



Cours Algorithmique

Prof. A. BOUZIDI

Année Universitaire : 2021/2022

Table des matières

1	Introduction	2
1.1	Définition	2
1.2	Etapes de résolution informatique d'un problème	2
1.3	Propriétés d'un algorithme	2
1.4	Représentation d'un algorithme :	2
1.4.1	Organigramme	3
1.4.2	Pseudo-code	3
2	Notion de base de l'algorithme:	4
2.1	Les entrées et les sorties :	4
2.2	Les variables	4
2.2.1	Types de d'une variable	5
2.2.2	Une constante :	5
2.2.3	Affectation :	5
2.2.4	Expression et Opérateurs	6
2.2.5	Exercice :	7
2.3	Instruction de contrôle :	8
2.3.1	La séquence :	8
2.3.2	La sélection :	9
2.3.3	Choix multiple	11
2.3.4	Les structures itératives	11
2.4	Les boucles :	12
2.5	Les Tableaux :	14
2.5.1	Problématique	14
2.5.2	Exemple :	14
2.6	Les Procédures et les Fonctions :	16
2.7	Les Procédures	16
2.8	Les Fonctions	17
2.8.1	Récursivité	18

1 Introduction

1.1 Définition

- Le terme **algorithme** prend sa racine du mot *algorisme*, qui définit le processus de faire l'arithmétique en utilisant les chiffres arabes.
- Le mot *algorisme* est issu de la transcription phonétique du nom du célèbre mathématicien AL-Khawarizmi (785 - 850).
- Un **algorithme** est une suite logique d'opérations élémentaires à traiter dans un ordre déterminé. Son implémentation permet de résoudre un problème déterminé.
- L'**algorithmique** est la logique d'écrire des algorithmes.

1.2 Etapes de résolution informatique d'un problème

Avant la réalisation d'un algorithme par n'importe quel langage de programmation, le programmeur doit passer par les étapes suivantes qui sont :

- 1) Comprendre l'énoncé du problème
- 2) Décomposer le problème en sous problèmes plus simples à résoudre
- 3) Associer à chaque sous-problème, une spécification :
 - des données nécessaires
 - des données résultantes
 - Démarche à suivre pour arriver au résultat en partant d'un ensemble de données

Exemple : préparer une recette de cuisine

- 4) Élaboration d'un algorithme
- 5) Validation de l'algorithme
- 6) Mise au point

1.3 Propriétés d'un algorithme

❑ Un algorithme doit:

- avoir un nombre fini d'étapes,
- avoir un nombre fini d'opérations par étape,
- se terminer après un nombre fini d'opérations,
- fournir un résultat.

❑ Chaque opération doit être:

- définie rigoureusement et sans ambiguïté
- effective, c.-à-d. réalisable par une machine

1.4 Représentation d'un algorithme :

- Il existe deux façons pour représenter un algorithme :
 - Une représentation graphique appelée : **Organigramme**
 - Une représentation textuelle appelée : **Pseudo-code**

1.4.1 Organigramme

Un organigramme représente le traitement de l'algorithme par l'utilisation de 4 formes géométriques, dont chacune à un rôle définit :

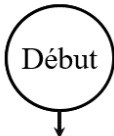
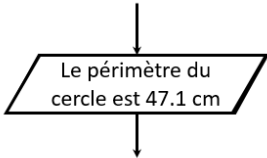
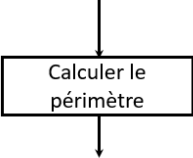
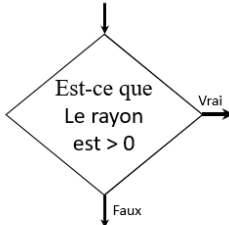
Le cercle est utilisé pour le *début* et la *fin* d'un organigramme, et on écrit le nom début ou fin à l'intérieur du cercle

Le parallélogramme représente (toutes les interactions avec l'extérieur) les instructions d'entrée et de sortie, par exemple pour calculer le périmètre l'algorithme va vous demander de saisir la valeur du rayon du cercle, c'est l'entrée. La sortie sera la valeur du périmètre si le rayon est égal à 15 cm.

Le rectangle représente l'action à réaliser par le système.

Un losange inscrit les conditions du système, par la considération du problème du calcul du périmètre d'un cercle, il faut que le rayon soit supérieur à 0.

Vous pouvez maintenant déduire que l'organigramme offre une vue d'ensemble de l'algorithme.

Forme géométrique				
Rôle	Début où fin	Entrées et Sorties	Traitement	Test

1.4.2 Pseudo-code

C'est une représentation textuelle avec une série d'instructions et structures de contrôle.

