République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université Abdelhamid Ibn Badis de Mostaganem

Faculté des Sciences & de la Technologie

Service du Tronc commun Sciences & Technologies

Polycopié

Du cours : *Informatique I*

Elaboré par :

Dr. **BOUHARAOUA Farouk**

Domaine : Sciences & Technologies

Expertisé par :

Dr. **ROUBA Baroudi**	Maître de conférences	Université Abdelhamid Ibn Badis Mostaganem
Dr. **OULD MAMMAR Madani**	Maître de conférences	Université Abdelhamid Ibn Badis Mostaganem
Dr. **HENNI Sid Ahmed**	Maître de conférences	Université Abdelhamid Ibn Badis Mostaganem

Table des matières

Avant-propos

Ce polycopié est le support du cours « Informatique I », destiné aux étudiants de la première année du socle commun en Sciences et Technologies (ST).

Ce cours ne nécessite aucun prérequis, et a pour objectif d'introduire les généralités sur l'informatique, de découvrir et comprendre les concepts de bases de l'algorithmique et de la programmation.

Ce polycopié est structuré en 6 chapitres qui abordent :

- Les généralités sur l'informatique (chapitres 1 et 2)
- La structure d'un algorithme, les types de données élémentaires et les instructions élémentaires (chapitres 3).
- Les structures de contrôles : conditionnelles et itératives (chapitres 4, 5 et 6).

Pour faciliter la translation des algorithmes en programmes, la présentation des chapitres de l'algorithmique suit un style pascalien. De ce fait, nous avons opté pour le langage Pascal pour écrire les programmes. Toutes les syntaxes sont présentées en algorithmique et en langage Pascal ; et la majorité des algorithmes sont traduits en programmes Pascal.

Les notions abordées dans ce cours sont plus ou moins relativement détaillées, mais l'ensemble des chapitres peuvent présenter un bagage minimal pour tout programmeur novice.

Nous avons rédigé ce cours en s'appuyant sur divers ouvrages reconnus dans la discipline, et sur des ressources en ligne. Une liste de références bibliographiques est répertoriée à la fin du polycopié.

1 Chapitre 1 : Introduction à l'informatique

1.1 Qu'est-ce que l'informatique ?

L'informatique est la science d'automatisation du traitement de l'information. D'où la naissance du mot **Informatique** qui est composé des deux mots **Informa**tion et auto**matique**.

1.2 L'information

L'information est un élément de connaissance pouvant être communiqué, conservé et même traité.
Il faut distinguer dans une information sa **forme** et son **sens :**
La **forme** représente le code.
Alors que le **sens** représente l'interprétation de ce code.

Dans la vie courante, pour être communiquée, l'information est codée sous différentes formes par des signaux ou des symboles de natures variée (chiffres, lettres, signaux lumineux, symbole graphiques,…).
Le sens attribué à une information est obtenu par interprétation du code établi par l'homme.
Le tableau ci-dessous présente quelques exemples.

Forme de l'information	**Nature du signal**	**Sens associé à l'information**
Feu rouge	Lumineux	Arrêt des véhicules
Sonnerie de téléphone	Sonore	Quelqu'un veut joindre l'abonné dont le téléphone sonne
14/01/2021	Caractères alphanumériques	14ème jour du mois de janvier de l'année 2021

1.3 Le traitement de l'information

Le traitement automatique de l'information par machine (ordinateur, robot, cellulaire,…), consiste à faire élaborer par cette machine, des informations appelées **résultats**, à partir d'informations connues appelées **données**.
La figure ci-dessous représente un schéma simplifié du traitement de l'information.

Fig.1.1 : Principe du traitement de l'information

1.4 Système informatique

Un système informatique est composé de deux entités qui se complètent et qui sont tous les deux nécessaires à l'exécution des tâches qui lui sont confiées :

- **Le matériel (Hardware) :** c'est l'entité qui regroupe l'ensemble des circuits électroniques et des dispositifs qui forment la machine physique (ordinateur).

- **Le logiciel (Software) :** c'est l'entité qui regroupe tous les programmes devant être introduits dans la machine pour lui faire exécuter une tâche précise (facturation, gestion des stocks, calcul de la paye,…).

1.4.1 Le matériel (Hardware)

C'est l'ensemble du matériel constitutif de l'ordinateur.
L'ordinateur est la machine qui se charge du traitement automatique des informations (textes, images, sons, vidéos…). Il se compose généralement d'une unité centrale reliée à d'autres dispositifs appelés périphériques.

1.4.1.1 L'unité centrale (UC)

L'unité centrale est l'élément le plus important de la machine. C'est le boitier qui contient les dispositifs vitaux dont se compose un ordinateur. On y trouve entre autres :

- La **carte mère** : c'est le dispositif qui sert à regrouper et fixer tous les autres dispositifs de l'unité centrale.

- Le **processeur** : Il représente le cerveau (sens figuré) de l'unité centrale. Il est responsable des calculs et de l'exécution des instructions.

- Les **mémoires** : On distingue deux types de mémoire :
  - **La RAM** (Random Access Memory), aussi appelée mémoire vive. Son rôle est de stocker les programmes et les données lors de leur exécution. Elle a la particularité d'être volatile car elle perd son contenu dès qu'elle est mise hors tension (dès que l'ordinateur s'éteint). La RAM est accessible en lecture et écriture.

  - **La ROM** (Read Only Memory), aussi appelée mémoire morte. Elle est dédiée pour contenir les données et les programmes du constructeur (bios) indispensables au démarrage de l'ordinateur et du lancement du système d'exploitation. C'est une mémoire non volatile. La ROM a la particularité d'être accessible en lecture seulement.

- Les **cartes d'extension** (circuits électroniques) installées sur la machine :
  - **Carte graphique** : pour l'affichage.
  - **Carte sonore** : pour la restitution du son.
  - **Carte modem** : pour la transmission de données via les lignes téléphoniques.
  - **Carte réseau** : pour la transmission des données via un réseau.

- Les **lecteurs Disques** : lecteur disquette, lecteur Cd-Rom, lecteur de carte mémoire, etc…

1.4.1.2 Les périphériques

Les périphériques sont les dispositifs connectés à l'unité centrale. Ils permettent à l'ordinateur de communiquer avec l'extérieur.
Les périphériques sont regroupés en quatre catégories :

Les périphériques d'entrée : ils permettent d'introduire des données à l'ordinateur.
Par exemple : Clavier, souris, …

Les périphériques de sortie : ils permettent de restituer des informations à partir d'un ordinateur.
Par exemple : Imprimante, écran, ….

Les périphériques d'entrée-sortie : ils permettent d'introduire et de restituer des informations à partir d'un ordinateur.
Par exemple : Modem, carte réseau,…

Les périphériques de stockage : appelés aussi mémoires auxiliaires. Ils permettent de stocker des informations.
Par exemple : Disque dur, carte mémoire, flash disque….

1.4.1.3 Les ports de communications

Les ports matériels de communications, appelés aussi connecteurs, sont les endroits (interfaces physiques) se trouvant sur le boitier de l'unité centrale sur lesquels les périphériques peuvent être connectés.
Par exemple : port USB, port HDMI, port Ethernet (réseau), port VGA (graphique), port souris, port clavier…

1.4.1.4 Composition minimale d'un ordinateur

Afin d'assurer l'interaction (communication) entre l'homme et la machine, l'ordinateur doit être composé au moins de ces trois éléments :

1. de l'unité centrale ;
2. d'un clavier comme périphérique d'entrée :
   Pour permettre à l'humain de communiquer avec l'ordinateur (en introduisant des données) ;
3. et d'un écran comme périphérique de sortie :
   Pour permettre à l'ordinateur de communiquer avec l'humain (en affichant des informations).

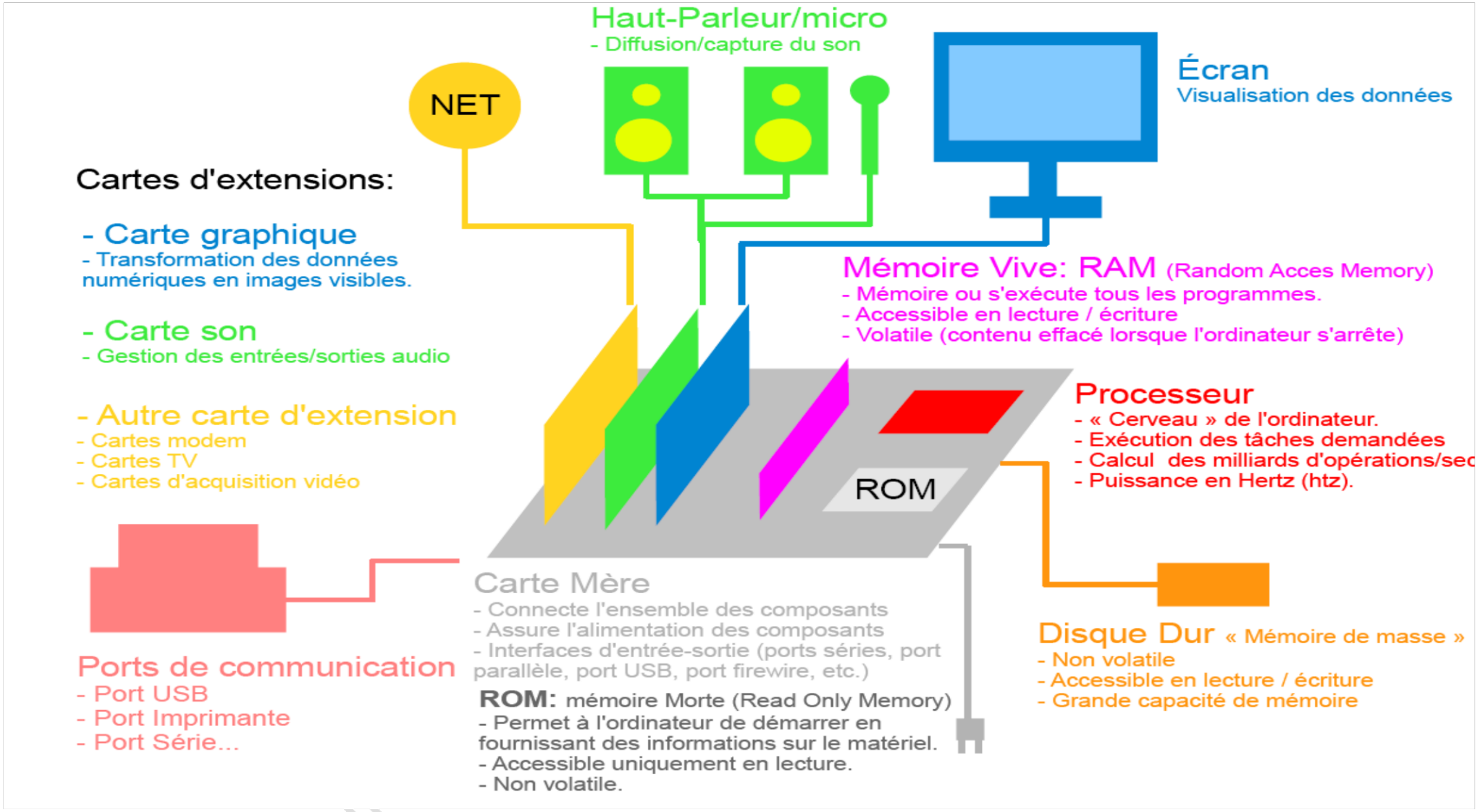


Fig.1.2 : Composant d'un Ordinateur.

1.4.2 Les logiciel (Software)

C'est l'ensemble des programmes qui s'exécutent sur un ordinateur. On distingue deux principales catégories :

1. Les logiciels de base : qui constituent les systèmes d'exploitation.
   La présence d'un système d'exploitation est nécessaire au démarrage et à l'utilisation de tout ordinateur.
   Par exemple : MS-DOS, Windows, Linux, Unix, OS2…
2. Les logiciels d'application : ce sont des programmes supplémentaires installés et utilisés selon les besoins des utilisateurs.

   Parmi les familles de ces logiciels qui existent on distingue :

   - Les logiciels de bureautique : On réunit dans cette catégorie les logiciels de traitements de textes, les tableurs et autres outils nécessaires au travail administratif.
     Par exemple : Word, Excel, WordPerfect….
   - Les antivirus : Ces programmes protègent l'ordinateur contre les virus, qui peuvent être dévastateurs, et vérifient toutes nouvelles données avant de l'enregistrer dans le disque dur.
     Par exemple : Norton antivirus, Avira…
   - Les systèmes de gestion de bases de données : Destinées à offrir à l'utilisateur un environnement de gestion de ses bases de données (Personnel, stock, facturation..).
     Par exemple : Access, DBase….

1.5 La quantité d'information

L'information doit être codée en binaire (suite des *0* et des *1*) pour qu'elle puisse être traitée par les composants électroniques de l'ordinateur :

- Etat *1* : présence du signal électrique (allumé).
- Etat *0* : absence du signal électrique (éteint).

Cette unité d'information (*0* ou *1*) est appelée **bit** (**bi**nary digi**t** en anglais).

- Avec un bit, on peut représenter (ou coder) $2^1=2$ informations (*0* ou *1*)
- Avec 2 bits, on peut représenter $2^2=4$ informations (*00*, *01*, *10* ou *11*).
- Avec 8 bits, on peut représenter $2^8=256$ informations.
- Donc avec n bits, on peut représenter 2^n informations.

Pour mesurer la capacité de stockage d'une mémoire ou d'un périphérique de stockage, on utilise l'octet qui peut être assimilé à la quantité de mémoire nécessaire pour stocker une lettre de l'alphabet. Mais la taille actuelle des mémoires est telle que l'on utilise plus souvent des multiples de l'octet :

- *1* octet= *8* bits
- Le kilo octet (ou ko)= *1024* octets=2^{10} octets.
- Le Méga octet (ou Mo)= *1024* ko.
- Le Giga octet (ou Go)= *1024* Mo.
- Le Téra octet (ou To)=*1024* Go.

