Chapitre 2 : Algorithmique

Partie I : Concepts de base de l'algorithmique

1. Définition d'un programme : un programme se présente sous la forme de séquences d'instructions, comportant souvent des données de base, devant être exécutées dans un certain ordre par un processeur, Un programme est la forme électronique et numérique d'un algorithme exprimé dans un langage de programmation - un vocabulaire et des règles de ponctuation destinées à exprimer des programmes.

2. Définition d'un algorithme : un algorithme est une suite finie d'instructions à exécuter dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données".

L'écriture de l'algorithme est une étape fondamentale pour résoudre un problème : il permet de réfléchir sur papier et représente une solution indépendante des langages de programmation. Un programme n'est que la traduction de l'algorithme dans un langage de programmation. Dans un algorithme :

- Les instructions sont en **nombre restreint** et doivent être communiquées dans **un ordre précis.**
- La description des actions doit être faite de **manière non ambigüe**.
- Le problème principal du programmeur est donc de décrire la suite des actions élémentaires permettant d'obtenir, à partir des **données fournies**, **les résultats escomptés**.
- Cette description doit envisager le moindre détail et **prévoir les diverses possibilités de solution.**
- La solution doit être généraliste et **non dépendante des données**.

3. Qualités d'un algorithme, d'un programme : pour obtenir un bon programme, il faut partir d'un bon algorithme. Il doit, entre autres, posséder les qualités suivantes :

1. être **clair** (lecture facile du code source)**.**
2. être **structuré** (entête, déclaration, traitement, affichage des résultats)**.**
3. être conçu de manière à **limiter le nombre d'opérations à effectuer** (solution optimale)**.**
4. être **bien documenté** (utiliser des commentaires pour faciliter sa compréhension)**.**

4. Déroulement d'un algorithme : Le déroulement d'un algorithme correspond à l'exécution de toutes ses instructions, ligne après ligne du début jusqu'à la fin.

5. Structure d'un algorithme : un algorithme se présente sous la structure suivante :

Algorithme nom_algorithme //entête de l'algorithme
    //partie déclaration
debut
    //séquence d'instructions } Corps de l'algorithme
Fin

5.1. Entête de l'algorithme : permet simplement d'identifier un algorithme.

5.2. Corps de l'algorithme : c'est la partie qui se trouve entre **début** et **fin**, dans cette partie sont placées les instructions à exécuter.

5.3. Commentaire : permet de documenter l'algorithme et faciliter sa compréhension, les commentaires sont vivement conseillés. Utiliser ‘‘ // ’’ pour écrire un commentaire sur une seule ligne et ‘‘ /* ...*/ ’’ pour un commentaire multi-lignes.

5.4. Partie déclaration :

5.4.1. Déclaration des constantes : une constante désigne un espace mémoire dont la valeur ne peut changer dans le temps. Cette valeur doit être fixée dès la déclaration, Exp : **const** PI ← 3.14 ;

5.4.2. Déclaration des variables : une variable désigne un emplacement mémoire qui permet de stocker une valeur. Une variable est définie par :

- ◊ Un nom unique qui la désigne.
- ◊ Un type de définition.
- ◊ Une valeur attribuée et modifiée au cours du déroulement de l'algorithme.

5.4.2.1. Nom d'une variable (identifiant) : le nom d'une variable est un nom qui permet de l'identifier de manière unique dans l'algorithme. Exp : **var** x, y : **entier** ;

La déclaration d'une variable alloue un espace mémoire dans la mémoire centrale de taille égale à celle spécifiée par son type.

Il faut respecter certaines règles lors du choix du nom d'un identifiant, à savoir :

1. Il est préférable que le nom d'une variable soit évocateur de l'information qu'elle désigne.
2. Le nom doit obligatoirement commencer par une lettre.
3. Le nom peut contenir des lettres, des chiffres et le caractère « _ ».
4. Le nom ne doit en aucun cas contenir des espaces.

5.4.2.2. Type d'une variable : le type d'une variable, appelé aussi domaine de définition, indique l'ensemble des valeurs que la variable peut prendre ainsi que la taille de l'espace mémoire à retenir. Il existe plusieurs domaines : entier, réel, caractère, chaine, booléen.

a. Le type numérique : Les variables de type numérique utilisées dans l'algorithmique ont comme domaines usuels : réel (**réel**) ou entier (**entier**). Les opérateurs arithmétiques utilisables dans ces domaines sont : l'addition (+), la soustraction (-), le produit (*), la division (/) et l'exponentiel (^). Les opérateurs : division entière (div) et reste de la division entière (mod) concernent les entiers. On peut aussi appliquer sur les éléments de type entier ou réel, les opérateurs de comparaison classique : >, <, ≠, =, ≥, ≤.

a.1. Fonctions arithmétiques standards pour les types numériques:

Fonction	**Type paramètre**	**Type résultat**	**Description**	**Exemple**
Tronc(x)	Réel	Entier	Extraire la partie entière de x.	Tronc(-5.125) vaut -5 Tronc(3.14) vaut 3
Arrondi(x)	Réel	Entier	Arrondit une valeur réelle à l'entier le plus proche.	Arrondi(7.499) vaut 7 Arrondi(7.50) vaut 8 Arrondi(7.99) vaut 8
Abs(x)	Entier ou réel	Entier/Réel	renvoie la valeur absolue de x.	Abs(-3) vaut 3
Carrée(x)	Entier ou réel	Entier/Réel	donne le carre de x.	Carré(3) vaut 9
RacineCarrée(x)	Entier ou réel	Réel	donne la racine carrée de x (x doit être positif)	RacineCarré(9) vaut 3.0

b. Type caractère et chaine de caractères : Il s'agit d'un domaine constitué des caractères alphabétiques, numériques et de ponctuation et les autres symboles &, #, …etc. Les opérations élémentaires pour les éléments de

type caractère (**char**) ou chaine de caractères (**chaine**) sont les opérations de comparaison : >, <, ≠, =, ≥, ≤ et le + pour la concaténation. Exp :

x : char ; // x ← 'A'
Nom : chaine ; // nom ← ''Brahim'' ;
Adresse : chaine[60] ; //Adresse ← ''Cité Badr, Boumahra, Guelma'' ;

