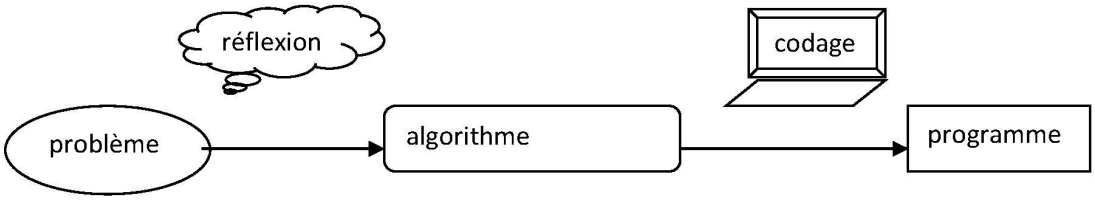
CHAP2 NOTION D'ALGORITHME ET DE PROGRAMME

L'algorithmique est un terme d'origine arabe, hommage à Al Khawarizmi (780-850) auteur d'un ouvrage décrivant des méthodes de calculs algébriques. Un algorithme est une méthode de résolution de problème énoncée sous la forme d'une série d'opérations à effectuer. La mise en œuvre de l'algorithme consiste en l'écriture de ses opérations dans un langage de programmation et constitue alors la brique de base d'un programme informatique.

1- Démarche d'analyse d'un problème et conception d'un algorithme et d'un programme:

De manière schématique, on peut représenter l'activité de développement pour résoudre un problème donné sous la forme suivante :



La réalisation d'un programme exécutable par un ordinateur, nécessite le suivi d'une démarche constituée d'un ensemble d'étapes.

Première étape : **Position du problème** :

Le problème est souvent posé par un demandeur de solution informatique. C'est le cas du pharmacien, d'un élève, d'un banquier,...

Parfois, ces demandeurs ne savent plus exprimer leurs besoins avec précision. L'objectif de cette étape est bien formuler le problème pour pouvoir le résoudre correctement.

Deuxième étape : **Spécification et analyse des problèmes** :

L'objectif de cette étape est bien comprendre l'énoncé du problème, déterminer les formules de calculs, les règles de gestion,...

L'analyse des problèmes s'intéresse aux éléments suivants :

- Les résultats souhaités (sorties),
- Les traitements (actions réalisés pour atteindre le résultat),
- Les données nécessaires aux traitements (entrées).

Troisième étape **: Ecriture de l'algorithme**.

Après avoir terminé l'analyse, il faut mettre les instructions dans leur ordre logique d'exécution. On obtient un algorithme.

Quatrième étape : **Ecriture du programme**.

Une fois l'algorithme du problème établit, on doit penser à son exécution par l'ordinateur. Il faut traduire l'algorithme à l'aide d'un langage de programmation.

Cinquième étape : **Exécutions et test du programme**.

Une fois compilé ou interprété, un programme doit être testé pour s'assurer de son fonctionnement et qu'il répond aux besoins exprimés par l'utilisateur.

Un programme est testé par un jeu de test (des valeurs différentes de données).

2-Concept d'un algorithme

Définition :

Un algorithme est une suite finie d'opérations élémentaires, à appliquer dans un ordre déterminé, à des données. Sa réalisation permet de résoudre un problème donné.

Exemples : suivre une recette de cuisine, suivre un plan, faire une division euclidienne à la main sont des exemples d'algorithme.

Remarques :

1.Un algorithme doit être lisible de tous. Son intérêt, c'est d'être codé dans un langage informatique afin qu'une machine (ordinateur, calculatrice, etc.) puisse l'exécuter rapidement et efficacement.

2.Les trois phases d'un algorithme sont, dans l'ordre :

-l'entrée des données

-le traitement des données

-la sortie des résultats.

3-Structure d'un programme :

3-1-Structure d'un algorithme :

L'algorithme doit avoir une structure bien définie. Cette structure doit comporter :

-L'en-tête qui comprend le nom de l'algorithme pour identifier l'algorithme.

- Les déclarations des variables et des constantes.

- Le corps de l'algorithme qui contient les instructions.

Toutes les instructions doivent situer entre le mot **Début** et le mot **Fin**, et chaque instruction doit comporter un **point-virgule** à la fin.

```
Algorithme nom_d'algorithme                    } L'en-tête

Variable nom_variable : type_variable ;        } Les déclarations
Constante nom_constante = valeur ;

Début                                          }
        Instruction 1 ;                        } Le corps
        Instruction 2 ;                        }
        Instruction 3 ;                        }
Fin .
```

3-2 -Structure d'un programme :

```
program {nom du programme};                          } entête

uses {liste des modules utilisés} ;
 Var nom-variable : type-variable ;                  } déclarations
Const nom-constante = valeur-constante;

begin
   instruction1 ;
   instruction2 ;                                    } corps
   instruction3 ;
end.
```

4-strusture de données

Un algorithme ou un programme qui manipule des informations : les données. On distingue deux types de données : les **variables** et les **constantes**. Comme leur nom l'indique, les "*variables*" sont des données qui pourront varier (dont la valeur pourra changer) durant l'exécution du programme. A l'inverse, les "constantes" ont une valeur constante tout au long de l'exécution du programme.

4.1. Les variables et les constantes

Définition et caractéristiques

- Une **VARIABLE** est une **donnée** (emplacement) stockée dans la mémoire de la calculatrice ou de l'ordinateur. Elle est repérée par un **identificateur** (nom de la variable constitué de lettres et/ou de chiffres, sans espace) et contient une **valeur** dont le **type** (nature de la variable) peut être un entier, un réel, un booléen, un caractère, une chaîne de caractères…

- Une **CONSTANTE**, comme une variable, peut représenter un chiffre, un nombre, un caractère, une chaîne de caractères, un booléen. Toutefois, contrairement à une variable dont la valeur peut être modifiée au cours de l'exécution de l'algorithme, la valeur d'une constante ne varie pas.