Exemples

5 ko= 5*1024 octet =5120 octet
256 ko=256/1024 Mo=0.25 Mo
125368 bits=125368/8 octets =15671 octets=15671/1024^2 Mo=0.015 Mo

2 Chapitre 2 : Systèmes de numération

2.1 Introduction

Dans la vie courante, nous avons pris l'habitude de représenter les nombres en utilisant dix symboles (chiffres) différents: *0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9*. Ce système est appelé le système décimal.
Il existe cependant d'autres formes de numération qui fonctionnent en utilisant un nombre de symboles distincts.

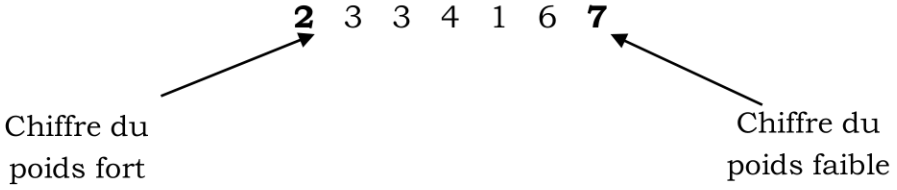
Exemple

- Le système binaire (bi : *2*),
- Le système octal (oct : *8*),
- Le système hexadécimal (hexa : *16*).
- Le système décimal (déci : *10*).

Ces quatre systèmes ci-dessus sont les systèmes les plus utilisés en informatique.
En fait, nous pouvons utiliser, dans un système de numération, n'importe quel nombre de symboles différents (pas nécessairement des chiffres). Le nombre de symboles distincts d'un système de numération est appelé **base.**

2.2 Système décimal

La base de ce système est 10.
Dans le système décimal, dix symboles différents sont utilisés : { *0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9* }.
N'importe quelle combinaison (par juxtaposition) de ces symboles nous forme un nombre.
Le chiffre qui se trouve le plus à gauche est appelé le chiffre du poids fort.
Le chiffre qui se trouve le plus à droite est appelé le chiffre du poids faible.



2.2.1 Développement polynômial d'un nombre décimal

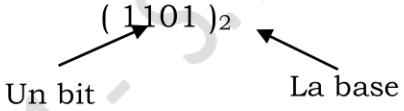
Soit le nombre *1978*.
Ce nombre peut être écrit sous la forme suivante :
1978 =1000 + 900 + 70 + 8
1978= 1*1000 + 9 *100 + 7 *10 + 8 *1
$1978 = 1*10^3 + 9*10^2 + 7*10^1 + 8*10^0$. ← Ce format s'appelle **la forme polynomiale** du nombre *1978*.

Le chiffre *1* est associé à la puissance la plus élevée (10^3), donc c'est le chiffre du poids fort. Tandis que le chiffre 8 est associé à la puissance la plus faible (10^0), donc c'est le chiffre du poids faible.

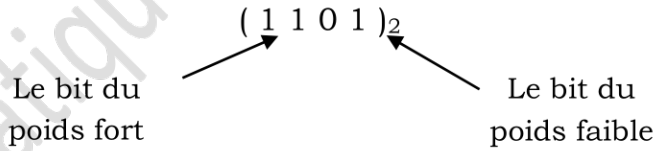
Un nombre réel peut aussi être écrit aussi sous sa forme polynomiale.
1978,265 = 1000 + 900 + 70 + 8 + 0,2 + 0,06 + 0,005
1978,265 = 1*1000 + 9 *100 + 7 *10 + 8 *1 + 2/10 + 6/100 + 5/1000
$1978{,}265 = 1*10^3 + 9*10^2 + 7*10^1 + 8*10^0 + 2*10^{-1} + 6*10^{-2} + 5*10^{-3}$

2.3 Système binaire

La base du système binaire est *2*.
Dans le système binaire, uniquement deux symboles {*0, 1*} sont utilisés pour exprimer n'importe quelle nombre.



Dans un nombre binaire, chaque chiffre est appelé « bit » (binary digit).



2.3.1 Forme polynomiale d'un nombre binaire

Le nombre binaire peut aussi être développé sous sa forme polynomiale.

Exemple

$(1110)_2 = 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0$

2.3.2 Conversion binaire décimal

En poursuivant les calculs du nombre $(1110)_2$ ci-dessus on obtiendra :
$(1110)_2 = 8+4+2+0$
$= (14)_{10}$

Le résultat du développement polynomial représente l'équivalent du nombre en décimal. C'est à dire l'équivalent du nombre binaire *1110* en base *10* est le nombre *14*.

Le même principe s'applique sur les nombres réels :
$(1110,101)_2 = 1*2^3+1*2^2+1*2^1+0*2^0+1*2^{-1}+0*2^{-2}+1*2^{-3}$
$= (14,625)_{10}$

Règle de conversion

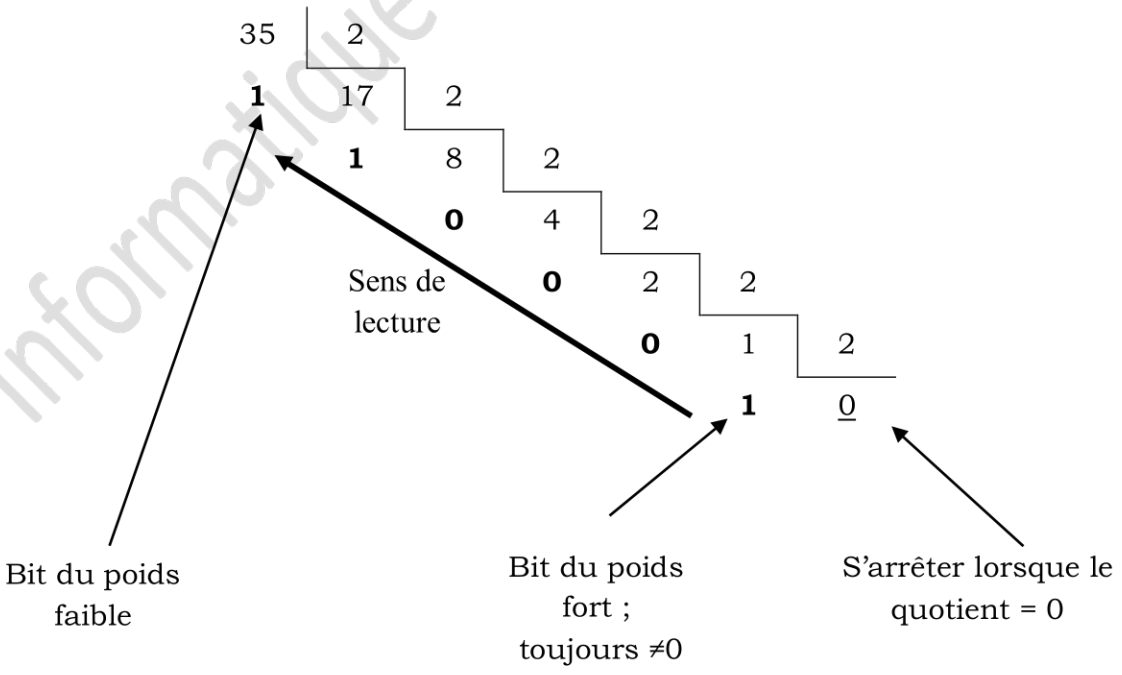
Pour convertir un nombre binaire au système décimal, il suffit d'utiliser son développement polynomial.
En poursuivant les calculs de la forme polynomiale d'un nombre binaire nous obtiendrons le nombre décimal équivalent.

2.3.3 Conversion décimal binaire

Le principe consiste à diviser successivement le nombre décimal par *2*, et prendre les restes des divisions (euclidiennes) dans l'ordre inverse.
Nous divisons de manière successive le nombre décimal par *2*, et nous ne s'arrêtions que lorsque le résultat de la division (le quotient) soit un zéro. Le nombre binaire équivalent sera la succession des restes obtenus ; où le bit de poids le plus fort correspondra au dernier reste et le bit de poids le plus faible correspondra au premier reste des divisions.

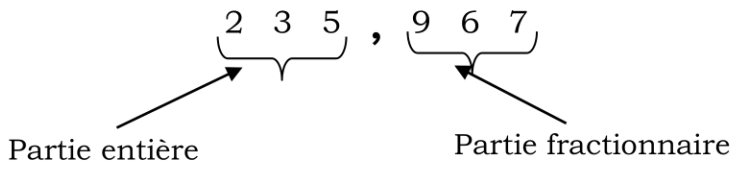
Exemple

$(35)_{10}=(100011)_2$



2.3.3.1 Conversion décimal binaire d'un nombre réel

Un nombre réel est constitué de deux parties : la partie entière et la partie fractionnaire (partie décimale).



La partie entière (PE) est convertie en effectuant des divisions successives par *2*, comme pour un nombre entier.
La partie fractionnaire (PF) est convertie en effectuant des multiplications successives par *2*. A chaque multiplication, la partie entière obtenue est prise en compte (elle correspond à un bit du nombre binaire). Et la partie fractionnaire résultante est utilisée pour la multiplication suivante.

Exemple

Soit le nombre $(35,625)_{10}$
PE : $(35)_{10}=(100011)_2$
PF : $(0,625)_{10}$

0,625*2 = **1**,25
0,25*2 = **0**,5
0,5*2 = **1**,0

Sens de lecture

On s'arrête puisque la partie fractionnaire égale *0*.
Donc $(0,625)_{10}=(0,101)_2$
Et $(35,625)_{10}=(100011,101)_2$

Remarques

- La précision est déterminée par le nombre de bits après la virgule.
- On doit arrêter les multiplications lorsque la partie fractionnaire égale *0*.
- On peut aussi arrêter les multiplications lorsque la partie fractionnaire a été déjà utilisée (partie fractionnaire cyclique).
- Sinon on s'arrête volontairement (pour les nombres irrationnels).

2.4 Système octal

La base de ce système est *8*.
Donc huit symboles sont utilisés dans le système de numération octal : { *0 , 1 , 2 , 3 , 4 , 5 , 6 , 7* }.

Exemple de la forme polynomiale d'un nombre octal

$(324)_8=3*8^2+2*8^1+4*8^0$

2.4.1 Conversion octal décimal

Pour convertir un nombre du système octal au système décimal, il suffit d'utiliser son développement polynomial.

Exemples

1. $(127)_8 = 1*8^2 + 2*8^1 + 7*8^0$
   $= 64 + 16 + 7$
   $= (87)_{10}$

2. $(104,65)_8 = 1*8^2 + 0*8^1 + 4*8^0 + 6*8^{-1} + 5*8^{-2}$
   $= 64 + 0 + 4 + 0,75 + 0,078125$
   $= (68,828125)_{10}$

2.4.2 Conversion décimal octal

On applique le même principe vu précédemment (conversion décimal binaire).
Pour convertir la partie entière d'un nombre décimal vers la base *8*, on divise successivement cette partie par *8* ; et ne s'arrête que lorsqu'on obtient un quotient nul. Finalement, on récupère les restes dans le sens inverse.
Pour convertir la partie fractionnaire, on multiplie successivement cette partie par *8*.

Exemple

$(24,26)_{10} = (\ ?\ )_8$

PE : $(24)_{10}$

24	8
0	3 | 8
	3 0

PF : $(0.26)_{10}$
0,26 * 8 = **2**.08
0.08 * 8 = **0**.64
0.64 * 8 = **5**.12
0.12 * 8 = **0**.96
0.96 * 8 = **7**.68
……..
On s'arrête et on obtiendra :
$(24,26)_{10} = (30,20507)_8$

2.5 Système hexadécimal

Dans le système hexadécimal (base *16*), nous utilisons seize symboles différents : les 10 premiers symboles sont les symboles du système décimal { *0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9* } plus les lettres { *A, B, C, D, E et F* }, car les symboles doivent être distincts. Les équivalents des symboles hexadécimal en décimal sont affichés dans le tableau suivant :

Décimal	Hexadécimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Tab. 2.1 : Correspondance hexadécimal décimal

Exemple de la forme polynomiale d'un nombre hexadécimal

$(95)_{16}=9*16^1+5*16^0$

2.5.1 Conversion hexadécimal décimal

Pour convertir un nombre hexadécimal au système décimal, il suffit d'utiliser son développement polynomial.

Exemples

1. $(17)_{16} = 1*16^1 + 7*16^0$
   $= 16 + 7$
   $= (23)_{10}$

2. $(ABE,2)_{16} = A*16^2 + B*16^1 + E*16^0 + 2*16^{-1}$
   $= 10*16^2 + 11*16^1 + 14*16^0 + 2*16^{-1}$
   $= 2560 + 176 + 14 + 0,125$
   $= (2750,125)_{10}$

2.5.2 Conversion décimal hexadécimal

Pour convertir un nombre décimal en hexadécimal. On effectue des divisions successives pour la partie entière et des multiplications successives pour la partie fractionnaire.

Exemple

$(31,25)_{10}$ = (?)$_{16}$

PE : $(31)_{10}$

31	16	
15	1	16
	1	0

PF : $(0.25)_{10}$

0,25 * 16 = **4**.0

Donc, $(31,25)_{10} = (1F,4)_8$

2.6 Système à base quelconque

Soit le nombre décimal suivant :

$1978,265 = 1*10^3 + 9*10^2 + 7*10^1 + 8*10^0 + 2*10^{-1} + 6*10^{-2} + 5*10^{-3}$

Ce nombre peut aussi être écrit comme suit :

$(1978,265)_B = 1*B^3 + 9*B^2 + 7*B^1 + 8*B^0 + 2*B^{-1} + 6*B^{-2} + 5*B^{-3}$

Avec B = *10* qui représente la base.

En générale, n'importe quel nombre N dans un système de numération à base B est exprimé comme suit :

$$N = (S_k\ S_{k-1}....S_1\ S_0)_B$$

Où:

B : base du système de numération.

