Chapitre 2: *Notions d'algorithme et de programme*

1. Concept d'un algorithme

- ✓ Le mot *algorithme* vient du nom du mathématicien arabe *Al Khawarizmi*.
- ✓ Un algorithme est une *méthode* pour *résoudre un problème* particulier.
- ✓ Un algorithme est une *séquence d'opérations* (instructions) à effectuer ou *étapes* pour résoudre un problème ou faire un traitement (l'ordre des opérations est important).
- ✓ Un algorithme prend en entrée des *données* et fournit un *résultat*.
- ✓ Un algorithme est écrit *indépendamment* des langages de programmation.
- ✓ L'implémentation (traduction) d'un algorithme dans un langage de programmation donne un *programme*.

Exemple: *pour sortir une voiture du garage:*

1. Ouvrir la porte du garage;
2. Prendre la clef;
3. Ouvrir la porte avant gauche;
4. Entrer dans la voiture;
5. Mettre au point mort;
6. Mettre le contact.

Exemple: *Addition de deux nombres A et B.*

1. Introduire le premier nombre A;
2. Introduire le deuxième nombre B;
3. Faire l'addition A+B;
4. Afficher le résultat.

2. Démarche de résolution d'un problème par ordinateur

Pour résoudre un problème par ordinateur on suit les étapes suivantes:

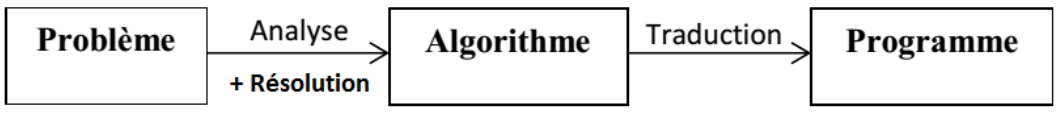
a. **Analyse du problème:** il s'agit de définir les objectifs. Qu'est-ce qu'on demande de faire? Quel est le travail à fournir ?

b. **Mettre une méthode de la résolution :** Il s'agit de dire, sans entrer dans les détails, comment le problème sera résolu (déterminer les opérations à effectuer pour avoir une solution).

c. **Formuler l'algorithme :** écrire les instructions qui composent l'algorithme.

d. **Traduction de l'algorithme:** traduire l'algorithme dans un langage de programmation.

e. **Exécution du programme :** on entre les données nécessaires, l'ordinateur fait le traitement et nous fournit le résultat.



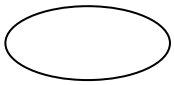
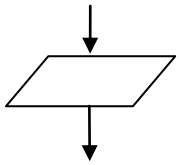
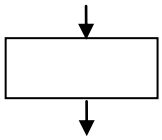
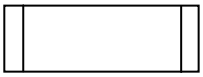
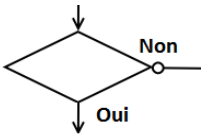
3. Représentation d'un algorithme

Il y a deux façons pour représenter l'algorithme : une représentation graphique par un *organigramme* et représentation textuelle par un *pseudo-code.*

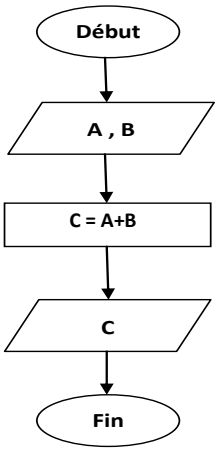
A. Représentation graphique (organigramme): un organigramme est composé:

- ✓ des *symboles* représentant le *type d'action* à effectuer. L'action est décrite à l'intérieur du symbole;
- ✓ des relations entre ces différents symboles représentées par des *traits* ou des *flèches*. Ils indiquent le *chemin* à suivre pour accéder au symbole suivant.