- 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,...)

Remarques :

 Ne pas confondre la variable et son identificateur. En effet, la variable possède une valeur (son contenu) et une adresse (emplacement dans la mémoire où est stockée la valeur). L'identificateur n'est que le nom de la variable, c'est-à-dire un constituant de cette variable.

 Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre et les opérations réalisables qu'elle peut subir.

 L'utilisation d'une variable doit être précédée de sa déclaration.

La syntaxe pour déclarer une variable est la suivante :

Var identificateur de la variable : type de la variable ;

La syntaxe pour déclarer une constante est la suivante :

const identificateur de la constante = valeur ;

 Si la valeur de la variable peut changer au cours du déroulement de l'algorithme, en revanche son type est figé lors de déclaration.

CONVENTIONS DE NOMMAGE

Le nom d'un algorithme, d'une variable ou d'une constante doit respecter les **règles** suivantes

- commencer par une lettre ;
- ne comporter ni caractère spécial (comme l'espace) ni ponctuation ;
- ne pas être un mot du langage algorithmique (comme « algorithme », « début », « fin », « variable », « non », « ou », « et », « si », « sinon », « pour »…)

4.2. Les types de données de bases

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(integer)**

b) **Type réel :** un autre type numérique qui représente les nombres réels, tels que : 0.5, -3.67, $1.5e^{+5}$,…

Mot clé : **réel(real)**

c) **Type caractère** : représente tous les caractères alphanumériques tels que :

'a', 'B', '*', '9', '@', ' ',…

Mot clé : **car(char)**

d) **Type chaînes de caractères** : concerne des chaines de caractères tels que des mots ou des phrases : "informatique", "la section B",…

Mot clé : **chaîne(string)**

e) **Type booléen :** ce type ne peut prendre que deux états : vrai ou faux

Mot clé **: booléen(boolean)**

Exemple de déclaration :

var A : **entier**

moyenne, note1, note2 : **réel**

nom : **chaîne**

lettre : **car**

const n = 5

arobase= '@'

e = "425"

5-Les opérateurs :

Un **OPERATEUR** est un outil qui permet d'agir sur une variable ou d'effectuer des calculs. Il existe plusieurs types d'opérateurs :

• L'**opérateur d'affectation**, représenté par le symbole «← » (**:=**), qui confère une valeur à une variable ou à une constante. (affectation de la valeur à la variable (ou à la constante)).

Syntaxe

identificateur_1 := identificateur_2 ;(Affecte à la variable 1 le contenu de la variable 2)
identificateur := expression ; (Affecte à la variable 1 le résultat de l'expression)

Exemples où A, B et C sont de type real , i, j de type integer et S de type string (chaîne de caractères)

A := B ; { assigne à A la valeur de B }
A := i ; { assigne à A la valeur de i }
i := A ; { donne une erreur de compilation : on ne peut assigner un real à un integer }
A := i/j ; { assigne à A le quotient (de type real) de i par j }
i := i+1 ; { i est incrémenté }
i := j div 10 { assigne à i le quotient (de type integer) de j par 10 (division entiere) }
S := S+'.' { ajoute le caractère '. ' à la fin de la chaîne S }

Remarques :

-Ne pas confondre ' := ' (affectation) avec ' = ' (comparaison des variables dans un test)
- Ne pas inverser les identificateurs ! ' A := B ' et ' B := A ' donnent des résultats différents

• **Les opérateurs arithmétiques** qui permettent d'effectuer des opérations arithmétiques entre opérandes numériques :

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

- **Les Opérateurs relationnels :**

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

• Les **opérateurs logiques** qui combinent des opérandes booléennes pour former des expressions logiques plus complexes :
  o Opérateur unaire : «non » (négation) (**not**)
  o Opérateurs binaires : « et» (conjonction) (**and**), «ou » (disjonction) (**or**)

• L'**opérateur de concaténation** qui permet de créer une chaîne de caractères à partir de deux chaînes de caractère en les mettant bout à bout. « + »

Remarque : Les opérateurs dépendent du type de la constante ou de la variable :

• **Opérateurs sur les entiers et les réels :** addition, soustraction, multiplication, division, division entière, puissance, comparaisons, modulo (reste d'une division entière)
• **Opérateurs sur les booléens :** comparaisons, négation, conjonction, disjonction
• **Opérateurs sur les caractères :** comparaisons
• **Opérateurs sur les chaînes de caractères :** comparaisons, concaténation

• **Priorité des opérateurs :** A chaque opérateur est associée une priorité. Lors de l'évaluation d'une expression, la priorité de chaque opérateur permet de définir l'ordre d'exécution des différentes opérations. Aussi, pour lever toute ambiguïté ou pour modifier l'ordre d'exécution, on peut utiliser des parenthèses.

- **Ordre de priorité décroissante des opérateurs arithmétiques et de concaténation :**
  - «^» (élévation à la puissance)
  - «* », « /» et « div»
  - « modulo»
  - «+ » et «-»
  - «+» (concaténation)

- **Ordre de priorité décroissante des opérateurs logiques :**
  - «not » (non)
  - « and» (et)
  - « or» (ou)

6-Les opérations d'entrée /sortie:

- La **lecture** de données correspond à l'opération qui permet de saisir des valeurs à partir du clavier pour qu'elles soient utilisées par le programme. Cette instruction est notée **lire «** *identificateur* » (**read** ou **readln**) .
- L'**écriture** des données permet l'affichage des valeurs des variables après traitement sur l'ecran ou l'imprimante. Cette instruction est notée **ecrire «** *identificateur* » (**write** ou **writeln**).

