent pas pour toutes les solutions possibles.

Par exemple, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles (solide, liquide ou gazeux).

Une première solution serait la suivante :

```
Variable Temp : entier
Début
    Ecrire "Entrez la température de l'eau :"
    Lire Temp
    Si Temp =< 0 Alors
        Ecrire "Etat solide"
    FinSi
    Si Temp > 0 Et Temp < 100 Alors
        Ecrire "Etat liquide"
    Finsi
    Si Temp >= 100 Alors
        Ecrire "Etat gazeux"
    Finsi
Fin
```

Une autre solution consiste à imbriquer les tests de la manière suivante :

```
Variable Temp : entier
Début
    Ecrire "Entrez la température de l'eau :"
    Lire Temp
    Si Temp =< 0 Alors
        Ecrire "Etat solide"
    Sinon Si Temp < 100 Alors
            Ecrire " Etat liquide"
        Sinon
            Ecrire "Etat gazeux"
        Finsi
    Finsi
Fin
```

Donc, on peut constater que l'utilisation des tests imbriqués a les avantages suivants :
- Economie de l'édition du programme, au lieu de devoir taper trois conditions, dont une est composée, nous n'avons plus que deux conditions simples.

Donc, un programme plus simple et plus lisible.

- Economie de temps d'exécution du programme, si le premier test est vrai, le programme passe directement à la fin, sans tester le reste qui est forcément faux.

Donc, un programme plus performant à l'exécution.

Remarque :

1- On peut remplacer les conditions dans un test par des variables booléennes.

Exemple :

```
Variables Temp : entier
          A, B : booléen
Début
    Ecrire "Entrez la température de l'eau :"
    Lire Temp
    A ← Temp =< 0
    B ← Temp < 100
    Si A Alors
            Ecrire "Etat solide"
    Sinon Si B Alors
            Ecrire " Etat liquide"
          Sinon
            Ecrire "Etat gazeux"
          Finsi
    Finsi
Fin
```

2- Toute structure de test avec condition composée faisant intervenir l'opérateur ET peut être exprimée de manière équivalente avec l'opérateur OU, et réciproquement.

Exemple :

```
Si A ET B Alors                    Si NON A OU NON B Alors
   instructions 1                     instructions 2
Sinon            est équivalent à  Sinon
   instructions 2                     instructions 1
Finsi                              Finsi
```

5. La structure de choix (choix multiples)

La structure de choix permet, en fonction de plusieurs conditions de type booléen, d'effectuer des actions différentes suivant les valeurs d'une seule variable.

Syntaxe :

Suivant (valeur ou expression) **faire**

Cas valeur1 : action1 ;

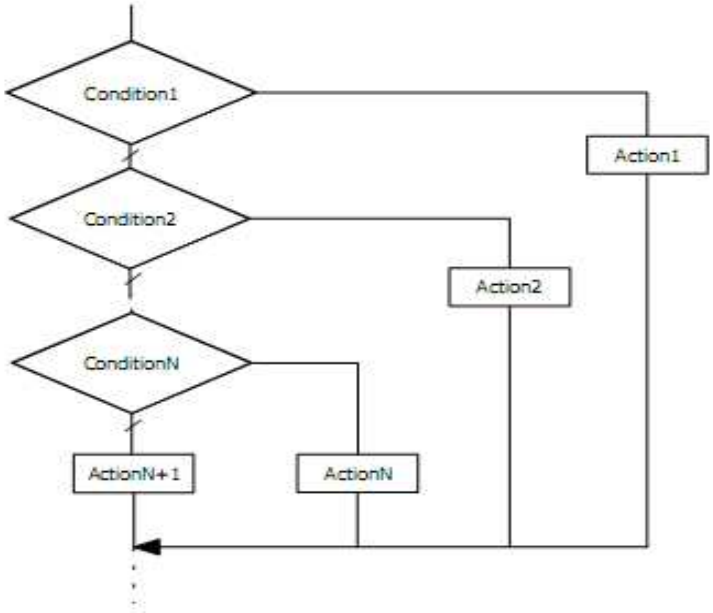
Cas valeur2 : action2 ;

…

Cas valeurN : actionN ;

Autres cas : actionN+1 ;

