République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de Mostaganem Abdelhamid Ibn Badis

Faculté des Sciences et de la Technologie

Département des sciences et techniques

Polycopié du cours

Informatique II

Elaboré par :

Dr. ROUBA Baroudi

Maître de conférences « A »

Expertisé par :

Dr. Laredj Mohammed Adnane

Dr. Mechaoui Moulay Driss

Résumé

Ce polycopié aborde des concepts avancés en algorithmique à savoir : les variables indicées (tableaux et matrices), la manipulation des données de différents types (les enregistrements), la programmation modulaire (procédures et fonctions) ainsi que la récursivité.

Mots clés

Algorithme, Programme, Pascal, Vecteur, Matrice, Enregistrement, Procédure, Fonction, récursivité**.**

Avant-propos

Ce polycopié est adressé aux étudiants de la 1ère année du tronc commun « sciences et technologies ». Il constitue une continuité du module informatique I où des concepts de base de la programmation ont été abordés à savoir :

- La notion d'algorithme et de programme
- La structure d'un algorithme.
- Les instructions de bases
- Les structures alternatives
- Les structures répétitives et itératives.

Les chapitres du présent polycopié abordent des concepts avancés liés à la programmation. Il est organisé comme suit :

- Les trois premiers chapitres abordent les notions de variable et de type de façon approfondis en introduisant deux nouveaux concepts : les variables indicées (les tableaux et matrices) et les enregistrements.
- Le quatrième chapitre représente une introduction à la programmation modulaire. Il définit la notion du sous-programme et met en évidence les principales différences en les procédures et les fonctions.
- Le dernier chapitre aborde la notion de récursivité.

Il est à noter que tous les concepts abordés dans ce polycopié sont présentés à la fois en langage algorithmique ainsi qu'en langage Pascal.

Table des matières

Chapitre 1 : Les tableaux 1
1.1. Introduction 1
1.2. Les tableaux 1
1.3. Tableau unidimensionnel (vecteur) 2
1.4. Accès aux éléments d'un vecteur (Accès indiciel) 3
1.5. La lecture des éléments d'un vecteur 5
1.6. Affichage des éléments d'un vecteur 6
1.7. Exercice d'application 7
Chapitre 2 : Tableaux à deux dimensions (Matrices) 9
2.1. Introduction 9
2.2. Tableaux à deux dimensions (Matrices) 9
2.3. Accès aux éléments d'une matrice 10
2.4. Remplissage d'une matrice 10
2.5. Affichage d'un tableau à deux dimensions 10
2.6. Exercice d'application 14
Chapitre 3 : Les enregistrements 14
3.1. Introduction 14
3.2. Définition d'un enregistrement 14
3.3. Déclaration d'un enregistrement 14
3.3.1. Déclaration du type 14
3.3.2. Déclaration de la variable 16
3.4. Manipulation des enregistrements 16
3.4.1. L'affectation des champs d'un enregistrement 17
3.4.2. La lecture des champs d'un enregistrement 17
3.4.3. L'ecriture des champs d'un enregistrement 17
3.5. Affectation d'un enregistrement à un autre 19
3.6. Les structures imbriquées 19
3.6.1. Un enregistrement comme champs dans un autre enregistrement 20
3.6.2. Un tableau comme champ dans un enregistrement 20
3.6.3. Les tableaux d'enregistrements 23

Chapitre 4 : Les procédures et les fonctions 25
4.1. Introduction 25
4.2. Les sous-algorithmes (sous-programmes) 25
4.3. Les procédures 27
4.3.1. Notion de paramètres 27
4.3.1.1. Procédure non paramétrée 28
4.3.1.2. Procédure paramétrée 29
4.3.2. Les paramètres formels et effectifs 29
4.4. Les fonctions 30
4.5. Passage (transmission) des paramètres 31
4.5.1. Passage des paramètres par valeur 31
4.5.2. Passage des paramètres par variable 32
4.6. Exercice d'application 34
7.1.1 Le déroulement 38
Chapitre 5 : La récursivité 40
5.1. Introduction 40
5.2. La récursivité 40
5.3. Exercice d'application 46
5.3.1. Méthode des soustractions 47
5.3.2. Méthode d'Euclide 49

Chapitre 1 : Les tableaux

1.1. Introduction

Les variables que nous avons vues sont jusqu'à présent élémentaires. Elles ne contiennent qu'une seule valeur de type simple.

Cependant, il arrive que nous soyons obligés de traiter plusieurs données de même type appartenant à la même entité .

Exemple

Imaginons que dans un programme, nous ayons besoin simultanément de 10 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer 10 variables (N1,N2,...N10). Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, cela donnera obligatoirement : Moy =(N1+N2+N3+N4+N5+N6+N7+N8+N9+N10)/10.

C'est tout de même ennuyeux, surtout si nous sommes face à un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter.

Heureusement, la programmation nous permet de rassembler toutes ces variables en une seule, appelée tableau.

1.2. Les tableaux

- Un tableau est un ensemble (limité) de données (éléments) de même type.
- Un tableau peut être :
  - à une dimension (vecteur),
  - à deux dimensions (matrice),
  - à plusieurs dimensions (tableau multidimensionnel).
- Le nombre maximal d'éléments, précisé à la déclaration, s'appelle la capacité du tableau.
- Le type du tableau est le type de ses éléments.
- Les éléments d'un tableau se différencient les unes des autres par leur positionnement dans ce tableau.
- La position d'un élément s'appelle indice ou rang de l'élément. Un tableau possède un ensemble d'indices. A chaque valeur de l'indice ne correspond qu'une et une seule case du tableau, donc un élément.

Remarques

La définition indique que :

- Tous les éléments d'un tableau portent le même nom (celui du tableau).
- Tous les éléments d'un tableau ont le même type ; on parlera d'un tableau d'entiers, d'un tableau de caractères, …
- Un tableau peut ne pas être entièrement rempli mais il ne pourra jamais contenir plus d'éléments que le nombre prévu lors de la déclaration.
- L'indice est forcément un entier positif.

1.3. Tableau unidimensionnel (vecteur)

Un tableau à une dimension (vecteur) est une variable indicée permettant de stocker plusieurs valeurs de même type.

Syntaxe

Pour déclarer un vecteur dans un langage algorithmique, on utilise le mot clé **tableau** selon la syntaxe suivante :

> *Nom_Tableau* : **tableau** [1.. *capacité*] **de** *Type_de_donnée ;*

Où :

Nom_Tableau: indique le nom du tableau
capacité : indique la capacité du tableau.
Type_de_données : indique le type des éléments du tableau.

Pour déclarer un vecteur dans un langage pascal, on utilise le mot clé **array** selon la syntaxe suivante :

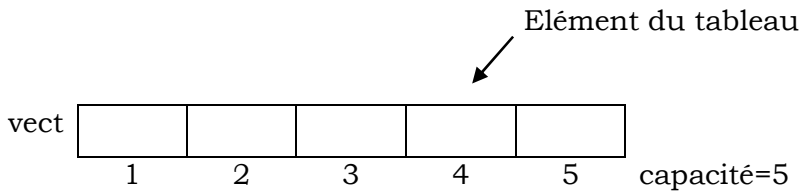
> *Nom_Tableau* : **array** [1.. *capacité*] **of** *Type_de_données ;*

Exemples

Variable
Vect : **tableau** [1..5] de entier ;
T1 : **tableau** [1..10] de caractère ;
Tab : **tableau** [1..100] de réel ;

- *Vect* est un tableau de 5 éléments (cases) de type entier
- *T1* est un tableau de 10 éléments de type caractère.

- *Tab* est un tableau de 100 cases de type réel.



Vect est un tableau de 5 éléments de type entier.

En langage Pascal, les déclarations précédentes sont traduites comme suit :

var

Vect : **array** [1..5] of integer ;

T1 : **array** [1..10] of char ;

Tab ; **array** [1..100] of real ;

1.4. Accès aux éléments d'un vecteur (Accès indiciel)

Pour repérer un élément parmi les autres, on utilise un indice qui représente un nombre entier permettant d'accéder à un élément.

Pour lire ou modifier la valeur d'un élément, on indique la valeur de son indice.

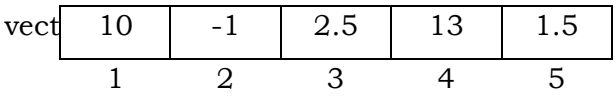
Exemple

Soient les déclarations suivantes :

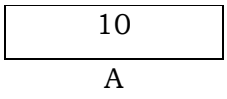
Vect : tableau [1..5] de réel ; (vect un tableau de capacité 5 de type réel).

a : réel ; (a une variable de type réel).

b :entier ; (b une variable de type entier).



- L'instruction a←vect[1] exprime que la variable *a* reçoit la valeur (10) de l'élément n°1 du tableau *vect*.



- L'instruction vect[5]←34.56 exprime que l'élément n°5 du tableau *vect* reçoit la valeur 34.56.

vect	10	-1	2.5	13	**34.56**
	1	2	3	4	5

- Les instructions b←3 et a←vect[b] exprime que *a* reçoit la valeur (2.5) de l'élément n° 3 du tableau *vect*

2.5	3
a	b

- Les instructions b←3 et a←vect[b-1] exprime que a reçoit la valeur (-1) de l'élément n° 2 du tableau *vect*

-1	3
a	b

Remarque

Attention au débordement : dans un tableau de capacité *n*, un indice *i* doit toujours être compris entre 1 et n (∀ i, $1 \leq i \leq n$).

Exemple

Soit Vect : tableau [1..5] de entier.

- Un tableau peut ne pas être entièrement rempli.

Vect	12	51	10		
	1	2	3	4	5

- Mais il ne pourra jamais contenir plus d'éléments que le nombre prévu lors de la déclaration (capacité).

vect	12	51	10	61	-1	**21**	**-4**
	1	2	3	4	5	***X***	***X***

1.5. La lecture des éléments d'un vecteur

Pour remplir la **i^{ème}** case d'un tableau *vect* par une valeur saisie par l'utilisateur, il faut utiliser la syntaxe (de lecture) suivante :

En langage algorithmique :

```
Lire (vect[i]) ;
```

En langage pascal:

```
Read (vect[i]) ;
```

Pour remplir tous les éléments d'un tableau, il est nécessaire d'utiliser une boucle. Puisque la capacité du tableau est connue, alors la boucle **Pour** est la plus appropriée pour ce traitement, mais rien n'interdit d'utiliser les autres boucles.

