# Informatique 1


République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieure et de la Recherche Scientifique

Université Abderrahmane Mira de Bejaia

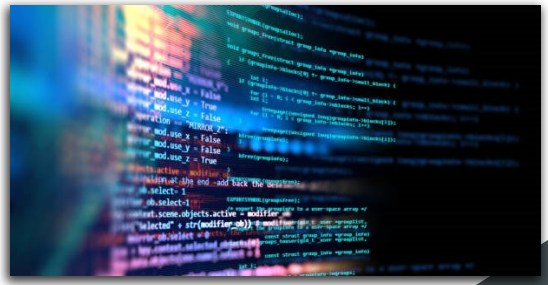
Faculté de la Technologie

Département de Technologie

**Dr. Cylia AMRANE**

cylia.amrane@univ-bejaia.dz


Mai 2024



# Table des matières

# Notion d'algorithme et de programme


## 1. Objectifs

A l'issu de ce chapitre, l'apprenant sera capable de :

➢ Comprendre la définition et le rôle des algorithmes dans la résolution de problèmes informatiques.

➢ Identifier les différentes étapes d'un algorithme, de la spécification à l'exécution.

➢ Appréhender les concepts de variables, de types de données et d'opérations élémentaires dans la programmation.

➢ Analyser des exemples concrets d'algorithmes simples et comprendre comment ils sont traduits en programmes exécutables.

## 2. Introduction

Un Algorithme est une **séquence d'instructions ordonnées**, qui permet de **résoudre un problème**. Le terme “Algorithme ” vient de l'arabe الخُوَارِزْمِيّ , nom du mathématicien perse Al-Khwarizmi.

Un algorithme prend, en entrée, un ensemble de données (Inputs) et délivre (produit, renvoie) un ensemble de données en sortie (Outputs), afin de résoudre un problème.

## 3. Concept d'un algorithme

Un algorithme[1] peut être schématisé comme suit :

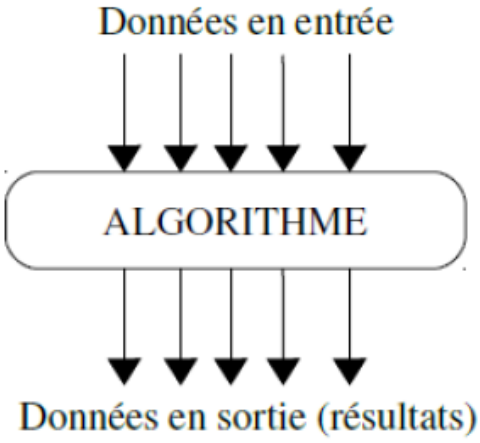



*Figure II.1 : Structure d'un algorithme*

Donc, un algorithme représente une solution pour un problème donné. Cette solution est spécifiée à travers un ensembles d'instructions (séquentielles avec un ordre logique) qui manipulent des données. Une fois l'algorithme est écrit (avec n'importe quelle langues : français, anglais, arabe, etc.), il sera transformé, après avoir choisi un langage de programmation, en un programme code source qui sera compilé (traduit) et exécuté par l'ordinateur.

[1] https://youtu.be/z9WVh2K0svU?feature=shared

Pour le langage de programmation qui sera utilisé, ça sera le langage **PASCAL**.

# 4. La démarche et analyse d'un problème

*Un algorithme représente une solution à un problème donné*. La résolution de ce problème applique un processus d'analyse et de résolution qui est constitué des étape suivantes :

**1. Définition du problème à traiter** : Cette phase demande plus d'attention à l'utilisateur :

- Un problème bien posé est déjà à moitié résolue, et d'autre part il faut savoir si le problème se prête à être résolu ou non sur ordinateur.

- La formulation mathématique du problème doit être définie.

- Définition de toutes les données ainsi que le choix des unités. La précision désirée, les résultats attendus et le contrôle des erreurs.

**2. Établissement de l'algorithme** : On formule la résolution du problème sous forme d'une succession logique d'instructions pour passer des données aux résultats.

**3. Établissement de l'organigramme** : C'est la représentation symbolique (graphique) de l'algorithme.

**4. Programmation** : C'est la traduction de l'algorithme en langage évolué.

**5. Compilation** : Correction de toutes les erreurs de programmation.

**6. Exécution** : jeu d'essai.

**7. Vérification** des résultats : Consiste à vérifier si les résultats obtenus correspondent aux solutions du problème.

# 5. Structure d'un algorithme

Un algorithme se compose de trois parties principales :

**- L'Entête** : dans cette partie on déclare le nom de l'algorithme à travers un identificateur.

**- La partie déclarative** : dans cette partie on déclare toutes les données utilisées par l'algorithme.

**- Le corps de programme** : représente la séquence d'actions (instructions) Pour écrire un algorithme.

Ces parties doivent respecter une syntaxe bien déterminée, définie comme suit :

| ***Algorithme*** | |
|---|---|
| Entête | **Algorithme** <identificateur_algo> ; |
| La partie déclarative | **Constantes** < identificateur_constante> = valeur ;<br>**Variables** < identificateur_variable> : Type; |
| Le corps du programme | **Début**<br><Instruction 1> ;<br><Instruction 2> ;<br>..<br><Instruction n> ;<br>**Fin.** |

*Tableau II.1 : Structure d'un algorithme*

| *Programme PASCAL* | |
|---|---|
| Entête | **Program** <identificateur_algo> ; |
| La partie déclarative | **Const** < identificateur_constante> = valeur ;<br>**Var** < identificateur_variable> : Type; |
| Corps du programme | **Begin**<br><Instruction 1> ;<br><Instruction 2> ;<br>..<br><Instruction n> ;<br>**End.** |

*Tableau II.2 : Structure d'un programme en Pascal*

## 5.1. L'Entête

L'entête sert à donner un nom à l'algorithme en utilisant un identificateur. Ce dernier est précédé par le mot clé "**Algorithme**". Alors qu'est-ce qu'un identificateur ?