Syntaxe en PASCAL :

Pour la lecture : READLN (identificateur_de_variable) ;

Permet l'entrée au clavier d'une variable

Remarques :

Attention à ce que l'entrée corresponde bien au type de la variable !

Seul un curseur clignotant indique que l'ordinateur attend une entrée. Il est judicieux d'afficher auparavant un message pour indiquer ce qu'on attend.

Pour l'écriture : WRITE (identificateur_de_variable) ;

Affiche à l'écran, à partir de la position courante du curseur, le contenu (la valeur) de la variable

WRITELN (identificateur_de_variable . . .) ;

Même chose, mais passe à la ligne après l'affichage

Exemples :

Algorithme qui calcule la somme de deux nombres réels

```
Algorithme sommedeuxnombres ;
Var x,y : real ;
début
  ecrire("donner deux nombres réels") ;
 lire(x,y) ;
 ecrire("la somme de",x,"et",y,"est",x+y) ;
fin.
```

Algorithme qui calcule l'âge

```
Algorithme calculage ;
Var annee : integer ;
Const ancourant = 2016;
début
  ecrire( "donner votre année de naissance") ;
 lire(annee) ;
 ecrire("votre age est", ancourant-annee) ;
fin.
```

algorithme qui calcule la surface d'un disque

```
Algorithme Surface d'un disque
Var  Rayon, Surface : real ;
Const  Pi = 3.14 ;
Début
   ecrire("Donner le rayon du disque : ") ;
   lire (Rayon) ;
  Surface := (Rayon ^ 2) * Pi ;
  ecrire ("La surface du disque est : ", Surface) ;
Fin.
```

SUITE CHAP2 NOTION D'ALGORITHME ET DE PROGRAMME

STRUCTURES CODITIONNELLES

Nous avons vu jusqu'à présent des algorithmes avec des instructions qui s'enchainent de façon séquentielles, c'est à dire qui vont s'exécuter les unes aprés les autres. Toutefois, cela n'est pas toujours satisfaisant.

Exemple : on veut écrire un algorithme permettant d'afficher le montant d'une commande d'imprimantes. Le prix unitaire d'une imprimante est de 5000 DA. A partir de cinq imprimantes achetées, le prix est de 4500 DA.

Que doit faire le traitement ?

*Il doit calculer le montant à payer, c'est à dire réaliser l'opération :quantité * prix unitaire . Or, le prix unitaire est variable et dépend de la quantité commandée. Le traitement doit donc* ***tester la quantité****.*

Pour effectuer des tests, nous utilisons les **structures conditionnelles**.

1-La structure alternative

Reprenons notre exemple, *nous pouvons exprimer la problématique de la manière suivante :*

Si *la quantité est inférieure strictement à 5* ***alors*** *le prix unitaire vaut 5000 DA,* ***sinon*** *le prix unitaire vaut 4500 DA.*

En algorithmique on utilise la **structure alternative** SI ... ALORS ... SINON, dont la syntaxe est :

```
Si condition(s)   Alors
    debut
      Traitement 1
   fin
   Sinon
   debut
      Traitement 2
   Fin Si ;
```

```
IF condition(s)   THEN
Begin
  Traitement 1 ;
End
ELSE
Begin
  Traitement 2 ;
End ;
```

A noter ici, l'**indentation** (décalage) des lignes pour permettre une meilleure lisibilité.

Ainsi l'algorithme correspondant à notre exemple est le suivant :

```
algorithme Commande
VAR
  quantite, montant : entier ;
DEBUT
      Ecrire( "Donner la quantité à acheter") ;
      Lire( quantite) ;
      SI quantite < 5 ALORS montant := quantite * 5000
       SINON   montant := quantite * 4500 ;
      Ecrire( " Prix des imprimantes et : ", montant) ;
FIN.
```

Remarque :

- L'instruction précédent le **sinon** ne termine pas avec un point-virgule(**;**).

- **debut** et **fin** ne sont pas nécessaires si le **si** ou le **sinon** comprend une seule instruction.

2. La structure conditionnelle simple

Parfois, il faut exécuter une instruction uniquement si une condition est vraie et ne rien faire de particulier si la condition est fausse.

Syntaxe

En algorithmique on utilise la structure conditionnelle simple SI … ALORS … ; dont la syntaxe est :

```
Si condition(s) Alors
 debut
   Traitement ;
 Fin Si ;
```

```
IF condition(s) THEN
Begin
  Traitement ;
End ;
```

3. Les structures alternatives imbriquées

L'alternative à une condition peut elle-même conduire à choisir de nouveau entre deux actions. On utilise alors les structures alternatives imbriquées.

Reprenons notre algorithme. Pour amèliorer le traitement, nous voulons que le traitement vérifie au préalable si la quantité saisie par l'utilisateur est positive.

Si la quantité est négative ou nulle, le programme affiche un message d'erreur.

Si la quantité est positive, le programme calcule puis affiche le montant de la commande selon la logique définie précédemment : le prix d'une imprimante est 5000DA. A partir de cinq imprimantes achetées, le prix est de 4500DA la pièce.

Dans un premier temps, le système teste la valeur saisie par l'utilisateur. Si elle est négative, un message d'anomalie est affiché.