La comparaison des caractères portent sur les valeurs des codes ASCII correspondants à ces caractères, le code ASCII est cohérent avec l'ordre lexicographique on a : "0" < "1" … <"9" < … "A" < "B" … < "Z" … "a" < "b" <"z" …, voir le tableau suivant :

Exemple de caractères	**Code ASCII**
Chiffres de 0 à 9	de 48 à 57
Lettres de A à Z	de 65 à 90
Lettres de a à z	de 97 à 122
La barre d'espace	32

c. Type booléen : Le domaine des booléens (**booléen**) est l'ensemble formé des deux seules valeurs {vrai, faux}. Les opérations admissibles sur les éléments de ce domaine sont réalisées à l'aide de tous les connecteurs logique notés : et (et logique), ou (ou logique), ouex (ou exclusif) et non (négation logique), ainsi que les deux opérateurs de comparaison =, ≠.

X	Y	x et y	x ou y	x ouex y	non x
Vrai	Vrai	Vrai	Vrai	Faux	Faux
Vrai	Faux	Faux	Vrai	Vrai	Faux
Faux	Vrai	Faux	Vrai	Vrai	Vrai
Faux	Faux	Faux	Faux	Faux	Vrai

De plus, les opérateurs : >, <, ≠, =, ≥, ≤ appliquée sur des opérandes entier, réel, caractère ou chaine de caractère produisent comme résultat une valeur booléenne.

d. Les types énumérés : un type énuméré est un type dont les variables associées n'auront qu'un nombre limité de valeurs à prendre. Définir un type énuméré consiste à énumérer la liste de ses valeurs possibles. Exp :

Type Jour = (Dimanche, Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi) ;
Type Couleur = (Rouge, Bleu, Vert, Jaune, Blanc, Noir) ;
Var j : Jour ; // j : (Dimanche, Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi) ;
c : Couleur ; // c : (Rouge, Bleu, Vert, Jaune, Blanc, Noir) ;

Les opérations applicables aux valeurs de type énumérés sont : >, <, ≠, =, ≥, ≤. De plus, on peut utiliser les fonctions suivantes :

Fonction	Description	Exemple
Succ(x)	Fournit le successeur de la valeur donnée	Succ(Lundi) = Maridi
Pred(x)	Fournit le prédécesseur de la valeur donnée	Pred(Lundi) = Dimanche
Ord(x)	Fournit l'ordre de la valeur donnée	Ord(Lundi) = 2

e. Le type Tableau : Un tableau est une structure de données qui désigne par une seule variable (nom du tableau) un ensemble de valeurs de même type accessibles via leurs positions dans le tableau.

La dimension et le type du tableau doivent être précisés lors de sa définition. Pour accéder aux éléments du tableau, un indice est utilisé, ce dernier indique le rang de l'élément.

e.1. Tableau à une dimension :

Const n ← 10 ;
Var tab : tableau[1..n] d'entier ;

tab	22	-7	56	12	0	23	-4	1	57	35
Indice	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Nous constatons que tab[6] = 23, tab[3] = 56, tab[9] = 57, …etc.

e.2. Tableau à deux dimensions :

Const nombre_lignes ← 3 ;
nombre_colonnes ← 3 ;
Var tab : tableau[1 .. nombre_lignes][1 .. nombre_colonnes] d'entier ;

[1][1]	[1][2]	[1][3]
[2][1]	[2][2]	[2][3]
[3][1]	[3][2]	[3][3]

Partie II

1. instructions de base

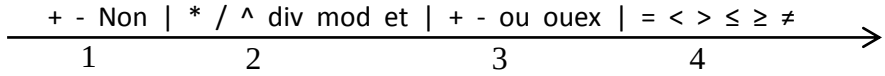
1.1. Instruction d'affectation : les variables changent de valeur grâce à l'opération d'affectation. Donc, l'affectation est une opération qui fixe une nouvelle valeur à une variable. Exp : x ← 5.

D'une manière générale, dans une instruction d'affectation, on trouve :

- à gauche de la flèche, un nom de variable déjà déclarée.
- à droite de la flèche, une expression.

a. Définition d'une expression : Une expression est une formule de calcul constituée **d'opérandes** et **d'opérateurs**. Les opérandes sont soit des constantes, des variables ou des fonctions.

b. Règles d'évaluation des expressions : Il existe une certaine priorité entre les opérateurs :



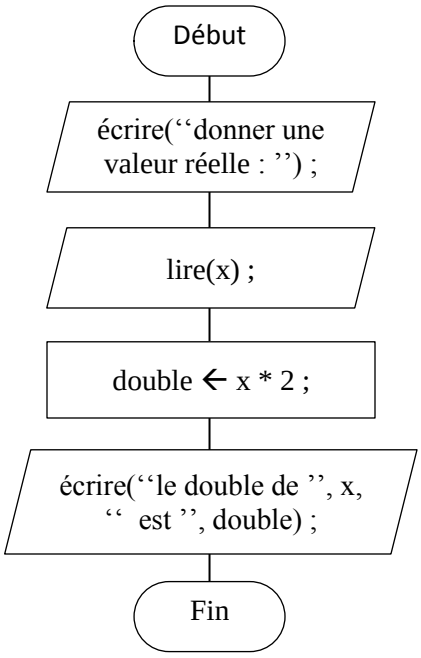
Le ''+'' et ''-'' dans 1 sont Les opérateurs arithmétiques unaires. Exp : +5 et -5.
En cas de priorités identiques (1 ou 2 ou 3 ou 4), les calculs s'effectuent de gauche à droite.
Les parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent.