- La structure générale d'un pseudo code est :

```
Algorithme nom_algorithme  
Début  
    <<instructions>>  
Fin
```

2 Notion de base de l'algorithme:

2.1 Les entrées et les sorties :

Pour chaque algorithme, les entrées et les sorties sont définies, dont les:

- Entrées : Sont les données saisies par l'utilisateur de programme.

Lire(<<donnée>>)

- Sorties : sont les données affichées par le programme.

Écrire(<<expression>>)

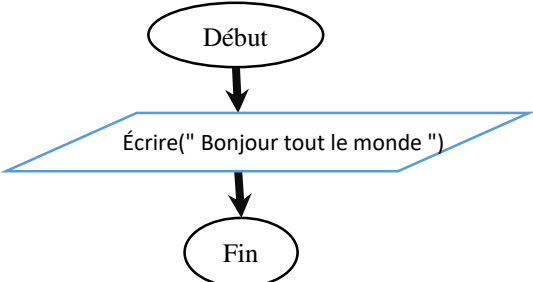
Par la représentation par organigramme, c'est le parallélogramme qui représente toutes les interactions avec l'extérieur du système.

2.1.1.1 Exemple

Ecrire un algorithme nommé *Mon_algorithme* qui affiche à l'écran le texte:

"Bonjour tout le monde"

La solution:

Pseudo-code	Organigramme
<pre>Algorithme Mon_algorithme Début Ecrire("Bonjour tout le monde") Fin</pre>	

2.2 Les variables

Une variable est un objet qui peut prendre une ou plusieurs valeurs durant son existence dans un algorithme donné, il désigne un emplacement mémoire dont le contenu peut changer au cours du processus d'un algorithme, à ajouter aussi qu'une variable doit être déclarée avant d'être utilisée.

Chaque variable est caractérisée par les trois propriétés suivantes :

- Son **nom** (identification ou désignation de l'emplacement mémoire de la variable) qui la différencie des autres et permet l'accès à sa valeur.
- Son **type**, qui spécifie le domaine de valeurs que peut prendre la variable.
- Sa **valeur** (c'est le contenu actuel de l'emplacement mémoire de la variable).

On déclare une variable **x** de type **T** entier par :

Notation **x : T**

Le choix des noms de variables est soumis à quelques règles qui varient selon le langage, mais en général:

- **doit commencer par une lettre**

Exemple ~~1ernote~~ ~~note 1^{er}~~ - noter1 ~~A&B~~

- **Doit être constitué uniquement de lettre, de chiffre et du caractère de soulignement**

Exemple : prix_TVA, A_B, ~~A-B~~

- Pour la lisibilité du code, **le choix nom doit être significatif**

Exemple : somme_totale, prix_TVA, note_generale, ...

2.2.1 Types de d'une variable

Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre, les types offerts par la plupart des langages sont:

- Entier
- Réel
- Booléen
- Caractère : lettres majuscules, minuscules, chiffres, symboles

Exemples: 'A', 'a', '1', '?', ...

- Chaîne de caractères : toute suite de caractères

Exemples: " Bonjour tout le monde", "code postale: 1000",...

2.2.2 Une constante :

C'est un cas particulier d'une variable dont la valeur ne varie pas pendant l'exécution.

Notation : **Identificateur_de_constant = valeur**

2.2.2.1 Exemple :

TVA =0.2

Pi=3.14

g =9.8

2.2.3 Affectation :

L'affectation consiste à attribuer une valeur à une variable, que ça soit remplir ou modifier le contenu de la zone mémoire utilisée.

Le pseudo code de l'affectation se note avec le signe flèche :.

Notation : «← ».

On affecte une variable par une expression du même type. Il faut noter qu'une expression du style : **x ← expression**, ne se lit que dans un seul sens : la variable x prend la valeur de l'expression à droite.

Exemples :

- **X← -2.56** , signifie que la variable X reçoit la valeur -2.56 (sa valeur actuelle). Son type est réel.
- **Y1 ← VRAI**. Son type est booléen { vrai, faux }.
- **Z ← "Bonjour"**. Son type est chaîne de caractères.

Il existe trois formes d'affectation :

- Variable \leftarrow constante.
- Variable \leftarrow variable (ex. $T \leftarrow Z$ et donc la valeur de la variable T est "Bonjour").
- Variable \leftarrow expression composée, arithmétique ou logique (ex. $M \leftarrow (x^2 + 1)/2$).

Faire attention à la concordance des types de variables pendant l'affectation. Supposons que x est de type réel et que n est de type entier. L'affectation $x \leftarrow n$ est valide alors que $n \leftarrow x$ ne l'est pas.

