Plan du Chapitre 2

Chapitre 2 : Notion d'Algorithme et de Programme....2
2.1. Concept d'un algorithme....2
2.2. La démarche et analyse d'un problème....2
2.3. Structures de données....3
2.3.1. Notion d'identificateur....3
2.3.2. Constantes et variables....4
2.3.3. Types de données....4
2.4. Structure d'un algorithme / programme....5
2.4.1. Déclarations....5
2.4.2. Corps....6
2.5.Types d'instructions....7
2.5.1. Instructions d'Entrées/Sorties (Lecture / Écriture)....7
2.5.1.1. Entrées (Lecture)....7
2.5.1.1. Sorties (Écriture)....8
2.5.2. Instruction d'affectation....8
2.5.3. Structures de contrôles....9
2.5.3.1. Structures de contrôle conditionnelle....9
a. Test alternatif simple....9
b. Test alternatif double....10
2.5.3.2. Structures de contrôle répétitives....10
a. Boucle Pour (For)....11
c. Boucle Tant-que (While)....11
c. Boucle Répéter (Repeat)....12
2.5.3.3. Structure de contrôle de branchements / sauts (l'instruction Goto)....12
2.6. Correspondance Algorithme <--> PASCAL....14
2.7. Représentation en organigramme....16
2.7.1. Les symboles d'organigramme....16
2.7.2. Représentation des primitives algorithmiques....17
2.7.2.1. L'enchaînement....17
2.7.2.2. La structure alternative simple....17
2.7.2.3. La structure alternative double....18
2.7.2.4. La structure itérative POUR (Boucle POUR)....18
2.7.2.6. La structure itérative Répéter (Boucle Répéter)....20
2.8. Les opérateurs....21
2.8.1. Les opérateurs arithmétiques 21
2.8.2. Les opérateurs relationnels....22
2.8.3. Les opérateurs logiques....22
2.8.4. Les fonctions....23
2.8.5. Calcul d'expressions : priorités dans les opérateurs....23
a) Les expression arithmétiques....23
b) Les expressions booléennes (logiques)....24
c) Règles de priorités des opérateurs....25
2.9. Exemples d'Application....25
Exemple 1 : Permutation de valeurs de deux variables....25
Exemple 2 : Somme de deux variables....27
Exemple 3 : Nombre pair ou impair....28
Exemple 4 : Afficher tous les nombres divisibles par n....29
Exemple 5 : Rechercher les diviseurs d'un nombre n....32

Chapitre 2 : Notion d'Algorithme et de Programme.

2.1. Concept d'un algorithme

– Le mot « Algorithme » est inventé par le mathématicien « AL-KHAWARISMI ». Un Algorithme est l'énoncé d'une séquence d'actions primitives réalisant un traitement. Il décrit le plan ou les séquences d'actions de résolution d'un problème donné.

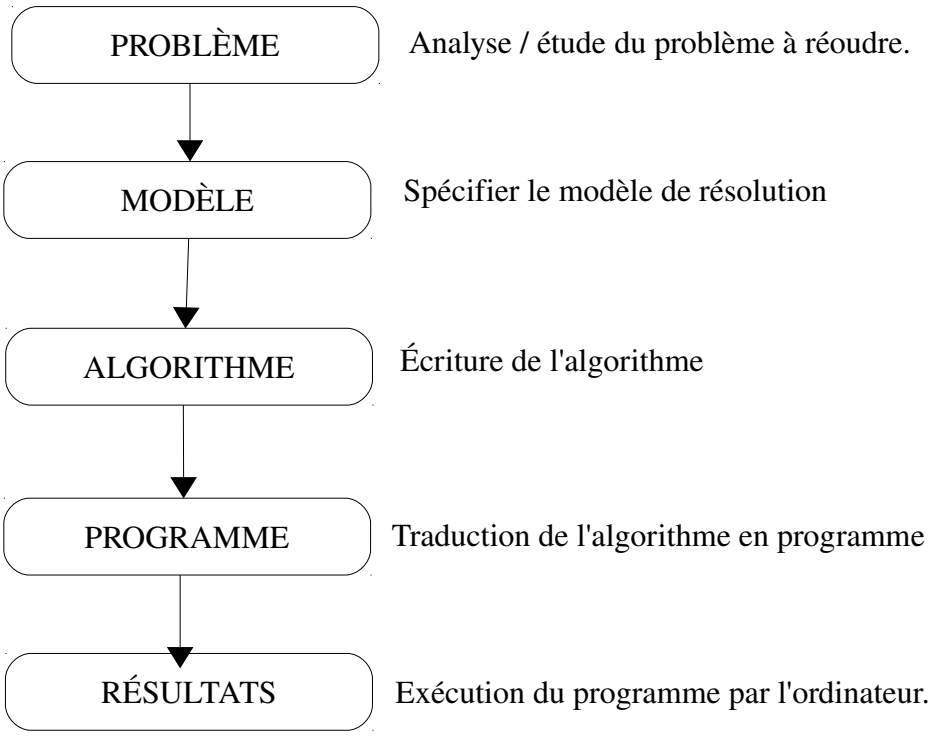
– Un algorithme est un ensemble d'*actions* (ou d'*instructions*) séquentielles et logiquement ordonnées, permettant de transformer des données en entrée (*Inputs*) en données de sorties (*outputs* ou les *résultats*), afin de résoudre un problème.

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.

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

2.2. La démarche et analyse d'un problème

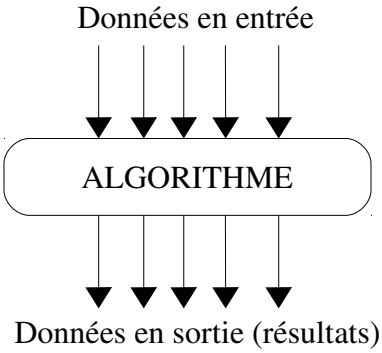
Comme vu dans le point précédent, un algorithme représente une solution à un problème donné. Pour atteindre à cette solution algorithmique un processus d'analyse et de résolution sera appliqué. Ce processus est constitué des étapes suivantes :



2.3. Structures de données

Un algorithme permet de réaliser un traitement sur un ensemble de données en entrées pour produire des données en sorties. Les données en sorties représentent la solution du problème traité par l'algorithme.

