Chapitre1 : Les sous-programmes : Procédures et Fonctions

1. Introduction

- La résolution d'un problème informatique se décompose en 4 Phases : Analyse, écriture d'algorithme, programmation, compilation et exécution.
- Donc, Un algorithme est une solution à une classe de problèmes.
- Parfois, le problème à résoudre est trop sophistiqué (compliqué) c.à.d:
  - ✓ Il devient difficile d'avoir une vision globale pour résoudre ce problème.
  - ✓ Le programme (l'algorithme) devient de grande taille (en un seul bloc), et généralement très difficile à comprendre, a trouver des erreurs, à développer et à lire.
- Dans ce cas, il est conseillé de décomposer le problème en sous problèmes, puis trouver une solution à chacun.

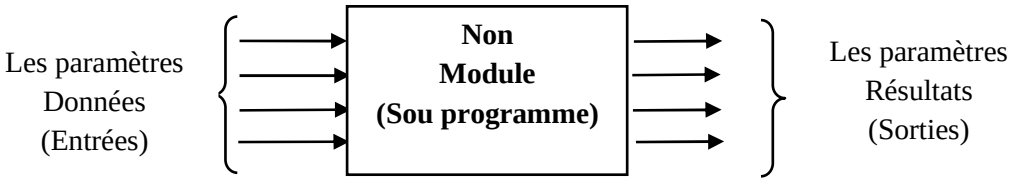
Exemple : **(problème et sous problèmes)**

On veut réaliser un programme permettant de lire les notes d'examen, TD et TP des étudiants en module initiation a l'algorithmique, calculer leurs moyennes et dire pour chaque étudiant s'il est admis ou ajourné dans ce module?

- On peut décomposer ce problème en 3 sous problèmes :
  - ✓ ***Sous problème 1*** : lire les notes des étudiants
  - ✓ ***Sous problème 2*** : calculer les moyennes des étudiants
  - ✓ ***Sous problème 3*** : dire si étudiant est admis ou ajourné

2. Sous programmes *(modules)* :

- Un sous-programme (sous algorithme) est un programme (algorithme) qui décrit la solution d'un sous problème.
- Schématiquement un module (sous-programme) est représenté par une boite noir qui a des entrées, des sorties et un rôle bien précis comme suit :

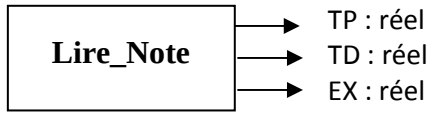


Rôle :……..

***Exemple* 1:**

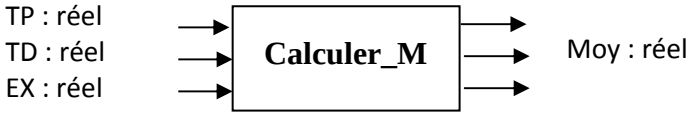
On veut écrire un algorithme permettant de lire les notes (examen, TD et TP) des étudiants dans le module Algorithmique et Structures de données 1, calculer leurs moyennes et dire pour chaque étudiant s'il est **admis** ou **ajourné** pour ce module.
Faire le découpage modulaire nécessaire ?

Module 1 :



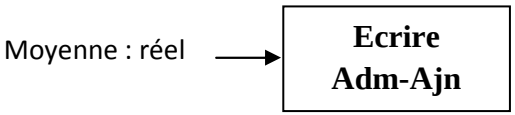
Rôle : lire les notes d'un étudiant

Module 2 :



Rôle : Calculer la moyenne depuis les notes

Module 3 :

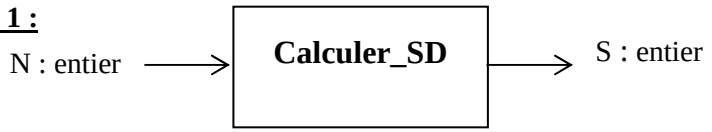


Rôle : Dire si l'étudiant est admis ou ajournée

Exemple 2:

Un nombre **déficient** est un nombre entier naturel **n** qui est strictement supérieur à la somme de ses diviseurs stricts. On veut écrire un algorithme qui lit un nombre entier **X** et affiche tous les nombres déficients **inférieurs à** X.
Faire le découpage modulaire nécessaire ?