Finsuivant ;



Exemple :

Soit une variable **Numéro** comprise entre 1 et 7. En fonction de la valeur de cette variable on peut écrire le jour de semaine correspondant :

Variable Num : entier ;

Début

Ecrire "Donner le numéro du jour :" ;

Lire Num **;**

Suivant Num **faire**

Cas 1 : **Ecrire** "C'est le Dimanche" ;

Cas 2 : **Ecrire** "C'est le Lundi" ;

Cas 3 : **Ecrire** "C'est le Mardi" ;

Cas 4 : **Ecrire** "C'est le Mercredi" ;

Cas 6 : **Ecrire** "C'est le Jeudi" ;

Cas 6 : **Ecrire** "C'est le Vendredi" ;

Cas 7 : **Ecrire** "C'est le Samedi" ;

Autres cas : **Ecrire** "Le numéro n'est pas comprise entre 1 et 7 !" ;

Finsuivant ;

Fin

IV. Les boucles

On appelle boucle ou structure itérative tout ensemble d'instructions qui peut être exécuté plusieurs fois. Par exemple, un programme de calcul de la paye répètera pour chaque employé les mêmes instructions pour établir une fiche de paye.

Généralement, on peut considérer deux cas pour les boucles. Dans le premier cas, le nombre de répétitions n'est pas connu ou est variable, il existe deux structures pour ce cas : la structure « Répéter… jusqu'à » et la structure « Tant que … faire ». Dans le deuxième cas, le nombre de répétitions est connu, la structure utilisée est « Pour ».

1. La boucle « Répéter… jusqu'à »

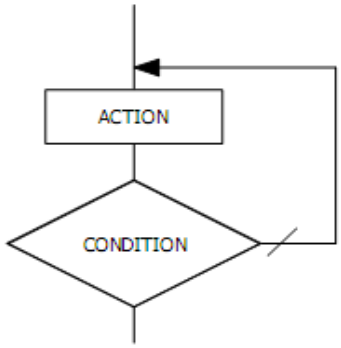
Dans cette structure, le traitement est exécuté une première fois puis sa répétition se poursuit jusqu'à ce que la condition soit vérifiée.

Syntaxe:

répéter

action ;

jusqu'à condition ;



L'action dans cette boucle est toujours exécutée au moins une fois.

Exemple : soit l'algorithme suivant

```
Variables n, p : entier ;
Début
   répéter
      Ecrire "Donner un nombre :" ;
      Lire n ;
      p ← n*n ;
      Ecrire p;
   jusqu'à n=0
   Ecrire "Fin de l'algorithme" ;
Fin
```

Les instructions encadrées par les mots **répéter** et **jusqu'à** constitue le bloc de la boucle qu'il faut répéter jusqu'à ce que la condition **n=0** soit vérifiée. Donc le nombre de répétitions de cette boucle dépend des données fournies par l'utilisateur.

Remarque :

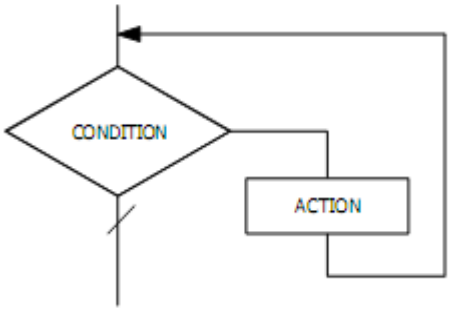
Dans la structure « répéter… jusqu'à », la condition telle qu'elle est exprimée ci-dessus, constitue une condition d'arrêt de la boucle ; mais réellement, cela diffère selon le langage de programmation utilisé. Par exemple, en langage Pascal, la condition de la boucle « répéter… jusqu'à » est une condition d'arrêt. Alors qu'en langage C, cette condition est exprimée en tant qu'une condition de continuation.

2. La boucle « Tant que… faire »

Dans cette structure, on commence par tester une condition, si elle est vérifiée, le traitement est exécuté.

Syntaxe:

tant que condition **faire**
    action ;
ftant que



L'action dans cette boucle peut ne jamais être exécutée.