Exemple

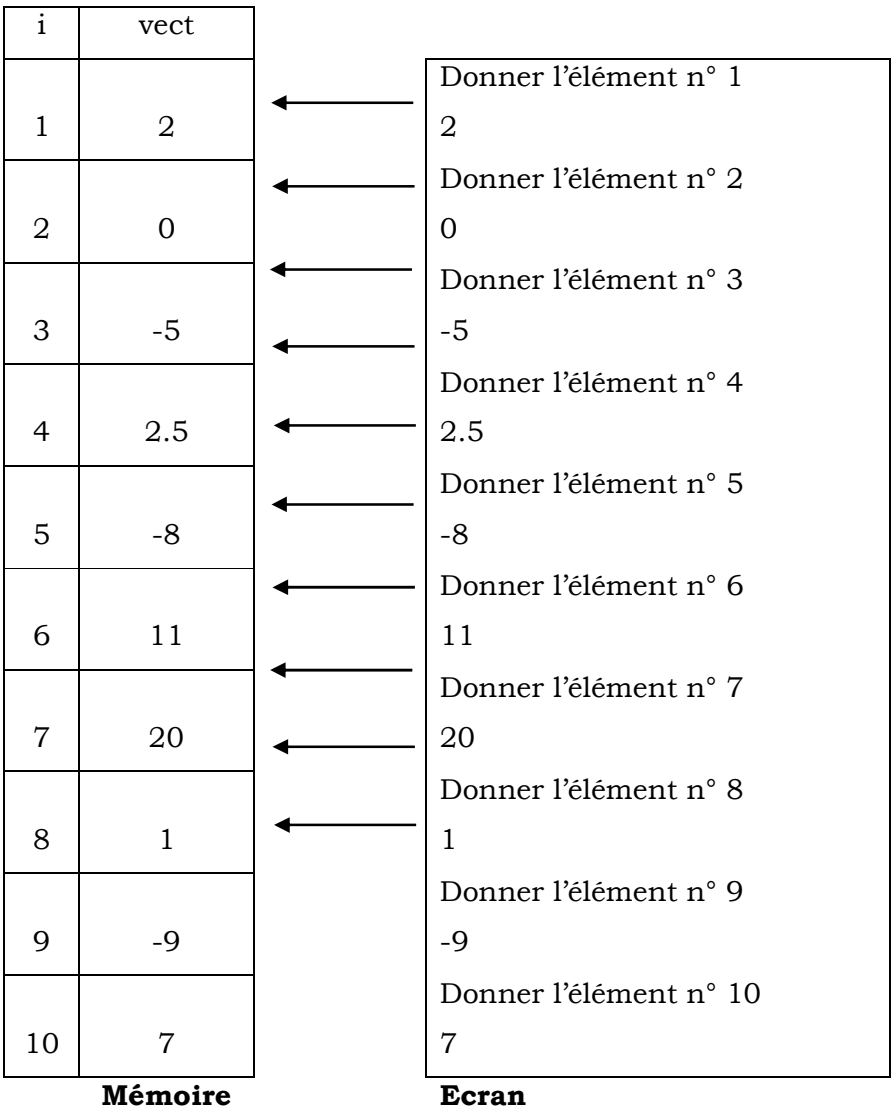
Soit *vect* un tableau de 10 éléments de type réel.

L'algorithme qui permet de remplir ce tableau est le suivant :

```
Algorithme lecture_tableau ;
Variable
i :entier ;
vect : tableau[1..10] de réel ;
Début
pour i de 1 à 10 faire
  écrire ('Donner l''élément n° ',i) ;
  lire (vect[i]) ;
finfaire
fin.
```

Exécution

Apres l'exécution de l'algorithme les états de la mémoire et de l'écran deviennent comme suit :



1.6. Affichage des éléments d'un vecteur

Pour afficher la **i^{ème}** case d'un tableau *vect*, il faut utiliser la syntaxe (d'écriture) suivante :

En langage algorithmique :

```
Ecrire (vect[i]) ;
```

En langage pascal:

```
Write (vect[i]) ;
```

Pour afficher tous les éléments d'un tableau, il est nécessaire d'utiliser une boucle. Là aussi la boucle **Pour** est la plus appropriée, puisque la dimension du tableau (le nombre d'éléments du tableau) est connue. On peut toujours utiliser les autres boucles.

Exemple

Soit *vect* un tableau de 10 éléments de type réel.

Vect

2	0	-5	2.5	-8	11	20	1	-9	7

L'algorithme qui permet d'afficher les éléments de ce tableau est le suivant :

```
Algorithme affichage_tableau ;
Variable
i :entier ;
vect : tableau[1..10] de réel ;
début
pour i de 1 à 10 faire
  écrire ('l"élément n° ', i , '=',vect[i]) ;
finfaire
fin.
```

Exécution

Apres l'exécution de l'algorithme, l'état de l'écran devient comme suit :

```
l'élément n° 1=2
l'élément n° 2=0
l'élément n° 3=-5
l'élément n° 4=2.5
l'élément n° 5=-8
l'élément n° 6=11
l'élément n° 7=20
l'élément n° 8=1
l'élément n° 9=-9
l'élément n° 10=7
```

Ecran

1.7. Exercice d'application

Ecrire un algorithme qui permet de :

1- Remplir un tableau « *temp* » par les températures quotidiennes d'une semaine.

2- Calculer et afficher la moyenne des températures.

On utilise :

- le tableau *temp* dont la taille (capacité) = 7.
- La variable *i* utilisée à la fois comme compteur pour les boucles et comme indice pour le tableau (avec $1 \leq i \leq 7$).
- Les variables *s* (pour le cumul des températures) et *m* (pour la moyenne des températures).

On utilise aussi 2 boucles :

- La 1ere boucle (7 itérations) pour le remplissage du tableau (remplir les 7 températures introduites par l'utilisateur).
- La 2eme boucle pour calculer *s* (cumul des températures).

L'algorithme sera comme suit :

```
Algorithme température ;
Variable
i :entier ;
temp : tableau [1..7] de réel ;
m,s : réel ;
début
pour i de 1 à 7 pas 1 faire
  écrire('Donner la température du jour n°', i) ;
  lire (temp[i]) ;
finfaire
s ←0 ;
pour i de 1 à 7 pas 1 faire
 s←s+temp[i] ;
finfaire
m←s/7;
écrire('La moyenne de la semaine=', m);
fin.
```

Chapitre 2 : Tableaux à deux dimensions (Matrices)

2.1. Introduction

L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par une seule coordonnée, mais par deux.

2.2. Tableaux à deux dimensions (Matrices)

L'utilisation d'une matrice (tableau bidimensionnel) s'avère très utile, lorsqu'on exécute le même traitement sur plusieurs vecteurs (tableaux unidimensionnels) ayant le même nombre d'éléments. En effet un tel tableau (matrice) peut regrouper tous ces vecteurs en une seule entité où chaque vecteur est représenté par une ligne. Ainsi, les éléments de la matrice seront repérés par 2 indices (ligne et colonne).

Syntaxe

Un tableau à 2 dimensions (matrice) se déclare en langage algorithmique comme suit:

NomMatrice : **tableau** [1..*nombre_Lignes* , 1.. *nombre_Colonnes*] **de** *Type_de_données*;

Où :

NomMatrice : indique le nom de la matrice.
Nombre_Lignes: indique le nombre de lignes de la matrice.
Nombre_Colonne: indique le nombre de colonnes de la matrice.
Type_de_données: indique le type de données des éléments de la matrice.

En langage pascal, la matrice est déclarée comme suit :

NomMatrice : **array** [1..*nombre_Lignes* , 1.. *nombre_Colonnes*] **of** *Type_de_données*;

Exemple

Mat : tableau [1..4,1..5] de entier ;

Cette déclaration veut dire qu'on demande à l'ordinateur de réserver un espace de mémoire pour 4 x 5 entiers, et quand on aura besoin de l'une de ces valeurs, on la repèrera par deux indices.

Le tableau *Mat* peut être assimilé à une matrice constituée de 4 lignes et 5 colonnes et qui peut être représenté comme suit :

Mat	1	2	3	4	5
1					
2					
3					
4					

2.3. Accès aux éléments d'une matrice

Pour accéder à un élément d'une matrice, on a besoin de spécifier deux indices :
Le 1er correspond au numéro de la ligne et le 2ème correspond au numéro de la colonne.
Ainsi, Mat[1][3] représente l'élément situé à la première ligne et la troisième colonne.

Mat	1	2	3	4	5
1			**8**		
2					
3					
4					

Mat[1][3]=8

Remarque

On peut utiliser deux notations pour repérer un élément.
La notation Mat[i][j] est équivalente à la notation Mat[i,j] pour désigner l'élément situé à la ieme ligne et la jeme colonne.

2.4. Remplissage d'une matrice

Pour remplir l'élément situé à la ième ligne et la jème colonne d'une matrice *Mat* par une valeur saisie par l'utilisateur, on utilise la syntaxe (de lecture) suivante :
En langage algorithmique :

```
Lire (Mat[i][j]) ;
```

En langage pascal:

```
Read (Mat[i][j]);
```

2.5. Affichage d'un tableau à deux dimensions

Pour afficher l'élément situé dans la ième ligne et la jème colonne d'une matrice *Mat,* on utilise la syntaxe (d'écriture) suivante :

En langage algorithmique :

```
écrire (Mat[i][j]) ;
```

En langage pascal:

```
write (Mat[i][j]);
```

Pour remplir une matrice ou afficher le contenu d'une matrice, nous aurons besoins 2 deux boucles :

- La première permet de parcourir les lignes
- et la deuxième est associée aux colonnes.