A ajouter que

l'affectation n'est pas commutative : $A \square B$ est différente de $B \square A$

l'affectation est différente d'une équation mathématique :

- $A \square A+1$ a un sens en langages de programmation
- $A+1 \square 2$ n'est pas possible en langages de programmation et n'est pas équivalente à $A \square 1$

2.2.4 Expression et Opérateurs

Il existe quatre types d'opérateurs, ils sont :

2.2.4.1 Les opérateurs arithmétiques

Ce sont les expressions dont le résultat est un nombre entier ou réel.

Opérateurs arithmétique	Exemple
+ (addition)	$8+4+3+1$ (résultat 16)
- (soustraction)	$7 - 47$ (résultat -47)
* (multiplication)	$9*5.2$ (résultat 46.8)
/ (division)	$18 \text{ div } 3$ (résultat 6)
% (reste de la division)	$17 \text{ mod } 2$ (résultat 1)
^ (puissance)	2^3 (résultat 8)

Pour les opérateurs arithmétiques donnés ci-dessus, l'ordre de priorité est le suivant (du plus prioritaire au moins prioritaire) :

^ (élévation à la puissance)

*, / (multiplication, division)

% (modulo)

+, - (addition, soustraction)

exemple $4 + 2 * 5$ vaut 14

En cas de besoin (ou de doute), on utilise les parenthèses pour indiquer les opérations à effectuer en priorité

2.2.4.2 Les opérateurs logiques

Ce sont les expressions dont le résultat est vrai ou faux

Opérateurs relationnelle	Exemple
= (égalité)	8 = 6 (résultat FAUX)
< > (différent)	7 < > 47 (résultat VRAI)
< (strictement inférieur)	9 < 5.2 (résultat FAUX)
> (strictement supérieur)	18 > 3 (résultat VRAI)
>= (supérieur ou égale)	17 >= 2 (résultat VRAI)
<= (inférieur ou égale)	2 <= 3 (résultat VRAI)

2.2.4.3 Les opérateurs relationnels

Opérateurs logique	Exemple
NON	NON(VRAI) (FAUX)
OU	VRAI OU FAUX (VRAI)
ET	VRAI ET FAUX (FAUX)

2.2.4.4 Les opérateurs sur les chaînes de caractères

Ce sont les expressions qui opèrent sur les chaînes de caractère. Parmi les opérations appliquées sur les chaînes de caractère on peut citer la concaténation &

Exemple :

A □ "Bonjour"

B □ "tout le monde"

C □ "A & B"

2.2.5 Exercice :

Quel est le résultat de l'algorithme suivant?

```
Algorithme Algol  
  
Variable  
  
a,b,c : entier;  
  
Début  
  
a ← 12
```



```

b ← 3
c ← a
a ← b
b ← c
Écrire(' a =', a, ' b =', b, ' c =', c)
Fin

```

Le résultat est : a=3 b=12 c=12

2.3 Instruction de contrôle :

Dans la programmation classique (structurée), on distingue trois structures :

- ✓ La séquence
- ✓ La sélection
- ✓ La répétition

2.3.1 La séquence :

On dit aussi instruction composée : c'est une suite d'instructions (arithmétiques ou autres), délimitée par les mots-clés, **DEBUT** et **FIN**, qui s'exécutent séquentiellement (c'est-à-dire l'une après l'autre).

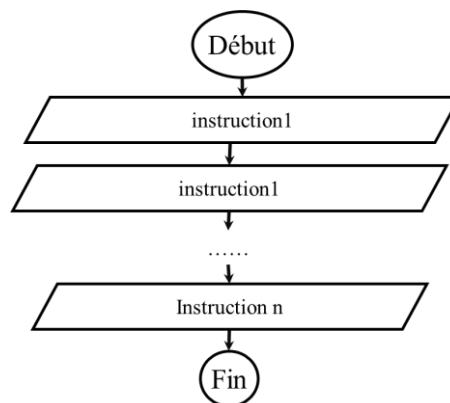


Figure 1: Organigramme de la séquence (instruction composée)

Exemple

L'algorithme suivant lit les variables x et y, en fait la somme z et affiche le résultat :

```

Algorithme Somme

Variables

x , y : entier

Debut

    Ecrire(" X= ")

    Lire(x)

    Ecrire("y= ")

    Lire(y)

```

```

    z ← x+y

    Ecrire(" z = ", z)

Fin

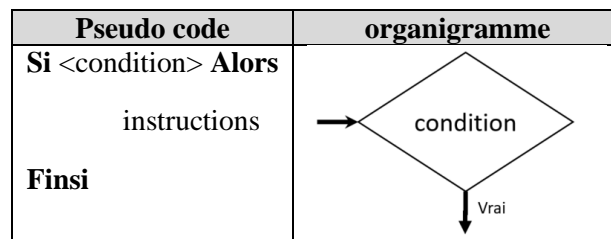
```

2.3.2 La sélection :

Elle comporte les instructions conditionnelles : c'est à dire qu'une instruction (simple ou composée) ne sera exécutée qu'après remplissage (satisfaction) de certaines conditions.