Un algorithme peut être schématisé comme suit :



Toute les données d'un programme sont des objets dans la mémoire vive (c'est un espace réservé dans la RAM). Chaque objet (espace mémoire) est désigné par une appellation dite : *identificateur.*

2.3.1. Notion d'identificateur

Un identificateur est une chaîne de caractères contenant uniquement des caractère alpha-numériques (alphabétiques de [a-z] et [A-Z] et numérique [0-9]) et tiré 8 '_' (trait souligné), et qui doit commencer soit par un une lettre alphabétique ou _.

Un identificateur permet d'identifier d'une manière unique un algorithme (ou un programme), une variable, une constante, une procédure ou une fonction.

Dans un langage de programmation donnée, on a pas le droit d'utiliser les mots réservés (mots clés) du langage comme des identificateurs. Parmi les mots clés du langage PASCAL :

```
program, begin, end, if, else, then, while, for, do, to, downto,
repeat, until, goto, procedure, function, label, var, const, type,
uses, array, of, real, integer, boolean, char, string, ...
```

Exemples :

`a1` : est un identificateur valide.
`a_1` : est un identificateur valide.
`A_1` : est un identificateur valide.
`x12y` : est un identificateur valide.
`x1 y` : est un identificateur non valide (à cause du blanc ou l'espace).
`x1-y` : est un identificateur non valide (à cause du signe -).
`x1_y` : est un identificateur valide.
`1xy` : est un identificateur non valide (commence un caractères numérique).

2.3.2. Constantes et variables

Les données manipulées par un algorithme (ou un programme) sont soit des constantes ou des variables :

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

Toute les données (variable ou constante) d'un algorithme possède un type de données (domaine de valeurs possibles).

2.3.3. Types de données

Dans l'algorithmique, nous avons cinq types de base :

- Entiers : représente l'ensemble {…, -4, -3, -2, -1, 0, 1, 2, 3, 4, ...}
- Réels : représente les valeurs numériques fractionnels et avec des virgule fixes (ou flottante)
- Caractères : représente touts les caractères imprimable.
- Chaînes de caractères : une séquence d'un ou plusieurs caractères
- Booléens (logique) : représente les deux valeurs *TRUE* et *FALSE*.

Le tableau suivant montre la correspondance entre les types d'algorithme et les types du langage de programmation PASCAL.

Algorithme	***PASCAL***
Entier	Integer
Réel	Real
Booléen	Boolean
Caractère	Char
chaîne	String

Exemples de Déclarations de constantes et variables :

```
CONST  PI = 3.14 ;              {constante réelle}
CONST  CHARGE = 1.6E-19;        {charge de l'électron}
CONST  MASSE = 0.9E-27;         {masse de l'électron}
```

```
CONST message = 'Bonjour tous le monde';     {constante chaîne de caractère}
```

Remarques

- Pour commenter un programme PASCAL, on écrit les commentaires entre les accolades { }. Par exemple : *{Ceci est un commentaire}*.
- Dans un programme PASCAL, on déclares les constantes dans une zones qui commence par le mot clé **const**.
- Dans un programme PASCAL, on déclares les variables dans une zones qui commence par le mot clé **var**.

2.4. Structure d'un algorithme / programme

Un algorithme manipule des données, les données avant de les utiliser il faut les identifier et les déclarer en utilisant les identificateur. Un algorithme est constitué de trois parties :

- ***Entête :*** dans cette partie on déclare le nom de l'algorithme à travers un identificateur.
- ***Déclarations :*** dans cette partie on déclare toutes les données utilisées par l'algorithme.
- ***Corps :*** représente la séquence d'actions (instructions)

Pour écrire un algorithme, il faut suivre la structure suivante :

```
Algorithme <identificateur_algo>          Program prog1;
      <Déclarations>                      uses wincrt;
                                                <déclarations>;
Début                                     Begin
   |  <Corps>                                    <Instructions>;
   |                                      End.
Fin.
```

2.4.1. Déclarations

Dans la partie déclaration, on déclare toutes les données d'entrées et de sorties sous forme de constantes et de variables.

- ***Constantes***

  Les constantes sont des objets contenant des valeurs non modifiables. Les constante sont déclarées comme suit :

  <identificateur> = <valeur>;

  Exemples :

  ```
  PI = 3.14;
  ```
  Constante réelle.

`MAX = 10;`	Constante entière.
`cc = 'a';`	Constante caractère.
`ss = 'algo';`	Constante chaîne de caractère.
`b1 = true;`	Constante booléenne.
`b2 = false;`	Constante booléenne.

- ***Variables***

Les variables sont des objets contenant des valeurs pouvant êtes modifiées. Les variables sont déclarées comme suit :

<identificateur> : <type>;

Une variable appartient à un type de données. On a cinq types de données de base :

- *Entiers*
- *Réels*
- *Caractères*
- *Chaîne de caractères*
- *Booléens*, contenant deux valeurs : *True* ou *False* ;

Exemples :

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

N.B. : On peut, en plus de constante et de variables, déclarer de nouveau types, des étiquettes et (dans un programme PASCAL) des fonctions et procédures.

2.4.2. Corps

Le corps d'un algorithme est constitué d'un ensemble d'actions / instructions séquentiellement et logiquement ordonnées. Les instructions sont de cinq types, à savoir :

- ***Lecture* :** L'opération de faire entrer des données à l'algorithme. Une lecture consiste à donner une valeur arbitraire à une variable.
- ***Écriture* :** L'opération d'affichage de données. Elle sert à afficher des résultat ou des messages.

- ***Affectation*** **:** ça sert à modifier les valeurs des variables.
- ***Structure de contrôle*** **:** Elle permet modifier la séquentialité de l'algorithme, pour choisir un chemin d'exécution ou répéter un traitement.
  - *Structure de* ***Test alternatif simple / double***
  - *Structure* répétitives (itérative)
    - *la boucle* ***Pour***
    - *la boucle* ***Tant-que***
    - *la boucle* ***Répéter***

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

2.5.Types d'instructions

Tous les instructions d'un programme sont écrits dans son corps. (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*)

2.5.1. Instructions d'Entrées/Sorties (Lecture / Écriture)

2.5.1.1. 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>);` `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.*

Il faut remarquer 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.

Exemples :

```
Lire (a, b, c)    read(a, b, c);              lire (hauteur)    read(hauteur);
```

2.5.1.1. 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.

Il faut remarquer 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 une valeur et sauter la ligne juste après à travers l'instruction : `writeln`.

Exemples :

```
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(5+2)            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, le résultat est la
                                               valeur booléenne FALSE}
écrire('La valeur de x : ', x)   write('La valeur de x :', x);
```

2.5.2. 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 :

```
<id_varialbe> ← <valeur>|<id_variable>|<expression>
<id_varialbe> :=  <valeur>|<id_variable>|<expression> ;
```

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.