- **Identificateur** : Un identificateur est une chaîne de caractère qui permet de donner un nom unique à un programme (algorithme), une constante, une variable, une procédure ou une fonction. Cette chaîne doit commencer soit par un caractère alphabétique ou par un tiret du 8 ( _ ) et ne peut contenir que des caractères alphanumériques. Aussi, les mots réservés (mots-clés) d'un langage de programmation ne peuvent être utilisés comme identificateurs. Voici quelques mots réservés au langage Pascal: *Begin, end, program, var, const, real, integer, char, if, then, else, while, for, do, repeat*.

? ***Exemple***

| Identificateur valide | Identificateur non valide |
|---|---|
| a1 | x1 y (contient de l'espace) |
| a_1 | x1-y (contient un caractère spécial ) |
| a_1 | 1xy (commence par un caractère numérique) |
| x12y | End (mots-clés) |
| x1_y | |

*Tableau II.3 : Exemples d'identificateurs valides et non valides*

## 5.2. La partie déclarative

La partie déclarative sert à déclarer les différentes données que l'algorithme utilise (Constantes, variables,.. etc.). Une donnée non déclarée et utilisée par l'algorithme engendre une erreur lors de la compilation. Alors qu'est-ce qu'une variable et qu'est-ce qu'une constante ?

- **Constantes** : une constante est un objet contenant une valeur qui ne peut jamais être modifiée. Son objectif est d'éviter d'utiliser une valeur d'une manière direct. Imaginons qu'un algorithme utilise la valeur 3.14 une dizaines de fois (le nombre d'occurrences de la valeur 3.14 est par exemple 15) et qu'on veut modifier cette valeur par une autre valeur plus précise : 3.14159. Dansce cas on est amené à modifier toutes les occurrences de 3.14. Par contre, si on utilise une constante PI=3.14 on modifier une seule fois cette constante.

**Exemple**

Ci-dessous quelques exemples de déclarations de constantes :

Const PI = 3.14 ; → Constante réelle.

Const MAX = 10 ; → Constante entière.

Const cc = 'a' ; → Constante caractère.

Const ss = 'algo '; → Constante chaîne de caractère.

Const b1 = true ; → Constante booléenne.

Const b2 = false ; → Constante booléenne.

- **Variables** : une variable est un objet contenant une valeur pouvant être modifiée.

Le tableau II.4 résume les 5 types de variables :

| Algorithme | Pascal | Valeurs |
|---|---|---|
| Entier | Integer | Représente l'ensemble {…, 4, 3, 2, 1, 0, 1, 2, 3, 4, ...} |
| Réel | Real | Représente les valeurs numériques fractionnels et avec des virgule fixes (ou flottante) |
| Booléen | Boolean | Représente les deux valeurs *TRUE* et *FALSE*. |
| Caractère | Char | Représente tous les caractères imprimable. |
| Chaîne de caractères | String | Une séquence d'un ou plusieurs caractères |

*Tableau II.4 : Les 5 types de base*

**Exemple**

| Algorithme | Pascal | Signification |
|---|---|---|
| x : réel | x : real; | variable réelle |
| n, m : entier | n, m : integer; | deux variables entières |
| s : chaîne de caractères | s : String; | variables chaîne de caractères |
| b1, b2, b3 : booleen | b1, b2, b3 : boolean; | 3 variables booléennes |
| c1 : caractère | c1 : char; | variable caractère |

*Tableau II.5 : Exemples de variables*

**Remarque**

- Pour commenter un programme Pascal, on écrit les commentaires entre les accolades { }. Par exemple : {Ceci est un commentaire}.

- Dans un programme Pascal, on déclare les constantes dans une section qui commence parle mot clé **const**.

- Dans un programme Pascal, on déclare les variables dans une section qui commence parle mot clé **var**.

- En plus des constantes et des variables, il est possible de déclarer de nouveaux types, des étiquettes, et (dans un programme Pascal) des fonctions et des procédures.

### 5.3. Le corps de programme

Le corps d'un algorithme est constitué d'un ensemble d'actions / instructions ordonnées de manière séquentielle et logique. Ces instructions se divisant en cinq types distincts:

**- Lecture** : Cette opération consiste à introduire des données dans l'algorithme. Une lecture consiste à donner une valeur arbitraire à une variable.

**- Écriture** : Cette opération implique l'affichage de données. Elle permet d'afficher des résultats ou des messages.

**- Affectation** : Elle permet de modifier les valeurs des variables en leur assignant de nouvelles valeurs.

**- Structures de contrôle** : Ces structures permettent de modifier la séquentialité de l'algorithme. Elles sont utilisées pour sélectionner différents chemins d'exécution ou pour répéter un traitement.