Algorithmes de lecture

Les algorithmes de remplissage d'une matrice *Mat* composée de 4 lignes et 5 colonnes, par les trois types de boucles sont comme suit :

La boucle Pour	**La boucle Tant que**	**La boucle Répéter**
Algorithme remplir_matrice ; Mat : tableau [1..4, 1..5] de entier ; i, j : entier ; début pour i de 1 à 4 faire pour j de 1 à 5 faire lire(mat[i][j]) ; finfaire finfaire fin .	Algorithme remplir_matrice ; Mat : tableau [1..4, 1..5] de entier ; i, j : entier ; début i ←1 ; tant que (i<=4) faire j←1 ; tant que (j<=5) faire lire (mat[i][j]) ; j←j+1 ; finfaire i←i+1 ; finfaire fin .	Algorithme remplir_matrice ; Mat : tableau [1..4, 1..5] de entier ; i, j : entier ; début i ←1 ; Répéter j←1 ; Répéter lire (mat[i][j]) ; j←j+1 ; jusqu'à (j>5); i←i+1; jusqu'à (i>4); fin .

Algorithmes d'affichage

Les algorithmes d'affichage du contenu d'une matrice *Mat* composée de 4 lignes et 5 colonnes, par les trois types de boucles sont comme suit :

La boucle Pour	**La boucle Tant que**	**La boucle Répéter**
Algorithme afficher_matrice ; Mat : tableau [1..4, 1..5] de entier ; i, j : entier ; début pour i de 1 à 4 faire pour j de 1 à 5 faire écrire(mat[i][j]) ; finfaire finfaire fin.	Algorithme afficher_matrice ; Mat : tableau [1..4, 1..5] de entier ; i, j : entier ; début i ←1 ; tant que (i<=4) faire j←1 ; tant que (j<=5) faire écrire (mat[i][j]) ; j←j+1 ; finfaire i←i+1 ; finfaire fin .	Algorithme afficher_matrice ; Mat : tableau [1..4, 1..5] de entier ; i, j : entier ; début i ←1 ; Répéter j←1 ; Répéter écrire (mat[i][j]) ; j←j+1 ; jusqu'à (j>5); i←i+1 jusqu'à (i>4); fin .

2.6. Exercice d'application

Ecrire un algorithme qui permet de :

- Remplir une matrice d'entiers de 5 lignes et 3 colonnes.
- Calculer et afficher la somme des nombres négatifs par lignes.

Traduire l'algorithme en langage pascal.

L'algorithme est comme suit :

```
algorithme negatif ;
Variable
Mat : tableau [1..5,1..3] de entier ;
T : tableau [1..5] de entier ;
i,j :entier ;
Debut
Pour i de 1 à 5 faire
  T[i] ← 0;
  pour j de 1 à 3 faire
    ecrire ('Donner mat[',i,'][',j,']');
    lire (mat[i][j]);
    si (mat[i][j]<0) alors
        T[i] ← T[i]+ mat[i][j];
    finsi
  finfaire
finfaire
Pour i de 1 à 5 faire
    ecrire ('T[',i,']=', T[i] );
finfaire
fin.
```

La traduction en langage pascal est comme suit :

```
program negatif ;
Var
Mat : array [1..5,1..3] of integer ;
T : array [1..5] of integer ;
i,j : integer ;
begin
for i := 1 to 5 do
begin
  T[i] := 0;
  for j := 1 to 3 do
  begin
    write ('Donner mat[',i,'][',j,']');
    read (mat[i][j]);
    if (mat[i][j]<0) then
        T[i] := T[i]+ mat[i][j];
  end;
end ;
for i := 1 to 5 do
    write ('T[',i,']=', T[i] );
end.
```

Chapitre 3 : Les enregistrements

3.1. Introduction

Jusqu'à présent nous avons utilisé des variables dont les types sont prédéfinis. C'est-à-dire des variables de type entier, réel, caractère, chaine de caractère ou logique. Lorsqu'il est nécessaire d'utiliser plusieurs variables de même type, il est possible d'utiliser des tableaux. Mais, parfois il serait nécessaire de manipuler plusieurs données de type différents en seul objet (ou entité). Pour y parvenir, on utilise les enregistrements.

3.2. Définition d'un enregistrement

Contrairement aux tableaux qui sont des structures de données dont tous les éléments sont de même type, les enregistrements sont des structures de données dont les éléments (les données) peuvent être de type différent et qui se rapportent à la même entité sémantique.

Les éléments qui composent un enregistrement sont appelés champs.

3.3. Déclaration d'un enregistrement

En algorithmique (ou en programmation), lorsqu'on veut déclarer une variable il faut s'assurer que son type est prédéfini (entier, réel, tableau…), sinon il faut déclarer ce nouveau type auquel la variable appartient.

Donc, avant de déclarer une variable enregistrement, il faut avoir au préalable définit son type (sa structure), c'est à dire le nom et le type des champs qui le composent. Le type d'un enregistrement est appelé type structuré.

L'enregistrement correspond à une nouvelle structure de données composée d'éléments de type déjà existants (prédéfinis ou nouvellement définis). L'enregistrement correspond, donc, à un nouveau type.

Pour pouvoir déclarer un enregistrement, il faut d'abord déclarer un nouveau type en utilisant les types existants.

La déclaration des types structurés se fait dans une section dédiée appelée **Type**, qui précède la section des variables.

3.3.1. Déclaration du type

Syntaxe

Pour déclarer un enregistrement dans un langage algorithmique, on utilise le mot clé **Type** selon la syntaxe suivante :

Type

nom_enreg = **Enregistrement**
  Champ1 : *type_de_données* ;
  Champ2 : *type_de_données* ;
  .
  .
  Champn : type_de_données ;
Fin;

Où :

nom_enreg : indique le nom de l'enregistrement ;

Champ1, …, Champn : indique les noms des champs de l'enregistrement.

type_de_données : indique les types associés aux champs.

Le mot clé **Enregistrement** indique que c'est une définition d'un enregistrement.

Le mot clé **Fin** indique la fin de la définition de l'enregistrement.

Pour déclarer un enregistrement dans un langage Pascal, on utilise aussi le mot clé **Type** selon la syntaxe suivante :

Type
nom_enreg = **record**
  Champ1 : *type_de_données* ;
  Champ2 : *type_de_données* ;
  .
  .
  Champn : *type_de_données* ;
end;

Exemple

La manipulation en une seule entité de certaines informations sur les étudiants (nom, prénom, âge, section) nécessite la déclaration d'un nouveau type.

Déclaration en langage algorithmique d'un nouveau type nommé etudiant.

```
Type
etudiant =enregistrement
  Nom : chaine de caractere ;
  Prenom : chaine de caractere ;
  Age : entier ;
  Section : caractère ;
Fin ;
```

La traduction de cette déclaration en langage pascal :

```
Type
etudiant =record
  Nom : string ;
  Prenom : string ;
  Age : Integer ;
  Section : char ;
end ;
```

3.3.2. Déclaration de la variable

Une fois le type de l'enregistrement déclaré, il est possible de déclarer des variables enregistrement appartenant au type déclaré.

La déclaration se fait de la même manière que la déclaration d'une variable de type prédéfini.

Exemple

Pour déclarer une variable enregistrement *ET1* de type *etudiant*, on utilise :

```
Variable
  ET1 : etudiant ;
```

ET1 est une variable appartenant au type etudiant préalablement déclaré.

3.4. Manipulation des enregistrements

La manipulation d'un enregistrement se fait via ses champs. Les enregistrements sont composés de plusieurs zones destinées à stocker les valeurs de chaque champ. Ainsi, la variable *ET1* de type *etudiant* déclarée précédemment peut être représentée comme suit :

	nom	prenom	age	section
ET1				

Le champ d'un enregistrement est accessible à travers son nom (identificateur) à l'aide de l'opérateur '.'. Ainsi, *ET1.nom* indique la valeur stockée dans le champ *nom* de la variable *ET1*.

Les champs d'un enregistrement sont considérés comme des variables. De ce fait, ils peuvent subir les mêmes opérations telles que l'affectation, la lecture et l'affichage.

3.4.1. L'affectation des champs d'un enregistrement

Pour affecter la valeur 'Rabah' au champ *nom* de l'enregistrement *ET1*, on utilise l'instruction suivante :

ET1.nom←'Rabah' ;

3.4.2. La lecture des champs d'un enregistrement

Pour remplir le champ *prenom* de l'enregistrement *ET1* par une valeur saisie par l'utilisateur, on utilise l'instruction suivante :

Lire(ET1.prenom) ;

3.4.3. L'ecriture des champs d'un enregistrement

Pour afficher la valeur stockée dans le champ *age* de l'enregistrement *ET1* sur l'écran, on utilise l'instruction suivante :

Ecrire (ET1.age) ;

Exemple

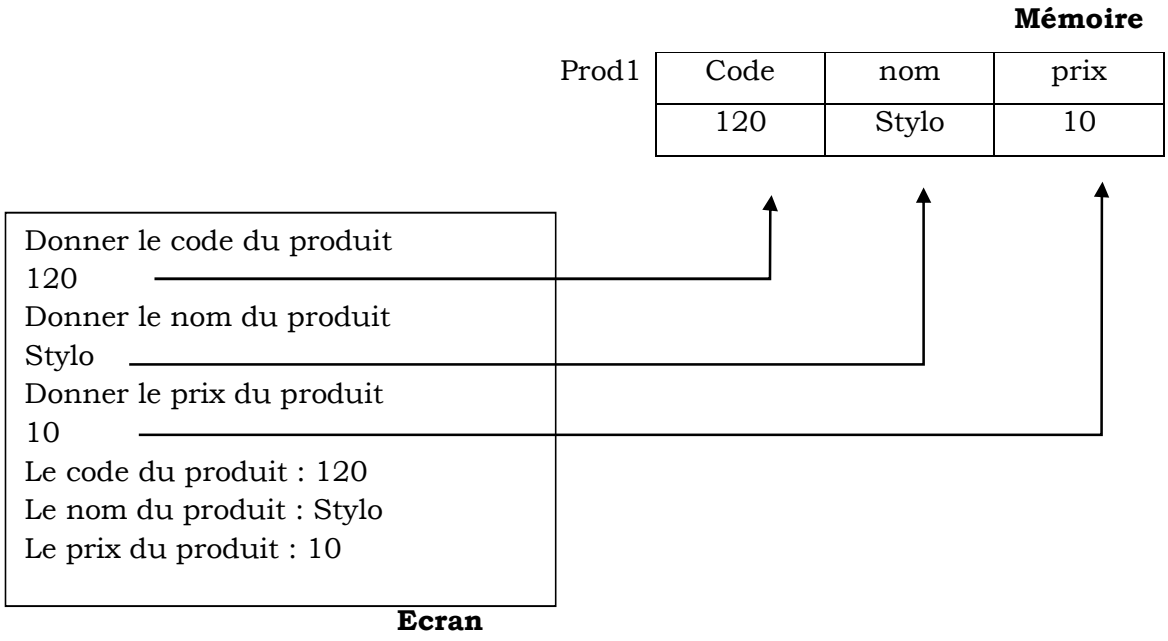
Ecrire un algorithme qui permet de :

- Définir le type d'un enregistrement nommé *Produit* composé des champs suivants : « Code » de type entier, « Nom » de type chaine de caractère, « Prix » de type réel.
- Déclarer une variable *prod1* de type produit.
- Remplir les champs de la variable *prod1* par des valeurs saisies par l'utilisateur.