Exercice : Evaluer les expressions suivantes sachant que : A = 3, B = 5, C = 2, D = 3, F = -1, H = 1, L = -2
E ← A+B*C ;
E ← A+B*C+D ;
E ← (A-B)+C*D ;
E ← (A+B/(H+L))*B-(C+F*D) ;

1.2. Instruction de lecture et écriture
Pour pouvoir **introduire** des données et **visualiser** des résultats, la mémoire centrale doit être mise en relation avec le monde extérieur : le clavier et l'écran. Les instructions permettant de gérer le transfert entre ces périphériques et la mémoire centrale sont :

- **lire** : ordonne le transfert d'information saisie par clavier vers la mémoire centrale.
- **écrire** : ordonne le transfert d'information depuis la mémoire centrale vers l'écran.

Dans ce contexte :
lire(A) : implique l'arrêt de l'exécution de l'algorithme. L'utilisateur doit saisir à l'aide du clavier une valeur du même type que A. cette valeur sera stocker dans l'espace mémoire désigner par A.
ecrire(A) : affiche la valeur de A sur l'écran.



Exemple complet :

```
algorithme doubleNombre
var x, double : réel ;
debut
        ecrire(‘‘donner une valeur réelle : ’’) ;
        lire(x) ;
        double ← x * 2 ;
        ecrire (‘‘le double de ’’, x, ‘‘ est : ’’, double) ;
fin
```

Voici le déroulement de cet algorithme :

Instructions	Mémoire		Ecran
variables x, double : réel ;	x = ?	double = ?	
ecrire(‘‘donner une valeur réelle ’’) ;	x = ?	double = ?	donner une valeur réelle
lire(x) ;	x = 5	double = ?	5↵
double ← x * 2 ;	x = 5	double = 10	
ecrire (‘‘le double de ’’, x, ‘‘ est : ’’, double) ;	x = 5	double = 10	le double de 5 est : 10

2. Structure de contrôle :

2.1. Structure linéaire (Séquence) : Dans une séquence, les instructions sont exécutées l'une après l'autre dans l'ordre dans lequel elles sont citées.

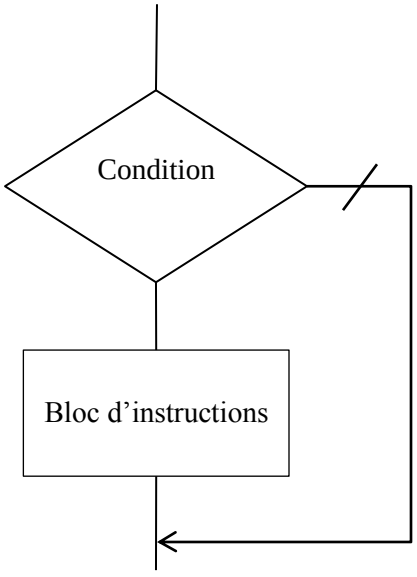
2.2. Structure alternative (instructions conditionnelles) : l'instruction conditionnelle détermine si le bloc d'instructions qui la suit sera exécuté ou non selon la valeur de sa condition. Cette dernière est une expression booléenne dont la valeur détermine le bloc d'instructions à exécuter (**condition = expression booléenne**).

Bloc d'instruction : suite d'instruction qui se trouve entre début et fin.

- **Syntaxe :**

```
si (condition) alors
        Bloc d'instructions ;
fin si
```

Le bloc d'instructions sera exécuté si la condition est vérifiée.



Exp :

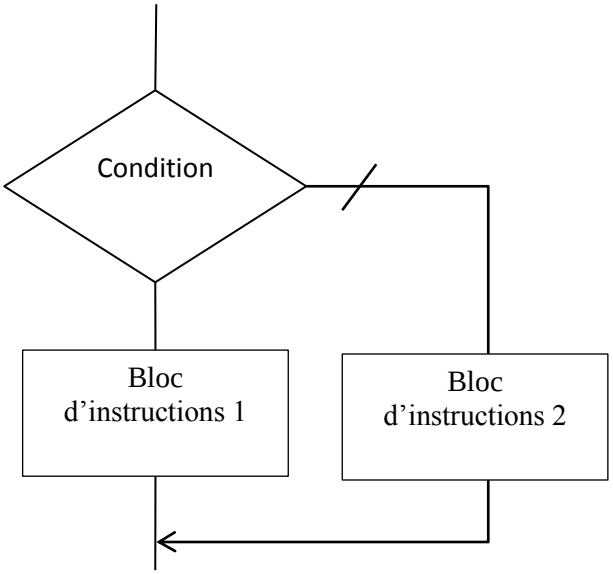
```
algorithme entier_positif
var x : entiers ;
debut
        écrire(‘‘donner un entier’’) ;
        lire(x) ;
        si (x > 0) alors
                écrire(x, ‘‘ est positif ’’) ;
        fin si
fin
```

<u>Exercice :</u> Donner le diagramme d’exécution de cet algorithme.

L’autre syntaxe est la suivante :

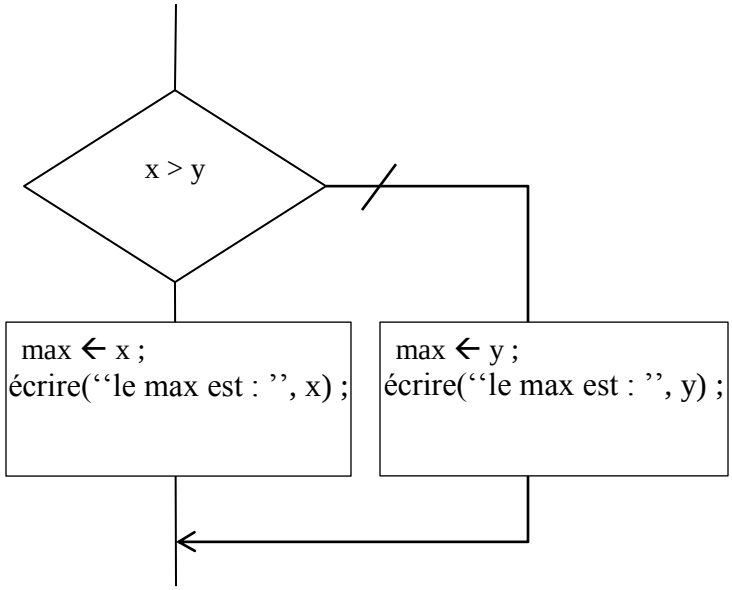
```
si (condition) alors
        Bloc d’instructions 1;
sinon
        Bloc d’instructions 2 ;
fin si
```

Si la condition est vérifiée alors le bloc d’instructions 1 est exécuté, dans le cas contraire c’est le bloc d’instructions 2 qui sera exécuté.