→ Structure de Test alternatif simple / double

→ Structure répétitives (itérative)

Dans le langage Pascal, chaque instruction se termine par un **point-virgule**. Sauf à la fin du programme, on met un **point**.

## 6. Types d'intructions

Toutes les instructions d'un programme sont écrites dans le corps du prgramme. (entre *Début* et *Fin*, i.e. *Begin* et *End*.). On peut regrouper ces instructions en trois types : les entrées/sorties (saisi de valeurs et l'affichage des résultat), l'affectation et les structures de contrôles (tests et les boucles)

### 6.1. Instructions d'Entrées/Sorties (Lecture / Écriture)

#### a) Entrées ( Lecture )

Une instruction d'entrée nous permet dans un programme de donner une valeur quelconque à une variable. Ceci se réalise à travers l'opération de lecture.

La syntaxe et la sémantique d'une lecture est comme suit :

| Algorithme | Pascal | Signification |
|---|---|---|
| Lire(<id_var>) ; | read(<id_var>);<br>Readln(<id_var>); | Donner une valeur quelconque à la variable dont l'identifiant <*id_var*>. |
| Lire(<iv1>, <iv2>, …); | read(<iv1>, <iv2>, …); | Donner des valeurs aux variables <iv1>, <iv2>, *etc.* |

*Tableau II.6 : La syntaxe et la sémantique d'une lecture*

***Remarque***

Il est important de noter que l'instruction de lecture concerne uniquement les variables, on peut pas lire des constantes ou des valeurs. Lors de la lecture d'une variable dans un programme Pascal, le programme se bloque en attendant la saisie d'une valeur via le clavier. Une fois la valeur saisie, on valide par la touche entrée, et le programme reprend l'exécution avec l'instruction suivante.

**Exemple**

| Algorithme | Pascal |
|---|---|
| Lire (a, b, c) ; | Read (a, b, c) ; |
| Lire (hauteur) ; | read (hauteur ) ; |

*Tableau II.7 : Exemples d'entrées*

### b) Sorties ( Écriture )

Une instruction de sortie nous permet dans un programme d'afficher un résultat (données traitées) ou bien un message (chaîne de caractères). Ceci se réalise à travers l'opération d'écriture.

La syntaxe et la sémantique d'une écriture est comme suit :

| Algorithme | Pascal | Signification |
|---|---|---|
| Lire(<id_var>\| <id_const>\| <valeur>, <expression>) | write(<id_var> \| <id_const> \| <valeur>, <expression>); writeln(<id_var> \| <id_const> \| <valeur>, <expression>); | Afficher une valeur d'une variable, d'une constante, valeur immédiate ou calculée à travers une expression. |

*Tableau II.8 : La syntaxe et la sémantique d'une écriture*

**Remarque**

Il est à noter que l'instruction d'écriture ne concerne pas uniquement les variables, on peut écrire des constantes, valeurs ou des expressions (arithmétiques ou logiques). On peut afficher unevaleur et sauter la ligne juste après à travers l'instruction : **writeln**.

**Exemple**

| Algorithme | Pascal | Signification |
|---|---|---|
| écrire('Bonjour') ; | write ('Bonjour'); | Afficher le message Bonjour |
| écrire(a, b, c) ; | write(a, b, c); | Afficher les valeurs des variables a, b et c |
| écrire(a, b, c) ; | write(5+2); | Afficher le résultat de la somme de 5 et 2 : afficher 7 |
| écrire(a+b-c) ; | write(a+b-c); | Afficher le résultat de l'expression arithmétique : a+b-c |
| écrire(5<2) ; | write(5<2); | Afficher le résultat de la comparaison 5 < 2 : **FALSE** |
| écrire('La valeur de x : ', x) ; | write('La valeur de x :', x); | |

*Tableau II.9 : Exemples de sorties*

## 6.2. Instruction d'affectation

Une affectation consiste à donner une valeur (immédiate, constante, variable ou calculée à travers une expression) à une variable.

La syntaxe d'une affectation est comme suit :

| Algorithme | Pascal |
|---|---|
| <id_varialbe> ← <valeur> \| <id_variable> \| <expression> | <id_varialbe> := <valeur> \| <id_variable> \| <expression> ; |

*Tableau II.10 : La syntaxe d'une affectation*

Une affectation possède deux parties : la partie gauche qui représente toujours une variable, et la partie droite qui peut être : une valeur, variable ou une expression. La condition qu'une affectation soit correcte est que : la partie droite doit être du même type (ou de type compatible) avec la partie gauche.

**Exemple**

| Algorithme | Pascal | Signification |
|---|---|---|
| *a ←5*<br>*b ← a+5*<br>*sup ← a>b* | *a:=5;*<br>*b:=a+5;*<br>*sup:=a>b;* | Mettre la valeur 5 dans la variable a<br>Mettre la valeur de l'expression a+5 dans la variable B<br>a>b donne un résultat booléen, donc sup est une variable booléenne |

*Tableau II.11 : Exemples d'affectation*

## 6.3. Structures de contrôles

En générale, les instructions d'un programme sont exécutés d'une manière séquentielle : la première instruction, ensuite la deuxième, après la troisième et ainsi de suite. Cependant, dans plusieurs cas, on est amené soit à choisir entre deux ou plusieurs chemins d'exécution (un choix entre deux ou plusieurs options), ou bien à répéter l'exécution d'un ensemble d'instructions, pour cela nous avons besoins de structures de contrôle pour contrôler et choisir les chemins d'exécutions ou refaire un traitement plusieurs fois. Les structures de contrôle sont de deux types : Structures de contrôles conditionnelles et structures de contrôle répétitives (itératives).