Exemple : reprenons l'exemple précédent avec cette boucle :

Variables n, p : entier ;
Début
  tant que n ≠ 0 **faire**
    Ecrire "Donner un nombre :" ;
    Lire n ;
    p ← n*n ;
    Ecrire p;
  ftant que
  Ecrire "Fin de l'algorithme" ;
Fin

3. La boucle « Pour »

Cette structure est utilisée lorsqu'on sait exactement combien de fois on doit répéter un traitement. Donc, la boucle s'arrête si le nombre souhaité d'itérations est atteint.

Cette structure utilise une variable (**indice**) de contrôle d'itérations caractérisée par :

- sa valeur initiale,
- sa valeur finale,
- son pas de variation.

Si la valeur initiale de l'indice est **inférieure** à sa valeur finale le pas de variation est **positif** et la structure est dite « **croissante** ». Dans le cas contraire, le pas est **négatif** et la structure est dite « **décroissante** ».

Syntaxe:

pour variable **de** début **à** fin **pas** n

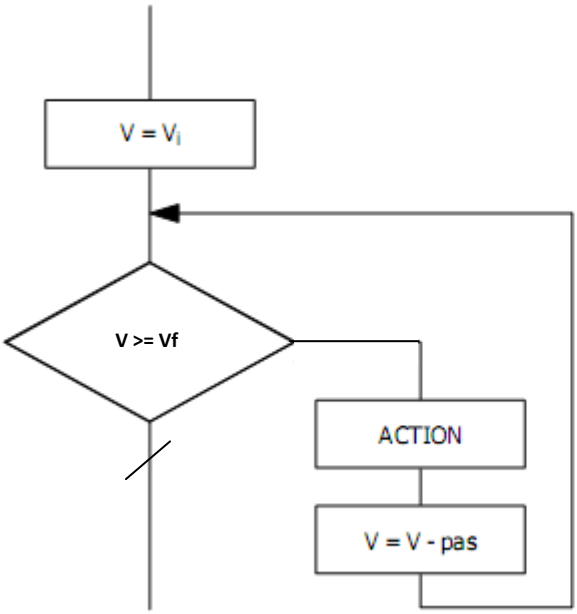
action;

fpour

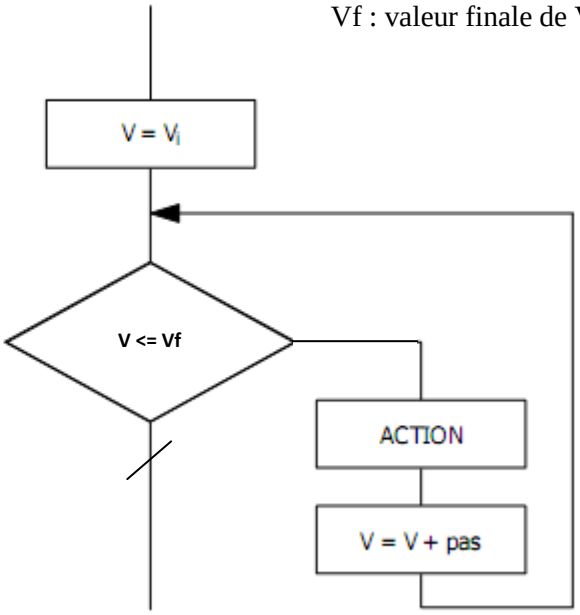
V : variable

Vi : valeur initiale de V

Vf : valeur finale de V



structure décroissante



structure croissante

Exemple :

Soit l'algorithme suivant qui compte jusqu'à 100.

Variables n, p : entier ;
Début
pour i **de** 1 **à** 100 */* équivalent à : **pour** i **de** 1 **à** 100* **pas** 1 */*
Ecrire i;
fpour
Fin

Remarque : si on ne précise pas la valeur du « **pas** » dans la boucle « pour » ; celle-ci fonctionne par défaut avec un pas de 1.

4. Notion de compteur

Un compteur est généralement associé à une boucle. C'est une variable dont la valeur est augmentée de 1 à chaque passage dans la boucle. Il sert donc à compter le nombre de fois que la boucle est exécutée.

Particulièrement, le compteur est associé aux structures itératives : « **répéter…jusqu'à** » et « **tant que …faire** ». Dans la structure « **pour** » la variable indiciaire (l'indice) joue le rôle du compteur.