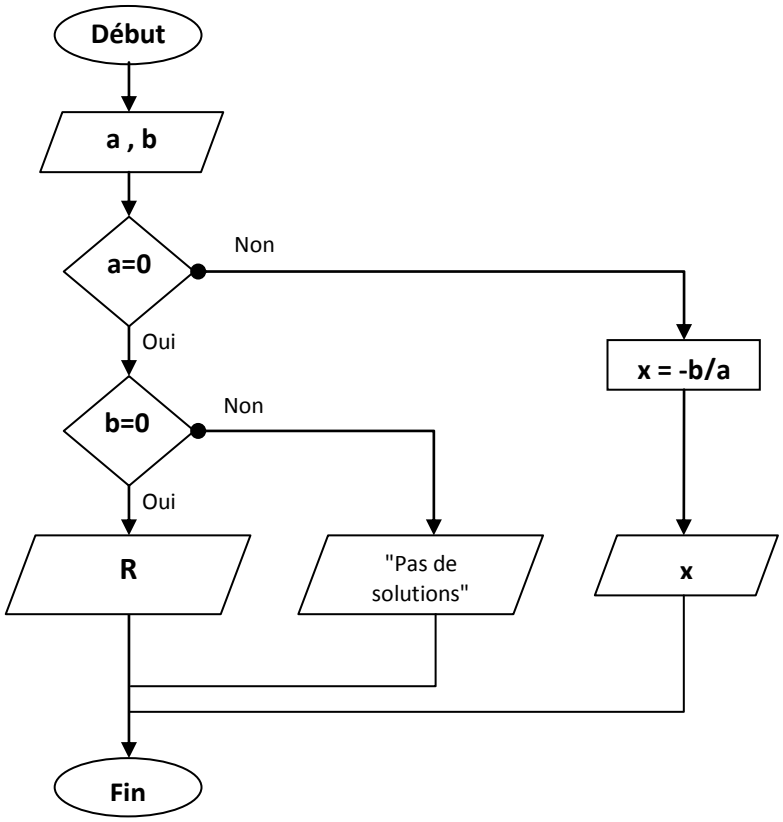
Un organigramme commence toujours par une case *Début* qui désigne le début le l'algorithme, et se termine pas une case *Fin* qui marque la fin de l'algorithme.

Symbole	Signification
	**L'ovale :** dans lequel se trouve soit le mot **DEBUT**, soit le mot **FIN,** qui représentent respectivement le début et la fin du de l'algorithme.
	**Le Parallélogramme:** il représente une action d'échange entre l'ordinateur et l'homme, soit introduction de *données* (entrées), soit affichage de *résultats* (sorties).
	**Le rectangle:** il représente une action, traitement de donnée ou calcul; cette action peut être simple ou complexe (c'est-à- dire un groupe d'actions).
	Sous-algorithme (portion d'algorithme considérée comme une simple opération).
	Représente une condition.

Exemple: *organigramme qui permet l'addition de deux nombres A et B.*



Exemple: *organigramme qui calcule la solution de l'équation a*x+b=0 dans R.*



B. Représentation textuelle (pseudo-code)

La deuxième méthode, fréquemment utilisée, sert à développer un algorithme, suivant la structure suivante :

Entête de l'algorithme

```
algorithme Nom_algorithme
```

Partie déclaration

```
Déclaration des constantes
Déclaration des variables
Définition des sous-algorithmes
```

Corps de l'algorithme

```
début

 Action 1
 Action 2
 …
 Action n

fin
```

- ✓ **L'entête de l'algorithme :** permet d'identifier l'algorithme comme suit: le mot *algorithme* suivi par le nom de l'algorithme**.**
- ✓ **La partie déclaration :** cette partie est réservée pour la déclaration de toutes les constantes et variables utilisées dans l'algorithme.
- ✓ **Le corps de l'algorithme :** cette partie contient les tâches de l'algorithme.