Exemples :

```
a ← 5       a:=5;        {mettre la valeur 5 dans la variable a}
b ← a+5     b:=a+5;      {mettre la valeur de l'expression a+5 dans la variable B}
sup ← a>b   sup:=a>b;    {a>b donne un résultat booléen, donc sup est une variable
                          booléenne}
```

2.5.3. 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).

2.5.3.1. 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 :

a. 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 :

```
si <Condition> alors
    <instruction(s)>
finsi;
```

```
if <condition> then
 begin
     <instruction(s)>;
 end;
```

Exemple :

```
lire(x)
si x > 2 alors
    x ← x + 3
finsi
écrire (x)
```

```
read(x);
if x > 2 then
begin
     x:= x + 3;
end;
write(x);
```

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).

b. 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 simple est :

```
si <Condition> alors
|     <instruction(s)1>
sinon
      <instruciton(s)2>
finsi
```

```
if <condition> then
 begin
      <instruction(s)1>;
 end
else
 begin
      <instruction(s)2>;
 end;
```

Exemple :

```
lire(x)
si x > 2 alors
|     x ← x + 3
sinon
|     x ← x - 2
finsi
écrire (x)
```

```
read(x);
if x > 2 then
 begin
      x:= x + 3;
 end
else
 begin
      x:= x - 2;
 end;
write(x);
```

Remarques :

- 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.

Exemples :

1- Écrire un algorithme (et un programme PASCAL) qui permet d'indiquer si un nombre entier est pair ou non.

2.5.3.2. 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) :

a. 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>.

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

c. 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
    <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.

c. 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 :

```
répéter                                  repeat
    <instruction(s)>                         <instruction(s)>;
jusqu'à <condition>;                     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.

2.5.3.3. 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 cible du 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 :

```
aller à <num_etiq>                       goto <num_etiq>;
.                                        .
.                                        .
.                                        .
<num_etiq> : .                           <num_etiq> : .
             .                                        .
             .                                        .
```

N.B. :

- 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 branchement
variables
      a, b, c : entier ;
début
      lire (a, b);
      2: c ← a;

      si (a > b) alors
            aller à 1;
      finsi
      a ← a + 5;
      aller à 2;

      1: écrire (c);
fin
```

```
program branchement;
uses wincrt;
var
      a, b, c:integer;
label 1, 2;
begin
      read(a,b);
      2: c:=a;

      if (a>b) then
            goto 1;
      a := a + 5;
      goto 2;

      1: write(c);
end.
```

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 au bloc du *si* *a ← a + 5;*	*7*	*5*	*2*
aller à 2 *c ← a;*	*7*	*5*	*7*
a > b → true puisque *a = 7* et *b =5* on entre au bloc du *si* *aller à 1 => écrire (c)*	*7*	*5*	*7* *(résultat affiché)*

Il y a deux types de branchement :

a. 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.

b. 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 à appartient au bloc *si*.

2.6. 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

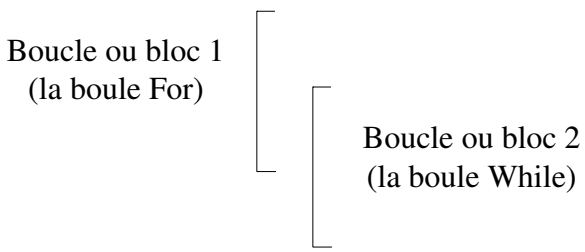
1. Langage PASCAL est insensible à la casse, c'est-à-dire, si on écrit begin, Begin ou BEGIN c'est la même chose.