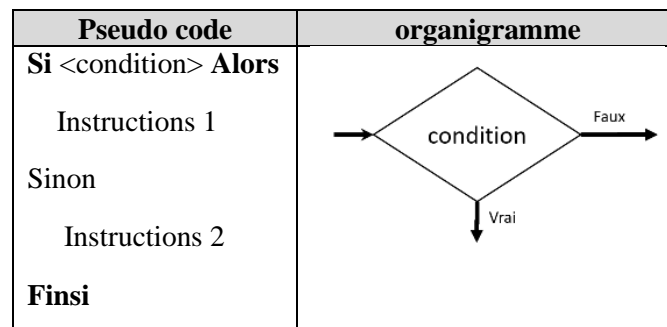
Dans ce qui suit <condition> a comme type les booléens {vrai, faux}; c'est une expression construite à partir des variables booléennes et des connecteurs (opérateurs) logiques **ET** , **OU**, **NON**.

Notation 1 :



Si la condition <condition> est vraie, alors exécuter les instructions; sinon on ne fait rien. On passe à l'exécution de l'instruction suivante (qui est juste après FSI)

Notation 2 :



Si la condition <condition> est vraie, alors exécuter les instructions 1; sinon exécuter les instructions 2

Exemple :

Ecrire un algorithme qui affiche la valeur absolue d'un nombre entier **x** saisie au clavier

```

Algorithme AffichageValeurAbsolue

Variable x : Entier

Début

    Ecrire (" Entrez svp une valeur entiere : ")

    Lire (x)

    Si(x<0) alors

        Ecrire ("la valeur absolue de ", x, "est:", (-1)*x)

    Sinon

```

```

        Ecrire ("la valeur absolue de ", x, "est:", x)

    Finsi

Fin

```

Remarque:

L'instruction après ALORS ou SINON est quelconque et peut être une instruction SI. Dans ce cas, on dit qu'on a des SI imbriqués.

Exemple :

```

SI (a = 0) ALORS

    SI (b = 0) ALORS

        x = x/2 ;

    FinSI

SINON

    x = 2*x ;

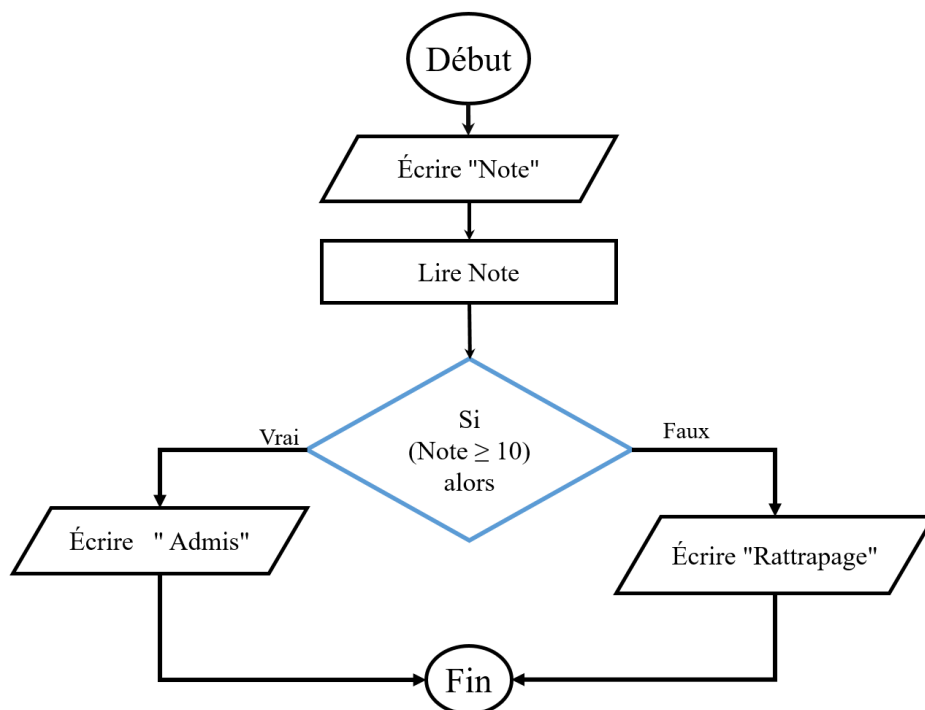
FinSI

```

On remarque que l'instruction ci-dessus affecte à la variable réelle x sa moitié si a et b sont nulles. Si a = 0, x est doublée.