4. Notions de constante et de variable

- Une *donnée* est une valeur introduite par l'utilisateur pendant l'exécution de programme. **Ex.** Dans l'exemple précédent (calcul de la somme de deux nombres A et B), les données sont A et B.
- Un *résultat* est une valeur produit par l'algorithme. Il peut être *définitif* ou *intermédiaire*. **Ex.** La somme de A et B dans l'exemple précédent est un résultat définitif. Le calcul de Δ dans une équation de deuxième degré est un résultat intermédiaire.

❖ **Les variables:** une variable est utilisée par un algorithme pour stocker une valeur (donnée ou résultat).
- ✓ Une variable peut changer de valeur au cours de l'algorithme.
- ✓ Une variable désigne en fait *un emplacement mémoire* dont le contenu peut changer au cours de l'algorithme.

❖ **Les constantes:** une constante est une valeur fixe utilisée par le l'algorithme. **Ex.** *Pi* (3.14), *la constante de gravitation* (9,81), etc.

❖ **L'identificateur :** un identificateur est le *nom* utilisé pour désigner une variable ou une constante ou l'algorithme (c.à.d. le nom d'une variable, d'une constante ou de l'algorithme).
- ✓ Un identificateur contient des chiffres **0** à **9**, des lettres de **A** à **Z** (majuscules ou minuscules) et le tiret du huit (**_**).
- ✓ Un identificateur doit commencer par une *lettre* ou par le tiret du huit.

Ex. Identificateurs valides: *aC2, Delta, longueur, _Taille, X*

Identificateurs non valides: *2Ca, a#, nom?, X 1, R-A*

❖ **Type:** le type d'une variable ou d'une constante représente la *nature des valeurs* qui peuvent être stockées dans cette variable ou dans cette constante (*nombres*, *texte*, etc.).

❖ **Mots clés:** les mots clés sont des mots réservés pour un usage bien défini. Un identificateur ne peut pas être un mot clé. **Ex.**
- **`algorithme :`** permet de définir ou de donner le nom à l'algorithme.
- **`début`** : marque le commencement de l'algorithme.
- **`fin :`** marque la fin de l'algorithme.
- **`var`**, **`const`**, **`si`**, **`sinon`**, …

5. Les types de bases

En algorithmique, les variables et les constantes peuvent avoir cinq types de base différents:

a. Le type *Entier* : un type numérique qui représente l'ensemble des entiers naturels et relatifs, tels que : *0, 45, -10,* etc. Mot clé : `entier`**.**

b. Le type *Réel* : un autre type numérique qui représente les nombres réels, tels que : *0.5,-3.67, 1.5e+5* , etc. Mot clé : `réel`**.**

c. Le type *Caractère*: représente tous les caractères alphanumériques, tels que : '*a*', '*B*', ' ', '*', '*9*', '@', etc. Mot clé : `car` **.**

d. Le type *Chaînes de caractères* : concerne des chaines de caractères tels que des mots ou des phrases : "*informatique*", "*la section B*", "*Bonjour*", etc. Mot clé : `chaîne`**.**

e. Le type *Booléen* : ce type ne peut prendre que deux états : *vrai* ou *faux*. Mot clé : `booléen`**.**

6. Déclaration des variables et des constantes

a. Déclaration des variables: pour déclarer des variables on utilise le mot clé `var`, comme suit:

```
var  identificateur_1 : Type1
     identificateur_2 : Type2
            …
     identificateur_n : Typen
```

Si plusieurs variables ont le même type, on sépare leurs noms par des virgules (**,**).