Exp :

```
algorithme max_deux_entiers
var x, y, max : entiers ;
debut
lire(x,y) ;
        si (x > y) alors
                max← x ;
                écrire(‘‘le max est : ’’, x) ;
        sinon
                max ← y ;
                écrire(‘‘le max est : ’’, y) ;
        fin si
fin
```



- **Conditionnelle imbriquée :** il est possible d'imbriquer des blocs de programme les uns dans les autres, essayons de résoudre le problème qui consiste à lire un caractère puis afficher le commentaire associé à ce caractère :
    - 'A' : ''Très Bien''.
    - 'B' : ''Bien''.
    - 'C' : ''Moyen''.
    - 'D' : ''Insuffisant''.

La première solution utilise des instructions conditionnelles simples :

```
algorithme commentaire_note
var x : char ;
debut
        lire(x) ;
        si(x = 'A') alors ecrire(''Très Bien'') ;
        si(x = 'B') alors ecrire(''Bien'') ;
        si(x = 'C') alors ecrire(''Moyen'') ;
        si(x = 'D') alors ecrire(''Insuffisant'') ;
        si((x ≠ 'A') et (x ≠ 'B') et (x ≠ 'C') et (x ≠ 'D')) alors ecrire(''valeur saisie incorrecte'') ;
fin
```

Dans cette solution, pour chaque note saisie x, cinq tests concernant sa valeur sont effectués même si le premier test est vérifié.

Exercice : Donner le diagramme d'exécution de cet algorithme.

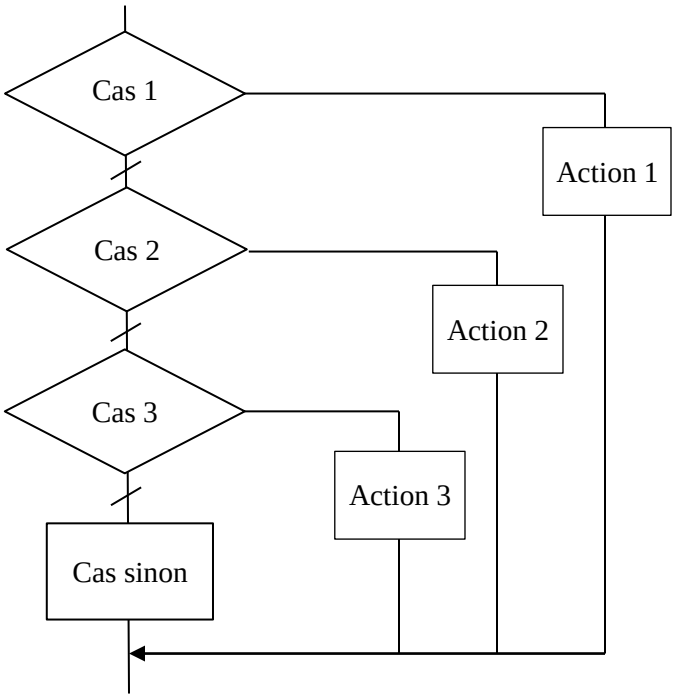
Voici une autre solution plus élégante :

```
algorithme commentaire_note
var x : char ;
debut
        lire(x) ;
        si(x = 'A') alors ecrire(''Très Bien'') ;
        sinon
                si(x = 'B') alors ecrire(''Bien'') ;
                sinon
                        si(x = 'C') alors ecrire(''Moyen'') ;
                        sinon
                                si(x = 'D') alors ecrire(''Insuffisant'') ;
                                sinon
                                        ecrire(''valeur saisie incorrecte'') ;
                                fin si
                        fin si
                fin si
        fin si
fin
```

Exercice : Donner le diagramme d'exécution de cet algorithme.

2.3. Structure à choix multiples (selon) : la structure de choix multiple permet d'effectuer des actions différentes suivant les valeurs que peut prendre une même variable (les actions peuvent être des blocs d'instructions).

Selon (variable ou exp)
    Cas Valeur1 : action1 ;
    Cas Valeur2 : action2 ;
    Cas Valeur3 :action3 ;
    Cas Sinon action par défaut ;
Fin Selon



Cette structure permet une présentation plus compact et claire des conditionnelles imbriquées.

Exp :

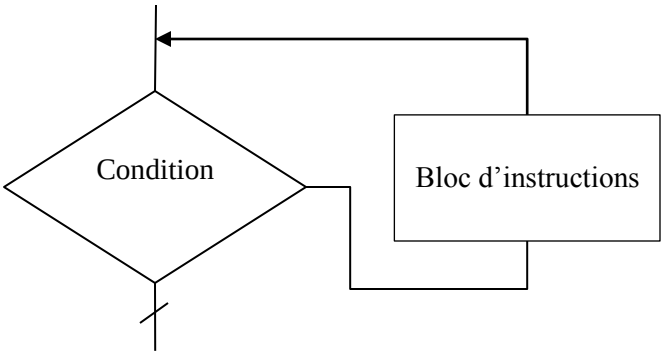
```
algorithme commentaire_note
var x : char ;
debut
        lire(x) ;
        selon (x)
                cas 'A' : ecrire(''Très Bien'') ;
                cas 'B' : ecrire(''Bien'') ;
                cas 'C' : ecrire(''Moyen'') ;
                cas 'D' : ecrire(''Insuffisant'') ;
                cas sinon ecrire(''valeur saisie incorrecte'') ;
        fin selon
fin
```