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 entre BEGIN 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;
```

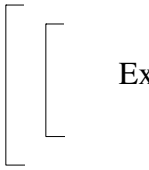
On a eu la forme suivante :



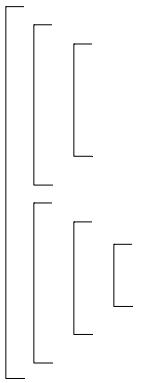
Ce qui est interdit.

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

Exemples de structures autorisées :



Exemple de deux blocs



Exemple de six blocs

2.7. 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.

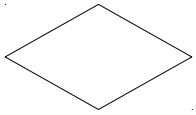
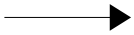
Les inconvénients qu'on peut rencontrer lors de l'utilisation des organigrammes sont :

- 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.

2.7.1. Les symboles d'organigramme

Les symboles utilisés dans les organigrammes :

	Représente le début et la Fin de l'organigramme

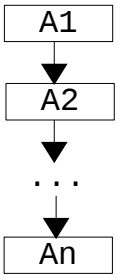
	Entrées / Sorties : Lecture des données et écriture des résultats.
	Calculs, Traitements
	Tests et décision : on écrit le test à l'intérieur du losange
	Ordre d'exécution des opérations (Enchaînement)
	Connecteur

2.7.2. Représentation des primitives algorithmiques

2.7.2.1. L'enchaînement

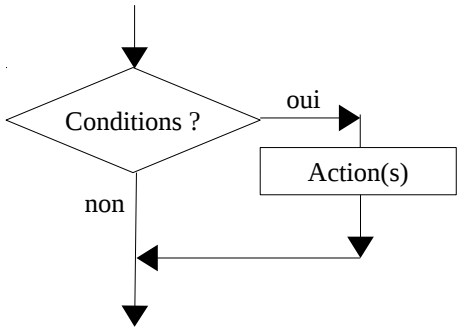
L'enchaînement permet d'exécuter une série d'actions dans l'ordre le leur apparition. Soit A_1, A_2, …,A_n une série d'actions, leur enchaînement est représenté comme suit :

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



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

2.7.2.2. 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.

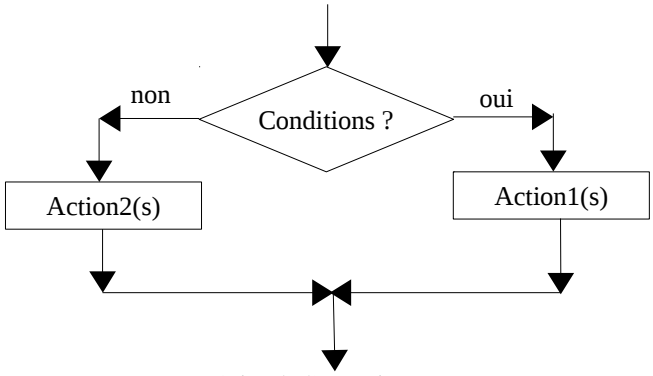
Exemples :

(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.

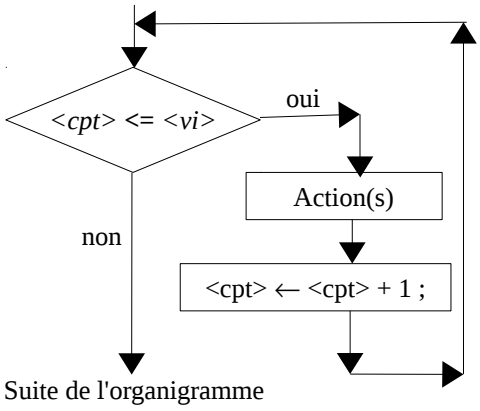
Not (x >= 5) : E.L. qui est vrai uniquement si la valeur de x est inférieur à 5.

(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.

2.7.2.3. 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)*>.	

2.7.2.4. La structure itérative POUR (Boucle POUR)

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

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 la boucle ;

3 – si <*cpt*> est inférieur ou égale à <*vf*> on exécute le bloc <*action(s)*> ;

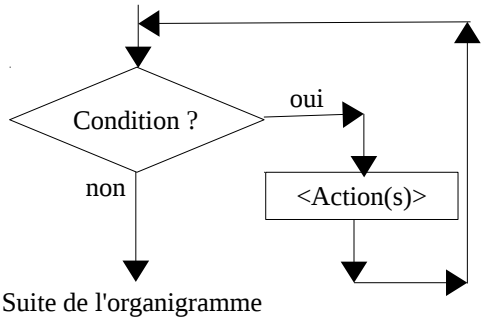
4 – la boucle *POUR* incrémente automatiquement le compteur <*cpt*>, c'est-à-dire elle lui ajoute *un* (<cpt> ← <cpt> + 1);

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).

2.7.2.5. 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 :

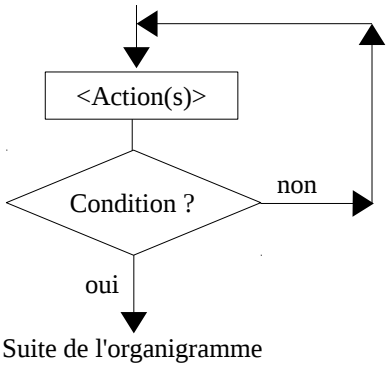
1 – On évalue la condition : si la condition est fausse on sort de la boucle ;

2 – Si la condition est vraie, on exécute le bloc <actions> ; sinon va à **4**.

3 – On revient à 1 ;

4 – On continue la suite de l'algorithme

2.7.2.6. 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

l'exécution du bloc.

✔ La boucle Tant-Que peut ne pas exécuter le bloc <action(s)> (dans le cas ou la condition est fausse dès le début), puisque le teste est avant l'exécution du bloc.

2.8. Les opérateurs

Les opérateurs dans l'algorithmique (ou dans la programmation) nous permettent d'écrire des expression qui seront évaluées par la l'ordinateur. La valeur d'une expression soit elle est affectée à une variable, affichée ou bien utilisé dans un teste. On trois type d'opérateurs : Opérateurs arithmétiques, opérateurs relationnels (de comparaison) et opérateurs logiques.

2.8.1. Les opérateurs arithmétiques

Nous distinguons les opérateurs arithmétiques suivants :

+	*(Addition)*
-	*(Soustraction)* **-** *(unaire)*
*****	*(Multiplications)*
/	*(Division)*
div	*(Division entière, elle fournit la partie entière d'une division)*
mod	*(Modulo ou reste de division)*

Les opérateurs (+, -, *) peuvent être appliqués à des opérandes de type entier ou réel.

Si les deux opérandes sont de type entier, le résultat est de type entier. Soit :

entier + entier = entier

entier – entier = entier

entier * entier = entier

Le moins unaire s'applique à un seul opérande. Exemple, le nombre négatif : *-5*

Dans le cas d l'opérateur / le résultat est réel quelque soit le type des deux opérandes. Soit :

entier / entier = réel.

Dans le cas des opérateurs DIV et MOD, les opérateurs doivent être obligatoirement des entiers et le résultat est une entier.

entier DIV entier = entier

entier MOD entier = entier

2.8.2. Les opérateurs relationnels

On distingue les opérateurs de relation suivants :

= (égale), <> (différent), < (inférieur), > (supérieur), <= (inférieur ou égale), >= (supérieur ou égale)

Appliqués aux opérandes, ils fournissent une résultat booléens. Soit :

12 = 5 fournit le résultat (faux) ou (false) en anglais

12 = 12 fournit le résultat (vrai) ou (true) en anglais

45 < 49 fournit le résultat (vrai)

etc.