Si la quantité est positive, le traitement doit alors tester la quantité pour savoir si elle est inférieure à 5.

```
algorithme Commande
VAR
  quantite, montant : entier ;
DEBUT
      Ecrire( "Donner la quantité à acheter") ;
      Lire( quantite) ;
      Si quantite > 0 alors
      debut
        SI quantite < 5 ALORS montant := quantite * 5000
        SINON   montant := quantite * 4500 ;
        Ecrire( " Prix des imprimantes et : ", montant) ;
      FIN
      Sinon ecrire ("erreur la quantité doit être positive") ;
FIN.
```

Il est possible d'imbriquer plusieurs niveaux : chaque SINON peut ouvrir sur un nouveau SI. Mais pour des raisons de compréhension et de lisibilité, il est préférable de se limiter à deux ou trois niveaux maximum : au-delà, il devient plus difficile d'analyser le code.

```
SI condition1 ALORS
debut
    traitement1 ;
fin
SINON
debut
  | SI condition2 ALORS
  | debut
  |   Traitement2 ;
  | fin
  | SINON
  | debut
  |   | SI condition3 ALORS
  |   | debut
  |   |   Traitement3 ;
  |   | fin
  |   | SINON
  |   | debut
  |   |   | SI condition4 ALORS
  |   |   | debut
  |   |   |   Traitement4 ;
  |   |   | fin
  |   |   | SINON
  |   |   |     ..........
  |   | FIN
  | FIN
FIN
```

```
IF condition1 THEN
begin
    Trait1,
end
ELSE
begin
  | IF condition2 THEN
  | Begin
  |   Trait2 ;
  | end
  | ELSE
  | begin
  |   | IF condition3 THEN
  |   | begin
  |   |   Trait3 ;
  |   | end
  |   | ELSE
  |   | begin
  |   |   | IF condition 4 THEN
  |   |   | begin
  |   |   |   Trait 4 ;
  |   |   | ELSE
  |   |   |     ..........
  |   | End ;
  | End ;
End ;
```

4.La structure alternative multiple

La structure **choix** permet de choisir le traitement à effectuer en fonction de la valeur ou de l'intervalle de valeurs d'une variable ou d'une expression. Cette structure permet de remplacer avantageusement une succession de structures SI … ALORS.

Syntaxe

```
choix expression dans
   valeur1 : trait1 ;
   valeur2 : trait2 ;
   valeur5, valeur8 : trait3 ;
   valeur10..valeur30 : trait4 ;
    ...
    valeur n-1 : trait n-1 ;
   Sinon trait n ;
FINchoix
```

```
CASE expression OF
  Valeur1 : trait1 ;
  valeur2 : trait2 ;
  valeur 5, valeur 8 : trait3 ;
  valeur 10..valeur 30 : trait4 ;
  ...
  valeur n-1 : trait n-1 ;
  Else trait n ;
END ;
```

- **La ligne sinon est facultative :** elle permet de définir le traitement à réaliser dans tous les cas non listés précédemment.
- **Attention,** les **valeurs** exprimées dans les **CAS** (valeur1, valeur2, …) **doivent être de même type que l'expression** évaluée dans le **choix.**

Exemple

Voilà l'algorithme qui affiche le mois en toute lettre selon son numéro.
Le numéro du mois est mémorisé dans la variable *noMois*.

```
...
VAR  noMois :entier ;

...
DEBUT
...
        Choix noMois dans
             1 : ecrire("Janvier") ;
             2 :ecrire("Février") ;
             3 :ecrire("Mars") ;
                    .....
             12 :ecrire("Décembre") ;
             sinon :ecrire("mois doit être compris entre 1 et 12") ;
        fin ;
...
```

Ce même algorithme écrit avec la structure SI aurait nécessité 13 appels de cette intruction soit 13x3=39 lignes de code au lieu de 15 utilisés avec la structure choix.

5.Complément sur les expressions conditionnelles

Une expression conditionnelle (ou expression logique, ou expression booléenne) est une expression dont la valeur est soit VRAI soit FAUX. Il existe plusieurs types d'expressions conditionnelles.

5.1. Les comparaisons simples

Une condition simple est une comparaison de deux expressions de même type.

Exemples

a < 0	comparaison d'entiers ou de réels
code = 's'	comparaison de caractères (code étant une variable de type caractère)

Attention, une condition simple ne veut pas dire une condition courte. Une condition simple peut être la comparaison de deux expressions comme:

(a + b - 3) * c ≤ (5 * y –2) / 3

5.2.Les expressions complexes

Les conditions (ou expressions conditionnelles) peuvent aussi être complexes, c'est-à-dire formées de plusieurs conditions simples ou variables booléennes reliées entre elles par les opérateurs logiques **ET**, **OU**, **NON**.

Exemples

SI a < 0 **ET** b < 0 ALORS
...
SI (a +3 = b et c < 0) **OU** (a = c *2 et b ≠ c) ALORS
...

❖ **ET**

Une condition composée de deux conditions simples reliées par **ET** est vraie si les deux conditions sont vraies

❖ **OU**

Une condition composée de deux conditions simples séparées par **OU** est vraie si au moins l'une des conditions simples est vraie.

❖ **NON**

Une condition précédée par NON est vraie si la condition simple est fausse et inversement.

Rappel sur les tables de vérité des opérateurs logiques ET et OU

ET	VRAI	FAUX
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

OU	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

5.3.Les variables booléennes

Les variables booléennes, comme les expressions conditionnelles, sont soit **vraies**, soit **fausses**. On peut donc affecter une expression conditionnelle à un booléen et on peut aussi trouver une variable booléenne à la place d'une expression conditionnelle.

Les variables booléennes et les expressions conditionnelles sont équivalentes. A chaque fois que l'on peut trouver une expression conditionnelle, on peut aussi trouver une variable booléenne.

Exercices

1. Ecrire l'algorithme *Calculer* permettant la saisie de deux nombres ainsi que la lettre représentant l'opération à effectuer (s pour somme, p pour produit). Le programme calcule puis affiche ensuite le résultat de l'opération. On considère que l'utilisateur ne fait aucune erreur de saisie.