2.4. Structure itérative ou répétitive (les boucles)

Définition : les boucles permettent d'exécuter plusieurs fois consécutives un même bloc d'instructions. La répétition s'effectue tant que la valeur de l'expression booléenne de la boucle (condition de la boucle) est vraie. Il existe différents types de boucle commençons par la boucle tant que :

2.4.1. La boucle Tant que … Faire : On utilise la boucle tant que quand l'algorithme doit répéter le même traitement tant que une condition (l'expression booléenne de la boucle) est vérifiée.

tant que (condition) **faire**
    Bloc d'instructions ;
Fin tq



Phases d'exécution :

1. Evaluation de l'expression logique.
    a. Evaluation = vrai : Exécuter les actions puis reprise de l'étape précédente (1).
    b. Evaluation = faux : Arrêt de l'itération et le programme poursuit son exécution après fin tq.

Remarque :

La condition d'arrêt doit être réalisable : sa valeur doit passer à faux après un nombre fini de tours de boucle. Cette condition est composée d'une variable dont la valeur change à chaque tour de boucle. Cette variable est appelée variable de contrôle ou d'itération. Elle doit être modifiée par une action dans la boucle. Exp :

```
Algorithme  Somme
var s, i, n : entier;
Début
        ecrire(''Entrer la valeur de n'') ;
        lire(n) ;
        s ← 0 ;
        i ← 1 ;
        tant que (i ≤ n) faire
                S ← s + i ;
                i ← i + 1 ;
        fin tq
        ecrire("La somme des ", n, " premiers entiers est : ", s) ;
Fin
```

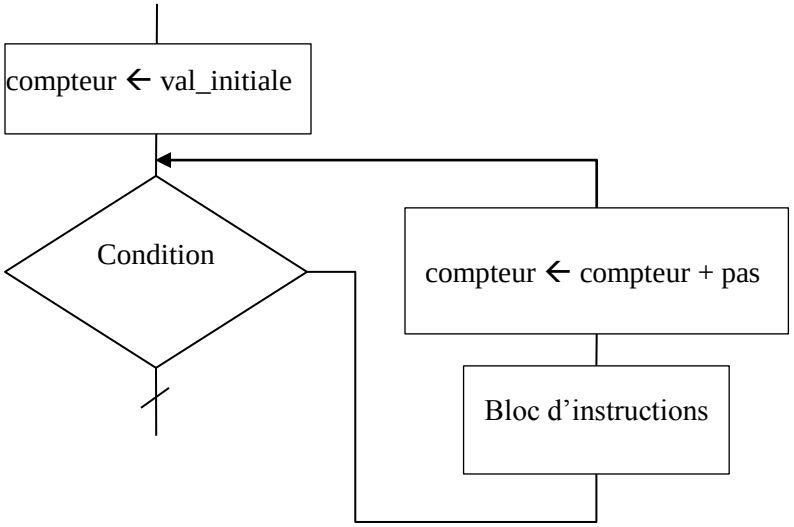
La variable s est utilisée pour additionner les différentes valeurs. Elle est initialisée à 0.

La variable i est appelée variable d'itération. Elle passe en revue les nombres de 1 à n

Cette instruction permet de faire évoluer i à la valeur suivante.

2.4.2. La boucle Pour … Faire : Cette boucle permet de répéter un ensemble d'actions un nombre connu de fois.

Pour (compteur ← val_initiale ; compteur ≤ val_finale ; compteur ← compteur + pas) **faire**
Bloc d'instructions ;
Fin p

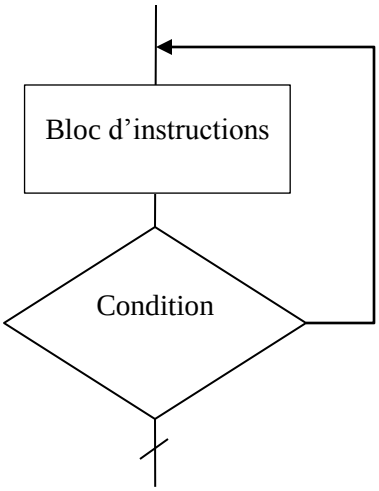


compteur est la variable **d'énumération des répétitions**. Elle part de **val_initiale**, elle passe automatiquement à la **valeur suivante** (compteur ← compteur + pas) et ainsi de suite jusqu'à ce que la condition **compteur ≤ val_finale devient fausse**. Et dans chaque itération le bloc d'instructions sera exécuté.

<u>Exp :</u>

```
Algorithme  Somme
var s, i, n : entier;
Début
        ecrire(''Entrer la valeur de n'') ;
        lire(n) ;
        s ← 0 ;
        pour (i ← 1 ; i ≤ n ; i ← i + 1) faire
                s ← s + i ;
        fin tq
        ecrire("La somme des ",n," premiers entiers est : ", s) ;
Fin
```

2.4.3. La boucle Faire … Tant que : cette boucle effectue l'évaluation de la condition booléen après avoir effectué le premier tour de la boucle.



faire
Bloc d'instructions ;
tant que (condition)

Voici un exemple d'utilisation de cette boucle :

```
Algorithme  Somme
var s, i, n : entier;
Début
        ecrire(‘‘Entrer la valeur de n’’) ;
        lire(n) ;
        S ← 0 ;
        i ← 1 ;
        faire
                S ← s + i ;
                i ← i + 1 ;
        tant que (i ≤ n)
        ecrire("La somme des ",n," premiers entiers est : ", s) ;
Fin
```

Comparaison des deux boucles faire … tant que et tant que … faire :

- Dans la boucle **faire … tant que**, le test est placé en fin d'itération, par conséquent les actions répétitives sont exécutées au moins une fois.
- Dans la boucle t**ant que … faire**, le test est placé avant le corps de la boucle. Dans certains cas, il est possible de ne jamais exécuter les opérations répétitives de la boucle.