En générale, on utilise les opérateurs relationnels dans les testes de conditions des boucles **tant-que** et **répéter**.

2.8.3. Les opérateurs logiques

On distingue les opérateurs logique suivant : AND, OR et NOT.

Il s'applique à des opérandes de type booléen, et fournissent une valeur de type booléen (logique). Les opérateurs AND et OR sont des opérateurs binaires, c'est-à-dire qu'ils s'appliquent à deux opérandes.

Par exemple, on écrit : opérande1 AND opérande2

L'opérateur NOT est un opérateur unaire, c'est-à-dire qu'ils s'applique à un seul opérande. Soit :

NOT opérande.

Exemple :

(45 > 59) AND (15 = 15) → faux AND vrai → Résultat : faux (FALSE)

(25 > 45) OR (47 < 50) → faux OR vrai → Résultat : vrai (TRUE)

NOT (25 > 45) → NOT (Faux) → Résultat : vrai (TRUE)

Les tableaux suivants récapitulent l'utilisation de ces opérateurs :

Opérande1	*Opérande2*	*Op1 AND Op2*
True	True	True
True	False	False
False	True	False
False	False	False

Opérande1	*Opérande2*	*Op1 OR Op2*
True	True	True
True	False	True
False	True	True
False	False	False

Opérande	*NOT Opérande*
True	False
False	True

True : Vrai

False : Faux

2.8.4. Les fonctions

Pour le moment on se limitera aux fonctions standards (ou prédéfinies).

Les fonctions standards 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*
Etc.		

Il existe également des fonctions standards appliquées aux caractères, aux chaîne de caractère et aux ensemble qu'il n'est pas urgent de les traiter.

2.8.5. Calcul d'expressions : priorités dans les opérateurs

On distingue les expressions arithmétiques et les expressions booléennes (ou logique) :

a) Les expression arithmétiques

Un expression arithmétique est constituée d'opérandes numériques reliés par des opérateurs arithmétiques.

Exemple :

5 + 6 / 3 donne *7* car *6/3* est évalué en premier, puisque */* est plus prioritaire que +.

(5+6) / 3 donne *3.66* car on évalue d'abord le contenu des parenthèses.

Règles d'évaluation des expression arithmétiques

Règle 1 : On évalue d'abord le contenu des parenthèses en commençant par les parenthèses les plus internes.

Règle 2 : On commence à effectuer les opérateurs de plus haute priorité. Dans le cas des opérateurs de même priorité, on commence par le plus à gauche.

Exemples :

a) 7 + 9 / 3 – 10 * 2
(1)3

7 + 3 – 10 * 2
(2)20

7 + 3 – 20
(3)10

10 – 20
(4)-10

-10

b) (5 * 6 + (8 + 2*7 – 4 / 2))
(5 * 6 + (8 + 2*7 – 4 / 2))
(1)
(5 * 6 + (8+14 – 4 / 2))
(2)
(5 * 6 + (8+14 – 2))
(3)
(5 * 6 + (22 – 2))
(4)
(5 * 6 + 20)
(5)
30 + 20
(6)
50

Exemples de Configuration :

((_____) + (_____))
(1) (2)
(3)

On commence par la plus à gauche de même profondeur.

((_____) - ((_____)))
(2) (1)
(3)
(4)

On commence par la plus profonde.

À l'intérieur des parenthèses, les priorités des opérateurs sont respectées.

b) Les expressions booléennes (logiques)

Une expression booléenne (ou expression logique) est une expression dont le résultat est de type booléen. Elle peut comporter des opérateurs arithmétiques, des opérateurs de relation et des opérateurs booléens.

Exemples :

(5+2 > 8-6) AND (7 < 9)
(1) (2) (4)
(3)
(5)
= (7 > 2) AND (7 < 9)
= TRUE AND TRUE
= TRUE

c) Règles de priorités des opérateurs

La priorité des opérateurs arithmétiques et logiques est dans l'ordre suivant :

1) Les parenthèses

2) Les fonctions

3) Le moins unaire, le NOT

*4) *, /, DIV, MOD, AND*

5) +, -, OR

6) =, <>, <, >, <=, >=

2.9. Exemples d'Application

Exemple 1 : Permutation de valeurs de deux variables

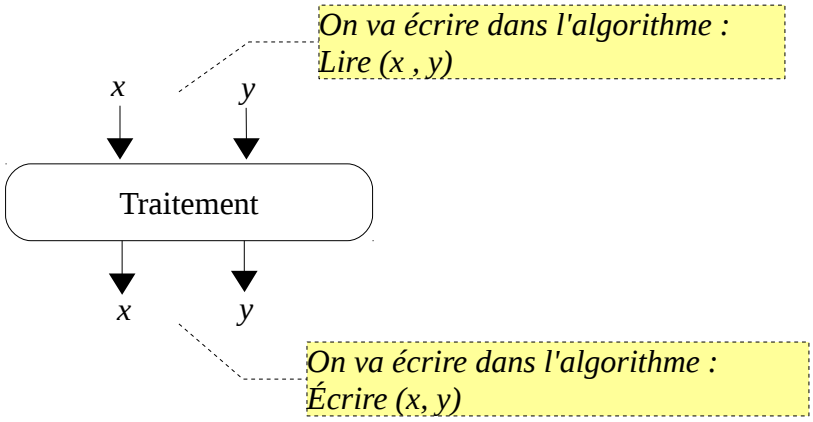
Écrire un algorithme qui permet de permuter les valeurs de deux variables réelles. Par exemple $x = 18$ et $y = -20$ au début de l'algorithme, à la fin on obtient $x = -20$ et $y = 18$.

Traduire l'algorithme en programme PASCAL, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : On veut réaliser la permutation de deux variables, on doit tout d'abord donner deux valeurs aux deux variables (on aura besoins de deux lectures). À la fin, on doit afficher les mêmes variables, après avoir été permutées.

Donc L'algorithme possède deux variables d'entrée et deux variables de sortie. Les variables de sortie sont les mêmes variables d'entrée. On peut schématiser l'algorithme comme suit :



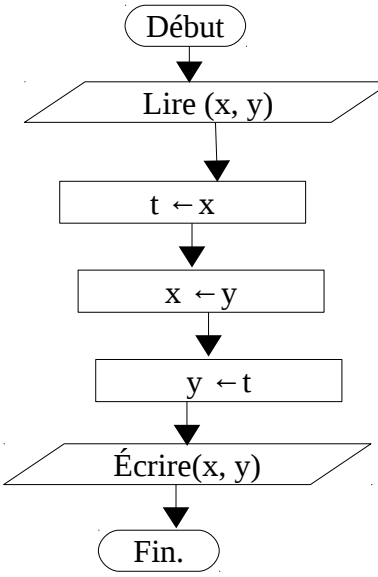
Après avoir déterminé les variables d'entrée et de sortie, il nous reste à trouver l'idée du traitement. La solution qui vient immédiatement à l'esprit est : $x \leftarrow y$ et $y \leftarrow x$. (On affecte y à x, enduite on affecte x à y). Le problème de cette solution est qu'on va perdre la valeur initiale de x.

(Explication : …). il faut conserver tout d'abord la valeur de x dans une autre variable, par exemple *t*, en suite, on affecte *t* à *y*. (t ← *x* et *x* ← *y* enfin *y* ← *t*).

Donc l'algorithme (et sa traduction en programme PASCAL) sera comme suit :