Exemples de ce qui sera visible à l'écran (en gris ce qui est saisi par l'utilisateur)

```
Saisir deux nombres entiers
100
20
Saisir s pour obtenir la somme et p pour
obtenir le produit s
Le résultat est : 120
```

```
Saisir deux nombres entiers
12
3
Saisir s pour obtenir la somme et p pour
obtenir le produit p
Le résultat est : 36
```

2. Ecrire l'algorithme permettant de déterminer puis d'afficher le maximum de deux nombres saisis (on considèrera que les valeurs saisies par l'utilisateur sont différentes)

Exemple de ce qui sera visible ƒ l'écran (en gris ce qui est saisi par l'utilisateur)

```
Saisir deux nombres entiers différents
-100
20
Le plus grand des deux nombres est : 20
```

3. Reprendre l'exercice précédent (affichage du maximum de deux nombres) en utilisant cette fois-ci une et une seule structure conditionnelle au sens strict (SI … ALORS …. FINSI).

4. Faire la trace (valeurs successives des variables) du programme ci-dessous dans les cas suivants. Vous présenterez les résultats dans les tableaux fournis ci-aprés.

Cas 1 : l'utilisateur saisit a=3, b=2 et c=1
Cas 2 : l'utilisateur saisit a=0, b=4 et c=5
Cas 3 : l'utilisateur saisit a=1, b=5 et c=3

Seules les variables qui changent de valeur d'une instruction à l'autre doivent être mentionnées dans le tableau.

```
PROGRAMME tester ;
VARIABLE
    a, b, c, d : entier
DEBUT
    ecrire ("Saisir trois nombres entiers : ")
    Lire( a) ; lire (b) ; lire( c) ;..... i1
    d ← 2. ;..... i2
    SI a = 3  ALORS
     debut
       a := 2 ;..... i3
        b := (a + c) * d;..... i4
     fin
     SINON
      debut
           SI a = 0  ALORS
           debut
            a ← 2 ;..... i5
             c ← d * a ; .....i6
           fin
           SINON
           debut
              c ← 1 + b ;  .....i7
              d ← b – a ; .....i8
          FIN ;
    FIN ;
   ecrire ("a=", a," b=", b," c=", c) ;.....i9
FIN.
```

Cas 1: l'utilisateur saisit a=3, b=2 et c=1

Aprés instruction	Valeur des variables			
	a	b	c	d
i1				
i2				
i3				
i4				
i5				
i6				
i7				
i8				
I9				

Cas 2 : l'utilisateur saisit a=0, b=4 et c=5

Aprés instruction	Valeur des variables			
	a	b	c	d
i1				
i2				
i3				
i4				
i5				
i6				
i7				
i8				
I9				

Cas 3 : l'utilisateur saisit a=1, b=5 et c=3

Apr€s instruction	Valeur des variables			
	a	b	c	d
i1				
i2				
i3				
i4				
i5				
i6				
i7				
i8				
I9				

5. Ecrire un algorithme qui demande 2 nombres non nuls à l'utilisateur et l'informe ensuite si leur produit est positif ou négatif. Vous ne devez pas calculer le produit de ces 2 nombres

SUITE CHAP2 NOTION D'ALGORITHME ET DE PROGRAMME

STRUCTURES CODITIONNELLES REPETITIVES

Une **structure répétitive** (ou **structure itérative**) répète l'exécution d'un traitement, dans un ordre précis, un nombre déterminé ou indéterminé de fois. Une structure itérative est aussi appelée boucle.
Deux cas sont cependant à envisager, selon que :

- le nombre de répétitions est connu à l'avance : c'est le cas des **boucles itératives**
- le nombre de répétitions n'est pas connu ou est variable : c'est le cas des **boucles conditionnelles**

1- Boucle Pour (For)

La structure de contrôle répétitive **Pour** (**For** en langage PASCAL) utilise un indice entier qui varie (avec un incrément = 1) d'une valeur initiale jusqu'à une valeur finale. À la fin de chaque itération, l'indice est incrémenté de 1 d'une manière automatique (implicite).

La syntaxe de la boucle pour est comme suit :

pour <indice> ← <vi> **à** <vf> **faire**

<instruction(s)>

finPour;

for <indice>:=<vi> **to** <vf> **do**
begin
<instruction(s)>;
end;

<indice> : variable entière
<vi> : valeur initiale <vf> : valeur finale

La boucle **pour** contient un bloc d'instructions (les instructions à répéter). Si le bloc contient une seule instruction, le begin et end sont facultatifs.

Le bloc sera répété un nombre de fois = (<vf> - <vi> + 1) si la valeur finale est supérieure ou égale à la valeur initiale. Le bloc sera exécuté pour <indice> = <vi>, pour <indice> = <vi>+1, pour <indice> = <vi>+2, …, pour <indice> = <vf>.

La sortie de la boucle s'effectue lorsque le nombre souhaité d'itérations est atteint, c'est-à-dire lorsque <indice> prend la valeur <vf>.
Il ne faut jamais mettre de point-virgule après le mot clé do. (erreur logique).

Exemple Faire la somme des 10 premiers entiers naturels.

```
Algorithme  utilisation_du_compteur1;
   Var somme,i : integer;
 debut
Somme←0 ;
For i ←1 to 9 Do somme←somme + i ;
end.
```

2- Boucle Tant-que (While)

La structure de contrôle répétitive **tant-que** (**while** en langage PASCAL) utilise une expression logique ou booléenne comme condition d'accès à la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) donc on entre à la boucle, sinon on la quitte.

La syntaxe de la boucle **tant-que** est comme suit :

tant-que <condition> **faire**
debut
<instruction(s)>
finTant-que;

while <condition> **do**
begin
<instruction(s)>;
end;

<condition> : expression logique qui peut être vraie ou fausse.

On exécute le bloc d'instructions tant que la condition est vraie. Une fois la condition est fausse, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après **fin Tant que** (après end).

Comme la boucle **for**, il faut jamais mettre de point-virgule après **do**.

