Chapitre3

INTRODUCTION A L'ALGORITHMIQUE

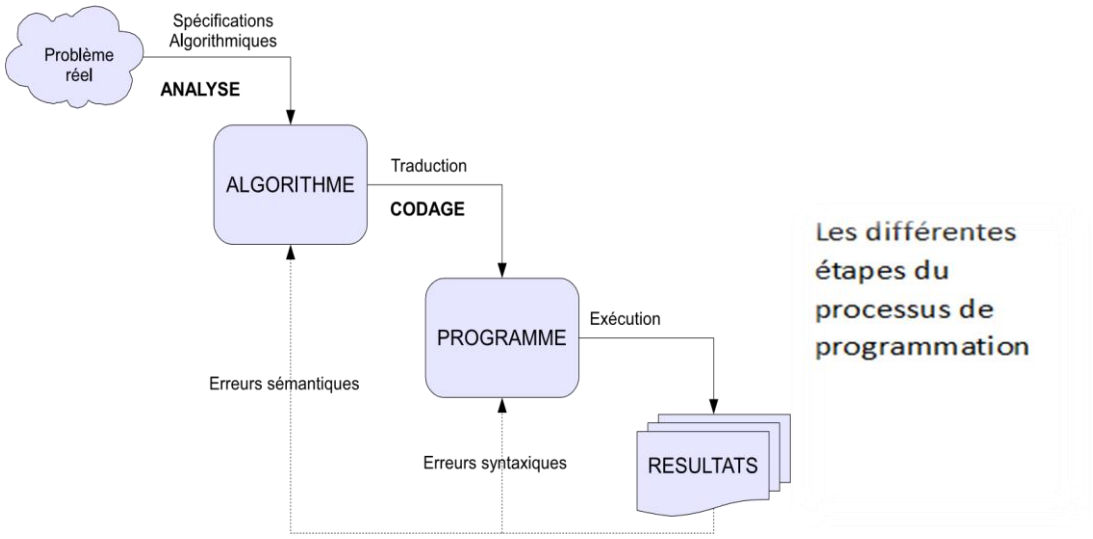
1- INTRODUCTION

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

- ***PRECIS*** : Il doit indiquer
  - L'ordre des étapes qui le constituent.
  - A quel moment il faut cesser une action.
  - A quel moment il faut en commencer une autre.
  - Comment choisir entre différentes possibilités
- ***DETERMINISTE***
  - Une suite d'exécutions à partir des mêmes données doit produire des résultats identiques.
- ***FINI DANS LE TEMPS***
  - C'est à dire s'arrêter au bout d'un temps fini.

2- Place de l'algorithme dans la résolution d'un problème informatique

Un algorithme doit être exprimé dans un langage de programmation pour être compris et exécuté par un ordinateur. Le programme constitue le codage d'un algorithme dans un langage de programmation donné, et qui peut être traité par un ordinateur. L'écriture d'un programme n'est qu'une étape dans le processus de programmation, comme le montre le schéma suivant :



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

Première étape : **Position du problème** :

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

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

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

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

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

- Les résultats souhaités (sorties).
- Les traitements (actions réalisées pour atteindre le but).
- Les données nécessaires aux traitements (entrées).

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

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

Pour obtenir un algorithme.

Quatrième étape : **Ecriture du programme**.

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

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

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

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

3- Structure d'un algorithme

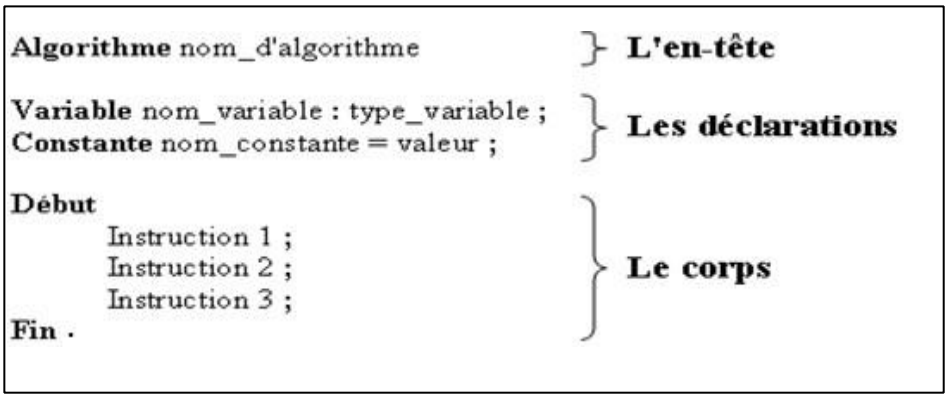
L'algorithme doit avoir une structure bien définie. Cette structure doit comporter :

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

- Les déclarations des variables et des constantes.