Exemple :

```
var  groupe : entier
     moyenne, note1, note2 : réel
     nom : chaîne
     section : car
     admis : booléen
```

b. Déclaration des constantes: pour déclarer des constantes on utilise le mot clé `const`, comme suit:

```
const identificateur_1 = valeur1
      identificateur_2 = valeur2
            …
      identificateur_n = valeurn
```

Exemple :

```
const pi = 3.14
      g  = 9.81
      Arobase='@'
      Adresse= "Mila-Algérie"
```

7. Les opérateurs

A. L'opérateur d'affectation

L'affectation consiste à attribuer une valeur à une variable (stocker une valeur dans une variable), en utilisant le symbole (←), de la manière suivante:

Variable ← valeur (c.à.d. affecter une valeur à une variable)

Ou bien

Variable1 ← *Variable2* (c.à.d. affecter à *variable1* la valeur de *variable2*)

Ou bien

Variable ← *Expression* (c.à.d. affecter à une variable la valeur d'une expression)

Exemples

- Affecter à la variable ***a*** la valeur **5.**
  - ✓ c'est-à-dire : stocker dans la variable ***a*** la valeur **5**.

```
a ← 5
```

- Affecter à la variable ***b*** la valeur de la variable ***a.***
  - ✓ c'est-à-dire : stocker dans la variable ***b*** la valeur de la variable ***a***.

```
b ← a
```

- Affecter à la variable ***a*** la valeur de l'expression **b+4**.
  - ✓ c'est-à-dire : stocker dans la variable ***a*** la valeur de l'expression **b+4**.

```
a ← b+4
```

Exemple :

Instruction	Valeur de A	Valeur de B
A ← 5	5	-
B ← A	5	5
A ← B+4	9	5
*B ← (A*A)/2*	9	40.5

B. Les opérateurs arithmétiques

+ : Addition
- : Soustraction
*** :** Multiplication
/ : Division
DIV: Division entière (**Ex.** `9 DIV 2 = 4`)
MOD: Reste de la division entière (**Ex.** `9 MOD 2 = 1`)

C. Les opérateurs de comparaison

= Egale à ≠ Différent de > Supérieur à

< Inférieur à ≥ Supérieur ou égale ≤ Inférieur ou égale

D. Les opérateurs logiques

`ET`: Et logique

`OU`: Ou logique

`NON`: Négation

A	B	**NON** A	**NON** B	A **ET** B	A **OU** B
faux	faux	vrai	vrai	faux	faux
faux	vrai	vrai	faux	faux	vrai
vrai	faux	faux	vrai	faux	vrai
vrai	vrai	faux	faux	vrai	vrai

E. L'opérateur de concaténation (&) : pour concaténer des chaînes de caractères.

Exemple: "ABC" & "DE" = "ABCDE", "12" & "3" = "123".

F. Les priorités dans les opérations

- On classe les opérateurs par ordre de priorité, les opérateurs de plus forte priorité étant réalisés avant ceux de plus faible priorité.
- Lorsque deux opérateurs sont de priorité égale, on évalue de gauche à droite.
- Voici la table des priorités classées par ordre décroissant, les opérateurs sur une même ligne ont la même priorité.

( )
not
/ * div mod and
+ - or
= < > < <= >=

Exemple:

A+B*C= A+ (B *C) et non pas (A+B) *C.

8. Les opérations d'entrée/sortie (lecture/écriture)

a. L'opération de lecture

- L'opération (l'instruction) de lecture permet de fournir des valeurs (des données) à notre algorithme (ou programme) à partir du *clavier* pendant l'exécution.
- On utilise le mot clé `Lire` de la manière suivante :

```
Lire (Nom_de_variable)
```

Exemple :

`Lire (A)`: signifie prendre une valeur à partir du clavier pendant l'exécution et la stocker (ranger) dans la variable A.

`Lire (A,B)`: signifie prendre deux valeurs à partir du clavier pendant l'exécution et les stocker (ranger) dans les variables A et B, respectivement.

b. L'opération d'écriture

- L'opération (l'instruction) d'écriture permet à l'algorithme d'*afficher* (écrire) quelque chose à l'écran. C.à.d. afficher la *valeur d'une variable*, la *valeur d'une expression* ou un *texte* (mot ou phrase) pendant l'exécution de l'algorithme.
- On utilise le mot clé `Ecrire` de la manière suivante :

```
Ecrire (Nom_de_variable)
Ecrire (Expression)
Ecrire ("Un texte")
```

Exemples:

```
Ecrire (A)
```

Signifie : afficher à l'écran la valeur de la variable `A`.

```
Ecrire ((A+B)/2)
```

Signifie : afficher (écrire) à l'écran la valeur de l'expression `(A+B)/2`.

```
Ecrire ("Bonjour")
```

Signifie : afficher (écrire) à l'écran le mot: `Bonjour`

- Pour afficher plusieurs choses à la fois, il suffit de les séparer par des virgules (**,**).