Toute boucle **Pour** peut être remplacée par une boucle **tant-que**, cependant l'inverse n'est pas toujours possible.

Exemple : Faire la somme des 10 premiers entiers naturels.

```
Algorithme utilisation_compteur2 ;
 Var i,somme :entier ;
debut
  i :=1 ; Somme :=0 ;
  While i < 10 Do
  begin
    somme := somme + i ; i := i + 1 ;
  end ;
end.
```

3-Boucle Répéter (Repeat)

La structure de contrôle répétitive **répéter** (**repeat** en langage PASCAL) utilise une expression logique ou booléenne comme condition de sortie de la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) on sort de la boucle, sinon on y accède (on répète l'exécution du bloc).

La syntaxe de la boucle répéter est comme suit :

repéter
<instruction(s)>
jusqu'à <condition>;

repeat
<instruction(s)>;
until <condition>;

<condition> : expression logique qui peut être vraie ou fausse.

On exécute le bloc d'instructions jusqu'à avoir la condition correcte. Une fois la condition est vérifiée, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après jusqu'à (après **until**). Dans la boucle **repeat** on utilise pas begin et end pour délimiter le bloc d'instructions (le bloc est déjà délimité par **repeat** et **until**).

La différence entre la boucle **répéter** et la boucle **tant-que** est :

– La condition de **répéter** et toujours l'inverse de la condition **tant-que** : pour **répéter** c'est la condition de sortie de la boucle, et pour **tant-que** c'est la condition d'entrer.

– Le teste de la condition est à la fin de la boucle (la fin de l'itération) pour **répéter**. Par contre, il est au début de l'itération pour la boucle **tant-que**. C'est-à-dire, dans **tant-que** on teste la condition avant d'entrer à l'itération, et dans **répéter** on fait l'itération après on teste la condition.

Exemple : Faire la somme des 10 premiers entiers naturels.

```
Algorithme utilisation_compteur3 ;
Var somme,i : entier ;
debut
  Somme :=0 ; i := 1 ;
  Repeat
    somme := somme + i ; i :=i+1 ;
  Until i = 10 ;
End.
```

Choix de la boucle

La règle est simple :

Si le nombre d'itérations est connu a priori, alors on utilise un for.

Sinon : on utilise le repeat (quand il y a toujours au moins une itération), ou le while (quand le nombre d'itérations peut être nul).

4-Correspondance Algorithme <--> PASCAL

Pour traduire un algorithme en programme PASCAL, on utilise le tableau récapitulatif suivant pour traduire chaque structure syntaxique d'un algorithme en structure syntaxique du PASCAL.

Vocabulaire / Syntaxe Algorithmique	Vocabulaire / Syntaxe du PASCAL
Algorithme	Program
Constantes	Const
Type	Type
Varaibles	Var
Etiquette	Label
Entier	Integer
Réel	Real
Caractère	Char
Booléen	Boolean

Chaîne de Caractères	String
Fonction	Function
Procédure	Procedure
Début	Begin
Fin	End
Si ... Alors ... Sinon ...	If ... Then ... Else ...
Tant-que ... Faire ...	While ... Do ...
Pour i ← 1 à N Faire ...	For i:= 1 To N Do ...
Pour i ← N à Pas- 1 Faire ...	For i:= 1 DownTo 1 Do ...
Répéter ... Jusqu'à ...	Repeat .... Until ...

Remarques Importantes

Langage PASCAL est insensible à la casse, c'est-à-dire, si on écrit begin, Begin ou BEGIN c'est la même chose.

Lorsque l'action après THEN, ELSE ou un DO comporte plusieurs instructions, on doit obligatoirement encadrer ces instructions entre BEGIN et END. Autrement dit, on les définit sous forme d'un bloc. Pour une seule instruction, il n'est pas nécessaire (ou obligatoire) de l'encadrer entre BEGIN et END (voir en travaux pratiques). Un ensemble d'instructions encadrées entre BEGIN et END, s'appelle un BLOC ou action composée. On dit qu'un programme PASCAL est structurée en blocs.

Voici quelque fonctions standards (ou prédéfinies) *appliquées aux entiers ou réels*

Fonction	*L'appel avec paramètre*	*Le résultat retourné*
ABS	*ABS(x)*	Retourne la valeur absolue d'un nombre *x*
EXP	*EXP(x)*	Retourne l'exponentiel d'un nombre *x*
LN	*LN(x)*	Retourne le logarithme népérien d'un nombre *x*
LOG	*LOG(x)*	Retourne le logarithme à base 10 d'un nombre *x*
SQRT	*SQRT(x)*	Retourne la racine carrée d'un nombre *x*
SQR	*SQR(x)*	Retourne le carré d'un nombre *x*
Arctan	*Arctan(x)*	Retourne l'arc tangente d'un nombre *x*
Cos	*Cos(x)*	Retourne le cosinus d'un nombre *x*
Sin	*Sin(x)*	Retourne le sinus d'un nombre *x*
Round	*Round(x)*	Retourne la valeur arrondie d'un nombre *x*
Trunc	*Trunc(x)*	Retourne la partie entière d'un nombre *x*

5-Représentation en organigramme

Un algorithme peut aussi être représenté de façon graphique. On parle ici d'*organigramme*. Un organigramme est une représentation normalisée de l'enchaînement des opérations et des décisions à effectuer par un programme d'ordinateur. Chaque type d'action dans l'algorithme possède une représentation dans l'organigramme.

Il est préférable d'utiliser la représentation algorithmique que la représentation par organigramme notamment lorsque le problème est complexe.