- Afficher les champs de la variable *prod1* sur l'écran.

```
Algorithme exemple1 ;
Type
produit =enregistrement
  Code : entier ;
  Nom : chaine de caractere ;
  Prix : réel ;
Fin ;
Variable
 Prod1 : produit ;
Début
Ecrire ('Donner le code du produit') ;
Lire (prod1.code) ;
Ecrire ('Donner le nom du produit') ;
Lire (prod1.nom) ;
Ecrire ('Donner le prix du produit') ;
Lire (prod1.prix) ;
Ecrire ('Le code du produit :', prod1.code) ;
Ecrire ('Le nom du produit :', prod1.nom) ;
Ecrire ('Le prix du produit :', prod1.prix) ;
Fin.
```

Apres l'exécution de l'algorithme les états de la mémoire et de l'écran deviennent comme suit :



3.5. Affectation d'un enregistrement à un autre

Il est possible d'affecter un enregistrement à un autre.

Ainsi, si nous avons deux variables *prod1* et *prod2* de type *produit*, il est possible d'écrire : prod2←prod1.

Cette affectation permet d'affecter les valeurs de tous les champs de *prod1* aux champs correspondants dans *prod2*.

En d'autres termes cette affectation s'exécute comme suit :

Prod2.code←prod1.code ;

Prod2.nom←prod1.nom ;

Prod2.prix←prod1.prix ;

3.6. Les structures imbriquées

Selon les besoins, les structures de données peuvent être imbriquées les unes dans les autres. On peut, par exemple, imbriquer un enregistrement dans la structure d'un autre enregistrement ou d'un tableau, comme on peut aussi imbriquer un tableau dans la structure d'un enregistrement.

3.6.1. Un enregistrement comme champs dans un autre enregistrement

Supposons que, dans le type *etudiant* déclaré précédemment, nous voulions ajouter l'adresse de l'étudiant.

Le champ adresse est composé du numéro de la rue, du nom de la rue et de la ville. L'adresse peut être aussi représentée par un type enregistrement composé de 3 champs. Donc, il faut déclarer le type *adresse* au préalable, pour pouvoir ensuite l'utiliser dans la déclaration du type *etudiant,* en ajoutant à sa définition un champ *adr* de type *adresse*.

La déclaration sera comme suit :

```
Type
Adresse = enregistrement
    Num : entier ;
    Rue : chaine de caractere ;
    Ville : chaine de caractere ;
Fin ;
etudiant =enregistrement
    Nom : chaine de caractere ;
    Prenom : chaine de caractere ;
    Age : entier ;
    Section : caractère ;
    Adr : adresse ;
Fin ;
```

La nouvelle représentation du type *etudiant* sera comme suit :

	nom	prenom	Age	section	adr		
ET1					adr.num	adr.rue	adr.ville

Pour accéder à la ville de l'étudiant *ET1*, il faut utiliser ET1.adr.ville

3.6.2. Un tableau comme champ dans un enregistrement

Il est possible de déclarer un tableau dans un champ d'un enregistrement.

Supposons que nous voulions stocker les notes d'un étudiant en plus de son nom, son prénom son âge, sa section et son adresse. L'étudiant a cinq modules à suivre, il aura donc cinq notes.

Pour associer les notes au type *etudiant*, il suffit d'ajouter un champ *notes* de type tableau à la définition du type *etudiant*.

La nouvelle définition du type *etudiant* sera comme suit :

```
Type
etudiant =enregistrement
    Nom : chaine de caractere ;
    Prenom : chaine de caractere ;
    Age : entier ;
    Section : caractère ;
    Adr : adresse ;
    Notes : tableau [1..5] de réel ;
Fin ;
```

Pour remplir les différents champs d'un étudiant, on utilise l'algorithme suivant :

```
Algorithme exemple2 ;
Type
adresse= enregistrement
  Num :entier ;
  Rue :chaine da caractere ;
  Ville : chaine de caractere ;
Fin ;
etudiant =enregistrement
  Nom : chaine de caractere ;
  Prenom : chaine de caractere ;
  Age : entier ;
  Section : caractère ;
  Adr : adresse ;
  Notes : tableau [1..5] de réel ;
Fin ;
Variable
ET1 : etudiant ;
i : entier ;
Début
Ecrire ('Donner le nom de l"étudiant') ;
Lire (ET1.nom) ;
Ecrire ('Donner le prénom de l"étudiant') ;
Lire (ET1.prenom) ;
Ecrire (Donner l"âge de l"étudiant') ;
Lire (ET1.age) ;
Ecrire ('Donner la section de l"étudiant') ;
Lire (ET1.section) ;
Ecrire ('Donner l"adresse de l"étudiant') ;
Ecrire ('Donner le numéro de la rue') ;
Lire (ET1.adr.num) ;
Ecrire ('Donner la rue') ;
Lire (ET1.adr.rue) ;
Ecrire ('Donner la ville') ;
Lire (ET1.adr.ville) ;
Ecrire ('Donner les notes de l"étudiant') ;
Pour i de 1 à 5 faire
  Ecrire ('Donner la note n°',i) ;
  Lire (ET1.notes[i]) ;
Finfaire
Fin.
```

3.6.3. Les tableaux d'enregistrements

Il arrive souvent que l'on veuille traiter non pas un seul enregistrement mais plusieurs. Par exemple, on veut traiter un groupe de 100 étudiants. On ne va donc pas créer 100 variables du type *étudiant*, mais, on va créer un seul tableau qui peut contenir tous les étudiants. Il s'agit d'un tableau d'enregistrements,

Pour déclarer un tableau d'enregistrements, il faut d'abord que le type de l'enregistrement soit créé.

Pour déclarer un tableau *tab* composé de 100 enregistrements de type *etudiant*, on procèdera comme suit :

```
Type
adresse= enregistrement
  Num :entier ;
  Rue :chaine da caractere ;
  Ville : chaine de caractere ;
Fin ;
etudiant =enregistrement
  Nom : chaine de caractere ;
  Prenom : chaine de caractere ;
  Age : entier ;
  Section : caractère ;
  Adr : adresse ;
  Notes : tableau [1..5] de réel ;
Fin ;
Variable
  tab : tableau [1..100] de etudiant ;
```

Chaque élément du tableau *tab* est un enregistrement de type *etudiant*, contenant les mêmes champs du type *etudiant*.

On accède à un enregistrement par son indice dans le tableau :

tab[2] représente le 2ème étudiant.

tab[2].nom représente le nom du deuxième étudiant.

Pour remplir tous les enregistrements du tableau *tab*, il suffit de répéter l'algorithme précédent 100 fois à l'aide d'une boucle.

```
Algorithme exemple2 ;
Type
adresse= enregistrement
   Num :entier ;
   Rue :chaine da caractere ;
   Ville : chaine de caractere ;
Fin ;
etudiant =enregistrement
   Nom : chaine de caractere ;
   Prenom : chaine de caractere ;
   Age : entier ;
   Section : caractère ;
   Adr : adresse ;
   Notes : tableau [1..5] de réel ;
Fin ;
Variable
tab : tableau [1..100] de etudiant ;
i,j : entier ;
Début
Pour j de 1 à 100 pas 1 faire
   Ecrire ('Donner les informations de l"étudiant N° ',j) ;
   Ecrire ('Donner le nom de l"étudiant') ;
   Lire (tab[j].nom) ;
   Ecrire ('Donner le prénom de l"étudiant') ;
   Lire (tab[j].prenom) ;
   Ecrire (Donner l"age de l"étudiant') ;
   Lire (tab[j].age) ;
   Ecrire ('Donner la section de l"étudiant') ;
   Lire (tab[j].section) ;
   Ecrire ('Donner l"adresse de l"étudiant') ;
   Ecrire ('Donner le numéro de la rue') ;
   Lire (tab[j].adr.num) ;
   Ecrire ('Donner la rue') ;
   Lire (tab[j].adr.rue) ;
   Ecrire ('Donner la ville') ;
   Lire (tab[j].adr.ville) ;
   Ecrire ('Donner les notes de l"étudiant') ;
   Pour i de 1 à 5 pas 1 faire
      Ecrire ('Donner la note n°',i) ;
      Lire (tab[j].notes[i]) ;
   Fin faire
Fin faire
Fin.
```

Chapitre 4 : Les procédures et les fonctions

4.1. Introduction

En pratique, les problèmes sont généralement complexes ; d'où la nécessité de les décomposer en sous-problèmes qui peuvent être plus ou moins facilement résolus.

Cette décomposition peut être réalisée par ce qu'on appelle programmation modulaire. Il s'agit de décomposer l'algorithme (programme en langage pascal) en sous-algorithmes (sous-programmes en langage pascal) appelés modules plus lisible et facilement maitrisables.

Exemple

Nous voulons écrire l'algorithme qui permet de calculer la somme suivante :

$$s = 1 + \frac{2!}{2^2} + \frac{3!}{3^3} + \frac{4!}{4^4} + \cdots + \frac{n!}{n^n}$$

Pour cela, on aura besoin d'une boucle avec un compteur i ($1 \leq i \leq n$).