On peut représenter l'utilisation d'un tel compteur dans les boucles « répéter…jusqu'à » et « tant que …faire » respectivement comme suit :

```
compt ← 0 ;
répéter
   instruction
   …
   compt ← compt +1 ;
jusqu'à condition
```

```
compt ← 0 ;
tant que condition faire
   instruction
   …
   compt ← compt +1 ;
ftant que
```

On remarque que la variable « compt »**,** utilisé ci-dessus comme compteur, a été initialisée à zéro (0) avant le début de chaque boucle. Donc, il faut toujours initialiser le compteur avant de commencer le comptage.

L'instruction d'affectation « compt ← compt +1 » signifie : augmenter la valeur de « compt » de un (1). Elle peut être placée n'importe où, l'essentiel qu'elle soit à l'intérieur de la boucle.

5. Notion d'accumulation

L'accumulation est une notion fondamentale en programmation. Elle est utilisée notamment pour effectuer la somme d'un ensemble de nombres.

Une instruction d'accumulation est une affectation qui se présente comme suit :

variable ← variable + valeur

Tel qu'il apparaît dans cette instruction, accumuler consiste à ajouter une valeur à une variable numérique, puis affecter le résultat dans la variable elle-même. En d'autres termes, la nouvelle valeur de variable égale à l'ancienne plus une certaine valeur.

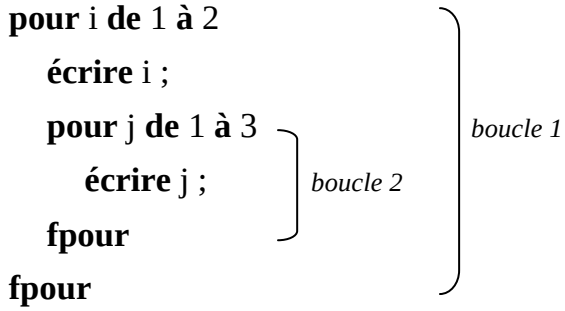
Exemple : calculer la somme de 10 valeurs saisies par l'utilisateur :

```
variables  i : entier ; som, val : réel ;
Début
   som ← 0 ;
   pour i de 1 à 10
         écrire "Enter une valeur :" ;
         lire val ;
         som ← som + val ;
   fpour
   écrire "la somme égale à :", som ;
Fin
```

6. Les boucles imbriquées

On parle de boucles imbriquées lorsqu'une boucle contient elle-même une autre boucle. Les deux boucles peuvent être les mêmes ou différentes.

Exemple :



Dans cet exemple, à chaque passage dans la boucle 1, le programme va exécuter la boucle intérieure (boucle 2) jusqu'à la fin avant de passer à l'étape suivante de la boucle 1, et ainsi de suite jusqu'à la fin des deux boucles.

Donc, généralement, l'exécution de la boucle intérieure est faite autant de fois qu'il y a d'étapes pour la boucle extérieure.

Le résultat d'exécution, de l'exemple ci-dessus, est le suivant :

1
1
2
3
2
1
2
3

Remarque :

Des boucles peuvent être successives ou imbriquées, mais elles ne peuvent jamais être croisées. Par exemple, l'algorithme suivant est faux puisqu'il comporte deux boucles croisées :

```
variables i, j : entier ;
Début
   i←1 ;
   répéter
      écrire i ;
      pour j de 1 à 3
         écrire j ;
   i←i+1 ;
   jusqu'à i>2
      fpour
Fin
```

V. Les tableaux

1. Qu'est ce qu'un tableau ?

Un tableau est un ensemble de variables de même type ayant toutes le même nom.

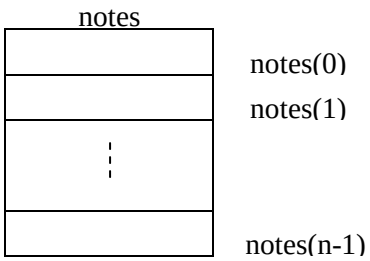
Suite à cette définition, deux questions se posent :

- Quelle est l'utilité d'avoir un ensemble de variables de même nom ?
- Et comment peut-on différencier entre ces variables ?