Exemple: supposons que: A = 5 et B = 4. L'instruction:

```
Ecrire ("La valeur de A est ", A, " et la valeur de B est ", B)
```

Permet d'afficher (écrire) à l'écran:

```
La valeur de A est 5 et la valeur de B est 4
```

Exemple : écrire un algorithme qui lit deux nombres entiers A et B, puis il calcule et affiche leur somme:

```
algorithme addition
var A, B, Somme : entier
début
    lire (A)
    lire (B)
    Somme ← A+B
    écrire (Somme)
fin
```

Exécution

```
3
8
11
```

Si on veut que l'exécution soit plus *lisible* et plus *compréhensible*, on affiche des *messages* en utilisant l'instruction `Ecrire`. L'algorithme précédent devient:

```
algorithme addition
var A, B, Somme : entier
début
    lire (A)
    lire (B)
    Somme ← A+B
    écrire ("La somme de A et B = ",Somme)
fin
```

Exécution

```
3
8
La somme de A et B = 11
```

Exemple : écrire un algorithme qui calcule et affiche la surface d'un cercle.

```
algorithme surface
const Pi = 3,14
var R,S: Réel
début
    lire (R)
    S ← Pi*R*R
    écrire ("La surface = ",S)
fin
```

Exécution

```
2
La surface = 12.56
```

9. Les structures de contrôle

- Les algorithmes précédents ont une structure *séquentielle*, car les instructions s'exécutent de façon séquentielle (étape par étape ou par ordre).
- Les structures de contrôle décrivent ou *contrôlent l'enchaînement d'exécution des instructions*.
- Deux types de structures de contrôle: *structures conditionnelles* et *structures itératives*.

9.1. Les structures conditionnelles (les tests)

A. Structure conditionnelle simple

La structure conditionnelle simple a la forme suivante :

```
si condition alors
    Bloc d'instructions
finsi
```

Ce qui signifie : si la condition est **vraie** le bloc d'instructions est *exécuté* et si la condition est **fausse** il est *ignoré*.

condition: est une expression booléenne.

Exemple: soient les deux cas suivant:

1er cas: A = 2.

2eme cas : A = 0.

```
        condition
si  A>0  alors
|   B ← A/2   }
|   C ← 2*A   }  Bloc d'instructions
finsi
```

- ***Dans le 1er cas* :** le bloc d'instructions sera exécuté.
- ***Dans le 2eme cas* :** le bloc d'instructions sera ignoré (ne sera pas exécuté).

B. Structure conditionnelle alternative

La structure conditionnelle alternative a la forme suivante :

```
si condition alors
|   Bloc d'instructions 1
sinon
|   Bloc d'instructions 2
finsi
```

Ce qui signifie : si la condition est **vraie** *le bloc d'instructions 1* sera exécuté et si la condition est **fausse**, c'est *le bloc d'instructions 2* qui sera exécuté.

Exemple: soient les deux cas suivant:

1er cas: A = 2.

2eme cas : A = 0.

```
si  A≠0  alors
|   B ← A/2   }
|   C ← 2*A   }  Bloc d'instructions 1
sinon
|   B ← 5     }
|   C ← 3     }  Bloc d'instructions 2
finsi
```

- ***Dans le 1er cas* :** c'est le *bloc d'instructions 1* qui sera exécuté.
- ***Dans le 2eme cas* :** c'est le *bloc d'instructions 2* qui sera exécuté.