Et pour chaque valeur de i (itération), on devra calculer le factoriel de i (i !) et la puissance i^i , et même ces deux calculs nécessitent à leur tour deux autres boucles.

Pour rendre la solution plus simple, on pourra écrire deux sous-algorithmes, le premier pour le calcul du factoriel et le deuxième pour le calcul de la puissance. Ensuite pour chaque valeur du compteur *i*, on utilise (on fait appel à) ces deux sous-algorithmes.

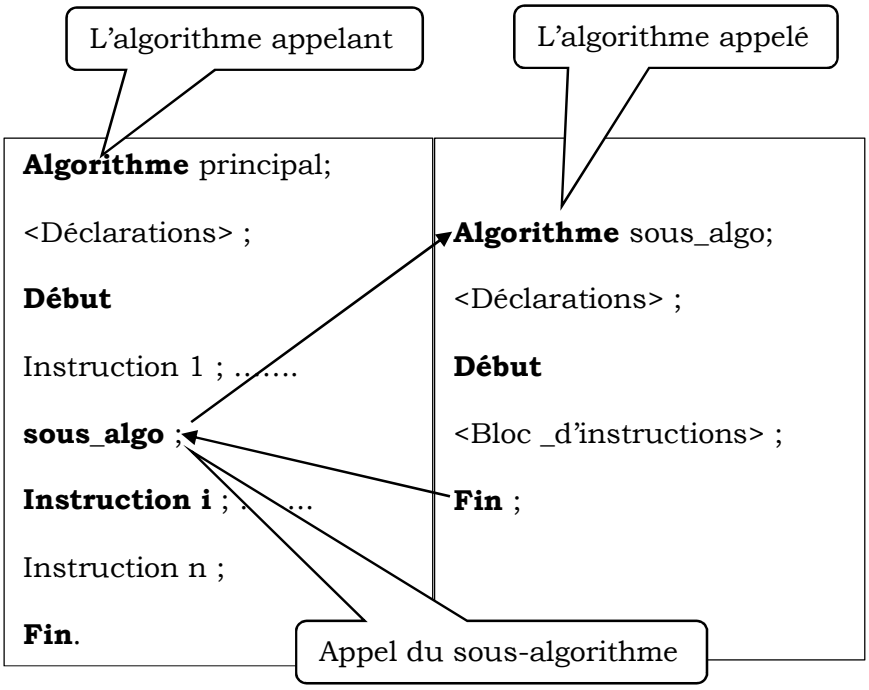
4.2. Les sous-algorithmes (sous-programmes)

Un sous-algorithme est un algorithme à l'intérieur d'un autre (appelé algorithme principal). Il possède donc la même structure qu'un algorithme (un entête, une partie déclarative et un corps de l'algorithme).

Un sous-algorithme peut être appelé (exécuter) par l'algorithme principal ou par un autre sous-algorithme pour réaliser le traitement qui lui est demandé et peut retourner des résultats.

Par exemple, en langage pascal, on fait souvent appel aux sous-programmes prédéfinis "*write*" et "*read*" qui réalisent les tâches qui leurs sont dédiées (de lecture et d'écriture) sans retour de résultats. Et on fait parfois appel à d'autres sous-programmes comme "*length*" et "*sqrt*" qui retournent des résultats (longueur d'une chaine, racine carré d'un nombre).

Le mécanisme d'appel d'un sous-algorithme peut être illustré comme suit :



L'algorithme principal s'exécute jusqu'à l'instruction **sous_algo** ; où il passe la main au sous-algorithme **sous_algo** (appel de **sous_algo**) pour qu'il puisse à son tour être exécuté.

Une fois l'exécution du sous-algorithme **sous_algo** terminée ; la main retourne à l'algorithme principal, et ce dernier poursuit son exécution à partir de l'instruction qui suit immédiatement l'instruction **sous_algo** (à partir de **Instruction i**).

- Les sous-algorithmes peuvent être différenciés par leurs comportements. Si le sous-algorithme retourne un résultat on parle alors de "fonction", sinon on parle de "procédure".
- La déclaration des sous-algorithmes se fait après la section des variables, dans la partie déclarative, de l'algorithme principal (ou de manière générale de l'algorithme appelant).
- Un sous-algorithme peut utiliser les variables déclarées dans l'algorithme principal ; on parle alors de variables globales.
- Les variables propres à un sous-algorithme (déclarées dans sa partie déclarative) ne peuvent être utilisées que dans ce sous-algorithme (elles ne peuvent pas être utilisées ailleurs) ; on parle alors de variables locales (pour le sous-algorithme).

- Seul l'algorithme (programme) principal se termine par un point après le mot **fin** ou **end** en pascal (**fin.** ou **end.** en pascal).
- Tous les autres sous-algorithmes (sous-programmes) se terminent par un point-virgule après le mot **fin** ou **end** en pascal (**fin;** ou **end;** en pascal).

4.3. Les procédures

Une procédure est un sous-algorithme déclaré en langage algorithmique comme suit :

Procedure *Nom_procedure* ([*prm1* : *type_prm*]….[;*prmn* : *type_prm*]) ;
<Déclarations> ;
Début
<Bloc _d'instructions> ;
Fin.

Où :

- *Nom_procedure* : indique le nom de la procédure.
- <*prm1*>, … <*prmn*> : indiquent les noms des paramètres
- *type_prm* : indique le type des paramètres de la procédure.

La structure d'une procédure est similaire à celle de l'algorithme, elle possède une partie déclarative et un bloc d'instructions. Sauf son entête (différent), qui est défini par le mot clé **procedure** en langages algorithmique et pascal, et peut être suivi par une liste de paramètres.

4.3.1. Notion de paramètres

Les paramètres d'une procédure sont déclarés comme si c'étaient des variables.
La liste des paramètres peut être vide, comme elle peut contenir un ou plusieurs paramètres, puisque leurs déclaration est optionnelle (dans la syntaxe, chaque paramètre est entre crochet).

Rappel

Tout ce qui est entre crochets "**[ ]**" est optionnel.

4.3.1.1. Procédure non paramétrée

Soit l'algorithme suivant, qui calcul et affiche la somme S=1+2+3+…+10.

```
Algorithme calcul ;
Variable
  S,i :entier ;
Debut
S←0 ;
Pour i de 1 à 10 faire
  s←s+i ;
finfaire
ecrire('somme=',s) ;
fin.
```

On peut modifier cet algorithme sans modifier sa sémantique, en introduisant une procédure qui effectue la somme. Le nouvel algorithme sera comme suit : Cet algorithme et le précèdent produisent le même résultat (ils affichent 'somme=55').

```
Algorithme calcul;
Variable
  S :entier ;
  Procedure somme;
  Variable
   i:entier;
  debut
   s←0 ;
   Pour i de 1 à 10 faire
     s←s+i ;
   finfaire;
  fin;
Debut
Somme ;
ecrire('somme=',s) ;
fin.
```

Variable globale

Déclaration de la procédure

Variable locale pour la procédure

Début de la procédure

Début de l'algorithme principal (calcul)

Appel de la procédure

- La déclaration de la procédure (*somme*) se fait dans la partie déclarative de l'algorithme principal (*calcul*).
- L'appel de la procédure s'effectue en précisant son nom (*l'instruction somme;).*
- La variable *S* peut être utilisée par la procédure (variable globale).
- La variable *i* ne peut être utilisée par l'algorithme principal (variable locale à la procédure).

4.3.1.2. Procédure paramétrée

La procédure déclarée précédemment est sans paramètres. On peut la modifier en ajoutant un paramètre.

```
Algorithme calcul;
Variable
  S :entier ;
  Procedure somme (n : entier);
  Variable
   i : entier;
  debut
   s←0 ;
   Pour i de 1 à n faire
     s←s+i ;
   finfaire;
  fin;
Debut
Somme(10);
ecrire('somme=',s) ;
fin.
```

Paramètre formel

Paramètre effectif

- La procédure *somme* possède un paramètre de type entier (*n*).
- L'appel de la procédure est aussi paramétré par la valeur *10*. La valeur 10 est transmise à *n*, ce qui nous permettra de calculer la somme de 1 à 10.
- Maintenant, on pourra tirer profit de cette nouvelle procédure, en faisant des appels avec comme paramètre n'importe quelle valeur entière et positive.
  Si on veut par exemple faire la somme de 1 à 50, il suffit d'appeler la procédure *somme* par un paramètre égale à 50 (l'instruction *somme(50);*) .

4.3.2. Les paramètres formels et effectifs.

- Les paramètres utilisés dans la déclaration de la procédure (listes des paramètres) sont appelés paramètres **formels**.
- Les paramètres utilisés lors de l'appel de la procédure sont appelés paramètres **effectifs**.

  Le paramètre effectif peut être :
  - une valeur : *somme (30);* .
  - une variable : somme *(a);* avec a=30.
  - une expression : *somme (20+10);* .

  Ces trois appels donnent le même résultat.

- Les paramètres effectifs doivent correspondre aux paramètres formels en type, en nombre et en ordre.

4.4. Les fonctions

Une Fonction est un sous-algorithme déclarée en langage algorithmique comme suit :

> **Fonction** *Nom_ fonction* ([*prm1* : *type_prm*]….[;*prmn* : *type_prm*]) : *Type_fonction* ;
> <Déclarations> ;
> **Début**
> <Bloc _d'instructions> ;
> *<Nom_ fonction>* ← *<resultat>*
> **Fin**.