```
Algorithme exemple1;
      Variables
            x, y, t : reel
Début
      Lire (x, y)
      t ← x
      x ← y
      y ← t
      Ecrire (x, y)
Fin
```

```
Program exemple1;
uses wincrt;
var
      x, y, t:real;
BEGIN
      write ('Donnez la valeur de x et y : ');
      read (x, y);
      t:=x;
      x:=y;
      y:=t;
      write('x = ', x, ' et y =', y);
END.
```



- Dans l'organigramme, on schématise le chemin d'exécution, et on indique le séquencement des actions à réaliser. On remarque dans l'organigramme à gauche, qu'il y a un seul chemin d'exécution (un seul sens pour les flêches). Dans ce cas l'exécution est déterministe (On connais au préalables toutes les actions qui seront exécutée).

- Dans les exemples qui suivent, on peut éventuellement trouver plusieurs chemins d'exécution.

N.B. : On peut généraliser le problème, en permutant, d'une manière circulaire, les valeurs de trois variable *x*, *y* et *z*. (aussi pour *4*, *5*, et ainsi de suite de variables).

Le déroulement : On déroule l'algorithme pour *x = 6.5* et *y = 17*

Variables / *Instructions*	*x*	*y*	*t*
Lire (x, y)	*6.5*	*17*	*/*
t ← x	*"*	*"*	*6.5*
x ←y	*17*	*"*	*"*
y ←t	*17*	*6.5*	*"*

Exemple 2 : Somme de deux variables

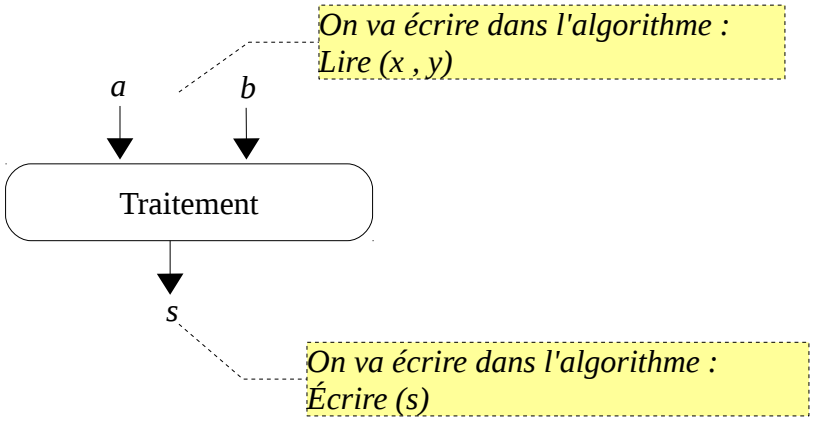
Écrire un algorithme qui permet de réaliser la somme de deux variables entières.

Traduire l'algorithme en programme PASCAL, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : On veut réaliser la somme de deux variables entière *a* et *b*, on doit tout d'abord donner deux valeurs aux deux variables (on aura besoins de deux lectures). À la fin, on doit calculer leur somme dans une troisième variable *s* et afficher ensuite la valeur de *s*.

Donc L'algorithme possède deux variables d'entrée (*a* et *b*) et une variable de sortie *s*. On peut schématiser l'algorithme comme suit :

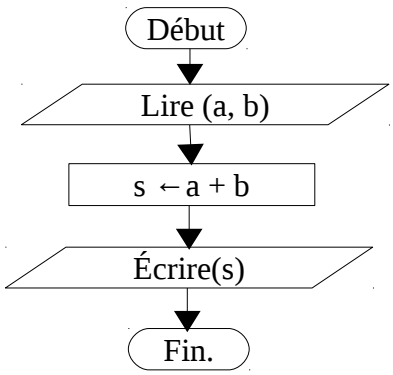


Après avoir déterminé les variables d'entrée et de sortie, il nous reste à trouver l'idée du traitement. Le traitement est simple, il suffit de réaliser l'affectation suivante : $s \leftarrow a + b$.

Donc l'algorithme (et sa traduction en programme PASCAL) sera comme suit :

```
Algorithme somme;
      Variables
            a, b, s : entier
Début
      Lire (a, b)
      s ← a + b
      Ecrire (s)
Fin
```

```
Program somme;
uses wincrt;
var
      a, b, s:real;
BEGIN
      write ('Donnez la valeur de a et b : ');
      read (a, b);
      s:=a+b ;
      write('Somme s =', s);
END.
```



Le déroulement : On déroule l'algorithme pour $a = 5$ et $y = -16$

Variables / *Instructions*	*a*	*b*	*s*
Lire (a,b)	*5*	*-16*	*/*
s ← a+b	*"*	*"*	*-11*

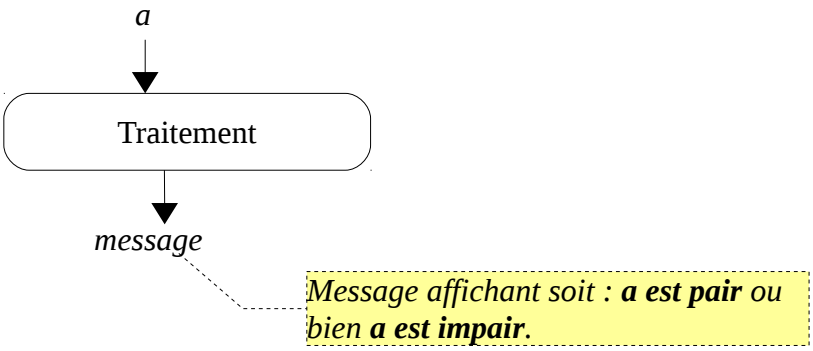
Exemple 3 : Nombre pair ou impair

Écrire un algorithme qui permet d'indiquer si le nombre est pair ou non (un nombre pair est divisible par 2).

Traduire l'algorithme en programme PASCAL, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : L'algorithme prend comme donnée d'entrée une variable entière et affiche un message indiquant si ce nombre là est pair ou non. Un nombre pair est divisible par 2. Donc, nous avons une variable d'entrée et un message comme sortie

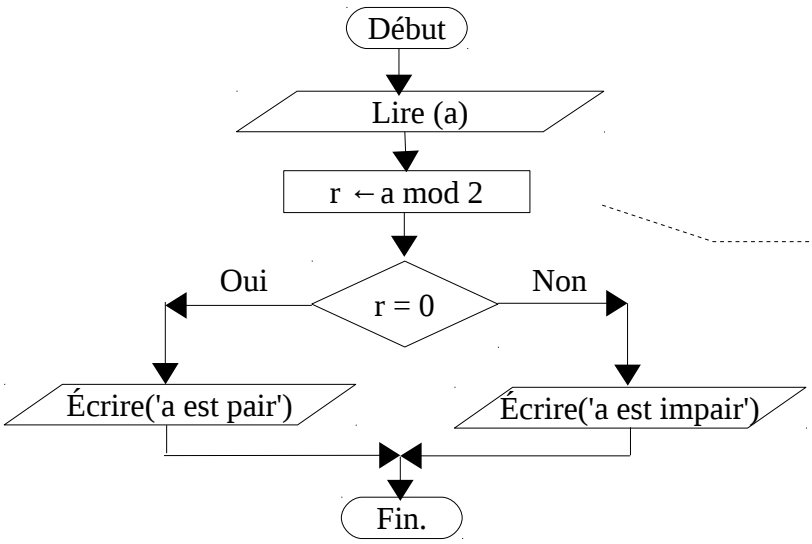


Pour savoir si un nombre est pair ou non, il suffit de calculer son reste de division par 2. On peut utiliser directement la fonction *mod* qui sécrit : *n mod b* = le reste de division de *n* sur *b*.

Donc l'algorithme (et sa traduction en programme PASCAL) sera comme suit :