Les Boucles Imbriquées : beaucoup de situations nécessitent l'utilisation des boucles imbriquées, voici un exemple : on demande de calculer le moyen des 5 notes saisies par l'utilisateur, sachant qu'une note est valide si elle est comprise entre 0 et 20 :

```
algorithme moyen
    var note, somme, moyen : réels ;
                compteur : entier ;
    debut
            somme ← 0 ;
            pour (compteur ← 1 ;  compteur ≤ 5 ; compteur ← compteur + 1) faire
                    faire
                            ecrire (‘‘entrer une note ’’) ;
                            lire (note) ;
                     tant que (note < 0 ou note > 20)
                    somme ←somme + note ;
            fin p
              moyen = somme /  5 ;
             ecrire(‘‘le mouen = ’’, moyen) ;
    fin
```

3. Procédure et Fonction : Une procédure ou une fonction est un algorithme indépendant, l'appel de la fonction déclenche l'exécution de son bloc d'instructions. Une fonction se termine en retournant ou non une valeur. Dans le cas où aucune valeur n'est retournée on parle de procédure. L'intérêt de l'utilisation des procédures et fonctions peut être résumé dans les points suivants :

- Le code des algorithmes devient plus simple, plus clair et plus court. Dans un algorithme, appeler une procédure ou une fonction se fait en une seule ligne et on peut l'appeler à maintes reprises.
- Une seule modification dans la procédure ou la fonction sera automatiquement répercutée sur tous les algorithmes qui l'utilisent.
- L'utilisation de procédures et fonctions génériques dans des algorithmes différents permet de réutiliser son travail et de gagner du temps.

La structure d'une fonction est la suivante :

```
fonction nomDeLaFonction(liste des paramères) : typeDeRetour
//partie déclaration ;
debut
        Instructions ;
Fin
```

La structure d'une procédure est la suivante :

```
procédure nomDeLaProcédure(liste des paramères)
//partie déclaration ;
debut
Instructions ;
Fin
```

Exécution d'une fonction : lors de l'appel d'une procédure ou d'une fonction dans un algorithme, trois étapes sont toujours nécessaires :

1. Interrompre l'exécution de l'algorithme appelant.
2. Exécution du bloc d'instructions de la procédure ou la fonction appelée jusqu'à son terme.
3. Reprise de l'exécution de l'algorithme appelant.

```
fonction lireNote( ) : réel
    var note : réel ;
    debut
            faire
        ecrire(''entrer une note : '') ;
        lire(note) ;
             tant que (note < 0 ou note > 20)
             lireNote ← note ;
    fin
```

Voici l'algorithme qui fait appel à cette fonction

```
algorithme moyen
    var n, somme, moyen : réels ;
                  compteur : entier ;
    debut
```

```
        somme ← 0 ;
        pour (compteur ← 1 ; compteur ≤ 5 ; compteur ← compteur + 1) faire
                n ← lireNote () ;
                somme ←somme + n ;
        fin p
          moyen = somme /  5 ;
         ecrire(‘‘le mouen = ’’, moyen) ;
fin
```

Les paramètres et les variables : L'exécution d'une fonction est paramétrable grâce à des valeurs qui lui sont passées, les paramètres de la fonction : il est faux donc de vouloir les redéfinir dans la zone de déclaration des variables de la fonction.

Définition : un paramètre est une variable locale à une fonction. Il possède dès le début de la fonction la valeur passée par l'algorithme appelant.

Une fonction peut accéder à trois types de variables :

- Les paramètres : leurs valeurs sont connues dès le début de la fonction, car elles sont passées en paramètres. Il est inutile de les nommer avec le même nom que les variables utilisées lors de l'appel de la fonction.
- Les variables locales : elles sont définies dans le bloc de déclaration des variables de la fonction elle-même.
- Les variables globales déclarées dans l'algorithme principal.

La valeur retournée par une fonction est unique. Il est impossible pour une fonction de retourner plusieurs valeurs.

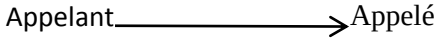
```
Fonction maxDeuxValeurs(p1 : entier, p2 : entier) : entier
debut
    si(p1 < p2) alors maxDeuxValeurs ← p2 ;
            sinon maxDeuxValeurs ← p1 ;
fin
```

Soit l'algorithme qui appelle cette fonction :

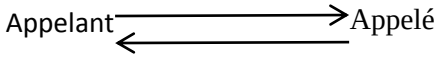
```
algorithme utiliseFonctionMax
var valeur, max : enier ;
debut
            lire(valeur) ;
        max ← maxDeuxValeurs(valeur, 25) ;
    ecrire(max) ;
fin
```

Passage de paramètres : dans certains cas, le programme appelant doit passer à une fonction des valeurs qui elle à son tour les utilise pour son exécution. En effet, le passage de paramètres consiste à définir le mode de transfert de données entre le programme appelant et le programme appelé. On distingue deux modes de passage de paramètres :

1) Passage par valeur : Dans ce mode de passage de paramètres, le programme appelant assure le transfert d'une valeur au programme appelé, mais si ce paramètre subit une modification de sa valeur dans le programme appelé, cette modification ne doit pas être transmise au programme appelant donc le transfert dans ce mode se fait d'une manière unidirectionnelle :



2) Passage par référence : Dans ce mode de passage de paramètres, le programme appelant et le programme appelé font un échange de données. En effet, toute modification de la valeur d'un paramètre au sein du programme appelé doit être communiquée au programme appelant, donc le transfert dans ce mode se fait dans les deux sens :



Pour un passage de paramètre par référence il faut précéder les paramètres formels qui seront transmis par référence par le mot **var**. Exp :

```
Procédure Saisie (var x : entier)
// Cette procédure permet de saisir un entier strictement positif
début
        faire
                écrire (''Donner un entier strictement positif : '')
                lire(x)
        tant que (x ≤ 0)
fin
```

L'algorithme qui utilise cette procédure est le suivant :

```
algorithme utiliseProcédureSaisie
var valeur : enier ;
debut
        Saisie(valeur) ;
    écrire(valeur) ;
fin
```

Dans ce contexte, quand un algorithme fait appel à une procédure avec un passage de paramètres par référence, on sait que les variables passées par référence subissent un changement de valeur dans l'algorithme appelant.