Où :

- *Nom_ fonction*: indique le nom de la fonction.
- <*prm1*>, … <*prmn*> : indiquent les noms des paramètres.
- *type_prm* : indique le type des paramètres de la fonction.
- *Type_fonction* : indique le type de la fonction.
- *Resultat :* la valeur du résultat que doit retourner la fonction

- La structure de la fonction est similaire à celle d'une procédure, elle est constituée d'une partie déclarative et d'un bloc d'instructions.
- Son entête est défini par le mot clé **fonction** en langage algorithme et **function** en langage pascal, et il peut être suivi par une liste de paramètres.
- Même pour les fonctions, les paramètres effectifs doivent correspondre aux paramètres formels en type, en nombre et en ordre.
- Contrairement aux procédures, une fonction doit retourner un résultat à l'algorithme appelant. Voilà pourquoi, la fonction a un type, et la dernière instruction dans le corps de la fonction doit toujours être : <Nom_ fonction> ← <resultat>.
  Donc la valeur du résultat doit être de même type que celle de la fonction.
- La fonction ne doit retourner qu'une seule valeur à l'algorithme appelant.
- L'appel de la fonction s'effectue par une instruction d'affectation du nom de la fonction à une variable de l'algorithme appelant, pour pouvoir récupérer le résultat (*Nom_variable* ←*Nom_Fonction ;*).
-

Exemple

Reprenant l'exemple précèdent de la somme.

En utilisant une fonction pour retourner la somme, l'algorithme devient comme suit :

```
Algorithme calcul;
Variable
  S :entier ;
  fonction sum (n : entier) : entier;
  Variable
   i, r: entier;
  debut
    r←0 ;
    Pour i de 1 à n faire
       r ← r+i ;
    finfaire;
    sum←r ;
  fin;
Debut
s ←Sum(10);
ecrire('somme=',s) ;
fin.
```

Type de la fonction

Affectation du résultat (*r*) à la fonction

Appel de la fonction par affectation de la fonction à la variable globale **s**

- La déclaration de la fonction (*sum*) se fait dans la partie déclarative de l'algorithme principal (*calcul*).
- L'appel de la fonction s'effectue par l'instruction d'affectation (*s ←Sum(10);).*
- L'appel de la fonction est paramétré par la valeur *10* qui est transmise à *n*.
- La fonction *sum*, la variable globale *s* et la variable locale *r* doivent être de même type (des entiers).

4.5. Passage (transmission) des paramètres

Lors d'un appel, un passage de paramètres (transmission) s'effectue entre l'algorithme appelant et l'algorithme appelé.

On distingue deux modes de passage : passage par valeur et passage par variable.

4.5.1. Passage des paramètres par valeur

Dans les exemples vus précédemment, tous les passages des paramètres s'effectuaient par valeur.

En effet, lorsque la procédure *somme(n :entier)* ou la fonction *sum(n :entier)* sont évoquées respectivement par les instructions *somme(10)* ou *sum(10)*, la valeur *10* du paramètre effectif est transmise à la variable *n* du paramètre formel.

Il serait de même pour :

- Les appels *somme(x)* ou *sum(x)*, avec *x=20* ; c'est la valeur *20* de la variable *x* qui est transmise à la variable *n.*
- Et les appels *somme(10+5)* ou *sum(10+5)*, la valeur *15* est transmise à la variable *n*, après évaluation de l'expression *10+5.*

Dans le mode de passage par valeur, les paramètres formels sont initialisés par les valeurs des paramètres effectifs.

Les valeurs des paramètres effectifs ne peuvent être modifiées : les modifications apportées sur les paramètres formels n'ont aucun effet sur les paramètres effectifs.

La distinction du mode de passage des paramètres s'effectue au niveau de la déclaration des paramètres formels de la procédure ou de la fonction.

La déclaration d'un paramètre par valeur s'effectue selon les mêmes syntaxes de déclaration vues précédemment (puisque le mode de passage utilisé précédemment était par valeur) :

Procedure *Nom_Procedure* ([*prm1* : *type_prm*]….[;*prmn* : *type_prm*]) ;

Fonction *Nom_ fonction* ([*prm1* : *type_prm*]….[;*prmn* : *type_prm*]) : *Type_fonction* ;

4.5.2. Passage des paramètres par variable

Dans ce mode de passage, les paramètres effectifs doivent être des variables, pour permettre des modifications sur leurs contenus (leurs valeurs).

Ce mode de passage est utilisé si on veuille que la variable du paramètre effectif soit affectée par la variable du paramètre formel. Autrement dit, le passage par variable est utilisé lorsqu'on veut récolter le résultat (dans la variable du paramètre effectif).

Il ne s'agit donc pas d'utiliser uniquement la valeur de la variable, mais aussi de son adresse mémoire (l'emplacement mémoire). En effet, lors d'un appel d'une procédure ou d'une fonction l'emplacement mémoire de la variable du paramètre effectif est transmis au paramètre formel. Par conséquent, toute modification du contenu du paramètre formel affectera le contenu du paramètre effectif.

Pour déclarer un paramètre par variable, on précède le paramètre formel par le mot clé **var** en langages algorithmique et pascal.
Ainsi les déclarations s'effectuent selon les syntaxes suivantes :

Procedure *Nom_Procedure* ([**var** *prm1* : *type_prm*]….[; **var** *prmn* : *type_prm*]) ;

Fonction *Nom_ fonction* ([**var** *prm1* : *type_prm*]….[; **var** *prmn* : *type_prm*]) : *Type_fonction*;

- En langage pascal

Function *Nom_ fonction* ([**var** *prm1* : *type_prm*]….[; **var** *prmn* : *type_prm*]) : *Type_fonction*;

Les paramètres formels *prm1*…. *prmn* sont des paramètres par variables.

Remarques

- La combinaison entre les deux modes de passage est possible.
  Par exemple dans les deux déclarations suivantes :
  Procedure *proc* (**var** *p1,p2* : *entier* ; *p3* : *entier*) ;
  Les paramètres *p1* et *p2* sont des paramètres par variable. Tandis que *p3* est un paramètre par valeur.
  Function *fonct* (**var** *p4* : *entier* ; **var** *p5* : *real ; p6: entier*) : *entier*;
  Les paramètres *p4* et *p5* sont des paramètres par variable. Et *p6* est un paramètre par valeur.

- Le paramètre par valeur ne doit jamais être précédé par le mot clé **var**.

Exemple

Reprenons l'exemple de la somme.
En utilisant le mode de passage par variable, l'algorithme présenté dans la section 4.3.1.2 peut être modifié comme suit :

A chaque fois que la valeur de la variable du paramètre formel (*r*) change le contenu du paramètre effectif (s) change.

```
Algorithme calcul;
Variable
  S :entier ;
  Procedure somme (n : entier ; var r : entier);
  Variable
   i: entier;
  debut
    r←0 ;
    Pour i de 1 à n faire
      r←r+i ;
    finfaire;
  fin;
Debut
Somme(10 , s);
ecrire('somme=',s) ;
fin.
```

Paramètre par variable

Paramètre par valeur

4.6. Exercice d'application

Reprenons l'exemple énoncé en introduction :

Écrire l'algorithme (et le programme) qui permet de calculer et afficher la somme:

$$s = 1 + \frac{2!}{2^2} + \frac{3!}{3^3} + \frac{4!}{4^4} + \cdots + \frac{n!}{n^n}$$

Pour simplifier la solution, on va utiliser :

- une fonction (*fact*) qui calcule le factoriel d'un nombre.
- une procédure qui calcule la puissance (*power*) .
- et une boucle avec un compteur *i* (1 ≤ i ≤ n).
- A chaque itération du compteur *i*, on fait appel à la fonction (*fact*) et à la procédure (*power*) pour calculer la valeur de $\frac{i!}{i^i}$

- Pour calculer f=m !=1*2*3*…*m, il suffit d'utiliser la boucle suivante :

```
pour k de 1 à m faire
   f← f*k;
finfaire ;
```

- Pour calculer $r=x^y=x*x*x……*x$ (y fois), il suffit d'utiliser la boucle suivante :

  pour j de 1 à y faire

  r ← r*x;

  finfaire;

- Mais, pour notre cas (*ïï*), on doit calculer $r=\mathbf{x^x}=x*x*x……*x$ (x fois), la boucle sera modifiée comme suit :

  pour j de 1 à x faire

  r ← r*x;

  finfaire;

L'algorithme principal et sa traduction en pascal seront comme suit :

Langage algorithmique	Langage Pascal

```
Algorithme calcul_somme ;
variable
  i,n,p :entier ;
  s :reel ;

  Fonction fact(m:entier): entier;
  variable
     k, f: entier;
  debut
    f: ←1;
    pour k de 1 à m faire
       f← f*k;
    finfaire ;
    fact ← f;
  fin;

  Procedure power (var r: entier; x: entier);
  variable
     j:entier ;
  debut
    r←1;
    pour j de 1 à x faire
       r ← r*x;
    finfaire;
  fin;

debut
  ecrire ('Donner le nombre de termes');
  lire (n) ;
  s←0 ;
  pour i de 1 à n faire
     power(p,i);
     s ← s+fact(i)/p;
  finfaire;
  ecrire ('La somme =', s);
fin.
```

```
program calcul_somme ;
var
  i,n,p :integer ;
  s :real ;

  Function fact(m:integer): integer;
  var
     k, f: integer;
  begin
    f:=1;
    for k := 1 to m do
       f:= f*k;
    fact := f;
  end;

  Procedure power (var r: integer; x: integer);
  var
     j:integer ;
  begin
    r:=1;
    for j := 1 to x do
       r := r*x;
  end;

begin
  write ('Donner le nombre de termes');
  read (n) ;
  s:=0 ;
    for i := 1 to n do
    begin
       power(p,i);
        s := s+fact(i)/p;
    end;
  write ('La somme =', s);
end.
```

Remarque

Dans cet exemple, on a développé une procédure pour le calcul de la puissance (i^i), juste pour présenter un exemple de procédure. Mais, puisque le résultat de la puissance est une seule valeur, il vaut mieux utiliser une fonction.