S_i : symbole du système, tel que $\forall\ i \in [0..k]$, $S_i < B$

Donc, dans un système de numération à base *B*, on utilise *B* symboles distincts *{0, 1, …. B-2, B-1}* pour représenter les nombres.

La valeur de chaque symbole doit être strictement inférieure à la base *B*.

2.6.1 Forme polynomiale d'un nombre dans un système à base B

Chaque nombre dans un système à base *B* peut être développé sous sa forme polynomiale.

$$N = (S_k\ S_{k-1}....S_1\ S_0)_B = S_k*B^k + S_{k-1}*B^{k-1} + ...+S_1*B^1+ S_0*B^0$$

Exemple

$(534)_6 = 5*6^2 + 3*6^1 + 4*6^0$

2.6.2 Conversion d'un système à base quelconque au système décimal

De manière générale, pour convertir un nombre d'un système à base quelconque (base *B*) au système décimal (base *10*), il suffit d'utiliser son développement polynomial.

Exemple

$(341)_6 = 3*6^2 + 4*6^1 + 1*6^0$
$= 108 + 24 + 1$
$= (133)_{10}$

2.6.3 Conversion du décimal à un système à base quelconque

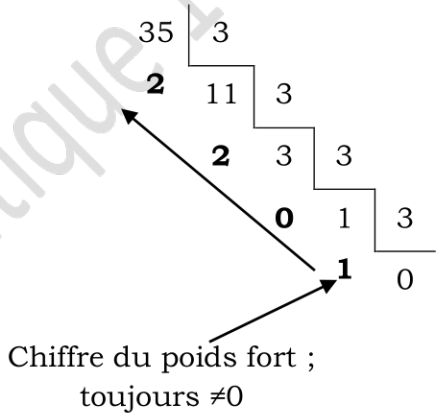
Le même principe est appliqué :
Pour convertir la partie entière d'un nombre décimal vers un système à base *B* :

- On effectue des divisions successives de la partie entière par la base *B* ;
- On n'arrête les divisions que lorsqu'on obtient un quotient nul ;
- Ensuite on récupère les restes dans le sens inverse.

Il est à noter que le dernier reste (chiffre du poids fort) doit toujours être différèrent de zéro.
De même, pour convertir la partie fractionnaire d'un nombre décimal vers un système à base *B*, on effectue des multiplications successives de cette partie par la base *B*.

Exemple

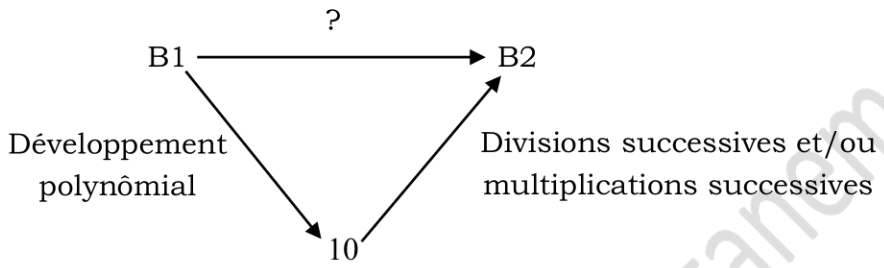
$(35,125)_{10} = (\ ?\ )_3$
PE : $(35)_{10}$



PF : $(0.125)_{10}$
0,125 * 3 = **0**.375
0.375 * 3 = **1**.125 ← partie fractionnaire déjà utilisée
On peut s'arrêter puisque la partie fractionnaire **0,125** a été déjà utilisée, et on obtiendra $(35,125)_{10} = (1022,01)_3$
Sinon on obtiendra $(35,125)_{10} = (1022,01\ 01\ 01\ .....)_3$

2.7 Conversion d'une base B1 vers une base B2

Il n'existe pas de méthode pour convertir directement un nombre de la base *B1* vers la base *B2*.
Pour effectuer la conversion nous devrions passer par la base *10*.
L'idée est de convertir le nombre de la base *B1* vers la base *10*, ensuite convertir le nombre décimal obtenue vers la base *B2* .



Exemple

$(34)_5 = (\ ?\ )_7$
On commence par le développement polynomial
$(34)_5 = 3*5^1 + 4*5^0$
$=(19)_{10}$

Ensuite on divise successivement *19* par *7*

19	7	
5	2	7
	2	0

$(19)_{10} = (25)_7$
Donc $(34)_5=(25)_7$

2.8 Conversion octal binaire et vice-versa

Nous avons deux possibilités pour faire une conversion de la base *8* à la base *2* et vice-versa.
La première consiste de passer par la base *10* vue précédemment.
La deuxième possibilité nous permet de passer directement de la base *8* à la base 2 et inversement, sans passer par la base *10*.
Il est à remarquer que $8=2^3$: Cela veut dire que chaque symbole de la base *8* est représenté par *3* symboles (bits) de la base *2*.

2.8.1 Conversion octal binaire

Pour convertir un nombre octal en binaire, il suffit de remplacer chaque symbole du nombre octal par sa valeur correspondante en binaire sur 3 bits (faire des éclatements sur 3 bits) selon le tableau suivant :

Octal	Binaire		
	2^2	2^1	2^0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Tab.2.2 : correspondance octal binaire.

Le remplissage du tableau Tab.2.2 ci-dessus est simple :
On commence par le remplissage des symboles de la base *8* {*0, 1, 2 , 3 , 4 , 5 , 6 , 7*}.
Ensuite on remplit les *3* colonnes (qui représentent les *3* bits) de la base *2*.
La valeur $2^0=1$ de la 1ere colonne (de droite), signifie que chaque symbole binaire doit être affiché alternativement une fois (*0* affiché une fois, *1* affiché une fois, *0* affiché une fois, *1* affiché une fois, …).
La valeur $2^1=2$ de la 2eme colonne, signifie que chaque symbole binaire doit être affiché alternativement deux fois (*0* affiché deux fois, *1* affiché deux fois, *0* affiché deux fois, *1* affiché deux fois, …).
La valeur $2^2=4$ de la 3eme colonne (de gauche), signifie que chaque symbole binaire doit être affiché alternativement quatre fois (*0* affiché quatre fois, *1* affiché quatre fois).

Exemples

1. $(345)_8=(\ ?\ )_2$

On remplace chaque symbole octal par les *3* symboles binaires correspondants.

$$(\ 3 \quad 4 \quad 5\ )_8$$
$$\downarrow \quad \downarrow \quad \downarrow$$
$$(011 \quad 100 \quad 101\ )_2$$

Donc $(345)_8 = (\ 011100101\ )_2$

2. $(65,76)_8 = (\underline{110}\ \underline{101}, \underline{111}\ \underline{110})_2$

3. $(35,34)_8 = (\underline{011}\ \underline{101}, \underline{011}\ \underline{100})_2$

2.8.2 Conversion binaire octal

L'idée de base est de faire des groupes de 3 bits.
Le regroupement s'effectue de droite à gauche pour la partie entière. Si le dernier groupe le plus à gauche ne contient pas *3* bits, il sera complété par des *0*.

Le regroupement s'effectue de gauche à droite pour la partie fractionnaire. Si le dernier groupe le plus à droite ne contient pas *3* bits, il sera complété par des *0*.
Ensuite en utilisant le tableau Tab.2.2, on remplace chaque groupe (de *3* bits) par le symbole octal correspondant.

Exemples :

1. $(1100101001011 0)_2 = (\mathbf{0}11\ 001\ 010\ 010\ 110)_2 = (31226)_8$
2. $(110010100,10101)_2 = (110\ 010\ 100\ ,\ 101\ 01\mathbf{0})_2 = (624,52)_8$

2.9 Conversion hexadécimal binaire et vice-versa

Pour ces conversions, nous pourrons passer directement de l'hexadécimal vers le binaire sans transiter par le décimal.
Puisque $16=2^4$, nous pourrons représenter chaque symbole hexadécimal par *4* symboles (bits) binaires.

2.9.1 Conversion hexadécimal binaire

Pour convertir un nombre hexadécimal en binaire, il suffit de remplacer chaque symbole du nombre hexadécimal par sa valeur correspondante en binaire sur 4 bits selon le tableau suivant :

Hexadécimal	**Binaire**			
	2^3	2^2	2^1	2^0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1
E	1	1	1	0
F	1	1	1	1

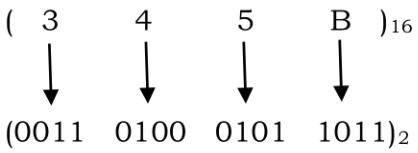
Tab.2.3 : correspondance hexadécimal binaire.

Pour remplir le tableau Tab.2.3 ci-dessus, on commence par le remplissage des symboles de la base *16* { *0* , *1* , *2* , *3* , *4* , *5* , *6* , *7* , *8* , *9* , *A* , *B* , *C* , *D* , *E* , *F* }. Ensuite on remplit les *4* colonnes (qui représentent les *4* bits) de la base *2*.
Pour remplir les *3* premières colonnes, on applique le même principe que celui décrit précédemment (section 2.8.1).
Pour remplir la 4eme colonne : la valeur $2^3=8$ signifie que chaque symbole binaire doit être affiché alternativement huit fois (*0* affiché huit fois, *1* affiché huit fois).

Exemples

1. $(345B)_{16}=(\ ?\ )_2$

On remplace chaque symbole hexadécimal par les *4* symboles binaires correspondants.



Donc $(345B)_{16}=(0011010001011011)_2$

2. $(AB3,4F6)_{16}=(\underline{1010}\ \underline{1011}\ \underline{0011},\ \underline{0100}\ \underline{1111}\ \underline{0110})_2$

2.9.2 Conversion binaire hexadécimal

Pour convertir un nombre binaire en hexadécimal, on forme des groupes de *4* bits.
Le regroupement s'effectue de droite à gauche pour la partie entière et de gauche à droite pour la partie fractionnaire. Les derniers groupes de droite et de gauche seront complétés par des *0* s'ils ne contiennent pas *4* bits.
En utilisant le tableau Tab.2.3, on remplace chaque groupe (de *4* bits) par le symbole hexadécimal correspondant.

Exemples

1. $(11001010100110)_2 = (\ \underline{\mathbf{00}11}\ \ \underline{0010}\ \ \underline{1010}\ \ \underline{0110}\ )_2 = (32A6)_{16}$
2. $(110010100,10101)_2 = (\ \underline{\mathbf{000}1}\ \ \underline{1001}\ \ \underline{0100}\ ,\ \underline{1010}\ \ \underline{1\mathbf{000}}\ )_2 = (194,A8)_{16}$

2.10 Conversion directe d'une base X vers une base Y et vice-versa

En règle générale, nous pourrons toujours effectuer des conversions directes de la base *X* vers la base *Y* et vice-versa, sans passer par la base *10*, si la base *X* est une puissance de la base *Y*.
Si $X = Y^Z$, alors on pourra toujours remplacer chaque symbole de la base *X* par *Z* symboles de la base *Y*.

2.10.1 Conversion de la base 2 vers la base 4 et vice-versa

Base 4	Base 2	
	2^1	2^0
0	0	0
1	0	1
2	1	0
3	1	1

$(12,03)_4 = (\underline{01}\ \underline{10}, \underline{00}\ \underline{11})_2$

2.10.2 Conversion de la base 3 vers la base 9 et vice-versa

Base 9	Base 3	
	3^1	3^0
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2
6	2	0
7	2	1
8	2	2

$(54,82)_9 = (\underline{12}\ \ \underline{11}\ ,\ \underline{22}\ \underline{02})_3$

2.10.3 Conversion de la base 4 vers la base 16 et vice-versa

Base 16	Base 4	
	4^1	4^0
0	0	0
1	0	1
2	0	2
3	0	3
4	1	0
5	1	1
6	1	2
7	1	3
8	2	0
9	2	1
A	2	2
B	2	3
C	3	0
D	3	1
E	3	2
F	3	3

$(AE,61)_{16} = (\underline{22}\ \ \underline{32}\ ,\ \underline{12}\ \underline{01})_4$

3 Chapitre 3 : Introduction aux algorithmes

3.1 Introduction

Le terme algorithme vient du nom du mathématicien « *Al Khawarismi* », dont le traité d'algèbre décrit des méthodes de calcul à suivre pour résoudre des problèmes qui se ramènent souvent à la résolution d'équations.
Par exemple, pour résoudre l'équation $8x + 3 = 4x - 5$;
Il faut suivre les instructions ci-contre :

- soustraire $4x$ des deux membres,
- soustraire 3 des deux membres,
- diviser par 4 les deux membres,
- afficher la solution.

3.2 L'algorithme

L'écriture d'un algorithme est essentielle pour la résolution d'un problème. Elle permet de décrire la démarche à suivre pour atteindre les résultats souhaités à partir d'un ensemble de données.

Un algorithme est une suite d'instructions élémentaires qui s'appliquent dans un ordre déterminé à des données (en entrée) et qui fournissent en un nombre fini d'étapes des résultats (en sortie).

Un algorithme correspond à ces trois étapes de fonctionnement :
1- Introduction des données nécessaires (les entrées).
2- Exécution d'instructions élémentaires et ordonnées sur ces données (traitements).
3- Edition des résultats obtenus (les sorties).

Exemple

Pour préparer un café, il faut :
1. Faire chauffer de l'eau
2. Mettre le café
3. Remuer
4. Servir

Ces 4 étapes ne sont pas assez détaillées. Il serait préférable de les mieux décrire, en divisant chaque instruction en une suite d'instructions plus compréhensibles (élémentaires) :

1. Faire chauffer de l'eau
    1.1 prendre une casserole
    1.2 mettre de l'eau dans la casserole
        1.2.1 mettre la casserole sous le robinet
        1.2.2 ouvrir le robinet
        1.2.3 remplir la casserole
        1.2.4 fermer le robinet
    1.3 déposer la casserole sur un feu de la cuisinière
    1.4 allumer un feu de la cuisinière
    1.5 attendre l'ébullition
    1.6 éteindre le feu de la cuisinière
2. Mettre le café
3. Remuer
4. Servir

Division de l'instruction en une suite d'instructions élémentaires.