Les inconvénients qu'on peut rencontrer lors de l'utilisation des organigrammes sont :

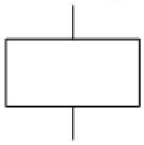
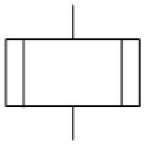
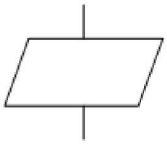
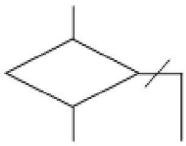
– Quand l'organigramme est long et tient sur plus d'une page,

– problème de chevauchement des flèches,

– plus difficile à lire et à comprendre qu'un algorithme.

Les principaux symboles rencontrés dans un organigramme sont représentés dans le tableau ci-dessous :

Les principaux symboles rencontrés dans un organigramme sont représentés dans le tableau ci-dessous :

SYMBOLE	DESIGNATION	SYMBOLE	DESIGNATION
Symboles de traitement		Symboles auxiliaires	
	**Symbole général** Opération sur des données, instructions, ...		**Renvoi** Connecteur utilisé à la fin et en début de ligne pour en assurer la continuité
	**Sous-programme** Portion de programme		**Début, fin ou interruption** d'un algorithme
	**Entrée-Sortie** Mise à disposition ou enregistrement d'une information		**Liaison** Les différents symboles sont reliés entre eux par des lignes de liaison. Le cheminement va de haut en bas et de gauche à droite. Un cheminement différent est indiqué à l'aide d'une flèche.
Symbole de test			
	**Branchement** Décision d'un choix parmi d'autres en fonction des conditions		

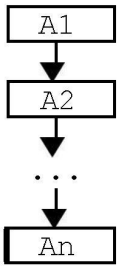
6. Représentation des primitives algorithmiques

6.1. L'enchaînement

L'enchaînement permet d'exécuter une série d'actions dans l'ordre le leur apparition.

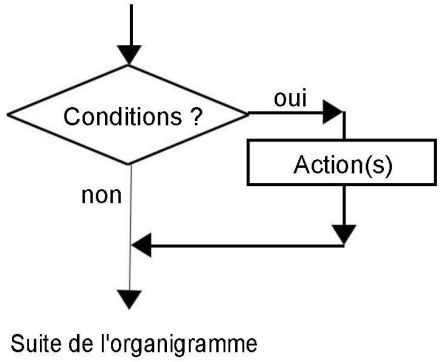
Soit $A_1, A_2, \ldots, A_n$ une série d'actions, leur enchaînement est représenté comme suit :

```
A1;
A2;
 .
 .
An;
```



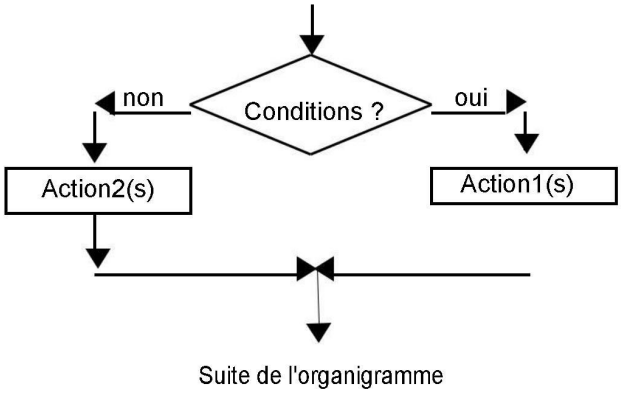
A1, A2, …, An : peuvent être des actions élémentaires ou complexes.

6.2. La structure alternative simple

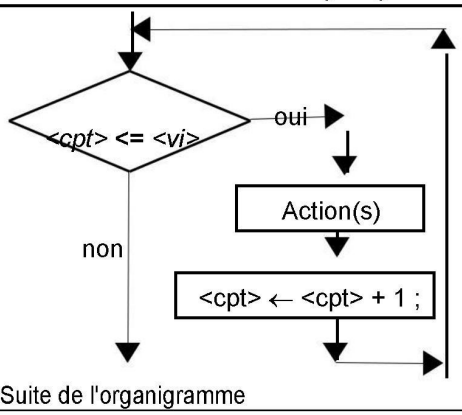
Représentation algorithmique	*Représentation sous forme d'organigramme*
si <Condition> **alors** <action(s)>; **finsi;** Si la condition est vérifiée, le bloc *<action(s)>* sera exécuté, sinon rien, et on continue l'exécution de l'instruction après fin si.	

Les condition utilisées pour les teste (simple ou double) sont des expression logique ou booléennes, ça veut dire des expression dont leur évaluation donne soit TRUE (Vrai) ou FALSE (faux). Toute comparaison entre deux nombre représente un expression logique. On peut former des expressions logiques à partir d'autres expressions logique en utilisant les opérateurs suivant : Not, Or et And.

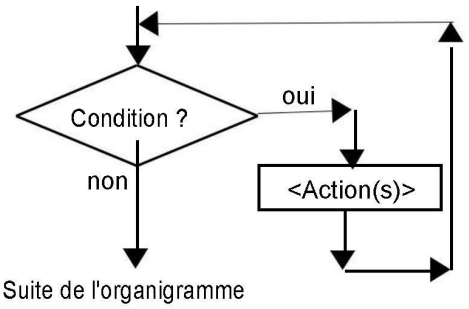
6.3. La structure alternative double

Représentation algorithmique	*Représentation sous forme d'organigramme*
si <Condition> **alors** <action1(s)>; **sinon** <action2(s)>; **finsi;** Si la condition est vérifiée, le bloc *<action1(s)>* sera exécuté, sinon (si elle est fausse) on exécute *<action2(s)>*.	