Signature d'une fonction : La signature d'une fonction décrit les éléments permettant de l'appeler correctement, à savoir :

- Le nom de la fonction ;
- Le type et l'ordre des paramètres ;
- Le type de la valeur retournée.

Un programmeur qui souhaite utiliser une fonction n'as pas besoin de connaître le corps de la fonction, ni même le nom ou les types des variables internes à la fonction, seulement sa signature lui suffit.

Polymorphisme paramétrique : Deux fonctions peuvent avoir le même nom et des paramètres différents en nombre ou en type. Le polymorphisme paramétrique garantit automatiquement l'exécution de la bonne fonction associée au bon nombre de paramètres et à leurs types**.** En effet, les programmes identifient une fonction par sa signature et pas uniquement par son nom.

```
Fonction maxDeuxValeurs(p1 : entier, p2 : entier) : entier
debut
    si(p1 < p2) alors maxDeuxValeurs ← p2 ;
            sinon maxDeuxValeurs ← p1 ;
fin
```

Voici la même fonction mais pour les réels :

```
Fonction maxDeuxValeurs(p1 : réel, p2 : réel) : réel
debut
    si(p1 < p2) alors maxDeuxValeurs ← p2 ;
            sinon maxDeuxValeurs ← p1 ;
fin
```

La récursivité : Une fonction est dite récursive si elle s'appelle elle-même. Exemple :

```
fonction factorielle (nb : entier) : entier
debut
        si (nb = 1) alors factorielle ← 1 ;
        sinon factorielle ← nb * factorielle(nb-1) ;
fin
```

L'algorithme qui utilise cette fonction est le suivant :

```
algorithme manipulationFactorielle
debut
    ecrire(factorielle(5)) ;
fin
```

Partie III : les structures de données

1. Les tableaux

1.1. Définition : un tableau est une structure de données qui désigne par une seule variable (nom du tableau) un ensemble de valeurs de même type accessibles via leurs positions dans le tableau.

La dimension et le type du tableau doivent être précisés lors de sa définition. Pour accéder aux éléments du tableau, un indice est utilisé, ce dernier indique le rang de l'élément dans le tableau.

a. Tableau à une dimension :

Const n ← 10 ;
Var tab : tableau[1..n] d'entier ;

Tab	22	-7	56	12	0	23	-4	1	57	35
Indice	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Nous constatons que tab[6] = 23, tab[3] = 56, tab[9] = 57, …etc.

Une boucle permet de parcourir ce tableau et afficher ses éléments.

```
pour (i ← 1 ; i ≤ 10 ; i ← i+1) faire
        ecrire (tab[compteur]) ;
  fin p
```

b. Tableau à deux dimensions :

```
Const nombre_lignes ← 3 ;
      nombre_colonnes ← 3 ;
Var tab : tableau[1 .. nombre_lignes, 1 .. nombre_colonnes] d'entier ;
```

[1][1]	[1][2]	[1][3]
[2][1]	[2][2]	[2][3]
[3][1]	[3][2]	[3][3]

Deux boucles imbriquées sont nécessaires pour le parcours et l'affichage des éléments d'un tableau à deux dimensions :

```
pour (i ← 1 ; i ≤ nombre_lignes ; i ← i + 1) faire
        pour (j ← 1 ; j ≤ nombre_colonnes ; j ← j +1) faire
                ecrire (tab[i][j]) ;
        fin p
fin p
```

1.2. Algorithmes pour le traitement des tableaux

a. Remplissage d'un tableau

```
Procédure remplissage(var tab : tableau[1..n] d'entier)
Var i : entier ;
début
    pour (i ← 1 ; i ≤ n ; i ← i + 1) faire
        lire (tab[i]) ;
     fin p
```

fin

b. Affichage des éléments d'un tableau

```
Procédure affichage(var tab : tableau[1..n] d'entier)
Var i : entier ;
début
    pour (i ← 1 ; i ≤ n ; i ← i + 1) faire
        écrire (tab[i]) ;
    fin p
fin
```

c. Algorithme de tri :

c.1. Tri par sélection

1. On se pointe à la 1ère case du tableau et on le parcourt afin de sélectionner l'indice du plus petit élément du tableau.
2. permuter le plus petit élément trouvé avec l'élément de la 1ère case du tableau.
3. Refaire les étapes 1 et 2 dès la 2ème case jusqu'à l'avant dernière case du tableau.

```
Procédure tri_Selection(var tab : tableau[1..n] d'entier)
Var i, k, indice, x : entier ;
début
pour (i de 1 à n-1 pas 1) faire
        indice ← i ;
        pour (k de i+1 à n pas 1) faire
        si(tab[indice] > tab[k]) alors indice ← k ;
        fin p
        si(indice ≠ i) alors
                x ← tab[i] ;
                tab[i] ← tab[indice] ;
                tab[indice] ← x ;
        fin si
finp
fin
```