3.3 Les Langages

La description des instructions ainsi que les objets manipulés par ces instructions, s'effectue via un langage.
Le langage utilisé pour formuler les instructions doit être compréhensible par l'exécuteur.
Pour un humain, il suffit d'utiliser le langage courant (la langue maternelle), mais pour une machine, il faut passer par un langage qui lui est compréhensible.

3.3.1 Le langage de programmation

Un langage de programmation est un langage évolué constitué d'un ensemble de commandes et de mots clés nécessaires pour l'écriture d'instructions compréhensibles par la machine.
Exemple : Pascal, C, Java, Basic, Fortran…

3.3.2 Le programme

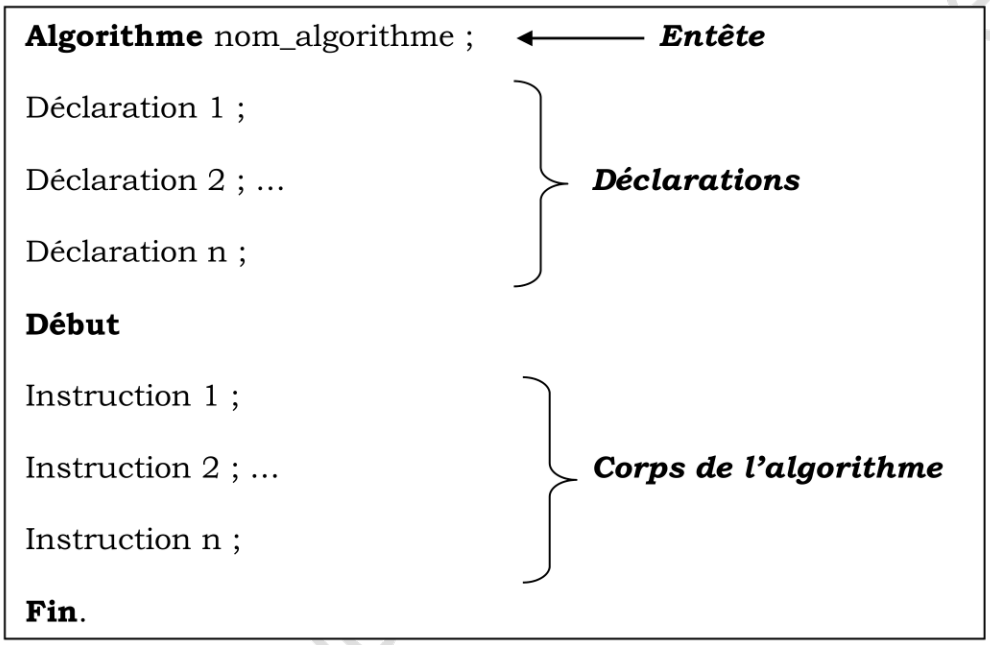
Un programme est la traduction d'un algorithme dans un langage de programmation (c'est-à-dire la traduction dans un langage compréhensible par la machine).

3.3.3 Le langage algorithmique

Le langage algorithmique est un pseudo-langage qui prend en considération les caractéristiques de la machine. Il reste plus souple qu'un langage de programmation. En effet, le langage algorithmique est un mélange (arrangement) entre le langage courant (maternelle) et un langage de programmation (le langage de programmation **Pascal** est jugé comme étant le langage le plus approprié).

3.4 La structure d'un algorithme

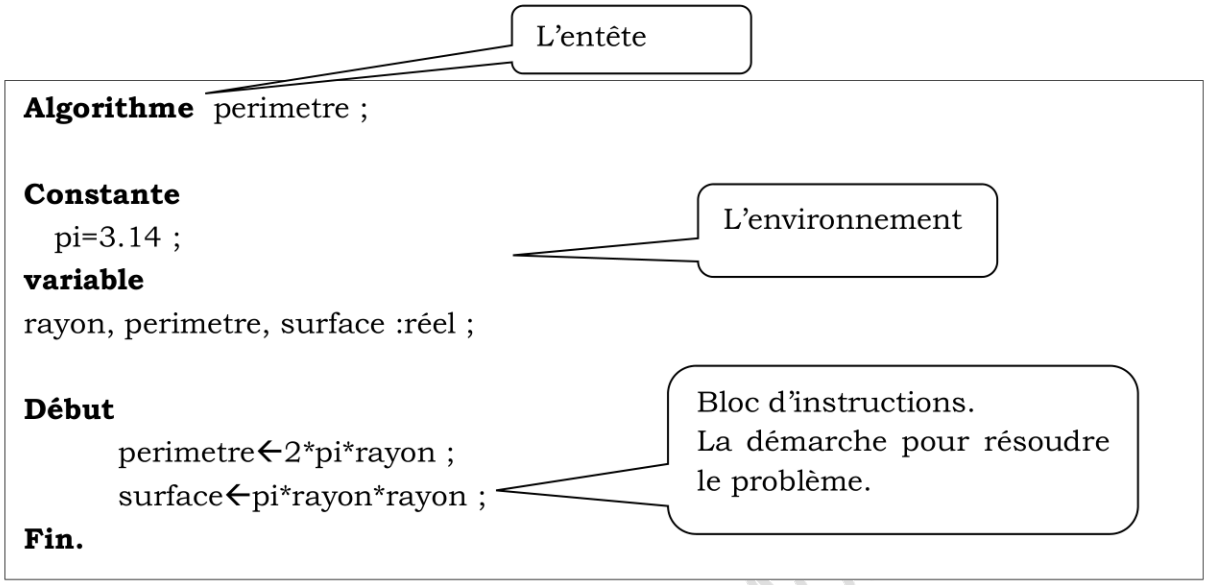
Un algorithme est composé de trois parties : l'entête, les déclarations (l'environnement) et le corps de l'algorithme (bloc d'instructions).



- L'entête : sert à identifier la tâche à résoudre (identification de l'algorithme). L'entête est défini par le mot clé **algorithme** suivi par un nom de la tâche considérée.
- La partie déclarative : est une liste de tous les objets utilisés dans le corps de l'algorithme (l'environnement de résolution du problème).
  Pour définir les objets de l'environnement, les langages de programmation utilisent des instructions déclaratives.
- Le corps de l'algorithme : décrit la méthode de résolution du problème (bloc d'instructions).
  Le corps de l'algorithme est délimité par les mots **Début** et **Fin** suivi par un "**.**".

Exemple

Le calcul du périmètre et de la surface d'un cercle dans le langage algorithmique est comme suit :



3.5 La déclaration d'un objet

Lorsqu'on déclare un objet dans son environnement, on le définit par certaines de ces caractéristiques en fonction de sa nature.
Selon la nature, un objet est soit une constante soit une variable.
Une variable est caractérisée par son nom et son type. Sa valeur, appelée à varier, ne la caractérise pas.
Tandis qu'une constante est caractérisée par un nom et sa valeur qui ne peut en aucun cas être modifiée.

3.5.1 Le nom d'un objet

Le nom (identificateur) d'un objet doit respecter une syntaxe particulière :

- Il est constitué d'une suite de caractères ne contenant que :
  - des lettres alphabétiques : 'a'..'z' et 'A'.. 'Z'.
  - des chiffres : 0..9.
  - et le caractère de soulignement : "_".
- Il doit commencer par une lettre.
- Il ne contient pas des indices ou des exposants.
- Il doit être différent de tous les mots clés.

Exemples

- *rayon, valeur_absolue, x1* : sont des noms corrects.
- *1x* : n'est pas un nom correct car il ne commence pas par une lettre.
- *Racine carrée* : n'est pas un nom correct car il contient l'espace et le "é".
- x_1 : n'est pas un nom correct car il est constitué d'un indice.
- *Begin* : n'est par un nom correct car c'est un mot clé.

Pour des raisons de lisibilité, il serait préférable d'attribuer aux objets des noms significatifs.

3.5.2 Déclaration des variables

Dans l'écriture d'un algorithme, on déclare une variable en définissant son nom et son type (numériques, alphanumérique, ...).
Cette étape est appelée déclaration des variables.
Pour déclarer une variable dans un langage algorithmique, on utilise le mot clé **variable** selon la syntaxe suivante :

Variable
Nom_variable : type_de_donnée ;

Où :

Nom_variable : est le nom (l'identificateur) attribué à la variable ;
type_de_donnée : est le genre d'information auquel appartient cette variable (type de cette variable).

Exemple

```
variable
  rayon :réel ;
```

Pour déclarer une variable dans un programme en pascal, on utilise le mot clé **var** selon la syntaxe suivante :

Var
Nom_variable : type_de_donnée ;

3.5.3 Déclaration d'une constante

On déclare une constante en définissant son nom et sa valeur.
Pour déclarer une constante dans un algorithme, on utilise le mot clé **constante** selon la syntaxe suivante :

Constante
Nom_constante = valeur ;

Où :

Nom_ constante **:** est le nom (l'identificateur) attribué à la constante ;
valeur **:** est la valeur de cette constante.

Exemple

```
Constante
  pi=3.14 ;
```

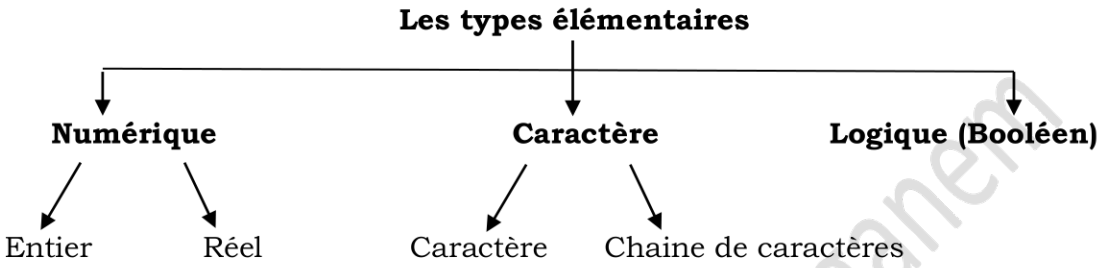
Pour déclarer une constante dans un programme en pascal, on utilise le mot clé **const** selon la syntaxe suivante :

Const
Nom_constante = valeur ;

3.6 Les types de données élémentaires

Un type de données est élémentaire, s'il définit un ensemble de valeurs simple.
Les types prédéfinis disponibles dans les langages de programmation sont :

- Les types numériques (entiers et réels),
- Le type des caractères (caractères, chaines de caractères)
- Le type logique (booléen).



3.6.1 Le type entier

Il définit un domaine de valeur limité, contrairement à l'ensemble des entiers en mathématiques qui est infini. Le domaine de valeurs du type entier est un intervalle fermé bornés ([-32 768 .. 32 767] en occupant 2 octets).
Les opérateurs associés au type entier sont (opérateurs usuels) :

- L'addition (+).
- La soustraction (-)
- La multiplication (*)
- La division euclidienne (entière) notée par **div.**
- L'opérateur modulo (reste de la division entière) noté par **mod**.

Exemples

27 div 5 = 5 ; 27 mod 5= 2 ; 9 mod 3 = 0.

3.6.2 Le type réel

Le type réel prend ses valeurs dans un sous ensemble des réels (en mathématiques).
Il est muni des opérateurs suivants (opérateurs usuels) :

- L'addition (+).
- La soustraction (-)
- La multiplication (*)
- La division réelle (/)

Ainsi que d'autres fonctions mathématiques prédéfinies, comme par exemple :

- Sqrt() : est une fonction qui fournit la racine carré d'un nombre.
- Abs() : est une fonction qui fournit la valeur absolue d'un nombre.

Exemples

Sqrt(16) =4 ; Abs(-8) =8 .

3.6.3 Le type caractère

- Le type caractère est l'ensemble de tous les caractères imprimables (touches du clavier).
  Il comporte :
  - les lettres 'a'..'z' et 'A'.. 'Z'.,
  - les chiffres 0..9,
  - les signes de ponctuations (., ; , ?, !, …),
  - les symboles utilisés en tant qu'opérateurs (+, *, -, /, <, =, …),
  - les caractères spéciaux (%, @, $, &, …)
- Un caractère doit être encadré par 2 apostrophes.
- L'ensemble des caractères est ordonné.
- Parmi les fonctions prédéfinies pour la manipulation des caractères, on cite :
  - Succ() : une fonction qui fournit le successeur immédiat d'un caractère.
  - Pred() : une fonction qui fournit le prédécesseur immédiat d'un caractère.

Exemples

Succ('f')= 'g' ; Pred('f')='e' .

3.6.4 Le type chaine de caractères

- Le type chaine de caractères est une suite de caractères.
- Une chaine est encadrée par 2 apostrophes.
- Toute apostrophe figurant dans une chaine doit être doublée.
- Plusieurs fonctions prédéfinies sont dédiées à la manipulation des chaines de caractères, comme par exemple :
  - Length() : une fonction qui fournit la longueur d'une chaîne.
    La longueur d'une chaîne peut être *0* (chaine vide).
  - Concat() : une fonction qui permet de concaténer deux chaînes de caractères.

Exemples

Length (Module)=6 ; Concat('Module', ' algorithmique') = 'Module algorithmique' .

3.6.5 Le type logique (booléen)

Les deux valeurs de ce type sont **vrai** et **faux.**
Les opérateurs logiques les plus utilisés sont :
- La conjonction (et)
- La disjonction (ou)
- La négation (non)

Les opérateurs *ET*, *OU* et *NON* sont définis par la table de vérité ci-dessous :

A	B	A ET B	A OU B	NON A
Vrai	Vrai	Vrai	Vrai	Faux
Faux	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Vrai	
Faux	Faux	Faux	Faux	

Tab.3.1 : Table de vérité

Remarque

Le résultat obtenu d'une comparaison (en utilisant les opérateurs relationnels : <, ≤, =, ≠, >, ≥) est une valeur booléenne.