### a) Structures de contrôle conditionnelle

Ces structures sont utilisées pour décider de l'exécution d'un bloc d'instruction : est ce que ce bloc est exécuté ou non. Ou bien pour choisir entre l'exécution de deux blocs différents. Nous avons deux types de structures conditionnelles :

#### i) Test alternatif simple

Un test simple contient un seul bloc d'instructions. Selon une condition (expression logique), on décide est ce que le bloc d'instructions est exécuté ou non. Si la condition est vraie, on exécute le bloc, sinon on l'exécute pas.

La syntaxe d'un test alternatif simple est comme suit :

| Algorithme | Pascal |
|---|---|
| **si** <condition> **alors**<br>  <instruction(s)><br>**finsi**; | **if** <condition> **then**<br>**begin**<br>  <instruction(s)>;<br> **end**; |

*Tableau II.12 : La syntaxe d'un test alternatif simple*

**Exemple**

| Algorithme | Pascal |
|---|---|
| lire(x)<br>**si** x > 2 **alors**<br>  x ← x + 3<br>**finsi**<br>écrire (x) | read(x);<br>**if** x > 2 **then**<br>  **begin**<br>  x:= x + 3;<br>**end**;<br>write(x); |

*Tableau II.13 : Exemple d'un test alternatif simple*

**Remarque**

Dans le langage Pascal, un bloc est délimité par les deux mots clés **begin** et **end**.

Si le bloc contient une seule instruction, **begin** et **end** sont facultatifs (on peut les enlever).

### ii) Test alternatif double

Un test double contient deux blocs d'instructions : on est amené à décider entre le premier bloc ou le seconds. Cette décision est réalisée selon une condition (expression logique ou booléenne) qui peut être vraie ou fausse. Si la condition est vraie on exécute le premier bloc, sinon on exécute le second.

La syntaxe d'un test alternatif double est comme suit :

| Algorithme | Pascal |
|---|---|
| **si** <condition> **alors**<br><instruction(s)1><br>**sinon**<br><instruciton(s)2><br>**finsi** | **if** <condition> **then**<br>**begin**<br><instruction(s)1>;<br>**end**<br>**else**<br>**begin**<br><instruction(s)2>;<br>**end** |

*Tableau II.14 : La syntaxe d'un test alternatif double*

***Exemple***

| Algorithme | Pascal |
|---|---|
| lire(x)<br>**si** x > 2 **alors**<br>x ← x + 3<br>**sinon**<br>x ← x - 2<br>**finsi**<br>écrire (x) | read(x);<br>**if** x > 2 **then**<br>**begin**<br>x:= x + 3;<br>**end**<br>**else**<br>**begin**<br>x:= x - 2;<br>**end**;<br>write(x); |

*Tableau II.15 : Exemple d'un test alternatif double*

***Remarque***

- Dans le langage Pascal, il faut jamais mettre de point-virgule avant **else**.

- Dans l'exemple précédent, on peut enlever **begin end** du **if** et ceux du **else** puisqu'il y a une seule instruction dans les deux blocs.

## b) Structures de contrôle répétitives

Les structures répétitives nous permettent de répéter un traitement un nombre fini de fois. Par exemple, on veut afficher tous les nombre premier entre 1 et N (N nombre entier positif donné). Nous avons trois types de structures itératives (boucles) :

### i) 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 :

| Algorithme | Pascal |
|---|---|
| **pour** <indice> ← <vi> **à** <vf> **faire**<br>  <instruction(s)><br>**finPour**; | **for** <indice>:=<vi> **to** <vf> **do**<br>  **begin**<br>  <instruction(s)>;<br>  **end**; |

*Tableau II.16 : La syntaxe de la boucle pour*

<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>.

**Attention**

Il ne faut jamais mettre de point-virgule après le mot clé **do**.

### ii) Boucle Tant-que (While)

La structure de contrôle répétitive **tantque** (**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 **tantque** est comme suit :

| Algorithme | Pascal |
|---|---|
| **tant-que** <condition> **faire**<br>  <instruction(s)><br>**fin tant-que**; | **while** <condition> **do**<br>  **begin**<br>  <instruction(s)>;<br>  **end**; |

*Tableau II.17 : La syntaxe de la boucle tantque*

<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**.

**Remarque**

Il est possible de remplacer toute boucle "**pour**" par une boucle "**tantque**", cependant, l'inverse n'est pas toujours réalisable.

### iii) 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 :

| Algorithme | Pascal |
|---|---|
| **repéter**<br>  <instruction(s)><br>**Jusqu'à** <condition>; | **repeat**<br>  <instruction(s)>;<br>**until** <condition>; |

*Tableau II.18 : La syntaxe de la boucle répéter*

<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 **tantque** est :

- La condition de **répéter** et toujours l'inverse de la condition **tantque** : pour **répéter** c'est la condition de sortie de la boucle, et pour **tantque** 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 **tantque**. C'est-à-dire, dans **tantque** on teste la condition avant d'entrer à l'itération, et dans **répéter** on fait l'itération après on teste la condition.