c.2. Tri à bulles

1. Comparer le premier pair d'éléments : Si tab[1] > tab[2] alors permuter tab[1] et tab[2] et tenir compte de cette action.
2. Aller au pair suivant et répéter les étapes 1 et 2 jusqu'à comparer le dernier pair.
3. Si une permutation a été réalisé (ou plusieurs) alors répéter ce qu'on vient de faire dès le début du tableau, sinon la liste est triée.

```
Procédure tri_Bulles(var tab : tableau[1..n] d'entier)
Var i, x : entier ;
    permut : booléen;
début
faire
        permut ← faux ;
        pour (i de 1 à n-1 pas 1) faire
                si(tab[i] > tab[i+1])alors
                        x ← tab[i] ;
                        tab[i] ←  tab[i+1] ;
```

```
                    tab[i+1] ← x ;
                    permut ← vraix ;
                fin si
        fin p
tant que(permut = vrai)
fin
```

c.3. Tri par insertion

1. On commence par le deuxième élément.
2. Comparer l'élément choisi avec tous ses précédents dans la liste et l'insérer dans sa bonne position.
3. Répéter l'étape 2 pour l'élément suivant jusqu'à arriver au dernier.

```
Procédure tri_Insertion(var tab : tableau[1..n] d'entier)
Var i, j, val : entier ;
début
        pour (i de 2 à n pas 1) faire
                val ← tab[i] ;
                j ← i ;
                tant que (j-1 > 0 et (tab[j-1] > val)) faire
                        tab[j] ←  tab[j-1] ;
                        j ← j-1;
                fin tq
                tab[j] ← val ;
        fin p
fin
```

c.4. Tri par comptage : L'algorithme de tri consiste à construire un vecteur intermédiaire indice dans lequel on indique la place de chaque élément du table à trier (pour trouver l'emplacement if faut compter le nombre des éléments inférieur à chaque élément).

```
Procédure tri_Comptage(tab1 : tableau[1..n] d'entier, var tab : tableau[1..n] d'entier)
Var i, j : entier ;
    Indice : tableau[1..n] d'entier ;
début
        pour (i de 1 à n pas 1) faire
        indice[i] ← 1 ;
        fin p
        pour (i de 1 à n-1 pas 1) faire
        pour (j de i+1 à n pas 1) faire
                si(tab1[j] < tab1[i]) alors indice[i] ← indice[i]+1 ;
                sinon indice[j] ← indice[j]+1 ;
        fin p
        fin p
        pour (i de 1 à n pas 1) faire
        tab[indice[i]]← tab1[i] ;
        fin p
fin
```

d. Algorithmes de recherche

d.1. Recherche séquentielle : consiste à comparer la valeur recherchée aux différents éléments du tableau jusqu'à trouver cette valeur ou atteindre la fin du tableau.

```
Fonction rechercheSéquentielle(tab : tableau[1..n] d'entier, x : entier) : entier
Var i : entier ;
debut
        tant que(i ≤ n et (tab[i] ≠ x)) faire   i ← i+1 ;
        si (i ≤ n) alors recherche ←i ;
        sinon recherche ← -1 ;
fin
```

Recherche dichotomique : la recherche dichotomique s'effectuer sur des tableaux triés, elle consiste à diviser chaque fois en deux l'intervalle de recherche et à rechercher la valeur souhaitée dans la moitié de l'intervalle dans laquelle elle peut être incluse. Pour cela, on procède de la façon suivante:

Soient inf et sup les extrémités de l'intervalle de cherche et x la valeur recherchée. On calcule la moitié de cette intervalle m ← (inf + sup) div 2. Il y a 3 cas possibles:

1. x = tab[m] : l'élément de valeur x est trouvé, la recherche est terminée.
2. x < tab[m] : l'élément x, s'il existe, se trouve dans l'intervalle [inf .. m - 1].
3. x > tab[m] : l'élément X, s'il existe, se trouve dans l'intervalle [m + 1 .. sup].

La recherche dichotomique consiste à itérer ce processus jusqu'à ce que l'on trouve x ou que l'intervalle de recherche soit vide.

```
Fonction rechercheDichotomique(tab : tableau[1..n] d'entier, x : entier) : entier
Var inf, sup, m : entier ;
debut
        inf ← 1 ;
        sup ← n ;
        m ← (inf + sup) div 2 ;
        tant que(inf < sup et (tab[m] ≠ x)) faire
                si (x < tab[m]) alors sup ← m - 1 ;
                sinon inf ← m + 1 ;
                m ← (inf + sup) div 2 ;
        fin tq
        si(x = tab[m]) alors recherche ← m ;
        sinon recherche ← -1 ;
fin
```

2. Les enregistrements : Un enregistrement (ou structure) est une structure de données permettant de désigner sous un seul nom un ensemble de champ pouvant être de types différents.

```
Type Element = enregistrement
        Champ1 : type1 ;
        Champ2 : type2 ;
        …..
        champN : typeN ;
fin Element
var e : Element ;
```

Element est un type d'enregistrement qui précise le **nom** et le **type** de chacun des **champs** constituant cette structure.

Accès à un enregistrement :

Accès global : e1 ← e2 ;
Accès à un champ : e.champ1 ← valeur ;

Exemple :

```
Type Sexe = {‘‘Masculin’’, ‘‘Féminin’’} ;
Type  Jour = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31} ;
Type Mois = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} ;
Type Personne = enregistrement
        nom : chaine[30] ;
        prénom : chaine[30] ;
        sexe : Sexe ;
        adresse : chaine[90] ;
        Type dateNaissance = enregistrement
                jour : Jour ;
                mois : Mois ;
                année : entier ;
        Fin dateNaissance
Fin Personne
Const n ← 10 ;
personne1 : Personne ;
tab : tableau[1..n] de Personne ;
personne1.nom ← ‘‘Abdi’’ ;
personne1.prenom ← ‘‘Hamid’’ ;
personne1.sexe ← ‘‘Masculin’’ ;
personne1.adresse ← ‘‘Boumahra, Guelma’’ ;
personne1.dateNaissance.jour ← 29 ;
personne1.dateNaissance.mois ← 9 ;
personne1.dateNaissance.année ← 1988 ;
tab[1] ← personne1 ;
tab[1].prenom ← ‘‘Rachid’’ ;
```