Exemple :

Dessiner l'organigramme d'un algorithme qui demande à l'utilisateur de saisir la note d'un étudiant (valeur réelle), par la suite affiche le message admis si la note ≥ 10 , sinon l'algorithme affiche le message Rattrapage.



2.3.3 Choix multiple

C'est une généralisation de la sélection qui permettra de choisir un traitement entre plusieurs alternatives (plus de deux).

```
Selon variable
  Val1 : groupe d'instruction1;
  Val2 : groupe d'instruction2;
  ...
  Valn : groupe d'instruction n;
Autrement : bloc
finselec
```

Exemple :

Ecrire un algorithme permettant la correspondance entre une couleur et un code. L'algorithme demande à l'utilisateur de saisir un code (une valeur entière) entre 1 et 6 et affiche la couleur correspondante. Le tableau suivant montre les couleurs possibles et les codes correspondants.

Couleur	code
rouge	1
vert	2
bleu	3
jaune	4
blanc	5
noir	6

Solution :

```
Algorithme couleur
variables
  code : ENTIER
début
  Ecrire("Saisir svp le code")
  lire(code)
  Selon (code)
    1: écrire("rouge")
    2: écrire("vert")
    3: écrire("bleu")
    4: écrire("jaune")
    5: écrire("blanc")
    6: écrire("noir")
    autrement : écrire("couleur
inconnue")
  finselec
fin
```

2.3.4 Les structures itératives

- Les structures itératives sont des instructions qui permettent de réaliser (répéter) un traitement plusieurs fois
- Pour réaliser et mettre en place une structure itérative on doit connaître:

- Soit le nombre de répétition (itération)
- Soit la condition d'arrêt qui est une expression logique qu'on vérifie avant ou après chaque répétition.
- On distingue trois sortes de boucles en langages de programmation :
 - Les boucles tant que : on y répète des instructions tant qu'une certaine condition est réalisée
 - Les boucles jusqu'à : on y répète des instructions jusqu'à ce qu'une certaine condition soit réalisée
 - Les boucles pour ou avec compteur : on y répète des instructions en faisant évoluer un compteur (variable particulière) entre une valeur initiale et une valeur finale

2.4 Les boucles :

Une boucle permet de décrire la répétition d'un traitement (bloc) à un certain nombre de fois.

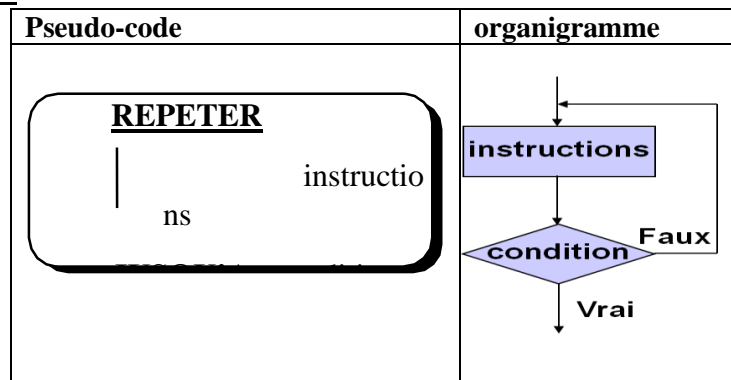
Le nombre de répétitions du bloc n'est pas toujours connu à l'avance, et l'arrêt de la boucle est décidé par la vérification d'une expression logique. Trois boucles sont définies :

- **répéter Jusqu'à**
- **Tantque**
- **pour.**

2.4.1.1 La boucle • répéter Jusqu'à

Nous commençons par créer le bloc, et ce n'est qu'après avoir évalué l'expression logique que nous pouvons décider de reprendre le bloc ou de quitter la boucle, et de traiter les instructions après cette boucle.