## c) Structure de contrôle de branchements / sauts (l'instruction Goto)

Une instruction de branchement nous permet de sauter à un endroit du programme et continuer l'exécution à partir de cet endroit. Pour réaliser un branchement, il faut tout d'abord indiquer la cibledu branchement via une étiquette <num_etiq> : . Après on saute à cette endroit par l'instruction aller à <num_etiq> (en pascal : goto <num_etiq>).

La syntaxe d'un branchement est comme suit :

| Algorithme | Pascal |
|---|---|
| **aller à** <num_etiq><br>  .<br>  .<br>  .<br><num_etiq> : .<br>  .<br>  . | **goto** <num_etiq>;<br>  .<br>  .<br>  .<br><num_etiq> : .<br>  .<br>  . |

*Tableau II.19 : La syntaxe d'un branchement*

**Remarque**

- Une étiquette représente un numéro (nombre entier), exemple : 1, 2, 3, etc.

- Dans un programme Pascal, il faut déclarer les étiquettes dans la partie déclaration avec le mot clé **label**. (on a vu **const** pour les constantes **var** pour les variables)

- Une étiquette désigne un seule endroit dans le programme, on peut jamais indiquer deux endroits avec une même étiquette.

- Par contre, on peut réaliser plusieurs branchement vers une même étiquette.

- Un saut ou un branchement peut être vers une instruction antérieure ou postérieure (avant ou après le saut).

***Exemple***

| Algorithme | Pascal |
|---|---|
| **algorithme** branchement<br>**variables**<br>a, b, c : entier ;<br>**début**<br>lire (a, b);<br>2: c ← a;<br>**si** (a > b) **alors**<br>**aller à** 1;<br>**finsi**<br>a ← a + 5;<br>**aller à** 2;<br><br>1: écrire (c);<br>**fin** | **program** branchement;<br>**uses** wincrt;<br>**var**<br>a, b, c:integer;<br>**label** 1, 2;<br>**begin**<br>read(a,b);<br>2: c:=a;<br><br>**if** (a>b) **then**<br>**goto**1;<br>a := a + 5;<br>**goto** 2;<br><br>1: write(c);<br>**end.** |

*Tableau II.20 : Exemple de branchement*

Dans l'exemple ci-dessus, il y a deux étiquettes : 1 et 2. L'étiquette 1 fait référence la dernière instruction de l'algorithme / programme (écrire(c) / write(c) ;), et l'étiquette 2 fait référence la troisième instruction de l'algorithme / programme (c ← a; / c := a;). Pour le déroulement de l'algorithme, on utilise le tableau suivant (a = 2 et b = 5) :

| Variables / Instructions | a | b | c |
|---|---|---|---|
| Lire (a, b) Donner deux valeur quelconque à a et b | 2 | 5 | ? |
| c ← a ; | 2 | 5 | 2 |
| a > b → false puisque a = 2 et b =5 on entre pas aubloc du si<br>a ← a + 5; | 7 | 5 | 2 |
| aller à 2<br>c ← a; | 7 | 5 | 7 |
| a > b → true puisque a = 7 etb =5 on entre au blocdu si<br>aller à 1 =>écrire (c) | 7 | 5 | 7<br>(résultat affiché) |

*Tableau II.21 : Déroulement de l'algorithme*

Il y a deux types de branchement :

- **Branchement inconditionnel** : c'est un branchement sans condition, il n'appartient pas à un bloc de si ou un bloc sinon. Dans l'exemple précédent, l'instruction aller à 2 (goto 2) est un saut inconditionnel.

- **Branchement conditionnel** : Par contre, un branchement conditionnel est un saut qui appartient à un bloc si ou un bloc sinon. L'instruction aller à 1 (goto 1), dans l'exemple précédent est un saut conditionnel puisque il appartient un bloc si.

[cf. ]

# 7. 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 |
| Variables | 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 … |

*Tableau II.22 : Correspondance Algorithme-Pascal*

***Remarque***

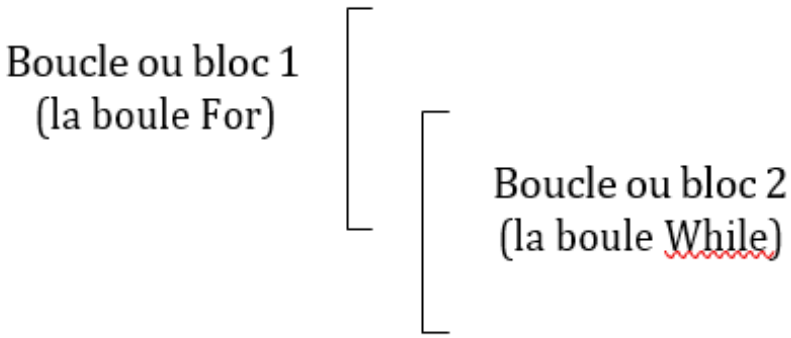
1. Langage Pascal est insensible à la casse, c'est-à-dire, si on écrit begin, Begin ou BEGIN c'est la même chose.