Alors, supposons qu'on a besoin de stocker et de manipuler les notes de 50 étudiants. On doit, par conséquent, déclarer 50 variables nommées par exemple : note1, note2,…, note50 ou pour simplifier : n1, n2,…, n50. Vous pouvez remarquer que c'est un peu lourd de manipuler une cinquantaine de variables (avec 50 fois de lecture et d'écriture…). Imaginons maintenant le cas pour une promotion de 1000 étudiants, alors là devient notre cas un vrai problème.

C'est pourquoi l'algorithmique (et la programmation) nous permet de regrouper toutes ces variables en une seule, par exemple **notes**. Un seul nom vaut mieux que 50 ou 1000 noms. Mais comment les différencier ?

La réponse est dans la notion de tableau lui-même où chaque élément de ce tableau est repéré par un numéro qui s'appelle l'**indice**. Ce dernier permet de différencier une variable d'une autre. En effet, elles ont toutes le même nom, mais pas le même indice. Donc, chaque fois qu'on doit désigner un élément du tableau, on utilise le nom de ce tableau, suivi de l'indice de l'élément entre parenthèses. On parle ici de tableau à une dimension.



2. Tableaux à une dimension

Un tableau à une dimension est un tableau dont on accède aux éléments (pour lire ou modifier leur valeur) en utilisant un seul indice. Un tableau est donc caractérisé par :

- Le type et le nombre des éléments qu'il contient.
- L'indice pour accéder à ces éléments de type entier généralement.

2.1. Déclaration d'un tableau

Syntaxe :

tableau nom du tableau (dimension) : **type**

Exemple : tableau notes (50) : **réel**

Remarque :

L'indice qui sert à désigner les éléments d'un tableau peut être exprimé directement comme un nombre, mais il peut être aussi une variable, ou une expression calculée.

Dans un tableau, la valeur d'un indice doit toujours :

- **être égale au moins à 0** : dans quelques langages, le premier élément d'un tableau porte l'indice 1(comme en Pascal). Mais nous avons choisi ici de commencer la numérotation des indices à zéro, comme c'est le cas en langage C. Donc attention, notes(1) est le deuxième élément du tableau notes.
- **être un nombre entier** : quel que soit le langage, l'élément notes(1,…) n'existe jamais.
- **être inférieure ou égale au nombre d'éléments du tableau** (moins 1, si on commence la numérotation à zéro). En langage C, si un tableau T est déclaré comme ayant 25 éléments, la présence dans une ligne, sous une forme ou sous une autre, de T(25) déclenchera automatiquement une erreur.

2.2. Manipulation d'un tableau

Une fois déclaré, un tableau peut être utilisé comme un ensemble de variables simples. Les trois manipulations de base sont l'affectation, la lecture et l'écriture.

a- L'affectation :

Pour affecter une valeur à un élément i d'un tableau nommé par exemple T, on écrira :

T(i) ← valeur.

Par exemple, l'instruction : T(0) ← 20 ; affecte au premier élément du tableau T la valeur 20.

Pour affecter la même valeur à tous les éléments d'un tableau T de type numérique et de dimension 100, on utilise une boucle.

Par exemple : **pour** i de 0 à 99

T(i) ← 0 ;

fpour

Cette boucle permet de parcourir le tableau T élément par élément et affecter à chacun la valeur 0. La variable i est appelée indice.

b- La lecture :

Comme les variables simples, il est possible aussi d'assigner des valeurs aux éléments d'un tableau lors de l'exécution c.à.d. les valeurs sont saisies par l'utilisateur à la demande du programme.

Exemple : **écrire** "Enter une note :" ;
lire T(5) ;

Dans cet exemple, la valeur saisie est affectée au sixième ($6^{ème}$) élément du tableau T.

c- L'écriture :

De façon analogue à la lecture, l'écriture de la valeur d'un élément donné d'un tableau s'écrira comme suit : **écrire** T(i)

Cette instruction permet d'afficher la valeur de l'élément i du tableau T.

Remarque :

Les éléments d'un tableau sont des variables comme les autres. C.à.d. on peut créer un tableau de numériques comme on peut créer un tableau de caractères, de booléens …etc.