2.4.1.1.1 Notation



Exemple : un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100 (version avec répéter jusqu'à)

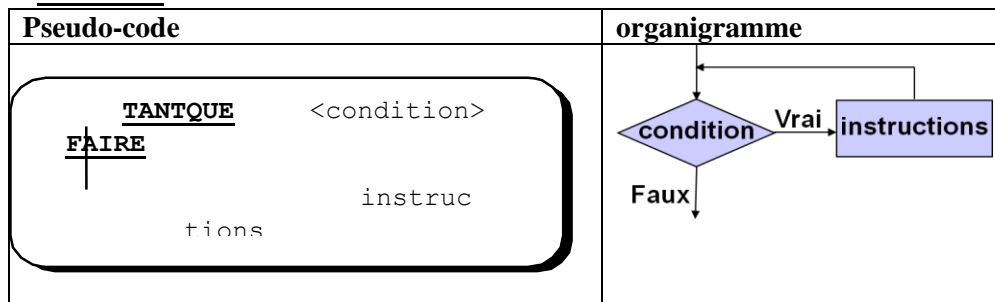
Algorithme Exemple_repeter**Variables**

som, i : ENTIER

Debutsom \leftarrow 0i \leftarrow 0**Répéter**i \leftarrow i+1som \leftarrow som+i**Jusqu'à** (som > 100)**Ecrire** (" La valeur cherchée est N= ", i)**Fin****2.4.1.2 La boucle tantque**

Le nombre d'itération n'est pas connu à priori. La boucle **tantque** fonctionne de la manière suivante :

- Evaluer une expression logique
 - *Vraie* : elle fait le bloc et recommence
 - *Fausse* : elle sort

2.4.1.2.1 Notation :

On note qu'on évalue d'abord la condition *<condition>* ; si elle est vraie on exécute les instructions « *instructions* » et on retourne pour réévaluer la condition. Dès que la condition est fausse, on exécute l'instruction qui suit la boucle **TANT QUE ... FAIRE...**

2.4.1.3 La boucle pour

Le nombre de répétition du bloc est prédéterminé pour la boucle Pour. On distingue deux cas :
Le pas = 1 et le pas \neq 1

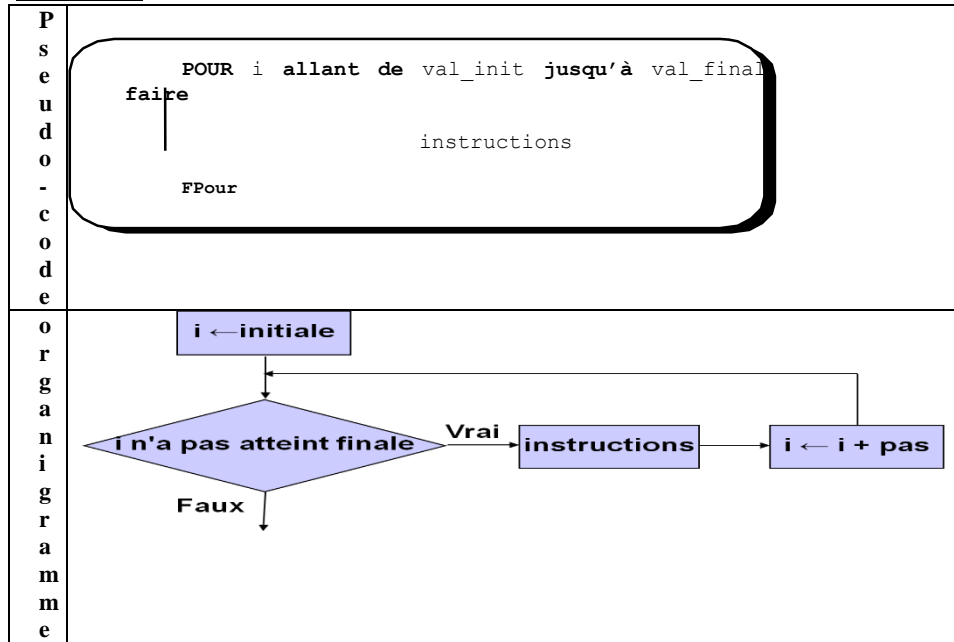
Pas = 1 :

On initialise v par vin et à chaque itération on incrémente (resp. décrément) v par 1 jusqu'à ce que v devienne égale à vfin. Le nombre de répétitions du bloc est donc égal à vfin- vin+1. v, vin et vfin sont entiers.

Pas \neq 1 :

Dans ce cas, on doit sauter par un pas qui est différent de 1. Le pas est appelé incrément (resp. décrétement). On initialise v par vin et à chaque itération on incrémente (resp. décrément) par la valeur d'incrément (resp. décrétement) jusqu'à ce que v devienne supérieure ou égale (resp. inférieure ou égale) à vfin.