2. 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 défini 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 entreBEGIN et END, s'appelle un BLOC ou action composée. On dit qu'un programme Pascal est structurée en blocs.

3. Il est interdit de chevaucher deux structures de boucles ou de blocs. Par exemple :

```
FOR..... DO
  BEGIN
        ......
        WHILE..... DO
          BEGIN
             .......
  END;
             .......
          END;
```

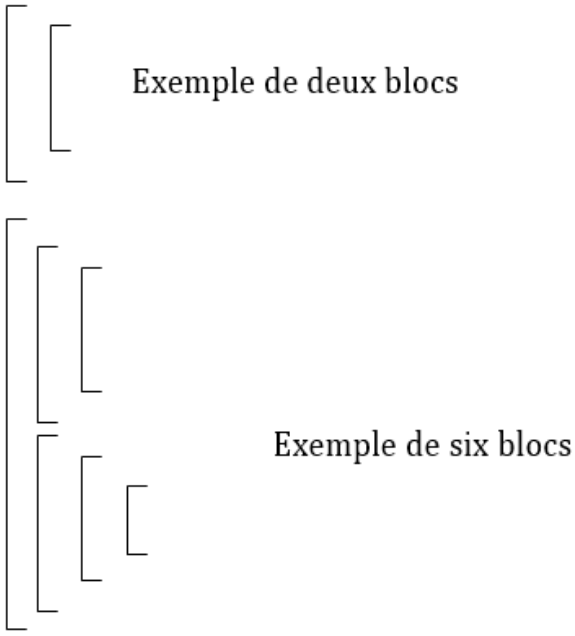
On a eu la forme suivante :




Ce qui est interdit.

Les boucles et blocs ne doivent en aucun cas chevaucher, ils doivent êtres imbriqués.

**→ structures autorisées :**




***Conseil***

Pour une initiation à la programmation avec le langage PASCAL, je vous invite à consulter OPENCLASSROOMS[1].

[1] https://openclassrooms.com/fr/

# 8. Représentation en organigramme

Un organigramme est la représentation graphique de la résolution d'un problème. Il est similaire à un algorithme. 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.

## 8.1. Les symboles d'organigramme

Les symboles utilisés dans les organigrammes sont illustrés dans le tableau II.23

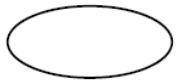
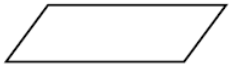
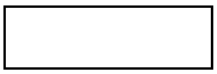
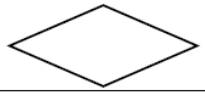
| | |
|---|---|
|  | Représente le début et la Fin de l'organigramme |
|  | Entrées / Sorties : Lecture des données et écriture des résultats. |
|  | Calculs, traitements |
|  | Tests et décision : on écrit le test à l'intérieur du losange |
|  | Ordre d'exécution des opérations (Enchaînement) |
|  | Connecteur |

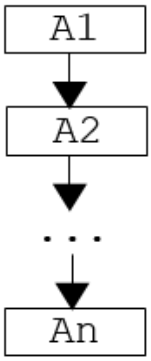
*Tableau II.23 : Les symboles d'organigramme*

# 9. Représentation des primitives algorithmiques

## 9.1. L'enchaînement

L'enchaînement permet d'exécuter une série d'actions dans l'ordre le leur apparition. Soit $A_1$, $A_2$, ...,$A_n$ une série d'actions, leur enchaînement est représenté comme suit :

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




$A_1$, $A_2$, ...,$A_n$ : peuvent être des actions élémentaires ou complexes.

## 9.2. La structure alternative simple

La syntaxe et l'organigramme de la structure alternative simple sont présentés dans le tableau II.24.

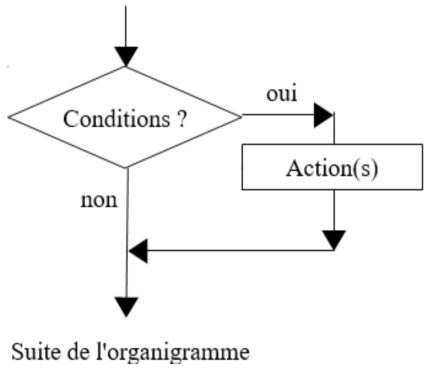
| Représentation algorithmique | Représentation sous forme d'organigramme |
|---|---|
| **si** <condition> **alors**<br><action(s)>;<br>**finsi**;<br><br>Si la condition est vérifiée, le bloc *<action(s)>* sera exécuté, sinonrien, et on Continue l'exécution de l'instruction après fin si. |  |

*Tableau II.24 : La syntaxe et l'organigramme de la structure alternative simple*

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 desexpressions logiques à partir d'autres expressions logique en utilisant les opérateurs suivant : Not, Or et And.

***Exemple***

(x >= 5) : est une expression logique, elle est vrai si la valeur de x est supérieur ou égale à 5. elle est fausse dans le cas contraire.

Not (x >= 5) : E.L. qui est vrai uniquement si la valeur de x est inférieur à 5.

(x >=5) And (y<=0) : E.L. qui est vrai si x est supérieur ou égale à 5 et y inférieur ou égale à 0.