Après modification, l'algorithme et le programme seront comme suit :

```
Algorithme calcul_somme ;
variable
  i,n :entier ;
  s :reel ;

  Fonction fact(m:entier): entier;
  variable
     k, f: entier;
  debut
    f: ←1;
    pour k de 1 à m faire
       f← f*k;
    finfaire ;
    fact ← f;
  fin;

  Fonction power (x: entier) : entier;
  variable
     r, j:entier ;
  debut
    r←1;
    pour j de 1 à x faire
       r ← r*x;
    finfaire;
    power←r ;
  fin;

debut
  ecrire ('Donner le nombre de termes');
  lire (n) ;
  s←0 ;
  pour i de 1 à n faire
     s ← s+fact(i)/power(i);
  finfaire;
  ecrire ('La somme =', s);
fin.
```

```
program calcul_somme ;
var
  i,n :integer ;
  s :real ;

  Function fact(m:integer): integer;
  var
     k, f: integer;
  begin
    f:=1;
    for k := 1 to m do
       f:= f*k;
    fact := f;
  end;

  Function power (x: integer): integer;
  var
     j,r:integer ;
  begin
    r:=1;
    for j := 1 to x do
      r := r*x;
    power:=r;
  end;

begin
  write ('Donner le nombre de termes');
  read (n) ;
  s:=0 ;
  for i := 1 to n do
     s := s+fact(i)/power(i);
  write ('La somme =', s);
end.
```

7.1.1 Le déroulement

calcul_somme				power			fact		
i	**n**	**p**	**s**	**r**	**x**	**j**	**f**	**M**	**k**
	3		0						
1		1		1	1				
		1		1		1			
							1	1	
			1				1		1
2		1		1	2				
		2		2		1			
		4		4		2			
							1	2	
							1		1
			1.5				2		2
3		1		1	3				
		3		3		1			
		9		9		2			
		27		27		3			
							1	3	
							1		1
							2		2
			1.7222				6		3

memoire

```
donner le nombre de termes
3
la somme =1.7222
```

Ecran

L'exécution est faite pour (3 itérations) :

- Pour la 1ere itération, s= 1.
- Pour la 2eme itération s= 1.5=$1 + \frac{2!}{2^2} = 1 + \frac{2}{4}$
- Et pour la 3eme itération s= 1.7222=$1 + \frac{2!}{2^2} + \frac{3!}{3^3} = 1 + \frac{2}{4} + \frac{6}{27}$

On remarque que la valeur de la variable *p* change à chaque fois que la valeur de la variable *r* change puisque le passage du paramètre *r* est par variable.

Chapitre 5 : La récursivité

5.1. Introduction

Pour simplifier l'écriture d'un algorithme plus ou moins complexe (pour résoudre un problème), on le décompose en sous-algorithmes en se basant sur la programmation modulaire.

Jusqu'à maintenant on n'a traité que les cas où des sous-algorithmes sont appelés par l'algorithme principal ou par d'autres sous-algorithmes. Mais, qu'en est-il des appels récursifs : lorsqu'on veut qu'un sous-algorithme fasse appel à lui-même ?

5.2. La récursivité

En littérature, un objet est dit récursif par définition, s'il est réutilisé directement ou indirectement dans sa définition.

En informatique, on parle de récursivité lorsqu'un sous-algorithme (procédure ou fonction) effectue un ou plusieurs appels à lui-même par une relation dite de récurrence. Autrement dit, cette situation correspond au cas où le nom d'un sous-algorithme figure parmi les instructions de son propre corps.

Généralement, les sous-algorithmes récursifs sont utilisés pour résoudre les problèmes de récurrences en mathématiques tels que la puissance, le factoriel, …

La récursivité peut souvent représenter une solution de remplacement pour les structures itératives. Elle consiste à remplacer (selon le cas), la boucle utilisée (Pour, Tant-que ou Répéter) dans le corps d'un sous-algorithme par un simple appel du nom de ce même sous-algorithme.

Exemple

Considérons la somme vue précédemment : S=1+2+3+……..+n.

Et reprenons le code de la procédure *somme* présenté dans la section10.3.1.2. où **s** est une variable globale.

```
Procedure somme (n : entier);
Variable
 i : entier;
debut
 s←0 ;
 Pour i de 1 à n faire
    s←s+i ;
  finfaire;
fin;
```

Pour n=5, S=1+2+3+4+5=15.

Ce calcul est réalisé par un simple appel de la procédure *somme(5).*

Il est à remarquer que :

Somme(5) = 5+ Somme(4)

= 5+ 4+ Somme(3)

= 5+ 4+ 3+ Somme(2)

= 5+ 4+ 3+ 2+ Somme(1)

= 5+ 4+ 3+ 2+ 1+ Somme(0)

= 5+ 4+ 3+ 2+ 1+ 0

De manière générale, *somme(n)=n + somme(n-1).*

somme(n-1)=n-1 + somme(n-2)

…….

somme(1)=0 + somme(0).

et *somme(0)=0*

Donc, la somme est exprimée comme suit :

$$Somme(n) = \begin{cases} \mathrm{n} + \mathrm{somme}(\mathrm{n}-1)\,, & n > 1 \\ 0\,, & n = 0 \end{cases}$$

En exploitant ces formules, le code de la procédure ci-dessus peut être modifié pour la rendre récursive.

```
Procedure somme (n : entier);
Variable
debut
 si n=0 alors
    s←0
 sinon
    somme(n-1) ;
    s←s+n ;
  finsi
fin;
```

Ainsi, un simple appel de la procédure *somme(5)* par l'algorithme principal génèrera une suite d'appels récursifs, illustrée par le schéma suivant :

1er appel	2eme appel	3eme appel	4eme appel	5eme appel	6eme appel	
algorithme principal	n=5	n=4	n=3	n=2	n=1	n=0
somme(5) →	somme(4) →	somme(3) →	somme(2) →	somme(1) →	somme(0) ⇄	s=**0**
s=15	s=10+5=**15**	s=6+4=**10**	s=3+3=**6**	s=1+2=**3**	s=0+1=**1**	
6eme retour	5eme retour	4eme retour	3eme retour	2eme retour	1er retour	

Sachant que la variable *s* est déclarée dans l'algorithme principal (variable globale).

1er appel :

L'algorithme principal effectue le premier appel (*somme(5)*),

La procédure teste la valeur de *n*.

Puisque *n=5≠0* la procédure effectuera un autre appel d'elle-même (*somme(4)*).

2eme appel :

La procédure teste la valeur de *n*.

Puisque *n=4≠0* la procédure effectuera un autre appel d'elle-même (*somme(3)*),

……

Et ainsi de suite jusqu'au 6eme appel (*somme(0)*).

6eme appel :

La procédure teste la valeur de *n* .

Puisque *n=0* la procédure n'effectuera aucun appel et affecte la valeur *0* à la variable *s*.

1er retour :

Une fois l'exécution de la procédure (*somme(0)*) terminée, on retourne vers la procédure appelante (*somme(1)*)

La nouvelle valeur de *s=s+n* est calculée : (*s=0+1=1*)

2eme retour :

Une fois l'exécution de la procédure (*somme(1)*) terminée, on retourne vers la procédure appelante (*somme(2)*)

La nouvelle valeur de *s=s+n* est calculée : (*s=1+2=3*)

…….

Et ainsi de suite jusqu'au 6eme retour (*algorithme principal*).

6eme retour :

Une fois l'exécution de la procédure (*somme(5)*) terminée, on retourne vers algorithme principal.

Et on obtiendra *s= 15*.

Remarque

- Il est impératif de prévoir une condition d'arrêt à la récursivité, sinon l'exécution du sous-algorithme récursif ne s'arrête jamais (une récurrence infinie). Cette condition d'arrêt doit assurer qu'à une étape donnée, le sous-algorithme ne fait plus appel à lui-même.

La condition (*si n=0*) est une condition d'arrêt à la récursivité. Elle assure que pour une valeur de *n=0* la procédure *somme* ne fera plus d'appel à elle-même et affecte directement la valeur *0* à la variable *s*.

Exemple1

Modifier les solutions de l'exercice de la section 10.6 du chapitre précédent, en utilisant des fonctions et des procédures récursives.

La somme à calculer est :

$$s = 1 + \frac{2!}{2^2} + \frac{3!}{3^3} + \frac{4!}{4^4} + \cdots + \frac{n!}{n^n}$$

Le factoriel est exprimé comme suit :

$$n! = \begin{cases} n * (n-1)!, & n > 1 \\ 1, & n = 0 \; ou \; n = 1 \end{cases}$$

Donc, la condition d'arrêt de la fonction récursive *fact* est "*si n=1*" .

Par exemple fact(3)= 3*fact(2)= 3*2*fact(1)= 3*2*1.

Donc si on veut calculer fact(3), cette dernière fait appel à fact (2), et fact (2) fait appel à fact(1) et ainsi de suite.

La puissance est exprimée comme suit :

$$x^y = \begin{cases} x * x^{y-1}, & y > 0 \\ 1, & y = 0 \end{cases}$$

Donc, la condition d'arrêt de la procédure récursive *power* est "*si y=0*" .

Par exemple 3^3=3*3^2= 3*3*3^1= 3*3*3*3^0= 3*3*3*1.

La procédure développée dans la solution du chapitre précédent, calculait la puissance x^x. Ce calcul est irréalisable en récursivité et la procédure doit être modifiée pour calculer la puissance x^y ; car la valeur de x doit rester fixe et seulement la valeur de y qui doit changer (décrémentée).

Après modification, l'algorithme et le programme seront comme suit :

```
Algorithme recursif ;
variable
  i,n,p :entier ;
  s :reel ;

Fonction fact(m:entier): entier;
debut
  si m=1 alors
    fact ←1
  sinon
    fact←m*fact(m-1) ;
  finsi ;
fin;

Procedure power(var r:entier;x,k: entier);
debut
  si k=0 alors
    r←1
  sinon
    power(r,x,k-1) ;
    r←x*r;
  finsi
fin;


debut
  ecrire ('Donner le nombre de termes');
  lire (n) ;
  s←0 ;
  pour i de 1 à n faire
    power(p,i,i);
    s ← s+fact(i)/p;
  finfaire;
  ecrire ('La somme =', s);
fin.
```

```
program recursif ;
var
  i,n,p :integer ;
  s :real ;

Function fact(m:integer): integer;
begin
  if m=1 then
    fact:=1
  else
    fact:= m*fact(m-1) ;
end;


Procedure power(var r:integer;x,k: integer);
begin
  if k=0 then
   r:=1
  else
  begin
    power(r,x,k-1) ;
    r:=x*r;
  end;
end;

begin
  write ('Donner le nombre de termes');
  read (n) ;
  s:=0 ;
  for i := 1 to n do
  begin
   power(p,i,i);
   s := s+fact(i)/p;
  end;
  write ('La somme =', s);
end.
```

Exemple2

Modifier la solution ci-dessus en remplaçant la procédure récursive *power* par une fonction récursive.