2.4.1.3.1 Notation :



Exemple :

Calcul de x à la puissance n où x est un réel non nul et n un entier positif

```

Algorithme ExemplePour
Variables x, puiss : réel
            n, i : entier
Debut
    Ecrire (" Entrez la valeur de x ")
    Lire (x)
    Ecrire (" Entrez la valeur de n ")
    Lire (n)
    puiss ← 1
    Pour i allant de 1 à n faire
        puiss ← puiss * x
    FinPour
    Ecrire (x, " à la puissance ", n, " est égal à ", puiss)
Fin
        
```

2.5 Les Tableaux :

2.5.1 Problématique

Soit $t = (x_1, x_2, \dots, x_n)$ une suite finie d'éléments d'un certain type X . Si n est suffisamment grand ($n > 100$) et que l'on veut représenter la suite t , il n'est pas pratique de déclarer n variables de type X . Une structure est donc nécessaire pour gérer des séquences finies et accéder à chaque élément de la séquence. On observe qu'une séquence t est caractérisée par un ensemble d'indices de représentation (ici l'intervalle $1..n$) et de valeurs prises par les éléments de la séquence (ici, X). Des exemples typiques sont les vecteurs, les matrices et les polynômes

2.5.2 Exemple :

Supposons que l'on veuille conserver les notes d'une classe de 100 élèves pour en extraire quelques informations. Exemple : Comptez le nombre d'élèves ayant des notes supérieures à 10.

La seule façon dont nous disposons actuellement est de déclarer 100 variables, disons N_1, \dots, N_{100} . Après avoir lu 100 instructions, nous devons écrire 100 instructions SI pour faire le calcul

```
nbre ← 0
```

Si (N1 >10) alors nbre ←nbre+1 FinSi ... Si (N100>10) alors nbre ←nbre+1 FinSi
--

Heureusement, les langages de programmation offrent la possibilité de rassembler toutes ces variables dans une structure de données appelée tableau.

Un tableau est un groupe d'éléments du même type spécifié par un identifiant unique. Une variable entière nommée index utilisée pour indiquer la position d'un élément donné dans le tableau et déterminer sa valeur

Un tableau est créé en spécifiant le type de ses éléments et leur dimension (nombre de ses éléments).

Notation en Pseudo Code

Variables

Tableau identificateur[<<dimension>>] : <<Type_elements>>

Exemple :

Variables Tableau score[30] : Réel

Vous pouvez définir les éléments du tableau comme pour les variables : tableaux entiers, de nombres réels, de caractères, de booléens, de chaînes.

L'accès aux éléments du tableau se fait par indexation. Par exemple, score[i] donne la valeur de l'élément à la position i dans ce tableau, et le premier indice du tableau est 0 ou 1, selon la langue. La plupart du temps c'est 0 (c'est ce qu'on va faire en pseudocode). Dans ce cas, score[i] représente l'élément i+1 du tableau score.

Un grand avantage des tableaux est qu'on peut traiter les données qui y sont stockées de façon simple en utilisant des boucles.

Exemple :

Écrire un algorithme qui déclare un tableau notes d'une taille 50 dont on demande par la suite la saisie des notes par l'utilisateur, à la fin l'algorithme va afficher le nombre d'étudiants ayant la note supérieure ou égale à 10.

Algorithme nbr_valider Variables i ,nbre : entier tableau notes[50] : réel Debut Pour i allant de 0 jusqu'à 49 faire Ecrire("Note[" ,i, "]=") Lire(note[i]) FinPour Pour i allant de 0 jusqu'à 49 faire Si (notes[i] >=10) alors nbre ←nbre+1 FinSi FinPour Ecrire ("le nombre de notes supérieures à 10 est : ", nbre) Fin
--

2.6 Les Procédures et les Fonctions :

Jusqu'à maintenant vous avez appris pas mal de notions de programmation. Mais lors de la réalisation de votre code, vous avez remarqué quelque limitation frustrante, et une certaine lourdeur lorsqu'il s'agit de répéter quelque bloc d'instruction.

Pour cela vous pouvez envisager un sous-programme qui sera lancé par le programme principal.

Les algorithmes que nous avons étudiés jusqu'à maintenant se composent d'un seul bloc d'instructions appelé l'algorithme principal.

Pour éviter la répétition d'un ensemble d'instructions, on les répartit sous forme de Module où sous-programme.

Les **procédures** et les **fonctions** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

- ✓ Permettent de "factoriser" les programmes, c-à-d de mettre en commun les parties qui se répètent
- ✓ Clarté dans la programmation: les fonctions et les procédures permettent de réaliser des programmes bien structurés, lisibles et faciles à écrire et à comprendre.
- ✓ Facilitent la maintenance du code (il suffit de le modifier une seule fois)
- ✓ Réduction de la taille des programmes: l'ensemble des instructions concernant le traitement réalisé par une fonction ou une procédure est écrit une seule fois dans le code d'un programme d'où l'élimination des duplications d'un même code.