3.7 Les instructions élémentaires

Une instruction élémentaire est une instruction exprimée par un ordre simple.
Toute instruction est définie par :

- Une syntaxe qui précise le formalisme d'écriture adopté pour formuler l'ordre.
- Une sémantique qui définit les actions effectuées par la machine en réponse à cet ordre.

Nous allons fréquemment utiliser des expressions pour formuler les instructions.
Une **expression** est une suite d'opérations effectuées sur des opérandes (variables, constantes, valeurs …). Chaque expression a une valeur et un type.

3.7.1 L'affectation

Une variable peut être assimilée à une boîte, repérée par son nom et peut contenir une information.
Pour utiliser le contenu de cette boîte, il suffit de l'appeler par son nom.
L'affectation permet de stocker la valeur d'une expression dans une variable.

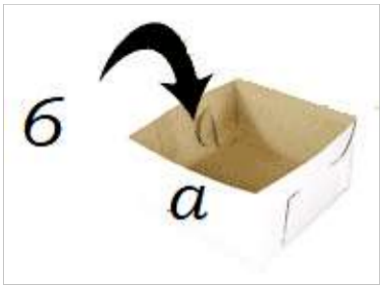


Fig.3.1 : Affectation du nombre *6* à la variable *a*

Syntaxe

En langage algorithmique, l'instruction d'affectation est donnée par la syntaxe suivante :

$X \leftarrow E$;

Où :

X : le nom de variable.
←: le symbole de l'affectation.
E : une expression.

On dit que la variable *X* reçoit la valeur de l'expression *E*.

En langage Pascal, l'instruction d'affectation est symbolisée par « **:=** » selon la syntaxe suivante :

```
X :=E ;
```

Sémantique

L'exécution de l'affectation s'effectue en deux étapes :

1- L'expression *E* est évaluée, soit *v* la valeur obtenue.
2- La valeur *v* est stockée dans la variable *X*.

L'affectation n'est valide que si le type de l'expression *E* est compatible avec le type de la variable *X*.

Exemples

1. Soit *A* une variable de type entier dont l'état à un instant donné est schématisé par :

A	32

L'exécution de l'opération A←5 a pour effet :

A	5

La valeur *5* est stockée dans *A*.
La valeur *5* écrase la valeur *32* qui est alors définitivement perdue.

2. Soit *B* une variable de type entier,

B	9

L'exécution de l'opération B←A a pour effet :

B	5

La valeur *5* de *A* est affectée à *B*.
On remarque que cette affectation est sans effet sur la variable *A* (la valeur de *A* est toujours *5*).

3. Soit *X* une variable de type réel, l'instruction X←A est valide.
Elle respecte l'inclusion de l'ensemble des entiers dans l'ensemble des réels.
La valeur *5* est convertie en réel avant d'être affectée à la variable *X*.
Par contre l'affectation A←X provoque une erreur d'incompatibilité de type.

4. L'exécution de l'opération B←(B-2)*A a pour effet :

B	15

La valeur initiale de la variable *B* (*5*) est utilisée dans le calcul de l'expression. La valeur obtenue est ensuite affectée à cette même variable (*B*).

3.7.1.1 Incrémentation et décrémentation

Soit la variable *A* de type entier.
L'instruction A ← A+1 est appelée opération d'incrémentation.
On dit qu'on incrémente *A*.
L'opération inverse de l'incrémentation est la décrémentation (A←A-1).
On dit qu'on décrémente *A*.

3.7.2 L'instruction de lecture

L'instruction de lecture permet à l'utilisateur de transmettre des données à la machine (au processus).

Syntaxe

En langage algorithmique, l'instruction de lecture est donnée par la syntaxe suivante :

Lire (X) ;

Où :
X : nom de variable.

On peut regrouper plusieurs ordres de lecture en un seul :
L'instruction lire (A, B, C) est équivalente à lire(A) ; lire(B) ; lire(C).

En langage Pascal, l'instruction de lecture est donnée par la syntaxe suivante :

Read(X) ;

Sémantique

La valeur transmise par l'utilisateur, par l'intermédiaire d'un dispositif d'entrée (clavier) est affectée à la variable *X*.
A chaque fois que la machine (le processus) reçoit un ordre de lecture *Lire(X)*, l'exécution du programme est suspendue en attendant la saisie d'une valeur. L'utilisateur doit alors saisir une valeur via le clavier. Une fois la saisie validée (par la touche entrée ↵), l'exécution du programme se poursuit et la valeur transmise par l'utilisateur est affectée à la variable *X*.

Si la valeur saisie est incompatible avec le type de la variable, la lecture provoquera une erreur.

3.7.3 L'instruction d'écriture

L'instruction d'écriture permet au processus (machine) de communiquer des informations (des messages ou des résultats de traitement) à l'utilisateur (affichage sur écran ou sur imprimante…).

Syntaxe

En langage algorithmique, l'instruction d'écriture est donnée par la syntaxe suivante :

Ecrire(E) ;

Où :
E : Expression.

On peut regrouper plusieurs instructions d'écriture en une seule :
L'instruction écrire(E1, E2, E3) est équivalente à écrire(E1) ; écrire(E2) ; écrire(E3).

En langage Pascal, l'instruction d'écriture est donnée par la syntaxe suivante :

| *Write(E) ;*

Sémantique

L'exécution de l'instruction *écrire(E)* s'effectue en 2 étapes :

1- L'expression *E* est évaluée.
2- Sa valeur est communiquée à l'utilisateur par l'intermédiaire d'un dispositif de sortie (écran ou imprimante).

Exemples

1. *Ecrire (2*x-3)* a pour effet d'afficher la valeur de l'expression 2*x-3 ; Si par exemple *x* vaut 10, cette instruction a pour effet d'afficher la valeur **17**.
2. Ecrire ('Donner un entier supérieur à 10') a pour effet d'afficher sur écran la constante chaine : **Donner un entier supérieur à 10**

Remarque

L'affichage des messages (écriture) est très utile en programmation.
Il permet d'orienter l'utilisateur, suite à un ordre de lecture, en lui indiquant ce que la machine attend de lui. Et il permet aussi d'expliquer les résultats obtenus des traitements.

3.8 Les commentaires

Pour rendre un algorithme (ou un programme) plus lisible et plus compréhensible, on doit le commenté.
Les commentaires sont des textes courts insérés dans un algorithme pour expliquer certaines parties de son corps, et ils n'ont aucun effet sur l'exécution de l'algorithme.

Par convention, les commentaires s'écrivent comme suit :

1. Commentaire sur une seule ligne : qui débutent par //.

   Exemple

   // Ceci est un commentaire sur une seule ligne.

2. Commentaire sur plusieurs lignes : qui est encadré par { et }, par **(*** et ***)** ou par **/*** et ***/**.

   Exemple

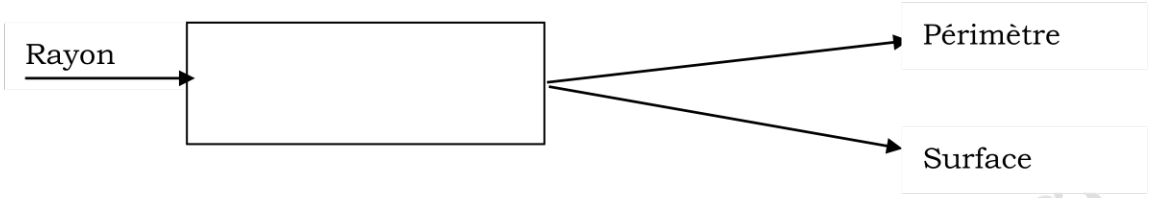
   { Ceci est un commentaire
   sur plusieurs lignes }.

3.9 Exercice d'application

Ecrire l'algorithme ainsi que le programme équivalent en langage Pascal qui permettent de calculer et d'afficher le périmètre et la surface d'un cercle dont le rayon est saisi par l'utilisateur.

Solution

La solution de cet exercice nécessite une seule valeur en entrée (donnée) qui est le rayon et deux valeurs en sorties (résultats) qui sont le périmètre et la surface d'un cercle




```
Algorithme  perimetre_surface_cercle ;
Constante
pi=3.14 ;
variable
r, p, s :reel ;
Début
Ecrire ('donnez la valeur du  rayon du cercle') ;
Lire (r) ;
p←2*pi*r ;
s←pi*r*r;
Ecrire('le périmètre du cercle est :', p) ;
Ecrire('la surface du cercle est :', s) ;
Fin.
```

Déclaration de pi (π), r(rayon), p(périmètre) et s(surface)

La machine demande à l'utilisateur de saisir la valeur du rayon.

La machine lit la valeur du rayon.

Calcul des valeurs du périmètre et de la surface

Affichage des valeurs du périmètre et de la surface

Le programme suivant est la traduction de l'algorithme précédent en langage Pascal.

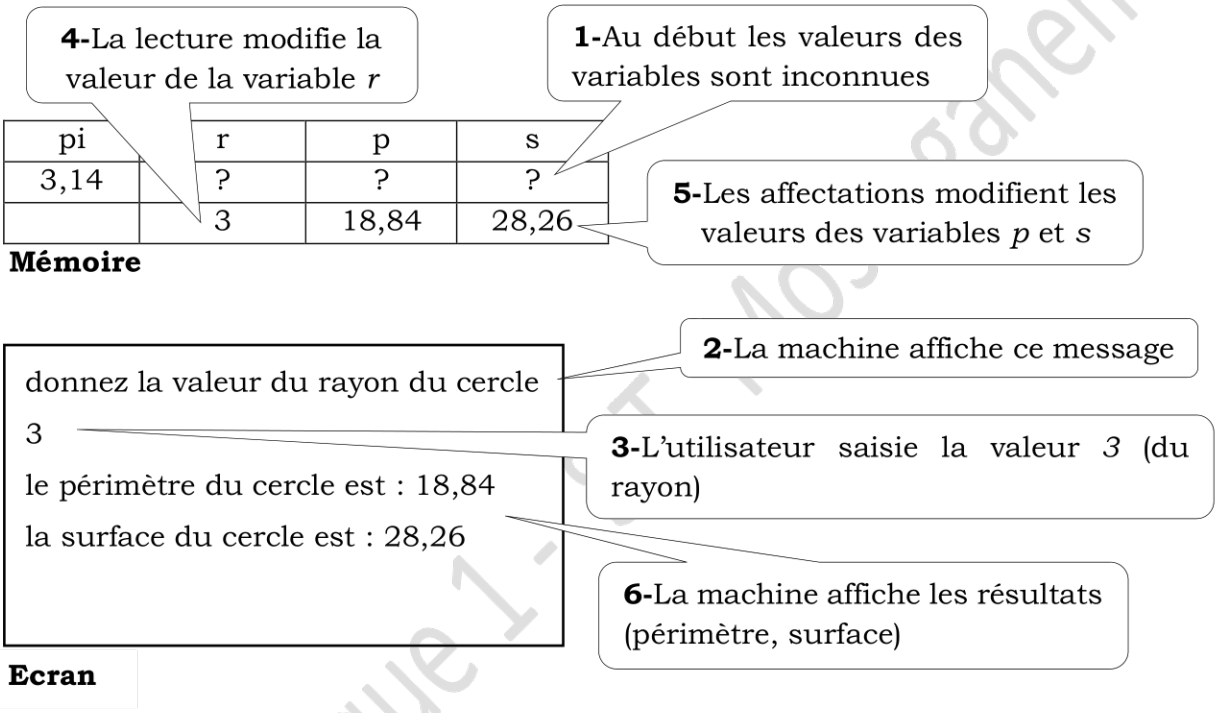
```
Program  perimetre_surface_cercle ;
Const
pi=3.14 ;
Var
r, p, s :real ;   { r: rayon,   p : périmètre,   s : surface }
Begin
Write ('donnez la valeur du rayon du cercle') ;
Read (r) ;
p :=2*pi*r ;
s :=pi*r*r;
Write('le périmètre du cercle est :', p) ;
Write('la surface du cercle est :', s) ;
End.
```

Ceci est un commentaire

3.9.1 Le déroulement (éxecution) du programme

L'exécution d'un algorithme (programme) s'effectue instruction par instruction (à partir du mot clé **Début** jusqu'au mot clé **Fin.**).
Le déroulement illustre l'effet de l'exécution d'un algorithme (programme) sur l'état de la machine, et plus particulièrement sur les états de l'écran et de la mémoire.
L'état de l'écran ne peut être modifié que par l'exécution des instructions de lectures et d'écritures. La modification de la valeur d'une variable modifie l'état de la mémoire.

L'exécution du programme précèdent est schématisé par les états suivants :



1- Au début les valeurs des variables sont inconnues. Seule la valeur de la constante *pi* est connue. Elle est initialisé par *3.14.*
2- La machine affiche ce message « donnez la valeur du rayon du cercle ».
3- L'utilisateur saisie la valeur *3* (du rayon).
4- La lecture modifie la valeur de la variable *r :* La machine lit la valeur *3* et la stocke dans la variable *r.*
5- Les affectations modifient les valeurs des variables *p* et *s :* La machine calcule les valeurs de *s* et *p.*
6- La machine affiche les résultats de *p* et *s* (périmètre, surface).

4

Chapitre 4 : Structures de contrôle conditionnelles

4.1 Introduction

Nous avons vu jusqu'à présent des algorithmes avec des instructions qui s'enchaînent de façon séquentielles (structures séquentielles) ; c'est-à-dire les instructions vont s'exécuter, du début jusqu'à la fin, les unes après les autres.
Toutefois, cela n'est pas toujours satisfaisant. Parfois, il faut décider de l'exécution d'une instruction ou d'un bloc d'instructions (est-ce qu'un bloc d'instructions doit être exécuté ou pas). Pour cela, on utilise les structures de contrôle conditionnelles, qui offrent la possibilité de sélectionner pendant l'exécution le bloc d'instructions à exécuter sur la base d'une condition.
Il existe 3 structures de contrôle conditionnelles :

- Structure conditionnelle simple
- Structure conditionnelle alternative
- Structure conditionnelle à choix multiples.