Module 1 :

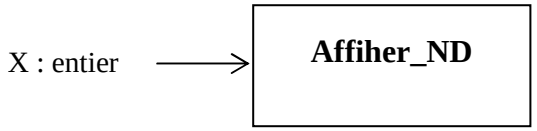


Rôle: calculer la somme des diviseurs stricts d'un nombre entier N

Module 2:

Rôle : vérifier si un nombre entier N est déficient ou non

Module 3 :



Rôle : lire un nombre entier X et affiche tous les nombres déficients inférieurs à X

Remarque :

- Au moment de l'élaboration d'un découpage modulaire on ne cherche pas la réponse de la question ***comment faire?*** Mais plus tôt ***Quoi Faire*** ? i.e. identification du rôle précis de chaque module.
- Il existe deux types de sous programmes (modules) : **les procédures** et **les fonctions**.

2.1. Procédures:

- Une procédure est un sous-programme (sous algorithme), qui peut être appelé (utilisé) dans un autre programme (algorithme) ou dans différents lieux du même programme (algorithme).
- Une procédure peut être appelée comme une instruction dans un programme (algorithme) par l'intermédiaire de **son nom**.

a) Déclaration d'une procédure :

- Une procédure est définie dans la **partie déclarative de l'algorithme**.
- Une procédure est constituée **d'un entête**, des **déclarations de variables** (si elles existent), et **d'un corps**.

Syntaxe :

Procédure <nom de la procédure> (Liste des paramètres)

<Partie déclaration>

Début

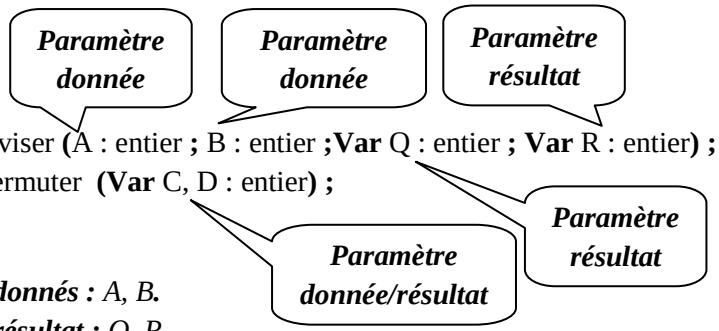
<Corps de la procédure>

Fin ;

Les paramètres d'une procédure : Les paramètres sont des ***variables***. Chaque paramètre est décrit par :

- Un nom,
- Un type,
- Mode de transmission : paramètre donnée, paramètre résultat, ou bien paramètre donnée /résultat.

Exemple :



Les paramètres donnés : *A, B.*
Les paramètres résultat : *Q, R.*
Les paramètres donnée /résultat : *C, D.*

Pour l'exemple précédent :
Procédure calculer_M (TD, TP, EX : réel ; **Var** Moy : réel) ;
ProcédureVérifier_ND(N : entier ; **Var**R : booléen) ;

Remarque :

Le mot clé **Var** indique que les paramètres sont des sorties (résultats), et qui peut être en entrées aussi.