Exercices:

a. Ecrire un algorithme qui détermine et affiche le minimum de deux nombres entiers.

b. Ecrire un algorithme qui calcule et affiche la valeur absolue d'un nombre réel X.

Solutions:

```
algorithme minimum
var A, B, Min: Entier
début
| lire (A,B)
| si A<=B alors
| |  Min ← A
| sinon
| |  Min ← B
| finSi
| ecrire (Min)
fin
```

```
algorithme val_absolue
var X, Abs: entier
début
| lire (X)
| si X<0 alors
| |  Abs ← -X
| sinon
| |  Abs ← X
| finsi
| ecrire (Abs)
fin
```

Remarque: dans une structure conditionnelle, un bloc d'instructions peut être lui-même une structure de conditionnelle. On parle alors des *structures conditionnelles imbriquées* (tests imbriqués). Cette structure aura les formes suivantes :

```
Si condition1 Alors
|  Si condition2 Alors
|  |  Bloc d'instructions 1
|  Sinon
|  |  Bloc d'instructions 2
|  FinSi
Sinon
|  Bloc d'instructions 3
FinSi
```

⇨

Ce qui signifie :

- ✓ si la *condition1* est **vraie** et la *condition2* est **vraie** le *bloc d'instructions 1* sera exécuté.
- ✓ si la *condition1* est **vraie** et la *condition2* est **fausse** le *bloc d'instructions 2* sera exécuté.
- ✓ si la *condition1* est **fausse**, c'est le *bloc d'instructions 3* qui sera exécuté.

Ou bien

```
Si condition1 Alors
|  Bloc d'instructions 1
Sinon
|  Si condition2 Alors
|  |  Bloc d'instructions 2
|  Sinon
|  |  Bloc d'instructions 3
|  FinSi
FinSi
```

⇨

Ce qui signifie :

- ✓ si la *condition1* est **vraie** le *bloc d'instructions 1* sera exécuté.
- ✓ si la *condition1* est **fausse** et la *condition2* est **vraie** le *bloc d'instructions 2* sera exécuté.
- ✓ si la *condition1* est **fausse** et la *condition2* est **fausse**, c'est le *bloc d'instructions 3* qui sera exécuté.

Exercices:

a. Ecrire un algorithme qui calcule la solution de l'équation a*x+b=0 dans R.

b. Ecrire un algorithme qui calcule le prix de photocopies, sachant que le prix unitaire est:

- 4 DA si le nombre de copies est inférieur à 10.
- 3DA pour un nombre compris entre 10 et 19.
- 2DA si le nombre de copies est supérieur ou égale à 20.

Solutions :

```
algorithme équation
var a, b, x: Réel
début
 lire (a,b)
 si a=0 alors
   si b=0 alors
     ecrire ("La solution est R.")
   sinon
     ecrire ("Pas de solutions.")
   finsi
 sinon
  x ← -b/a
  ecrire (x)
 finsi
fin
```

```
algorithme photocopies
var nb_copies, prix: Entier
début
 lire (nb_copies)
 si nb_copies <10 alors
   prix ← nb_copies*4
 sinon
   si nb_copies <20 alors
     prix ← nb_copies*3
   sinon
     prix ← nb_copies*2
   finsi
 finsi
 ecrire (prix)
fin
```

9.2. Les structures itératives (les boucles)

Les boucles ou les structures itératives (répétitives) servent à *répéter plusieurs fois* l'exécution d'un bloc d'instructions, le nombre de répétions peut être *connu à l'avance* ou *non*. On distingue trois types de boucles, qui sont: la boucle `Pour`, la boucle `TantQue` et la boucle `Répéter`.

A. La boucle `pour`

Lorsque on connait à l'avance le nombre de répétitions, on utilise la boucle `Pour`. Elle se présente sous la forme suivante:

```
pour compteur = vi à vf faire
  Bloc d'instructions
finpour
```

- ✓ *Compteur* est une variable de type Entier;
- ✓ *vi* est la valeur initiale du compteur (sa première valeur);
- ✓ *vf* est la valeur finale du compteur (sa dernière valeur);
- ✓ Les instructions situées entre Pour et FinPour, s'exécutent N fois, tel que: N = vf-vi +1.