Les deux premières structures seront présentées dans ce chapitre, la troisième structure sera présentée dans le prochain chapitre.

4.2 La structure conditionnelle simple

La structure conditionnelle simple nous permet, sur la base d'une condition, de choisir entre l'exécution ou l'ignorance d'un bloc d'instructions.
Parfois, il faut exécuter une ou plusieurs instructions uniquement si une condition est vraie et ne rien faire si la condition est fausse.

Syntaxe

En algorithme, la syntaxe de la structure simple est comme suit :

Si *condition* **alors**
<*Bloc d'instructions*> ;
Finsi ;

Notons qu'un bloc d'instructions peut contenir une ou plusieurs instructions.

Sémantique

L'exécution de la structure s'effectue en 2 étapes :

1. Evaluer la condition.
2. Si elle est vérifiée (vraie), le *Bloc d'instructions* est exécuté ;
   Et si elle n'est pas vérifiée (fausse), le *Bloc d'instructions* est ignoré (non exécuté).

En langage pascal, on utilise l'une de ces deux syntaxes :

1ere syntaxe

```
if   condition then
     Instruction1;
```

2eme syntaxe

```
if   condition  then
begin
  Instruction1 ;
  Instruction2 ;
end;
```

Le choix de la syntaxe dépend du nombre d'instructions par bloc (une ou plusieurs) :

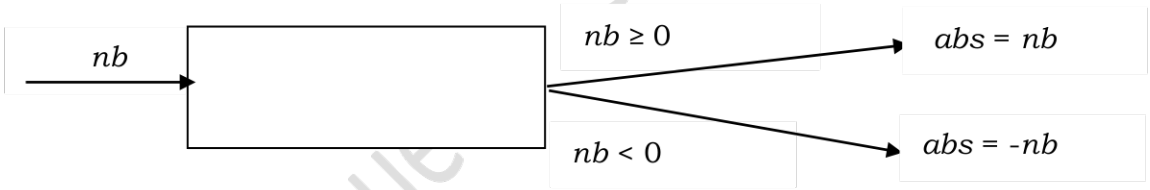
1. Si le bloc contient une seule instruction alors on utilise la première syntaxe ;
2. Et si le bloc contient plusieurs instructions (2 instructions ou plus) alors on utilise la deuxième syntaxe.

Si plusieurs instructions (2 ou plus) dépendent de la condition alors il faut utiliser **begin** et **end** (délimiter les instructions par **begin** et **end**). Cependant si une seule instruction dépend de la condition alors on n'utilise pas **begin** et **end**.

Exemple

Ecrire un algorithme qui affiche la valeur absolue d'un nombre entier.

Soit *nb* le nombre à introduire par l'utilisateur (via clavier).
Et *abs* la valeur absolue à afficher comme résultat.



L'algorithme de cet exemple est comme suit :

```
Algorithme valeur_absolue ;
Variable
nb, abs : entier ;
Début
Ecrire ('donnez un nombre entier ') ;
Lire (nb) ;
Abs←nb ;
Si nb<0 alors
   Abs←-nb ;
Finsi
Ecrire ('la valeur absolue du nombre est :', abs) ;
Fin.
```

Dans cet algorithme, on suppose que la valeur absolue est égale au nombre *nb* ; et on ne modifie la valeur absolue que si le nombre *nb* est négatif.
Donc, il n'y a aucun traitement à faire si la condition est fausse (*nb*>0).

La traduction de cet algorithme en langage pascal est comme suit :

```
program valeur_absolue ;
Var
nb, abs : integer ;
begin
write ('donnez un nombre entier ') ;
read (nb) ;
Abs :=nb ;
if nb<0 then
   Abs := -nb ;
write ('la valeur absolue du nombre est :', abs) ;
end.
```

4.3 La structure conditionnelle alternative

La structure conditionnelle alternative nous permet, sur la base d'une condition, de sélectionner le bloc d'instructions à exécuter parmi deux blocs.

Syntaxe

En algorithme, la syntaxe de la structure alternative est comme suit :

Si *condition* **alors**
  <*Bloc d'instruction1*>
Sinon
  <*Bloc d'instruction2*> ;
Finsi ;

Sémantique

L'exécution de la structure s'effectue en 2 étapes :

1. Evaluer la condition.
2. Si elle est vérifiée (vrai), seulement le *Bloc d'instruction1* sera exécuté ;
   Et si elle n'est pas vérifiée (fausse), seulement le *Bloc d'instruction2* sera exécuté.

En langage pascal, la syntaxe est comme suit :

if *condition* **then**
  <*Bloc d'instruction1*>
else
  <*Bloc d'instruction2*> ;

Rappel

Il faut toujours utiliser **begin** et **end** pour délimiter le bloc d'instructions s'il contient plusieurs instructions (2 ou plus).

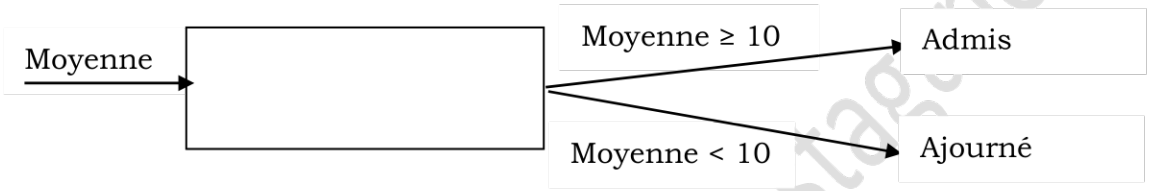
Remarque

En pascal le mot clé **else** ne doit jamais être précédé par le point-virgule (**;**).

Exemple1

Ecrire un algorithme permettant de lire la moyenne d'un étudiant et d'afficher la mention « Admis » si la moyenne est supérieure ou égale à *10* et la mention « Ajournée » dans le cas contraire.
Traduire cet algorithme en langage pascal.

Pour afficher l'une de ces deux mentions, il faut tester la valeur de la note introduite par l'utilisateur (via un clavier). Pour effectuer ce test, on utilise la structure alternative.
On peut exprimer la problématique de la façon suivante :
Si la note est supérieure ou égale à *10* **alors** afficher «Admis» **sinon** afficher «Ajourné»**.**



L'algorithme est comme suit :

```
Algorithme   mention ;
Variable
Moyenne : réel ;
Début
Ecrire ('donnez la moyenne de l''étudiant');
Lire (moyenne) ;
Si  (moyenne ≥ 10)    alors
   Ecrire ('Admis')
Sinon
   Ecrire ('Ajourné') ;
Finsi
Fin.
```

La traduction en langage pascal est comme suit :

```
program   mention ;
 Var
Moyenne : real ;
Begin
write ('donnez la moyenne de l''étudiant');
read (moyenne) ;
if  (moyenne>=10)    then
      write('Admis')
else
      write ('Ajourné') ;
End.
```

Exemple 2

Ecrire l'algorithme qui lit un nombre et détermine si ce nombre est multiple de 3.

L'algorithme est comme suit :

```
Algorithme multiple ;
Variable
nb : entier ;
Début
Ecrire ('donner un nombre entier') ;
Lire (nb) ;
Si   (nb mod 3 =0)    alors
   Ecrire (nb, ' est multiple de 3')
Sinon
   Ecrire (nb ,' n"est pas multiple de 3') ;
Finsi
Fin.
```

La traduction en langage pascal est comme suit :

```
program multiple ;
Var
nb : integer ;
begin
write ('donner un nombre entier') ;
read (nb) ;
if  (nb mod 3 =0)    then
     write (nb, ' est multiple de 3')
else
   write (nb ,' n"est pas multiple de 3') ;
end.
```

La variable *nb* qui représente le nombre à lire doit être de type entier, puisque la fonction **mod** (modulo) ne s'applique que sur les entiers.

Rappel

La fonction **mod** permet de retourner le reste de la division entière. Ainsi, *nb mod 3* exprime le reste de la division entière de *nb* par *3*.

4.4 Les structures alternatives imbriquées

Dans certains traitements, il faut sélectionner un bloc d'instructions à exécuter parmi plusieurs (plus de deux blocs).
Lorsque le nombre de cas possibles est supérieur à deux, il serait nécessaire d'utiliser les structures alternatives imbriquées (une structure à l'intérieur d'une autre).

Remarque

(Le nombre de structures "**si**…**alors**… **sinon**") = (nombre de cas) -1.

Exemple

Ecrire un algorithme qui lit la moyenne d'un étudiant est affiche l'un des messages suivants :

- 'Bon' si la moyenne est supérieure à *13*.
- 'Moyen' si la moyenne est comprise entre *10* et *13*.
- 'Faible' si la moyenne est inférieure à *10*.

Dans cet exemple, le nombre de cas possible est *3*. La structure alternative vue dans la section précédente ne permet pas de résoudre le problème. On utilise alors la structure des alternatives imbriquées.
Notons que :

(Nombre de structures " **si**...**alors**... **sinon** ") = (nombre de cas) -1 = 3 -1 = **2**.

L'algorithme est comme suit :

```
Algorithme   mention ;
Variable
Moyenne : réel ;
Début
Ecrire ('donnez la moyenne de l"étudiant');
Lire (moyenne) ;
Si  (moyenne > 13) alors
     Ecrire ('Bon')
Sinon
     Si  (moyenne ≥ 10) alors
          Ecrire ('Moyen')
     Sinon
          Ecrire ('Faible') ;
     Finsi
Finsi
Fin.
```

La traduction en langage pascal est comme suit :

```
program  mention ;
Var
Moyenne : real ;
begin
write ('donnez la moyenne de l"étudiant');
read (moyenne) ;
if  (moyenne > 13) then
     write ('Bon')
else
     if  (moyenne > =10) then
          write ('Moyen')
     else
          write ('Faible') ;
end.
```

4.5 Exercice d'application

Ecrire l'algorithme et le programme en pascal qui permettent de résoudre une équation du deuxième degré.

Solution

Une équation du deuxième degré est comme suit : $ax^2 + bx + c = 0$.
Où x est l'inconnue et a, b et c sont les coefficients, avec a différent de 0.
Les données en entrée sont les coefficients a, b et c.
Les résultats en sortie sont les différentes valeurs de l'inconnue x, selon la valeur du discriminant (Δ).

$$x = \begin{cases} aucune\ solution, & \Delta < 0 \\ soluiton\ double: \dfrac{-b}{2 * a}, & \Delta = 0 \\ \dfrac{-b \pm \sqrt{\Delta}}{2 * a}, & \Delta > 0 \end{cases}$$

On va utiliser la structure des alternatives imbriquées avec *2* " **si**…**alors**… **sinon** ", puisque le nombre de cas =*3* (selon la valeur de Δ).
Plus un autre test sur la valeur de a afin de s'assurer qu'elle est différente de zéro, pour ne pas avoir une équation du premier degré.

L'algorithme est comme suit :

```
algorithme  equa_2_degres ;
Variable
a,b,c,delta,x1,x2 : réel ;
debut
  écrire ('donnez le 1er coefficient a=') ;
  lire (a) ;
  si a=0 alors
    écrire ('donnez une valeur de a≠0')
  sinon
    écrire ('donnez le 2eme coefficient b=') ;
    lire (b) ;
    écrire ('donnez le 3eme coefficient c=') ;
    lire (c) ;
    delta←b*b-4*a*c;
    si delta <0 alors
     écrire ('pas de solution')
    sinon
      si delta=0 alors
        x1:=-b/(2*a);
        écrire ('Solution double, x=', x1)
      sinon
        x1:=(-b-sqrt(delta))/(2*a);
        écrire ('x1=', x1) ;
        x2:=(-b+sqrt(delta))/(2*a);
        écrire ('x2=', x2) ;
      finsi;
    finsi;
  finsi;
Fin.
```

Le programme en pascal est comme suit :

```
program equa_2_degres ;
Var
a,b,c,delta,x1,x2 : real ;
begin
  write (‘donnez le 1er coefficient a=’) ;
  read (a) ;
  if a=0 then
    write (‘donnez une valeur de a≠0’)
  else
  begin
    write (‘donnez le 2eme coefficient b=’) ;
    read (b) ;
    write (‘donnez le 3eme coefficient c=’) ;
    read (c) ;
    delta:=b*b-4*a*c;
    if delta <0 then
     write (‘pas de solution’)
    else
      if delta=0 then
      begin
        x1:=-b/(2*a);
        write (‘Solution double, x=’, x1) ;
      end
      else
      begin
        x1:=(-b-sqrt(delta))/(2*a);
        write (‘x1=’, x1) ;
        x2:=(-b+sqrt(delta))/(2*a);
        write (‘x2=’, x2) ;
      end;
  end;
End.
```

5

Chapitre 5 : Structure à choix multiple « selon cas »

5.1 Introduction

Il est utile de pouvoir choisir un traitement parmi plusieurs en fonction de la valeur d'une variable. S'il n'existe que deux actions possibles, la structure alternative (si...alors...sinon) sera la plus approprié et convient très bien à la solution de ce type de problème. Sinon, un branchement direct vers le traitement voulu peut souvent être réalisé à l'aide de l'instruction **selon cas**.

5.2 Structure à choix multiple « selon cas »

La structure à choix multiple noté « **selon cas** » est une instruction qui permet d'effectuer un branchement à partir de la valeur d'une expression. Elle peut dans certains cas remplacer les structures alternatives imbriquées. On l'utilise lorsque les choix à gérer sont multiples.
La structure à choix multiple est une structure équivalente à une série de **si...alors...sinon** imbriqués. Elle doit être préférée, lorsque cela est possible, à un emboîtement de **si...alors...sinon** dont la lisibilité n'est en fait pas optimale.
La structure à choix multiples ne peut être utilisée que dans les cas où les expressions logiques (conditions) sont sous forme d'égalités entre des valeurs de type discret (entier, caractère).
A la différence de **si...alors...sinon** imbriqués, avec **selon cas**, la condition de branchement doit porter sur la comparaison d'égalité d'une expression avec des valeurs constantes, et non sur une condition quelconque (le branchement en fonction de conditions comme *delta*>0, *delta*<0 ou *mesure=0.01*, doit être réalisé avec **si...alors...sinon** imbriqués).
Par exemple, les structures de contrôle conditionnelles contenant des expressions logiques (conditions) de type *a>0*, *a=0.1*, *b/2=1*, ne doivent être réalisées que par les structures alternatives imbriquées (impossible d'utiliser la structure à choix multiple).