2.7 Les Procédures

Dans certains cas, on peut avoir besoin de répéter une tâche dans plusieurs endroits d'un algorithme, dans ces cas on peut utiliser les procédures.

Une procédure est un module (groupe d'instructions distinctes) indépendants désigné par un nom.

Une fonction s'écrit en dehors de l'algorithme principal sous la forme :

```
Procedure nom_procedure(parametre1:typeP1,..., parametren:typePn)

Variable

    <<variables>>

Début

    <<instructions>>

FinProcedure
```

Exemple :

Écrire la procédure Fill qui permet de demander les éléments d'entrée d'un tableau d'éléments réels de taille entière n.

```
Procedure Fill(Tableau T[]:Réel, n: Entier)
```

```
Variable
```

```
    i: Entier
```

```
Début
```

```
    Pour i allant de 0 jusqu'à n-1 faire
```

```
        Ecrire("T[" , i , "]=")
```

```
        Lire(T[i])
```

```
    FinPour
```

```
FinProcedure
```

Écrire une procédure Display pour afficher les éléments d'un tableau d'éléments réels

```
Procedure Display(Tableau T[]:Réel, n: Entier)
```

```
Variable
```

```
    i: Entier
```

```
Début
```

```
    Pour i allant de 0 jusqu'à n-1 faire
```

```
        Ecrire("T[" , i , "]=", T[i])
```

```
    FinPour
```

```
FinProcedure
```

Écrire un algorithme qui déclare un tableau T de 20 éléments de type réel, dont on fait ensuite saisir les valeurs par l'utilisateur, à la fin l'algorithme affichera les éléments du tableau par l'utilisation des procédures FILL et Display.

```
Algorithme exemple_procedure_tableau
```

```
Variables
```

```
    tableau notes[20] : réel
```

```
Debut
```

```
    Fill(T,20)
```

```
    Display()
```

```
Fin
```

2.8 Les Fonctions

Une fonction agit dans un algorithme comme une fonction en mathématiques : elle calcule un résultat basé sur les valeurs de ses arguments d'entrée.

Une fonction écrite en dehors de l'algorithme principal est de la forme :

```

Fonction nom_fonction(parametre1:typeP1,..., parametren:typePn): Type_retour
Variable

    <<variables>>

Début

    <<instructions>>

    retourne valeur

FinFonction

```

Exemple :

Écrire la fonction *somme* permettant de calculer et afficher la somme de deux réels fournis en argument

```

Fonction somme (val1 : Réel, val2 :Réel): Réel
Variable

    resultat :Réel

Début

    resultat ← val1 + val2

    retourne resultat

FinFonction

```

L'utilisation de la fonction se fera en écrivant simplement son nom dans l'algorithme principal. Lors de l'appel, tous les paramètres doivent être définis. L'ordre, le nombre et les types des paramètres d'appel doivent correspondre aux paramètres d'ordre et formels et à leurs types. Le résultat est une valeur qui doit être affectée ou utilisée dans les expressions, l'écriture...

Exemple :

```

Algorithme exemple_somme
Variables

    result: entier

Debut

    result ← somme (3,5)
    Ecrire ("le somme = ", result)

Fin

```

Lors de l'appel `somme(3,5)` le paramètre **formel** `val1` est remplacé par le paramètre **effectif** 3 et le paramètre formel `val2` est remplacé par le paramètre effectif 5.

2.8.1 Réversivité

Une fonction est visible par elle-même; ce qui permet d'utiliser le nom d'une fonction dans le corps de cette fonction. Ceci est possible puisque à chaque appel de la fonction, de nouvelles variables locales sont créées. Ce mécanisme s'appelle la réversivité. Il faut toujours utiliser une condition pour arrêter les appels récursifs. Cette condition s'appelle condition d'arrêt.

Une fonction qui peut s'appeler lui-même est dite fonction récursive et toute fonction récursive doit posséder un cas limite (cas trivial) qui arrête la réversivité.

Exemple :

➤ La factorielle $U_n = n!$ Avec $u_1 = 1$ et $u_n = n \times u_{n-1}$

```
Fonction factorielle(n : Entier) : Entier
//-----initialisation

    Si (n==1) alors

        retourne 1


    Sinon

//-----Hérédité

        retourne n * factorielle(n-1)

FinFonction
```

Dans cette fonction, les calculs s'effectuent au retour des appels récursifs

fact(3)	→	fact(2)	→	fact(1)
				
6	←	2	←	1