Dans le cas d'un tableau de type numérique par exemple, les éléments du tableau peuvent être utilisés dans l'évaluation des expressions numériques du genre :

x ← (T(1)+T(2))/2 ;

T(0) ← x + 10 ;

Exercice :

Ecrire un algorithme qui permet de saisir les prix dans un tableau de taille 5, et les afficher par la suite.

Exemple de solution :

```
variables i : entier ;
tableau Prix(5) : réel ;
Début
    pour i de 0 à 4
        écrire "Donner un prix :" ;
        lire Prix(i) ;
    fpour
    pour i de 0 à 4
        écrire  Prix(i) ;
    fpour
Fin
```

L'exécution du programme permet d'initialiser le tableau Prix comme suit :

Prix	
10	Prix (0)
120	Prix (1)
10.6	Prix (2)
99.99	Prix (3)
250	Prix (4)

Où les valeurs : 10, 120, 10.6, 99.99, 250 sont saisies par l'utilisateur.

Simulation d'exécution :

Lecture :

Donner un prix : 10

Donner un prix : 120

Donner un prix : 10.6

Donner un prix : 99.99

Donner un prix : 250

Affichage :

10.00

120.00

10.60

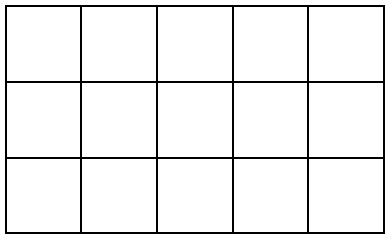
99.99

250.00

3. Tableaux à deux dimensions

Un tableau à deux dimensions, comme celui à une dimension, est un ensemble de variables de même type (numériques, caractères,…) ayant toutes le même nom. Il est dit à deux dimensions car il se présente sous forme d'un ensemble de ligne et de colonnes.

Exemple : le tableau T ci-dessous possède 3 lignes et 5 colonnes.



Ce genre de tableaux est caractérisé donc par l'utilisation de deux indices : indice de la ligne et indice de la colonne.

Par exemple, T(2,1) désigne l'élément de la ligne 3 et la colonne 2 (en commençant de zéro bien sûr).

3.1. Déclaration d'un tableau à deux dimensions

Syntaxe :

tableau nom du tableau (dimension1, dimension2) : **type**

Exemple : tableau notes (50,10) : **réel**

Dans cet exemple, nous avons déclaré un tableau notes qui contient 50 lignes et 10 colonnes. De tableau pourra contenir donc 50x10 soit 150 valeurs réelles.

L'utilité d'un tableau à deux dimensions réside dans la possibilité de déclarer un seul tableau au lieu d'en déclarer plusieurs identiques. En effet, le tableau notes ci-dessus équivaut à 50 tableaux simples de 10 éléments chacun. En d'autres termes, la déclaration :

tableau notes (50,10) : **réel**

remplace : **tableau** notes1 (10),…, notes50 (10) : **réel**

3.2. Manipulation d'un tableau à deux dimensions

Une fois déclaré, un tableau à deux dimensions peut être manipulé de la même façon qu'un tableau simple soit pour l'affectation, la lecture ou l'écriture.

Reprenons la solution de l'exercice précédent pour illustrer ces trois manipulations mais cette fois, pour un tableau de prix de 10 lignes et 5 colonnes :

```
variables i, j : entier ;
tableau Prix(10,5) : réel ;
Début
    pour i de 0 à 9
        pour j de 0 à 4
            écrire "Donner un prix :" ;
            lire Prix(i,j) ;
        fpour
    fpour
    pour i de 0 à 9
        pour j de 0 à 4
          écrire  Prix(i,j) ;
        fpour
    fpour
Fin
```

4. Tableaux à n dimensions

On peut déclarer des tableaux de dimension n>2

Syntaxe :

tableau nom du tableau (dimension 1,…, dimension N) : **type**

Exemple : tableau notes (50,10,20) : **réel**

Le principe de manipulation d'un tableau à plusieurs dimensions est le même que pour un tableau à deux dimensions. Si on utilise des boucles imbriquées pour parcourir le tableau, la 1ère boucle correspondra à « dimension 1 », la 2ème à « dimension 2 »,… et la Nème à « dimension N ».