La structure **selon cas**, ne convient pas pour les conditions :

- *a>0* : car ce n'est pas une comparaison d'égalité.
- *a=0.1* : car la valeur *0.1* est de type réel (qui n'est pas discret).
- *b/2=1* : car l'expression *b/2* est de type réel (qui n'est pas discret).

Donc la structure **selon cas** peut être utilisée pour des comparaisons d'égalité entre des entiers ou entre des caractères… (*a=0, a='x'* …).

Syntaxe

```
Selon Cas express
  cas_1 : <Bloc d'instructions_1>;
  cas_2 : <Bloc d'instructions_2>;
  …
  cas_n : <Bloc d'instructions_n>
[ Sinon   <Bloc d'instructions_n+1> ];
Fin ;
```

La section du **Sinon** est optionnelle.

Où :

Express : une expression dont la valeur est de type discret.

cas1, cas2, …. casn : des listes de constantes du même type que l'expression *express*.

Chaque liste est constituée d'une suite de constantes, séparées par des virgules.

Remarque

En informatique, tout ce qui est entre crochets "**[ ]**" est optionnel. Il peut ne pas être utilisé (selon les besoins).

Sémantique

Après l'évaluation de l'expression, le programme est branché vers la liste qui contient la valeur de cette expression et le Bloc d'instructions correspondant sera exécuté.
Si la valeur de cette expression ne figure dans aucune de ces listes alors le programme est branché vers le **sinon** et le *Bloc d'instructions_n+1* est exécuté.
La présence du **sinon** dans l'instruction est facultative.

En langage pascal, la syntaxe est comme suit :

```
Case   express   of
  cas _1 : <Bloc d'instructions_1>;
  cas _2 : <Bloc d'instructions_2>;
  …
  cas _n : <Bloc d'instructions_n>
[else    <Bloc d'instructions_n+1> ] ;
end ;
```

5.3 Exercices d'application

Exercice 1

Nous désirons connaitre la réduction applicable à un usager de chemin de fer. La réduction est déterminée en fonction de la catégorie de l'usager :

Si la catégorie=1 alors la réduction=10%
Si la catégorie=2 alors la réduction =50%
Si la catégorie=3 alors la réduction=100%
Si la catégorie=4 alors la réduction =30%
Pour toute autre valeur de la catégorie, la réduction=0%.

1. En utilisant les instructions alternatives, écrire un algorithme qui permet de :
   - Lire le montant du billet et la catégorie de l'usager.
   - Calculer et afficher le montant de la réduction.
2. Réécrire le même algorithme en utilisant l'instruction à choix multiple.

Solution1 :

Dans l'algorithme suivant, on utilise la structure "**si…alors…sinon**" imbriqués.

Il existe 5 cas possibles selon la catégorie "*cat*", donc il y aura 4 **si…alors…sinon.**

```
Algorithme   reduction ;
Variable
mtb, red : réel ;
cat : entier ;
Début
 Ecrire ('Donner le montant du billet :') ;
 Lire (mtb) ;
 Ecrire ('Donner la catégorie de l"usager :') ;
 Lire (cat) ;
Si  (cat= 1) alors
    Red ← mtb*0.1
Sinon
   Si  (cat=2) alors
       Red ← mtb*0.5
   Sinon
       Si  (cat=3) alors
           Red ← mtb
       Sinon
           Si  (cat=4) alors
               Red ← mtb*0.3
           Sinon
               Red←0
            Finsi
         Finsi
    Finsi
Finsi
Ecrire (' Le montant de la réduction=', red) ;
Fin.
```

Solution2 :

Dans l'algorithme qui suit, on utilise la structure **"Selon Cas".**

La solution suivante illustre le cas où les listes (cas1, cas2, cas3, cas4) sont simples et ne contiennent qu'une seule constante (1, 2 , 3 , 4) .

```
Algorithme   reduction ;
Variable
mtb, red : réel ;
cat : entier ;
Début
 Ecrire ('Donner le montant du billet :') ;
 Lire (mtb) ;
 Ecrire ('Donner la catégorie de l"usager :') ;
 Lire (cat) ;
  Selon Cas cat
    1 :  Red ← mtb*0.1 ;
    2 :  Red ←  mtb*0.5 ;
    3 :  Red ← mtb;
    4 :  Red ← mtb*0.3
  Sinon
   Red ← 0 ;
  Fin ;
Ecrire (' Le montant de la réduction=', red) ;
Fin.
```

La traduction en langage pascal de cet algorithme est comme suit :

```
program   reduction ;
Var
mtb, red : real ;
cat : integer ;
begin
 write ('Donner le montant du billet :') ;
 read (mtb) ;
 write ('Donner la catégorie de l"usager :') ;
 read (cat) ;
  Case cat of
    1 :  Red := mtb*0.1 ;
    2 :  Red :=  mtb*0.5 ;
    3 :  Red := mtb;
    4 :  Red := mtb*0.3
  else
     Red := 0 ;
  end ;
  write (' Le montant de la réduction=', red) ;
end.
```

Exercice 2

Ecrire l'algorithme et le programme qui lisent un caractère et affichent à quel ensemble il appartient :

1. Aux "Lettres" si le caractère lu est 'A'..'Z', 'a'..'z'
2. Aux "Chiffres" si le caractère lu est '0'..'9'.
3. Aux "Opérateurs" si le caractère lu est '+', '-', '*', '/'
4. Aux "Comparateurs" si le caractère lu est '<','>','='
5. Sinon aux "Caractère spéciaux" .

L'algorithme suivant illustre le cas où les listes sont composées : chaque liste est constituée d'une suite de constantes, séparées par des virgules.

```
algorithme caracteres;
Variable
car :caractere;
debut
écrire('Tapez un caractère');
lire (car);
selon cas car
  'A'..'Z', 'a'..'z':  Écrire('Lettres');
  '0'..'9':            Écrire('Chiffres');
  '+', '-', '*', '/':  Écrire('Opérateurs');
  '<','>','=':         Écrire('Comparateurs')
sinon
  Écrire('Caractère spéciaux');
fin;
fin.
```

La traduction de l'algorithme en langage pascal est comme suit :

```
Program caracteres;
Var
car :char;
Begin
Write('Tapez un caractère');
read (car);
case car of
  'A'..'Z', 'a'..'z' :  Write('Lettres');
  '0'..'9' :            Write('Chiffres');
  '+', '-', '*', '/' :  Write('Opérateurs');
  '<','>','=' :         Write('Comparateurs')
else
  Write('Caractère spéciaux');
end;
End.
```

6 Chapitre 6 : Structures de contrôle répétitives (les boucles)

6.1 Introduction

Dans les problèmes quotidiens, on n'exécute pas uniquement des traitements (blocs d'instructions) avec ou sans conditions, mais il peut être fréquent d'exécuter un traitement plusieurs fois.
En effet, pour saisir les *N* notes d'un étudiant et calculer sa moyenne, on est amené à saisir *N* variables, puis faire la somme et ensuite diviser la somme par *N*. Cette solution nécessite la réservation de l'espace par la déclaration des variables, et une série d'instructions de lecture et d'écriture.
Ce problème peut être remédié en utilisant les structures de contrôle répétitives, appelées aussi structures de contrôle itératives ou tout simplement les **boucles**. Celles-ci permettent de donner un ordre de répétition d'une d'instruction ou d'un bloc d'instructions zéro ou plusieurs fois.

6.2 Les itérations (les boucles)

Les itérations (les boucles) permettent de répéter plusieurs fois l'exécution d'un même traitement décrit par une instruction ou par un bloc d'instructions.
En littérature, le bloc d'instructions à répéter est aussi appelé corps de la boucle.
Dans une boucle, le nombre d'itérations doit être fini et doit être contrôlé par un compteur ou par une condition.

6.3 La boucle "Pour"

La boucle **Pour** utilise un compteur pour contrôler le nombre d'itérations du traitement.
Donc, ce nombre d'itérations doit être connu (fixé à l'avance).

Syntaxe

La syntaxe en langage algorithmique est comme suit :

> **Pour** *Compteur* **de** *Valeur_initial* **à** *Valeur_finale* **[Pas** *Valeur_pas***] Faire**
> <Bloc d'instructions> ;
> **Finfaire**

Où :

Compteur : est une variable entière, qui compte le nombre de répétition de l'exécution du <Bloc d'instructions> ;

Valeur_initiale : la valeur initiale à laquelle le *compteur* est initialisé,

Valeur_finale : la valeur finale sur laquelle le *compteur* s'arrête,
Valeur_pas : la valeur du pas, c'est la valeur qu'on rajoute au *compteur* à chaque itération.

Sémantique

La boucle **Pour** fonctionne comme suit :

1. Initialisation du C*ompteur* par la valeur de *Valeur_initiale* (comme si on avait *Compteur =Valeur_initiale*)
2. Tester *compteur :*
   - si la valeur du *Compteur* dépasse la *Valeur_finale* (du côté supérieur ou inférieur, selon la positivité ou la négativité du pas) ;
     alors la boucle s'arrête et l'exécution se poursuit après le **Finfaire**
     Sinon, exécution du < Bloc d'instructions>,
   - Incrémentation ou décrémentation de *Compteur* par la valeur du pas (selon la positivité ou la négativité du pas),
   - Retour à l'étape 2.

Remarques

- La boucle **Pour** est utilisée lorsqu'on connaît le nombre d'itérations d'avance.
- La valeur du pas peut être positive ou négative et par conséquent, il faut, au départ de la boucle, que la *Valeur_initiale* soit inférieure ou égale à la *Valeur_finale* ou que la *Valeur_initiale* soit supérieure ou égale à la *Valeur_finale* selon la positivité ou la négativité de cette valeur.
- La boucle **Pour** permet de répéter le traitement **zéro** ou plusieurs fois.
  En effet si la valeur du pas est positive et la *Valeur_initiale* est supérieure à la *Valeur_finale* alors aucune itération ne sera exécutée.
- La valeur du pas est facultative. Elle égale à *1* par défaut.

En langage Pascal, la syntaxe de la boucle **pour** sera comme suit :

```
For compteur :=valeur_initiale to valeur_finale do
  <Bloc d'instructions> ;
```

Rappel

Le bloc d'instructions doit être mis entre **begin** et **end** s'il contient plusieurs instructions (2 ou plus).

Remarques

- En langage Pascal, il n'est pas possible d'utiliser la boucle **For** avec un pas différent de *1* ou de *-1*.
- En Pascal, si la valeur du pas=*-1* alors la syntaxe se présente comme suit :

```
For compteur :=valeur_initiale downto valeur_finale do
  <Bloc d'instructions> ;
```

Exemple 1

Ecrire un algorithme qui calcule et affiche la somme de *n* nombres.

Dans cet algorithme, on doit lire *n* nombres et calculer leur somme. L'opération de lecture doit se répétée *n* fois. Après chaque lecture, l'algorithme doit ajouter la valeur du nombre lu à la somme des nombres lus précédemment.
Donc le traitement qui consiste à lire le nombre et à l'ajouter à la somme doit se répéter *n* fois.

```
Algorithme moyenne ;
Variable
i, n :entier ;
s, x : réel ;
Début
Ecrire ('Donner le nombre de nombres') ;
Lire (n) ;
s←0 ;
Pour  i de 1 à n pas 1 faire
  Ecrire ('Donner un nombre') ;
  Lire(x) ;
  s←s+x ;
finfaire
Ecrire ('La somme =',s) ;
fin.
```

Le pas est facultatif. Puisque pas=1. On pouvait tout simplement écrire **Pour i de 1 à n faire**

La traduction en pascal est comme suit :

```
program moyenne ;
Var
i, n :integer ;
s, x : real ;
begin
write ('Donner le nombre de nombres') ;
read (n) ;
s :=0 ;
for  i := 1 to n do
begin
  write ('Donner un nombre') ;
  read(x) ;
  s :=s+x ;
end;
write ('La somme =',s) ;
end.
```

Notons que même si $n \leq 0$, l'exécution restera cohérente avec la valeur de n et s restera égale à 0 ($s=0$), puisque aucune itération ne sera faite.

Exemple 2

Ecrire un algorithme qui affiche les valeurs de *1* à *n* dans l'ordre inverse, où *n* est un nombre entier positif.
Dans cet algorithme, il s'agit d'écrire les instructions permettant d'afficher les nombre de *n* à *1* (*n , n-1,*.....3 , 2 , 1*)* . L'instruction d'affichage doit être répétée *n* fois.
Nous utilisons dans cet algorithme, la variable entière *i* pour compter le nombre de répétition. Cette variable est initialisée par la valeur *n* et elle est décrémentée à chaque itération de la boucle.
Donc, le pas de la boucle dans cet algorithme est égal à *-1*.

```
Algorithme inverse ;
Variable
n, i : entier ;
Début
Ecrire ('Donner un entier') ;
Lire(n) ;
Pour  i de n à 1 pas -1 faire
   Ecrire (i) ;
Finfaire
Fin.
```

Le pas doit obligatoirement être défini. Sinon sa valeur sera par défaut égale à 1

La traduction en pascal est comme suit :

```
program inverse ;
Var
n, i : integer ;
begin
write ('Donner un entier ') ;
read(n) ;
for  i := n downto 1 do
   write (i) ;
end.
```

Là aussi, si $n \leq 0$, aucun affichage ne sera effectué, puisque aucune itération ne sera faite. Car, on ne peut pas reculer à partir d'une valeur inférieure à *1* vers *1*. Par exemple : il n'y a aucune itération pour la boucle " **for** i := 0 **downto** 1 **do** **".**

6.4 La boucle "Répéter"

Cette structure permet de répéter le traitement **une** ou plusieurs fois et de s'arrêter sur une condition. En effet, lorsque la condition est vérifiée, la boucle s'arrête, sinon elle réexécute le traitement.