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

Toutes les instructions doivent se situer entre le mot **Début** et le mot **Fin**, et chaque instruction doit se terminer par un **point-virgule** .



Exemple : L'algorithme qui permet de permuter (échanger) les valeurs de deux nombres entiers X et Y.

Solution :

```
Algorithme permutation ;
Var X,Y,Z : entier ;
Debut
    Lire (X,Y) ;
    Z← X ;
    X← Y ;
    Y← Z ;
    Ecrire(X,Y) ;
Fin.
```

3-1 Les variables et les constantes

Définition et caractéristiques

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

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

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

• **L'identificateur** : c'est le nom de la variable ou de la constante, il est composé de lettres et de chiffres sans espaces.

• **Le type** : il définit la nature de la variable ou de la constante (entier, réel, caractère, ...) .

Remarques :

- Ne pas confondre la variable et son identificateur. En effet, la variable possède une valeur (son contenu) et une adresse (emplacement dans la mémoire où est stockée la valeur). L'identificateur n'est que le nom de la variable, c'est -à-dire un constituant de cette variable.
- Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre et les opérations réalisables qu'elle peut subir.
- L'utilisation d'une variable doit être précédée de sa déclaration.

La syntaxe pour déclarer une variable est la suivante :

Var identificateur de la variable : type de la variable ;

La syntaxe pour déclarer une constante est la suivante :

Const identificateur de la constante = valeur ;

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

Conventions de nommage

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

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

3-2 <u>Les types de données de base</u>

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

a) Type entier : un type numérique qui représente l'ensemble des entiers naturels et relatifs, tels que : 0, 45, -10, …

Mot clé : **entier .**

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

Mot clé : **reel.**

c) Type caractère : représente tous les caractères alphanumériques tels que : ′a′, ′B′, ′*′, ′9′, ′@′, ′ ′,…

Mot clé : **car.**

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

Mot clé : **chaine.**

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

booleen .

Exemple de déclaration :

var A : **entier ;**

moyenne, note1, note2 :**réel**

nom : **chaîne**

lettre : **car**

const n = 5 arobase= ′@′

e = ″425″

4-<u>Les opérateurs :</u>

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

- L'**opérateur d'affectation**, représenté par le symbole «← » (**:=**), qui confère une valeur à une variable . (affectation de la valeur à la variable).
  Syntaxe
  identificateur_1←identificateur_2 ;(Affecte à la variable 1 le contenu de la variable 2)
  identificateur_1 ←expression ; (Affecte à la variable 1 le résultat de l'expression)

Exemples où A, B et C sont de type reel , i, j de type entier et S de type chaine de caractères.
A← B ; { assigne à A la valeur de B }
A← i ; { assigne à A la valeur de i }
i← A ; { donne une erreur de compilation : on ne peut assigner un reel à un entier }
A← i/j ; { assigne à A le quotient (de type reel) de i/j
i ←i+1 ; { i est incrémenté }
i ← j div 10{assigne à i le quotient (de type integer) de j par 10 (division entiere)}
S ← S+'.' { ajoute le caractère '. ' à la fin de la chaîne S }

Remarques :
- Ne pas inverser les identificateurs ! ' A←B ' et ' B←A ' donnent des résultats différents

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

+	addition
-	soustraction
*	multiplication
/	division
mod	modulo
^	Puissance(en Pascal il n y a pas de puissance)
div	Division entiére

- **Les Opérateurs relationnels :**

>	supérieur
<	inférieur
>=	supérieur ou égal

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

▪ Les **opérateurs logiques** qui combinent des opérandes booléennes pour former des expressions logiques plus complexes :

- Opérateur unaire : «non » (négation)
- Opérateurs binaires : « et» (conjonction) , «ou » (disjonction)

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

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

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

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

- **Ordre de priorité décroissante des opérateurs arithmétiques et de concaténation :**

▪ «^» (élévation à la puissance)

▪ «* », « /» et « div»

▪ « modulo»

▪ «+ » et «-»

▪ «+» (concaténation)

- **Ordre de priorité décroissante des opérateurs logiques :**

▪ «not » (non)