Exemple :

Solution 1 : sans utilisation d'une procédure	**Solution 2 : utilisation d'une procédure**
Algorithme addition A, B, som : réel ; **Début** Lire (A, B) ; som←A+B ; Ecrire (som) ; **Fin.**	**Algorithme** addition A, B, som : réel ; **Procédure** somme (X, Y : réel ; Var S : réel) S : réel ; **Début** S ← X+Y ; **Fin** **Début** Lire (A, B) ; somme (A, B, som) ; Ecrire (som) ; **Fin.**

Appel de la procédure

b) Appel d'une procédure :

- L'appel d'une procédure se fait par le biais de son nom :
  - ✓ Les paramètres indiqués dans la déclaration des sous-programmes sont appelés **« Paramètres formels ».**
  - ✓ Les paramètres indiqués dans l'appel des sous-programmes sont appelés
  - ✓ **« Paramètres effectifs »**.
- Lors de l'appel il faut que l'ordre des paramètres effectifs soit conforme à celui des paramètres formels.

Exemple : exemple précédent

X, Y : paramètres **formels.**
A, B : paramètres **effectifs**.

2.2. Fonctions :

- Une fonction est un cas particulier de procédures, contrairement à la procédure, la fonction doit avoir un ***type*** c'est-à-dire doit remettre une ***valeur*** en sortie.

a) Déclaration d'une fonction :

Fonction <nom de la fonction> (Liste de paramètres en entrée) : **Type** ;

<Partie déclaration>

Début

<Corps de la fonction>

Retourner (valeur de la sortie) ;

Fin ;

b) Appel de fonction :

L'appel d'une fonction se fait de la même façon qu'une procédure sauf que le nom de la fonction contient directement la valeur de retour (valeur de sortie).

Exemples:

Fonction Carré (X : réel) **:** réel ; retourne le carré du nombre X

Pour les exemples précédents :

Fonction calculer_M (TD, TP, EX : réel) : réel ;

Fonction Vérifier_ND (N : entier) : booléen ;

Ecrire un algorithme qui fait l'adition de deux nombres réels ?

Solution 1 : sans utiliser la fonction	**Solution 2 : utiliser la fonction**
Algorithme addition A, B, som : réel ; **Début** Lire (A, B) ; som←A+B ; Ecrire (som) ;	**Algorithme** addition A, B, som : réel ; **Fonction** somme (X, Y : réel) : réel ; S : réel ; **Début** S ← X+Y ; Retourner (S) ; **Fin**

Fin.	**Début** Lire (A, B) ; som ← somme (A, B) ; Ecrire (som) ; **Fin.**

Remarque :

- L'entête d'une fonction se termine toujours par **le type** de la valeur remise par la fonction.
- Le corps d'une fonction se termine toujours par l'instruction ***Retourner*** (valeur de la sortie) qui permet de retourner la valeur de la sortie au ***programme appelant***.

3. Les variables globales et les variables locales :

- Une ***variable globale*** est une variable déclarée dans le programme (algorithme) principal et peut être utilisée par une ou plusieurs procédures ou fonctions.
- Une ***variable locale*** est une variable déclarée et utilisée dans un sous programme (procédure ou une fonction).

Exemple :

```
Algorithme util-Proc
    A, B : entier ;
Procédure Permuter (Var x, y: réel);
Z: réel ;
      Début
        Z ←x ;
        x← y ;
y ← Z ;
      Fin
Début
   Lire (A, B) ;
   Permuter (A, B) ;
   Ecrire (A, B) ;
Fin.
```

Variables globales

Variables Locales

Les variables globales : A, B.
Les variables locales : X, Y, **Z**.

4. Le passage des paramètres :

- Il faut rappeler qu'une ***variable*** dans un programme est un espace mémoire destiné à stocker une ***valeur***.
- Une variable possède un ***nom*** (identificateur) et une ***adresse*** mémoire.

Variable1 (adresse1)	Valeur1
Variable2 (adresse2)	Valeur2
.	.
.	.
.	.
Variable n (adresse n)	Valeur n

- Les identificateurs représentent les paramètres dans la ***déclaration*** des sous programmes ne sont pas les mêmes a ***l'appel***.
- Il existe deux types de passage (transmissions) de paramètres :
  a) Le passage par valeur
  b) Le passage par variable (ou par adresses)

4.1. Le passage Par valeur :

- Les valeurs des paramètres effectifs sont copiées dans les paramètres formels de sous programme sans toucher les valeurs d'origines.
- Dans ce cas, se sont les variables locales dans le sous programme appelée qui sont utilisées.
- La modification des variables locales dans le sous programme ne modifie pas les variables passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de ces dernières.