6.4. La structure itérative POUR (Boucle POUR)

Représentation algorithmique	*Représentation sous forme d'organigramme*
pour <cpt> ← <vi> **à** <vf> **faire** <action(s)>; **finpour;**	

6.5. La structure itérative Tant-Que (Boucle Tant-Que)

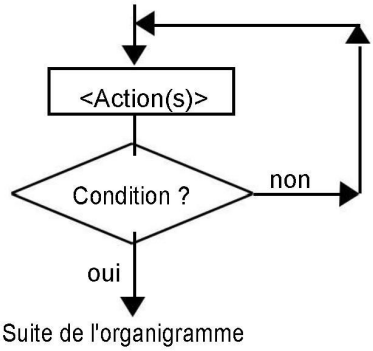
Représentation algorithmique	*Représentation sous forme d'organigramme*
Tant-que <condition> **faire** <action(s)>; **finpour**;	

On exécute le bloc d'instructions <actions> tant que la *<condition>* est vérifiée (c'est-à-dire elle est vraie). Le déroulement de la boucle est comme suit :

1 – On évalue la condition : si la condition est fausse on sort de la boucle

2 – Si la condition est vraie, on exécute le bloc <actions> ; et on réévalue la condition (on revient a 1).

6.6. La structure itérative Répéter (Boucle Répéter)

Représentation algorithmique	*Représentation sous forme d'organigramme*
Répéter <action(s)>; **Jusqu'à** <condition>;	

On répète l'exécution du bloc *<action(s)>* jusqu'à avoir la condition correcte. Le déroulement est comment suit :

1 – On exécute le bloc *<action(s)>* ;

2 – On évalue la condition : si la condition est vérifiée (elle est vraie) on sort de la boucle (on continue la suite de l'algorithme);

3- si la condition n'est pas vérifiée (elle est fausse) on revient à 1.

Remarques :

✔ N'importe quelle boucle *POUR* peut être remplacée par une boucle *Tant-Que*, cependant l'inverse n'est pas toujours correcte, c'est-à-dire, il y a des cas où la boucle *Tant-Que* ne peut pas

être remplacée par une boucle *POUR*.

✔ On transforme une boucle pour à une boucle Tant-Que comme suit :

*Boucle **POUR***	*Boucle **Tant-Que***
pour <cpt> ← <vi> **à** <vf> **faire** <action(s)>; **finpour;**	<cpt> ← <vi>; **Tant-que** <cpt> <= <vf> **faire** <action(s)>; <cpt> ← <cpt> + 1; **finTant-Que;**

✔ La boucle Répéter possède une condition de sortie (c'est-à-dire si elle est vraie on sort de la boucle), alors que la boucle Tant-que possède une condition d'entrée (c'est-à-dire si elle est vraie on entre dans la boucle).

✔ La boucle Répéter exécute le bloc *<action(s)>* au moins une fois, le teste vient après

Exemples

Exemple1

Ecrire un programme qui effectue la division de deux entiers par des soustractions successives.

Solution

```
Algorithme division ;
Var dividende, diviseur, quotient :entier ;
Début
  quotient←0 ;
  ecrire('entrer le dividende') ;
  lire(dividende) ;
  ecrire('entrer diviseur') ;
  lire(diviseur) ;
  tantque dividende ≥ diviseur faire
  début
     dividende←dividende-diviseur ;
     quotient←quotient+1 ;
  fin ;
  ecrire('le quotient est ',quotient) ;
  ecrire('le reste est ',dividende) ;
fin.
```

```
program division;
uses crt;
var  dividende, diviseur, quotient: integer;
begin
  clrscr;
  quotient := 0;
   writeln('Entrez le dividende');
  readln(dividende);
  writeln('Entrez le diviseur');
  readln(diviseur);
  while (dividende >= diviseur) do
  begin
    dividende := dividende - diviseur;
    quotient := quotient + 1;
  end;
  writeln('Le quotient est : ', quotient);
  writeln('Le reste est : ', dividende);
 end.
```

Exemple2

Faire une variante de ce programme en utilisant la boucle repeat...until

Solution

```
Algorithme division ;
Var dividende, diviseur, quotient :entier ;
Début
  quotient←0 ;
  ecrire('entrer le dividende') ;
  lire(dividende) ;
  ecrire('entrer diviseur') ;
  lire(diviseur) ;
  si dividende ≥ diviseur alors
  début
    repéter
      dividende←dividende-diviseur ;
      quotient←quotient+1 ;
    jusqu'à  dividende < diviseur ;
 fin ;
  ecrire('le quotient est ',quotient) ;
  ecrire('le reste est ',dividende) ;
fin.
```

```
program division2;
uses crt;
var dividende, diviseur, quotient: integer;
begin
  clrscr; quotient := 0;
  writeln('Entrez le dividende');
  readln(dividende);
  writeln('Entrez le diviseur');
  readln(diviseur);
  if  dividende ≥ diviseur then
  begin
    repeat
     dividende := dividende - diviseur;
     quotient := quotient + 1;
    until dividende < diviseur;
  end ;
  writeln('Le quotient est : ', quotient);
  writeln('Le reste est : ', dividende);
  readln;
end.
```

Exercices d'application

Exercice 1 Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 2 Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

Exercice 3 Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

Exercice 4 Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) : Table de 7 : 7 x 1 = 7

7 x 2 = 14

7 x 3 = 21 …

7 x 10 = 70.

Exercice 5 Ecrire un algorithme qui demande un nombre positif de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer : 1 + 2 + 3 + 4 + 5 = 15 NB : on souhaite afficher uniquement le résultat, pas la décomposition du calcul.

Exercice 6 Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle. NB : la factorielle de 8, notée 8 !, vaut 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8

Exercice 7 Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dise ensuite quel était le plus grand parmi ces 20 nombres :

Entrez le nombre numéro 1 : 12

Entrez le nombre numéro 2 : 14 etc.

Entrez le nombre numéro 20 : 6

Le plus grand de ces nombres est : 14

Modifiez ensuite l'algorithme pour que le programme affiche de surcroît en quelle position avait été saisie ce nombre : C'était le nombre numéro 2.