Exemples:

– Un algorithme qui affiche cinq fois le mot *Bonjour*.

– Un algorithme qui affiche les nombres entiers de 1 à 10.

```
algorithme bonjour
var i: entier
début
   pour i = 1 à 5 faire
      ecrire ("Bonjour")
   finpour
fin
```

Exécution

```
Bonjour
Bonjour
Bonjour
Bonjour
Bonjour
```

```
algorithme nombres
var i: entier
début
   pour i = 1 à 10 faire
      ecrire (i)
   finpour
fin
```

Exécution

```
1
2
3
4
5
6
7
8
9
10
```

B. La boucle tantque

Lorsque le nombre de répétions n'est pas connu à l'avance on utilise la boucle TantQue. Elle se présente sous la forme suivante:

```
tantque condition faire
   Bloc d'instructions
finTQ
```

- ✓ Les variables de la condition doivent être initialisées *avant* la boucle.
- ✓ Tant que la condition est vraie, on répète l'exécution du bloc d'instructions;
- ✓ Dès qu'elle devient fausse, on sort de la boucle;
- ✓ Si la condition est fausse initialement, le bloc d'instructions ne sera jamais exécuté.

Exemple: un algorithme qui lit un nombre entier N et détermine son premier multiple qui est supérieur ou égale à 50.

```
algorithme multiple
var N,M: entier
début
   lire (N)
   M ← N
   tantque M<50 faire
      M ← M+N
   finTQ
   ecrire (M)
fin
```

Exécution

```
7
56
```

- **Déroulement de l'exécution:** si la valeur entrée (tapée) de N est 7:

M
7
14
21
28
35
42
49
56

Remarque: chaque boucle `Pour` peut être toujours remplacée par une boucle `TantQue`, mais l'inverse n'est pas vrai.

Exemple: un algorithme qui affiche les nombres entiers de 1 à 10 (en utilisant la boucle `TantQue`).

```
algorithme nombres
var i: entier
début
   i ← 1
   tantque i ≤ 10 faire
      ecrire (i)
      i ← i+1
   finTQ
fin
```

Exécution

```
1
2
3
4
5
6
7
8
9
10
```

C. La boucle `répéter`

La boucle répéter est utilisée elle aussi lorsque le nombre d'itérations n'est pas connu. On répète l'exécution d'un bloc d'instruction *jusqu'à ce qu'une condition soit vraie*. Elle a la forme suivante:

```
répéter
    Bloc d'instructions
jusqu'à condition
```

✓ Dans la boucle `Répéter`, le bloc instructions est exécuté au moins une fois quelle que soit la condition.

Exemple: un algorithme qui continue de demander à l'utilisateur de taper des valeurs entières, jusqu'à ce que l'utilisateur tape une valeur négatif.

```
algorithme val_entières
var v: entier
début
    répeter
        ecrire ("Tapez une valeur entière")
        lire (v)
    jusqu'à v < 0
fin
```

Exécution

```
Tapez une valeur entière
9
Tapez une valeur entière
7
Tapez une valeur entière
12
Tapez une valeur entière
-2
```

Exercices:

1. Ecrire un algorithme qui calcule X^n.

2. Ecrire un algorithme qui calcule la somme des nombres entiers supérieurs à 1 et inférieurs à 100.

Solutions:

```
algorithme puissance
var n, i: entier
    X,P : réel
début
    P ← 1
    pour i = 1 à n faire
        P ← P*X
    finpour
    ecrire (P)
fin
```

```
algorithme sommes
var S, i: entier
début
    S ← 0
    pour i = 2 à 99 faire
        S ← S+i
    finpour
    ecrire (S)
fin
```