Exemple :

```
Algorithme P_Valeur
    A, B : réel ;
    Procédure Permuter (X, Y: réel);
        Z : réel ;
```

Début

Z ← X ;
X ← Y ;
Y ← Z ;

Permutation des valeurs

A←X ;
B← Y ;

Restitution des valeurs

Fin ;

Début
A ← 3;
B ← 5;
Permuter (A, B) ;
Ecrire (A, B) ;
Fin.

Le passage des paramètres se fait par valeur :

- Les valeurs des paramètres effectifs 'A' et 'B' sont copier dans les paramètres formels 'X' et 'Y' lors de l'appel : A → X, B → Y
- Les modifications (permutation) sont apportés seulement sur les variables locales (X, Y, Z).
- Finalement, Les valeurs des paramètres formels 'X' et 'Y' sont restituées dans les paramètres effectifs 'A' et 'B'.

4.2. Le passage Par adresse :

- Cette technique consiste à passer non plus les valeurs des paramètres effectifs, mais à passer les variables elles-mêmes (son emplacement dans la mémoire).
- Il n'y a donc plus de copie, toute modification des paramètres formels dans le « sous programme appelé » entraîne la modification des paramètres effectifs (variables passés en paramètre).

Exemple :

Algorithme P_Variable
A, B : entier ;
Procédure Permuter (**Var** x, y: réel);
Z: réel ;
Début
Z ←x ;
x← y ;
y ← Z ;
Fin ;

Début
A ← 3;
B ← 5;
Permuter (A, B) ;
Ecrire (A, B) ;
Fin.

- Les adresses des paramètres effectifs 'A' et 'B' sont passées à la procédure lors de l'appel.
- Les modifications (permutation) sont apportées sur les variables A, B, et Z.

5. Différences entre procédures et fonctions

Procédure	Fonction
En-tête : Procédure <Nom_Proc> (liste de paramètres) ;	**En-tête :** Fonction <Nom_Fonc> (liste de paramètres) : <Type> ;
Exemple d'utilisation : (appel) Nom_proc (liste paramètres) ;	**Exemple d'utilisation : (appel)** ***Affectation*:** x ←nom_f (liste paramètres) ; ***Écriture*:** Ecrire (nom_f (liste paramètres)) ;
Passage de paramètres : - Passage par valeur - Passage par adresse	**Passage de paramètres :** - Passage par valeur

6. Avantages d'utilisation des procédures et fonctions

- Voici quelques avantages d'une programmation modulaire:
  - ✓ ***Minimisation de la duplication de code***
  - ✓ ***Meilleure lisibilité***
  - ✓ ***Diminution du risque d'erreurs***

- ✓ ***Possibilité de tests sélectifs :*** (module par module)
- ✓ ***Réutilisation de modules déjà existants :*** Il est facile d'utiliser des modules qu'on a écrits soi-même ou qui ont été développés par d'autres personnes.
- ✓ ***Simplicité de l'entretien :*** Un module peut être changé ou remplacé sans devoir toucher aux autres modules du programme.
- ✓ ***Favorisation du travail en équipe*** : Un programme peut être développé en équipe par division et affectation des modules à différentes personnes ou groupes de personnes.

Exemple **1:** Écrire un algorithme qui lie trois nombres positifs non nuls A, B, C puis calcule et affiche la somme suivante : (A ! + B !+ C !) !