## 9.3. La structure alternative double

La syntaxe et l'organigramme de la structure alternative double sont présentés dans le tableau II.25 :

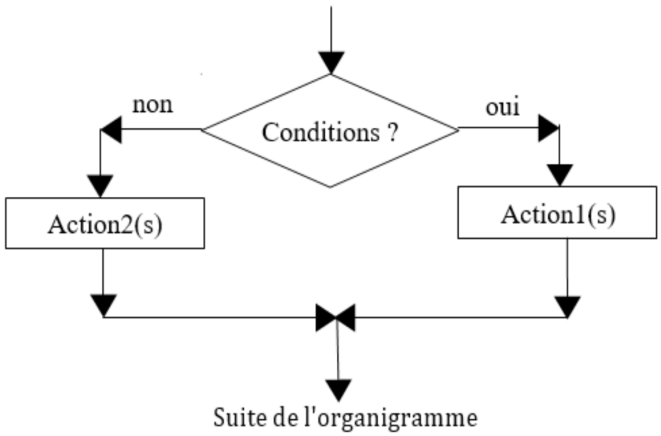
| Représentation algorithmique | Représentation sous forme d'organigramme |
|---|---|
| **si** <condition> **alors**<br><action1(s)>;<br>sinon<br><action2(s)>;<br>**finsi**;<br><br>Si la condition est vérifiée, le bloc *<action1(s)>* sera exécuté, sinon (si elle estfausse) onexécute *<action2(s)>*. |  |

*Tableau II.25 : La syntaxe et l'organigramme de la structure alternative double*

## 9.4. La structure itérative POUR (Boucle POUR)

La syntaxe et l'organigramme de la structure itérative POUR sont présentés dans le tableau II.26.

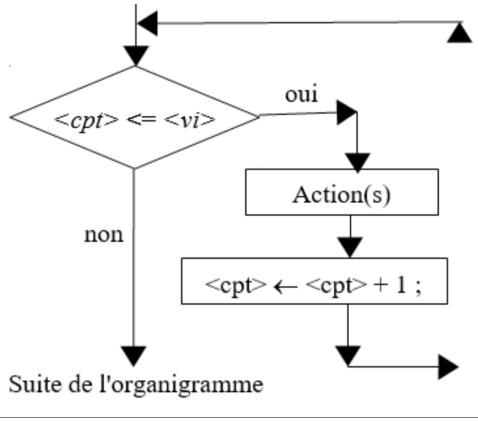
| Représentation algorithmique | Représentation sous forme d'organigramme |
|---|---|
| **pour** <cpt> ← <vi> **à** <vf> **faire**<br><action(s)>;<br>**finpour**; |  |

*Tableau II.26 : La syntaxe et l'organigramme de la structure itérative POUR*

Dans la boucle **POUR**, on exécute le bloc <acitons> (<vf> - <vi> + 1) fois. Ceci dans le cas où <vf> est supérieur ou égale à <vi>. Dans le cas contraire, le bloc d'actions ne sera jamais exécuté. Le déroulement de la boucle POUR est exprimé comme suit :

1. La variable entière <cpt> (le compteur) prends la valeur initiale <vi> ;

2. on compare la valeur de <cpt> à celle de <vf> ; si <cpt> est supérieur à <vf> on sort de laboucle ;

3. Si <cpt> est inférieur ou égale à <vf> on exécute le bloc <action(s)> ;

4. La boucle POUR incrémente automatiquement le compteur <cpt>, c'est-à-dire elle lui ajoute un (<cpt> <cpt> + 1);

5. On revient à 2 (pour refaire le teste <cpt> <= <vi> C'est pour cela qu'on dit la boucle);

***Remarque***

La boucle **POUR** est souvent utilisée pour les structures de données itératives (les tableaux et les matrices – variables indicées).

## 9.5. La structure itérative Tant-que (Boucle Tant-que)

La syntaxe et l'organigramme de la structure itérative Tant-que sont présentés dans le tableau II.27.

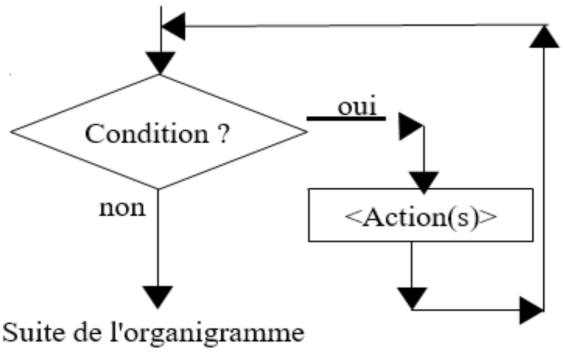
| Représentation algorithmique | Représentation sous forme d'organigramme |
|---|---|
| **Tant-que** <condition> **faire**<br><action(s)>;<br>**FinTant-que**; |  |

*Tableau II.27 : La syntaxe et l'organigramme de la structure itérative Tant-que*

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 ;

– Si la condition est vraie, on exécute le bloc <actions> ;

2. On revient à 1 ;

3. On continue la suite de l'algorithme

## 9.6. La structure itérative Répéter (Boucle Répéter)

La syntaxe et l'organigramme de la structure itérative répéter sont présentés dans le tableau II.28.