▪ « and» (et)

▪ « or» (ou)

5- Les operateurs d'entrée/sortie

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

Exemples :

Algorithme qui calcule la somme de deux nombres réels

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

Algorithme qui calcule l'âge

```
Algorithme calculage ;
Var annee : integer ;
Const ancourant = 2022;
 debut
    ecrire( ″donner votre année de naissance″) ;
    lire(annee) ;
    ecrire(″votre age est″, ancourant-annee) ;
fin.
```

algorithme qui calcule la surface d'un disque

```
Algorithme Surface d'un disque
Var  Rayon, Surface : real ;
Const Pi=3.14;
Debut
   ecrire("Donner le rayon du disque");
```

lire (Rayon) ;

Surface←(Rayon ^ 2) * Pi ;

ecrire ("La surface du disque est : ", Surface) ;

Fin.

5- Les structures de contrôle

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

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

Que doit faire le traitement ?

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

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

5-1-La structure alternative

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

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

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

```
Si condition(s)  Alors
 debut
   Traitement1 ;
 fin
Sinon
debut
   Traitement 2 ;
 Fin Si ;
```

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

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

Remarque :

- L'instruction précédent le **sinon** ne se termine pas avec un point-virgule (**;**).
- **debut** et **fin** ne sont pas nécessaires si le **si** ou le **sinon** comprend une seule instruction.

5-2. La structure conditionnelle simple

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

Syntaxe

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

```
SI condition(s) ALORS
Debut
    Traitement ;
Fin ;
```

5-3. Les structures alternatives imbriquées

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

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

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

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

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

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

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

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

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

5-4.La structure alternative multiple

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

Syntaxe

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

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

Exemple

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

```
…
VAR  noMois :entier ;

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

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

5-5-Complément sur les expressions conditionnelles

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

5-5.1. Les comparaisons simples

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

Exemples

a < 0 comparaison d'entiers ou de réels

code = 's' comparaison de caractères (code étant une variable de type caractère)

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

(a + b - 3) * c ≤ (5 * y –2) / 3

5-5.2.Les expressions complexes

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

Exemples

SI a < 0 **ET** b < 0 ALORS

…

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

6-STRUCTURES CODITIONNELLES REPETITIVES

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

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

1- Boucle Pour

La structure de contrôle répétitive **Pour** 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
Debut
  <instruction(s) ;
Fin ;
```

<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, les debut et fin sont facultatifs.

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

La sortie de la boucle s'effectue lorsque le nombre souhaité d'itérations est atteint, c'est-à-dire lorsque <indice> prend la valeur <vf>.

Exemple Faire la somme des 10 premiers entiers naturels.

```
Algorithme   utilisation_du_compteur1;
   Var somme,i : entier;
debut
 Somme ← 0 ;
 Pour i← 1 to 9 Faire somme←somme + i ;
 fin.
```

2- Boucle Tant-que

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

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

tant-que <condition> **faire**

debut
<instruction(s)> ;
fin;

<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 devient fausse, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après **fin** .

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

Exemple : Faire la somme des 10 premiers entiers naturels.

```
Algorithme utilisation_compteur2 ;
Var i,somme :entier ;
debut
 I← 1 ; Somme← 0 ;
tant-que  i  <  10  faire
debut
   somme←  somme + i ; i ←  i + 1 ;
fin; fin.
```

3-Boucle Répéter

La structure de contrôle répétitive **répéter** utilise une expression logique ou booléenne comme condition de sortie de la boucle : si la condition est vérifiée (elle donne un résultat vrai) 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
<instruction(s)> ;
Jusqu'à  <condition(s)> ;
```

<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 **.** Dans la boucle **répéter** on utilise pas debut et fin pour délimiter le bloc d'instructions (le bloc est déjà délimité par **répéter** et **jusqu'à**).

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

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

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

Exemple : Faire la somme des 10 premiers entiers naturels.

```
Algorithme utilisation_compteur3 ;
Var somme,i : entier ;
debut
  Somme ← 0 ; i ← 1 ;
  Repéter
    somme ← somme + i ;      i ← i+1 ;
  jusqu'à i = 10 ;
fin.
```

Choix de la boucle

La règle est simple :

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

Sinon : on utilise le répéter (quand il y a toujours au moins une itération le nombre d'itérations n'est pas connu d'avance), ou le tant que (quand le nombre d'itérations n'est pas connu d'avance et peut être nul).