```
Algorithme pair_impair
      Variables
            a, r : entier
Début
      Lire (a)
      r ← a mod 2
      si (r = 0) alors
         écrire ('a est pair')
      sinon
         écrire ('a est impair')
      finsi
Fin
```

```
Program pair_impair;
uses wincrt;
var
      a,r:integer;
BEGIN
      write ('Donnez la valeur de a : ');
      read (a);   r:= a mod 2;
      if (r = 0) then
            write('a est pair')
      else
            write('a est impair');
END.
```



- Dans cet exemple, on remarque dans l'organigramme que nous avons deux chemins d'exécution. Pour cela, on peut avoir deux scénario de déroulement de l'algorithme : le premier si a est pair et le second si a est impair.

Le déroulement : On déroule l'algorithme pour $a = 7$

Variables / *Instructions*	*a*	*r*
Lire (a)	*7*	*/*
r ← a mod 2	*"*	*1 (= 7 mod 2)*
R = 0 → FALSE *On suit la branche NON* *→ écrire ('a est impair')*	*a est impair*	

On déroule l'algorithme pour $a = 16$

Variables / *Instructions*	*a*	*r*
Lire (a)	*16*	*/*
r ← a mod 2	*"*	*0 (= 16 mod 2)*
R = 0 → TRUE *On suit la branche OUI* *→ écrire ('a est pair')*	*a est pair*	

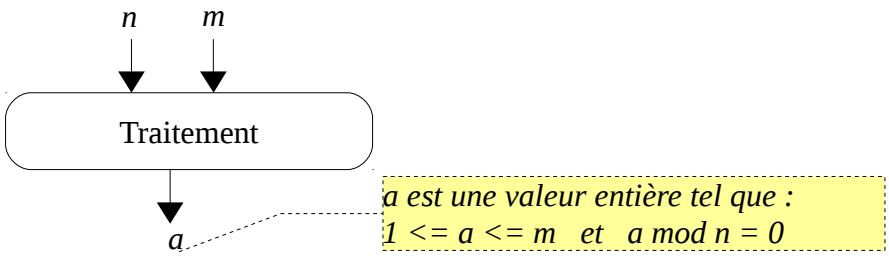
Exemple 4 : Afficher tous les nombres divisibles par *n*

Écrire un algorithme qui permet d'afficher tous les nombres entiers supérieurs ou égale à *1* et inférieurs à *m* (entier positif) et et qui sont divisible sur un nombre entier *n*.

Traduire l'algorithme en programme PASCAL, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : Pour chercher tous les nombres qui sont entre *1* et *m*, on doit donner une valeur pour *m* (donc *m* est une variable d'entrée). Et si on indique que ces nombres sont divisibles par *n*, on doit aussi donner une valeur pour *n* (donc *n* est aussi une variable d'entrée). Comme résultat, l'algorithme affiche tous les nombres entiers divisibles par *n* et qui sont entre *1* et *m*.

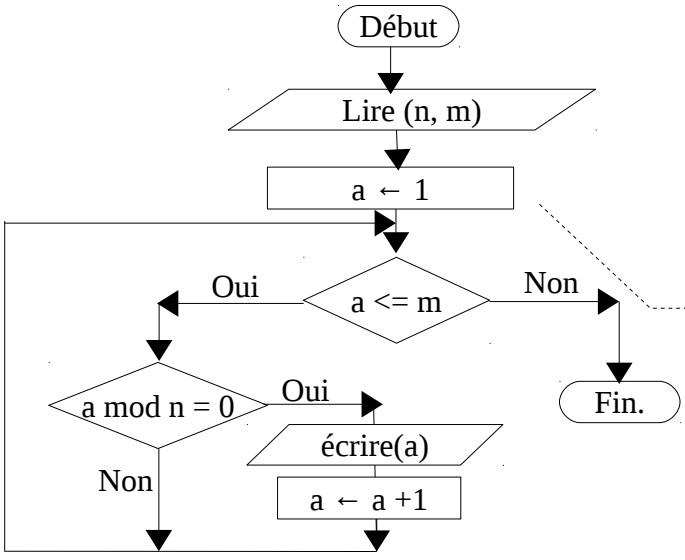


Pour le traitement, on parcours tous les nombres *a* entre *1* et *m* et à chaque itération (boucle) on teste si *a* est divisible par *n* ou non. Dans le cas ou *a* est divisible on l'affiche sinon on fait rien.

Donc l'algorithme (et sa traduction en programme PASCAL) sera comme suit :

```
Algorithme exemple_4
      Variables
            n,m,a : entier
Début
      Lire (n, m)
      pour a ← 1 à m faire
         si (a mod n = 0) alors
            écrire (a)
         finsi
      finPour
Fin
```