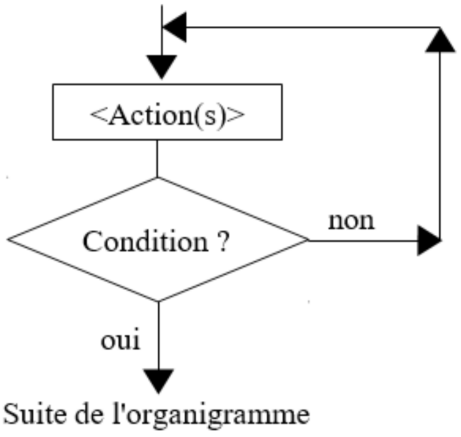
| Représentation algorithmique | Représentation sous forme d'organigramme |
|---|---|
| Répéter<br><action(s)>;<br>**Jusqu'à**<br><condition>; |  |

*Tableau II.28 : La syntaxe et l'organigramme de la structure itérative répéter*

n 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.

***Remarque***

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**<br><action(s)>;<br>**finpour**; | <cpt> ← <vi>;<br>**Tant-que** <cpt> <= <vf> **faire**<br><action(s)>;<br><cpt> ← <cpt> + 1;<br>**FinTant-que**; |

*Tableau II.29 : Transformation de la boucle POUR à la boucle Tant-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 onentre dans la boucle).
- La boucle Répéter exécute le bloc <action(s)> au moins une fois, le teste vient après l'exécution du bloc.
- La boucle Tant-Que peut ne pas exécuter le bloc <action(s)> (dans le cas où la condition est fausse dès le début), puisque le teste est avant l'exécution du bloc.

# 10. Exercices corrigés

**Exercice N°1 :**

Donner le type des variables suivantes : 2010 ; 124.5 ; 667.0E-8 ; 'A' ; TRUE ; False ; 'division par zéro'.

**Solution :**

Type des variables

| Variable | Type |
|---|---|
| 2010 | Entier / Integer |
| 124.5 | Réel / Real |
| 667.0E-8 | Réel / Real |
| 'A' | Caractère / Char |
| TRUE | Booléen / Boolean |
| False | Booléen / Boolean |
| 'division par zéro' | Chaîne de caractère / String |

**Exercice N°02 :** (Identificateurs)

Identifier les identificateurs valides et non valides : 1A ; R? ; K2 ; T280 ; 12R ; Hauteur ; Prix- HT ; Prix_HT ; Exo 04 ; Exo_04 ; Exo-04 ; Program ; read.

**Solution :**

Les variables valides et non valides :

| Variable valide | Variable non valide |
|---|---|
| K2 | 1A |
| T280 | R? |
| Hauteur | 12R |
| Prix_HT | Prix-HT |
| Exo_04 | Exo 04 |
| | Exo-04 |
| | Program |
| | read |

**Exercice N°03 :** (Enoncé du problème → Algorithme → Programme)

Écrire un algorithme, puis traduit le en programme PASCAL, pour chacun des problèmes suivants :

1) Permuter entre les deux variables X et Y ?

2) Calculer le quotient et le reste de la division euclidienne de a par b ?

**Solution :**

1) Permuter entre les deux variables X et Y ?

| Algorithme | Programme PASCAL |
|---|---|
| Algorithme Exo2_1;<br>**Variables**<br>X, y, t : **entier**; | **Program** Exo2_1;<br>**Var**<br>X, y, t : **integer**; |
| Début<br>{-*-*-*- Entrées -*-*-*-}<br>**Lire**(x, y) ; | **Begin**<br>{-*-*-*- Entrées -*-*-*-}<br>**Read**(x,y) ; |
| {-*-*-*- Traitement -*-*-*-}<br>t ← x;<br>x ← y; y ← t ;<br>{-*-*-*- Sorties -*-*-*-}<br>Écrire('x=', x, 'y=', y) ;<br>**Fin.** | {-*-*-*- Traitement -*-*-*-}<br>t := x;{on conserve la valeur de X dans t}<br>x := y; {pas de risque de perte de valeur}<br>y := t; {on récupère l'ancienne valeur de x}<br><br>{-*-*-*- Sorties -*-*-*-}<br>**Write**('x=', x, 'y=', y);<br>**End.** |

2) Calculer le quotient et le reste de la division euclidienne de a par b ?

| Algorithme | Programme PASCAL |
|---|---|
| **Algorithme** Exo2_4;<br>Variables<br>A, b, Q, R : **entier**; | **Program** Exo2_4;<br>Var<br>a, b, Q, R : **integer**; |
| **Début**<br>{-*-*-*- Entrées -*-*-*-}<br>**Lire**(a, b) ; | **Begin**<br>{-*-*-*- Entrées -*-*-*-}<br>**Read**(a, b) ; |
| {-*-*-*- Traitements -*-*-*-}<br>Q ← a div b;<br>R ← a mod b; | {-*-*-*- Traitements -*-*-*-}<br>Q := a div b;<br>R := a mod b; |
| {-*-*-*- Sorties -*-*-*-}<br>**Écrire**('Le quotient est : ', Q, 'et le reste est :', R) ;<br>**Fin.** | {-*-*-*- Sorties -*-*-*-}<br>**Write**('Le quotient est : ', Q, 'et le reste est :' , R) ;<br>**End.** |