Syntaxe

Répéter
<Bloc d'instructions> ;
Jusqu'à (*condition d'arrêt*) ;

Sémantique

La boucle **Répéter** fonctionne comme suit :

1. Exécution du < Bloc d'instructions >.
2. Tester la valeur de la <*condition d'arrêt*>.
3. Si elle est vérifiée alors la boucle s'arrête,
   Sinon retour à l'étape 1.

Remarques

- Dans cette boucle, le Bloc d'instructions est exécuté au moins une fois avant l'évaluation de la condition d'arrêt.
- Les paramètres de la condition doivent être initialisés avant la boucle (par lecture ou par affectation).
- Il doit y avoir une instruction dans le Bloc d'instructions qui modifie la valeur de la condition, sinon le nombre d'itérations devient infini.

En langage Pascal, la syntaxe de la boucle **Répéter** est donnée par :

Repeat
< Bloc d'instructions > ;
Until (*condition d'arrêt*) ;

Remarque

- En pascal, le bloc d'instructions ne doit pas être mis entre **begin** et **end**, même s'il contient plusieurs instructions, car il sera délimiter par **repeat** et **until**.

Exemples :

Reprendre les mêmes exemples précédents avec la boucle **Répéter**.

A la différence de la boucle **Pour**, dans la boucle **Répéter**, il faut commencer par l'initialisation du compteur avant la boucle.
De plus, à la fin du traitement, ce compteur doit être modifié (incrémenté ou décrémenté) afin d'atteindre la valeur finale.

1er Exemple :

L'algorithme du 1er exemple est comme suit :

```
Algorithme moyenne ;
Variable
i, n :entier ;
s, x : réel ;
Début
Ecrire('Donner le nombre de nombres (supérieur à 0)');
Lire (n) ;
s←0 ;
i←1 ;
Répéter
  Ecrire ('Donner un nombre') ;
  Lire(x) ;
  s←s+x ;
  i←i+1;
jusqu'à (i>n);
Ecrire ('La somme =',s) ;
fin.
```

La traduction en pascal est comme suit :

```
program moyenne ;
Var
i, n :integer ;
s, x : real ;
begin
write('Donner le nombre de nombres (supérieur à 0)');
read (n) ;
s :=0 ;
i :=1 ;
repeat
  write ('Donner un nombre') ;
  read(x) ;
  s:=s+x ;
  i:=i+1;
until (i>n);
write ('La somme =',s) ;
end.
```

Contrairement à la boucle **Pour**, dans la boucle **Répéter** *n* doit être supérieur à 0 (*n>0*), sinon l'exécution devient incohérente avec la valeur de *n*. Remarquons le message '*Donner le nombre de nombres (supérieur à 0)*'.

2eme Exemple

L'algorithme du 2eme exemple est comme suit :

```
Algorithme inverse ;
Variable
n, i : entier ;
Début
Ecrire ('Donner un entier positif non nul');
Lire(n) ;
i←n ;
Répéter
    Ecrire (i) ;
    i←i-1 ;
jusqu'à (i<1) ;
Fin.
```

La traduction de l'algorithme en pascal est comme suit

```
program inverse ;
Var
n, i : integer ;
begin
write ('Donner un entier positif non nul');
read(n) ;
i :=n ;
repeat
    write (i) ;
    i:= i-1 ;
until (i<1) ;
end.
```

Là aussi, *n* doit être doit être supérieur à *0* sinon l'exécution devient incohérente. Remarquons le message '*Donner un entier positif non nul*'.

Par exemple pour *n=-1*, l'exécution commence par l'affichage de *-1*, ensuite elle décrémente la valeur de *i* (*i=-2*) et elle s'arrête puisque *i<n*.

6.5 La boucle "Tant que"

Cette structure permet de répéter le traitement **zéro** ou plusieurs fois selon la condition d'exécution. En effet, lorsque la condition d'exécution est vérifiée, le traitement est exécuté, sinon la boucle s'arrête.

Syntaxe

Tant que (*condition d'exécution*)
Faire
< Bloc d'instructions > ;
FinFaire

Sémantique

La boucle **Tant que** fonctionne comme suit :

1. Tester la valeur de la <condition d'exécution>
2. Si elle est vérifiée alors
   exécution du < Bloc d'instructions >
   retour à l'étape 1.
   Sinon arrêt de la boucle.

Remarques

- Dans cette boucle, le traitement peut ne jamais être exécuté, c'est lorsque la condition d'exécution est à fausse dès le départ.
- Les paramètres de la condition doivent être initialisés avant la boucle (par lecture ou par affectation).
- Il doit y avoir une instruction dans le Bloc d'instructions qui modifie la valeur de la condition, sinon le nombre d'itérations devient infini.

En langage Pascal, la syntaxe de la boucle **Tant que** est comme suit :

While (*condition d'exécution*) **do**
< Bloc d'instructions > ;

Rappel

Le bloc d'instructions doit être mis entre **begin** et **end** s'il contient plusieurs instructions.

Exemples

Reprendre les même exemples précédents avec la boucle **Tant que**.

A la différence de la boucle **Répéter**, dans la boucle **Tant que**, la condition est évaluée au début.
Et à la différence de la boucle **Pour**, dans la boucle **Tant que**, il faut commencer par l'initialisation du compteur avant la boucle. De plus, à la fin du traitement, ce compteur doit être modifié afin d'atteindre la valeur finale.

1er Exemple :

L'algorithme du 1er exemple est comme suit :

```
Algorithme moyenne ;
Variable
i, n :entier ;
s, x : réel ;
Début
Ecrire ('Donner le nombre de nombres');
Lire (n) ;
s←0 ;
i←1 ;
Tant que (i≤n) faire
  Ecrire ('Donner un nombre') ;
  Lire(x) ;
  s←s+x ;
  i←i+1;
finfaire
Ecrire ('La somme =',s) ;
fin.
```

La traduction en pascal est comme suit :

```
program moyenne ;
Var
i, n :integer ;
s, x : real ;
begin
write ('Donner le nombre de nombres');
read (n) ;
s :=0 ;
i :=1 ;
while (i<=n) do
begin
  write ('Donner un nombre') ;
  read(x) ;
  s :=s+x ;
  i:=i+1;
end;
write ('La somme =',s) ;
end.
```

Comme pour la boucle **Pour**, avec la boucle **Tant que**, l'exécution restera cohérente quel que soit la valeur de *n* (pour *i<=0, s* restera égale à 0).

2eme Exemple :

L'algorithme du 2eme exemple est comme suit :

```
Algorithme inverse ;
Variable
n, i : entier ;
Début
Ecrire ('Donner un entier ');
Lire(n) ;
i←n ;
Tant que (i≥1) faire
    Ecrire (i) ;
    i←i-1 ;
finfaire
Fin.
```

La traduction en pascal est comme suit :

```
program inverse ;
Var
n, i : integer ;
begin
write ('Donner un entier ');
read(n) ;
i :=n ;
while (i>=1) do
begin
    write (i) ;
    i:= i-1 ;
end;
end.
```

Là aussi, si $n \leq 0$ alors aucune itération ne sera faite.

Remarques

1. Tout problème qui peut être résolu avec la boucle **Pour**, peut aussi être résolu avec les boucles **Repeter** et **Tant que** mais le contraire n'est pas vrai.
2. Si le nombre d'itérations est connu à l'avance, il est préférable d'utiliser la boucle **Pour**.
3. Si le traitement risque de ne pas s'exécuter (*0* itérations), il est recommandé d'utiliser les boucles **Pour** ou **Tant que**.

6.6 Exercice d'application

Ecrire un algorithme qui permet de :

- Lire une suite de lettres. La lecture se termine par la saisie du point '.' .
- Afficher le nombre de consonnes et le nombre de voyelles.

Dans cet exemple, le nombre de lettres à lire n'est pas connu à l'avance. L'opération de lecture est répétée jusqu'à ce que le '.' soit introduit. Donc, il n'est pas possible d'utiliser la boucle **Pour**.

Solution 1

Dans cette solution on utilise la bucle **Tant que**

```
Algorithme  exemple ;
Variable
L : caractère ;            { L : la lettre lue }
nbv, nbc : entier ;
Début
Nbv←0 ;
Nbc←0 ;
Ecrire('donner une L alphabétique') ;
Lire(L) ;
Tant que (L ≠ '.') faire
  Si (L='a') ou (L='e') ou (L='i') ou (L='o') ou (L='u') ou (L='y') alors
     nbv←nbv+1
  sinon
     nbc←nbc+1 ;
  finsi
  Ecrire('donner une autre L alphabétique') ;
  Lire(L) ;
Finfaire
Ecrire ('le nombre de voyelles=',nbv) ;
Ecrire ('le nombre de consonnes=',nbc) ;
Fin.
```

La traduction de l'algorithme en langage pascal est comme suit :

```
program  exemple ;
Var
L : char ;                          { L : la lettre lue }
nbv, nbc : integer ;
begin
Nbv :=0 ;
Nbc :=0 ;
Write('donner une L alphabétique') ;
read(L) ;
while (L <> '.') do
begin
  if (L='a') or (L= 'e') or (L='i') or (L='o') or (L='u') or (L='y') then
     nbv :=nbv+1
  else
     nbc :=nbc+1 ;
  Write('donner une autre L alphabétique') ;
  read(L) ;
end;
Write ('le nombre de voyelles=',nbv) ;
Write ('le nombre de consonnes=',nbc) ;
end.
```

La première opération dans cet algorithme consiste à initialiser les compteurs des voyelles et des consonnes (*Nbv* et *Nbc* respectivement). Afin de tester la condition de la boucle, il est nécessaire de faire une première lecture avant la boucle. Une autre lecture est nécessaire à l'intérieur de la boucle pour pouvoir reexécuter le traitement. La boucle risque ne pas s'exécuter si on introduit le '.' comme première lettre. L'algorithme affiche dans ce cas la valeur zéro pour le nombre de voyelles et le nombre de consonnes.

Solution 2

Dans cette solution on utilise la boucle **Répéter**

```
Algorithme  exemple ;
Variable
L : caractère ;
nbv, nbc : entier ;
Début
Nbv←0 ;
Nbc←0 ;
Ecrire('donner une L alphabétique') ;
Lire(L) ;
Si (L≠'.') alors
Répéter
  Si (L='a') ou (L='e') ou (L='i') ou (L='o') ou (L='u')ou (L='y') alors
    nbv←nbv+1
  sinon
    nbc←nbc+1 ;
  finsi
  Ecrire('donner une autre L alphabétique') ;
  Lire(L) ;
Jusqu'à (L='.') ;
finsi
Ecrire ('le nombre de voyelle=',nbv) ;
Ecrire ('le nombre de consonne=',nbc) ;
Fin.
```

La traduction de l'algorithme en langage pascal est comme suit :

```
program  exemple ;
Var
L : char ;
nbv, nbc : integer ;
begin
Nbv :=0 ;
Nbc :=0 ;
Write('donner une L alphabétique') ;
read(L) ;
if (L<>'.') then
repeat
  if (L='a') or (L= 'e') or (L='i') or (L='o') or (L='u') or (L='y') then
    nbv :=nbv+1
  else
    nbc :=nbc+1 ;
  Write('donner une autre L alphabétique') ;
  read(L) ;
until (L='.') ;
Write ('le nombre de voyelles=',nbv) ;
Write ('le nombre de consonnes=',nbc) ;
end.
```

Le même principe s'applique pour la boucle **Répéter**.
Le test (si…alors) avant la boucle permet de gérer le cas où on commence directement par la saisie du point (**'.'**).
Si ce test n'est pas utilisé, la boucle s'exécute même si on commence par la saisie du point (**'.'**) comme première lettre et l'algorithme affiche dans ce cas le nombre de consonne = 1 (au lieu d'afficher 0).

Bibliographie

- Mohamed Mezguiche. *Introduction à l'informatique : Historique et principe de fonctionnement.* OPU-Alger, 1987.
- Jacky Akoka et Isabelle Comyn-Wattiau. *Encyclopédie de l'informatique et des systèmes d'information* . Vuibert, 2006.
- Jacques Jorda et Abdelaziz M'zoughi. *Architecture de l'ordinateur : cours + exos corrigés.* Dunod, 2012.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. *Introduction to Algorithms.* MIT Press et McGraw-Hill edition. 2001.
- Thomas H. Cormen. *Algorithmes - Notions de base.* Collection : Sciences Sup, Dunod, 2013.
- Mohand Cherif BELAID. *Algorithmique & Programmation en PASCAL, Cours, Exercices, Travaux Pratiques Corrigés*. Pages Bleues Internationales, 2008.
- Claude Bauer et Pierre Vincenti, *Le langage Pascal appliqué à l'algorithmique : cours & exercices corrigés.* Ellipses, 2005.

Documents web

- http://zegour.esi.dz/Cours/Algo1/Plan1%20.htm
- http://zegour.esi.dz/Publication/Livre3/Livre3.htm

Annexe

Instructions et mots clés en langage algorithmique	**Instructions et mots clés équivalents en langage Pascal**
algorithme	program
début	begin
fin	end
variable	var
constante	const
ecrire	write
lire	read
entier	integer
réel	real
caractère	char
chaine de caractère	string
logique(booléen)	boolean
a ← 2	a :=2
a=2	a=2
a≠2	a <>2
a ≤2	a<=2
a ≥2	a >=2
si	*if*
alors	*then*
sinon	*else*
et	*and*
ou	*or*
non	*not*
tant que	*while*
faire	*do*
répéter	*repeat*
jusqu'à	*until*
pour i de v1 à v2 faire	*for i :=v1 to v2 do*
pour i de v2 à v1 faire	*for i :=v2 downto v1 do*