```
Program exemple_4;
uses wincrt;
var
      n,m,a : integer;
BEGIN
      write ('Donnez la valeur de n et m : ');
      read (n, m);
      for a:=1 to m do
        begin
            if (a mod n = 0) then
                  writeln(a);
        end;
END.
```



*- La boucle **pour** initialise le compteur (a←1), réalise le teste (a <= m) et incrémente le compte de 1 (a←a+1) à la fin de chaque itération.*

- L'organigramme montre le chemin de la boucle sous forme d'un circuit bouclé.

*- Si le teste a<=m est faux, on sort de la boucle **pour** (et dans cet exemple, on quitte l'algorithme).*

Le déroulement : On déroule l'algorithme pour $n = 3$ et $m = 7$

Variables / *Instructions*	*n*	*m*	*a*
Lire (n, m)	*3*	*7*	*/*
Pour a = 1 *a mod n = 0* *(1 mod 3 = 0) → **false*** *donc on fait rien*	*"*	*"*	*1*
Pour a = 2 *a mod n = 0* *(2 mod 3 = 0) → **false*** *donc on fait rien*	*"*	*"*	*2*
Pour a = 3 *a mod n = 0* *(3 mod 3 = 0) → **true*** *donc on affiche a = 3*	*"*	*"*	*3*
Pour a = 4 *a mod n = 0* *(4 mod 3 = 0) → **false*** *donc on fait rien*	*"*	*"*	*4*
Pour a = 5 *a mod n = 0* *(5 mod 3 = 0) → **false*** *donc on fait rien*	*"*	*"*	*5*
Pour a = 6 *a mod n = 0* *(6 mod 3 = 0) → **true*** *donc on affiche a = 6*	*"*	*"*	*6*
Pour a = 7 *a mod n = 0* *(7 mod 3 = 0) → **false*** *donc on affiche a = 3*	*"*	*"*	*7*
Pour a = 8 *on arrête la boucle (a>m 8 > 7)* *fin de l'algorithme*	*"*	*"*	*8*

N.B. :

– La boucle **Pour** initialise le compteur (c'est une variable entière) de la valeur initiale une seule fois (le premier passage).

– À la fin de chaque itération, on ajoute *1* au compteur de la boucle **Pour**.

– Avant d'accéder à une itération, on réalise une comparaison entre le compteur et la valeur

finale (est-ce-que le compteur est inférieur ou égale à la valeur finale. Si oui, on accède à la boucle, sinon on arrête la boucle).

- Si la valeur finale est strictement inférieure à la valeur initiale, dans ce cas, on accède jamais à la boucle **Pour**.
- On peut remplacer la boucle **Pour** soit par la boucle **Tant-Que** ou bien la boucle **Répéter**.

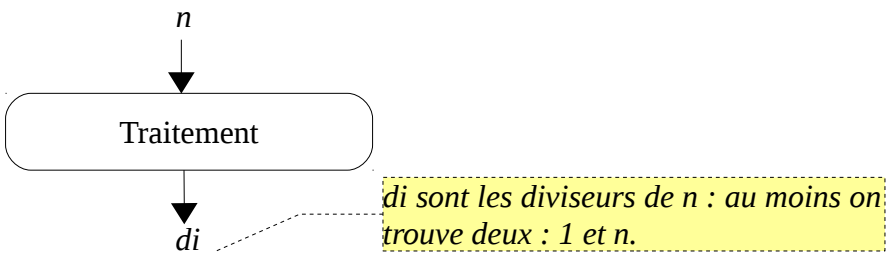
Exemple 5 : *Rechercher les diviseurs d'un nombre n*

Écrire un algorithme qui permet d'afficher tous les diviseurs positifs d'un nombre entier *n*.

Traduire l'algorithme en programme PASCAL, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : Pour trouver tous les diviseurs d'un nombre entier n, il suffit de parcourir les nombre de *1* jusqu'à *n*. Mais, il faut faire attention dans le cas où *n=0*. Dans ce cas tous les nombres entiers sont des diviseurs de *n*.

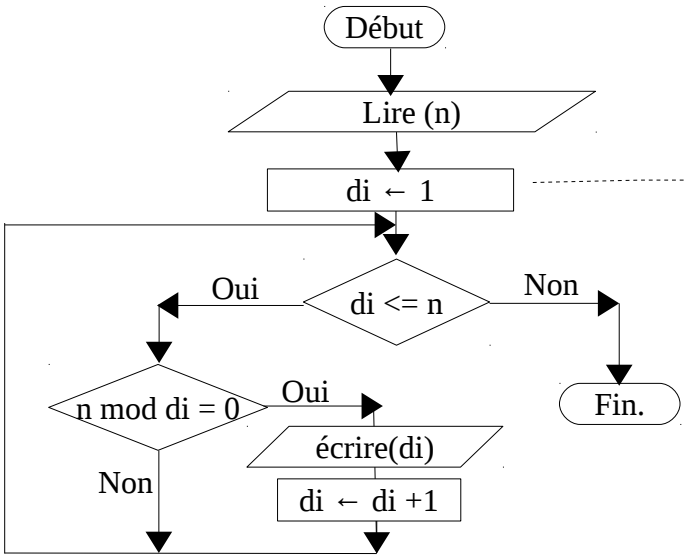


Pour le traitement, on parcours tous les nombres *di* entre *1* et *n* et à chaque itération (boucle) on teste si *n* est divisible par *di* ou non. Dans le cas ou *n* est divisible on affiche alors *di* sinon on fait rien.

Donc l'algorithme (et sa traduction en programme PASCAL) sera comme suit :

```
Algorithme exemple_5
      Variables
            n,di : entier
Début
      Lire (n)
      pour di ← 1 à n faire
         si (n mod di = 0) alors
            écrire (di)
         finsi
      finPour
Fin
```

```
Program exemple_5;
uses wincrt;
var
      n,di : integer;
BEGIN
      write ('Donnez la valeur de n : ');
      read (n);
      for di:=1 to n do
        begin
            if (n mod di = 0) then
                  writeln(di);
        end;
END.
```



*- Dans cette organigramme (algorithme), on a pas traiter le cas de n = 0 (**Donc, essayer de le faire**)*

- Aussi, il faut traiter le cas de n < 0.

Le déroulement : On déroule l'algorithme pour *n = 12*

Variables / *Instructions*	*n*	*di*
Lire (n)	*12*	*/*
Pour di = 1 *n mod di = 0* *(12 mod 1 = 0) → **true*** *donc on affiche di = 1*	*"*	*1*
Pour di = 2 *n mod di = 0* *(12 mod 2 = 0) → **true*** *donc on affiche di = 2*	*"*	*2*
Pour di = 3 *n mod di = 0* *(12 mod 3 = 0) → **true*** *donc on affiche di = 3*	*"*	*3*
Pour di = 4 *n mod di = 0* *(12 mod 4 = 0) → **true*** *donc on affiche di = 4*	*"*	*4*
Pour di = 5 *n mod di = 0* *(12 mod 5 = 0) → **false*** *donc on fait rien*	*"*	*5*
Pour di = 6 *n mod di = 0* *(12 mod 6 = 0) → **true*** *donc on affiche di = 6*	*"*	*6*

Pour di = 7 *n mod di = 0* *(12 mod 7 = 0) → **false*** *donc on fait rien*	*''*	*7*
Pour di = 8 *n mod di = 0* *(12 mod 8 = 0) → **false*** *donc on fait rien*	*''*	*8*
Pour di = 9 *n mod di = 0* *(12 mod 9 = 0) → **false*** *donc on fait rien*	*''*	*9*
Pour di = 10 *n mod di = 0* *(12 mod 10 = 0) → **false*** *donc on fait rien*	*''*	*10*
Pour di = 11 *n mod di = 0* *(12 mod 11 = 0) → **false*** *donc on fait rien*	*''*	*11*
Pour di = 12 *n mod di = 0* *(12 mod 12 = 0) → **true*** *donc on affiche di = 12*		*12*

Donc, l'algorithme affichera pour les diviseurs de *12 : 1, 2, 3, 4, 6, 12*