Après modification, l'algorithme et le programme seront comme suit :

```
Algorithme calcul_somme ;
variable
  i,n :entier ;
  s :reel ;

Fonction fact(m:entier): entier;
debut
  si m=1 alors
    fact ←1
  sinon
    fact←m*fact(m-1) ;
  finsi ;
fin;

Fonction power (x,k: entier) : entier;
debut
  si k=0 alors
    power ←1
  sinon
    power ← x*power(x,k-1) ;
  finsi
fin;

debut
  ecrire ('Donner le nombre de termes');
  lire (n) ;
  s←0 ;
  pour i de 1 à n faire
      s ← s+fact(i)/power(i,i);
  finfaire;
  ecrire ('La somme =', s);
fin.
```

```
program calcul_somme ;
var
i,n :integer ;
s :real ;

Function fact(m:integer): integer;
begin
  if m=1 then
    fact:=1
  else
    fact:= m*fact(m-1) ;
end;

Function power (x,k: integer): integer;
begin
  if k=0 then
    power:=1
  else
    power:= x*power(x,k-1) ;
end;

begin
  write ('Donner le nombre de termes');
  read (n) ;
  s:=0 ;
  for i := 1 to n do
      s := s+fact(i)/power(i,i);
  write ('La somme =', s);
end.
```

5.3. Exercice d'application

Écrire l'algorithme et le programme qui calculent le plus grand commun diviseur (PGCD) et le plus petit commun multiple (PPCM) de deux entiers positifs non nuls.

Remarque

Il existe 2 méthodes de calcul du PGCD :

- la méthode des soustractions.

- la méthode des divisions euclidiennes (d'Euclide).

Le PPCM de 2 nombres a et b se calcule comme suit :

PPCM(a,b)=(a*b) div PGCD(a,b).

5.3.1. Méthode des soustractions

Par la méthode des soustractions, le PGCD de 2 nombres x et y se calcule comme suit :

```
Tant que (x≠y) faire
  Si x>y alors
    x←x-y
  sinon
    y←y-x
  finsi
finfaire
pgcd←x
```

La condition d'arrêt pour cette méthode est "$si\ x=y$" .

On remarque qu'à chaque itération on remplace le plus grand des deux nombres par la différence entre eux. Et cela correspond au PGCD entre le plus petit nombre et la différence entre ces deux nombres.

Donc, le PGCD est calculé selon la formule suivante :

$$pgcd(x,y) = \begin{cases} x, & si\ x = y \\ pgcd(x-y,y), & si\ x > y \\ pgcd(x,y-x), & si\ x < y \end{cases}$$

1ere solution

Dans cette solution, on utilise une procédure récursive dans le code du calcul du PGCD. L'algorithme et le programme correspondants seront comme suit :

```
algorithme pgcd_ppcm ;
variable
  a,b,pg1,pp1 : entier;

procedure pgcd(x,y: entier; var pg2: entier);
debut
  si (x=y) alors
    pg2 ← x
  sinon
    si (x > y) alors
      pgcd (x-y, y ,pg2)
    sinon
      pgcd (x, y-x ,pg2);
    finsi
  finsi
fin;

debut
  ecrire ('donner le 1er entier');
  lire(a);
  ecrire ('donner le 2eme entier');
  lire(b);
  si (a<1) ou (b<1) alors
    ecrire ('donner des entiers >0')
  sinon
    pgcd(a, b, pg1);
    pp1← a*b div pg1;
    ecrire (' PGCD= ', pg1);
    ecrire (' PPCM= ', pp1);
  finsi
fin.
```

```
program pgcd_ppcm ;
var
  a,b,pg1,pp1 : integer;

procedure pgcd(x,y:integer; var pg2:integer);
Begin
  if (x=y) then
    pg2 := x
  else
    if (x > y) then
      pgcd (x-y, y ,pg2)
    else
      pgcd (x, y-x ,pg2);
End;

Begin
  Write ('donner le 1er entier');
  read(a);
  Write ('donner le 2eme entier');
  read(b);
  if (a<1) or (b<1) then
    write ('donner des entiers >0 ')
  else
  begin
    pgcd(a, b, pg1);
    pp1:= a*b div pg1;
    Write (' PGCD= ', pg1);
    write (' PPCM= ', pp1);
  end;
End.
```

2eme solution

Dans cette 2eme solution, on utilise une fonction récursive pour calculer le PGCD.

L'algorithme et le programme correspondants seront comme suit :

```
algorithme pgcd_ppcm ;
variable
  a,b,pg1,pp1 : entier;

Fonction pgcd (x,y:entier): entier ;
Début
  si (x=y) alors
    pgcd ← x
  sinon
    si (x > y) alors
      pgcd ← pgcd (x-y, y)
    sinon
      pgcd ← pgcd (x, y-x);
    finsi
  finsi
Fin.

debut
  ecrire ('donner le 1er entier);
  lire(a);
  ecrire ('donner le 2eme entier);
  lire(b);
  si (a<1) ou (b<1) alors
    ecrire ('donner des entiers >0')
  sinon
    pg1←pgcd(a, b);
    pp1:= a*b div pg1;
    ecrire (' PGCD= ', pg1);
    ecrire (' PPCM= ', pp1);
  finsi
fin.
```

```
program pgcd_ppcm ;
var
  a,b,pg1,pp1 : integer;

function pgcd(x,y:integer): integer;
Begin
  if (x=y) then
    pgcd := x
  else
    if (x > y) then
      pgcd:=pgcd (x-y, y)
    else
      pgcd:=pgcd (x, y-x);
End;



Begin
  Write ('donner le 1er nombre');
  read(a);
  Write ('donner le 1eme nombre');
  read(b);
  if (a<1) or (b<1) then
    write ('donner des entiers >0 ')
  else
  begin
    pg1:=pgcd(a, b);
    pp1:= a*b div pg1;
    Write (' PGCD= ', pg1);
    write (' PPCM= ', pp1);
  end;
End.
```

5.3.2. Méthode d'Euclide

La méthode d'Euclide calcule le PGCD de 2 nombres x et y comme suit :

```
Tant que (x mod y ≠ 0) faire
    x← y
    y← x mod y
finfaire
pgcd←y
```

La condition d'arrêt pour cette méthode est "*si x mod y =0*" .

On remarque qu'à chaque itération on remplace x par y et y par le reste de la division euclidienne entre x et y ($x \ mod \ y$).

Donc, le PGCD est calculé selon la formule suivante :

$$PGCD(x,y) = \begin{cases} y, & x \ mod \ y = 0 \\ PGCD(y, x \ mod \ y), & x \ mod \ y \neq 0 \end{cases}$$

1ere solution

Dans cette solution, on utilise une procédure récursive dans le code du calcul du PGCD.

L'algorithme et le programme correspondants seront comme suit :

```
algorithme pgcd_ppcm ;
variable
  a,b,pg1,pp1 : entier;

procedure pgcd(x,y:entier; var pg2:entier);
Début
  si (x mod y =0) alors
    pg2 ← y
  sinon
    pgcd (y, x mod y, pg2) ;
  finsi
Fin.

debut
  ecrire ('donner le 1er entier');
  lire(a);
  ecrire ('donner le 2eme entier');
  lire(b);
  si (a<1) ou (b<1) alors
    ecrire ('donner des entiers >0')
  sinon
    pgcd(a, b, pg1);
    pp1← a*b div pg1;
    ecrire (' PGCD= ', pg1);
    ecrire (' PPCM= ', pp1);
  finsi
fin.
```

```
program pgcd_ppcm ;
var
a,b,pg1,pp1 : integer;

procedure pgcd(x,y:integer; var pg2:integer);
Begin
  if (x mod y =0) then
    pg2 := y
  else
    pgcd (y, x mod y, pg2);
End;


Begin
  Write ('donner le 1er entier');
  read(a);
  Write ('donner le 2eme entier');
  read(b);
  if (a<1) or (b<1) then
    write ('donner des entiers >0 ')
  else
  begin
    pgcd(a, b, pg1);
    pp1:= a*b div pg1;
    Write (' PGCD= ', pg1);
    write (' PPCM= ', pp1);
  end;
End.
```

2eme solution

Dans cette 2eme solution, on utilise une fonction récursive pour calculer le PGCD. L'algorithme et le programme correspondants seront comme suit :

```
algorithme pgcd_ppcm ;
variable
  a,b,pg1,pp1 : entier;

Fonction pgcd (x,y:entier): entier ;
Début
  si (x mod y =0) alors
    pgcd ← y
  sinon
    pgcd←pgcd (y, x mod y)
  finsi
Fin.

debut
  ecrire ('donner le 1er entier);
  lire(a);
  ecrire ('donner le 2eme entier);
  lire(b);
  si (a<1) ou (b<1) alors
    ecrire ('donner des entiers >0')
  sinon
    pg1←pgcd(a, b);
    pp1:= a*b div pg1;
    ecrire (' PGCD= ', pg1);
    ecrire (' PPCM= ', pp1);
  finsi
fin.
```

```
program pgcd_ppcm ;
uses wincrt;
var
a,b,pg1,pp1 : integer;

function pgcd(x,y:integer): integer;
Begin
  if (x mod y =0) then
    pgcd := y
  else
    pgcd:=pgcd (y, x mod y)
End;

Begin
  Write ('donner le 1er nombre');
  read(a);
  Write ('donner le 1eme nombre');
  read(b);
  if (a<1) or (b<1) then
    write ('donner des entiers non nuls')
  else
  begin
    pg1:=pgcd(a, b);
    pp1:= a*b div pg1;
    Write (' PGCD= ', pg1);
    write (' PPCM= ', pp1);
  end;
End.
```

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.

- I.MALTSEV et M.ANDRé CHAUVIN, *Algorithmes et Fonctions Récursives*, office des publications universitaires, 1980.

- Mounira Belmesk, *Algorithmes et structures de données*, Khawarysm, 1991.

- Patrick Cousot, *Algorithmique et programmation en PASCAL : Exercices et corrigés*, Berti, 1993