Solution1	Solution2
Algorithmeexpemple1 A, B, factA, factB,factC,Somme, fact : entiers ; **Début** Lire (A, B); factA ← 1; **Pour** i**allant de** 1 **a Afaire** factA ← factA * i; **Fin pour** factB ← 1 ; **Pour** i**allant de** 1 **a** B **faire** factB ← factB * i; **Fin pour** factC ← 1 ; **Pour** i**allant de** 1 **a**C**faire** factC← factC * i; **Fin pour** Somme ← factA + factB + factC; fact ← 1 ; **Pour** i**allant de 1 a**somme **faire** fact ← fact * i; **Fin pour** Ecrire (fact) ; **Fin.**	**Algorithme**exemple1 A, B, C : entiers ; Fonction factorielle (N : entier ;) : entier ; factN, i : entier ; **Début** factN ← 1; **Pour** i**allant de** 1 **a Nfaire** factN ← factN* i; **Fin pour** Retourner (fact) ; **Fin**. **Début** Lire (A, B); Ecrire (factorielle (factorielle (A) +factorielle (B) factorielle (C))); **Fin.**

Exemple 2:

Revenons au problème des nombres déficients. On veut écrire un algorithme qui lit un nombre entier X et affiche tous les nombres déficients inférieurs à X ?

7. Appels imbriquées

Soit les fonctions suivantes :

$$\begin{cases} F(x) = 3\,x^2 \\ G(x) = 7\,x \\ H(x) = F(x) + G(x) \end{cases}$$

Q1) Écrire un algorithme qui lit un nombre réel **a** et affiche : F(a), G(a) et H(a).

Q2) Réécrire l'algorithme en remplaçant les deux fonctions F et H par deux procédures

8. La récursivité

8.1. Notion de la récursivité :

- En programmation, la ***récursivité*** est une méthode qui permet à un sous-programme (procédure ou fonction) de s'appeler elle-même.
- C'est Dans <u>la partie corps</u> (instructions) on retrouve un appel à la procédure (fonction) elle-même.

Exemple : écrire un algorithme (fonction) permettant de calculer la factorielle d'un entier N donné ?

a) Solution itérative (classique) :

N !=1*2* 3*4*5*…..... *(N-1)*N.

Fonction fact (N : entier):Entier ;

R, i : entier ;

Debut

R ← 1;

Pour i allant 2 à N **faire**

R ← R* i ;

FinPour

Retourner (R) ;

Fin.

b) Solution récursive:

N ! = N*(N-1) !

= N*(N-1)*(N-2) !

= N*(N-1)*(N-2)* .....* 0!

- la fonction factorielle « ***fact*** » est définie comme suit :
  - ✓ Si N=0: fact(0)=1.
  - ✓ Si N<> 0: fact(N)=N*fact(N-1).
- En algorithmique la fonction ***fact*** est définie comme suit :

Fonction fact (N : entier):Entier ;

R : entier ;

Début

Si (N = 0) alors

R ← 1;

Sinon

R ← N * fact (N - 1);

Appel récursive

Finsi

Retourner (R) ;

Fin.

Appel à fact(4)
4*fact(3) = ?
Apple à fact(3)
3*fact(2) = ?
Appel à fact(2)
2*fact(1) = ?
Appel à fact(1)
1*fact(0) = ?
Appel à fact(0)
Retour de la valeur 1
1*1
Retour de la valeur 1
2*1
Retour de la valeur 2
3*2
Retour de la valeur 6
4*6
Retour de la valeur 24

8.2. Types de récursivité :

- On distingue en général deux types de récursivité :
  - ✓ La récursivité **simple**
  - ✓ La récursivité **croisée**

a) La récursivité simple :

- C'est lorsqu'un sous-programme (fonction ou procédure) s'appelle lui-même. C'est en effet le cas général de récursivité comme on l'a déjà vu dans l'exemple précédent avec la fonction **factorielle**.

Syntaxe :

Procédure P
Début
…
Appel de P ;
…
Fin ;

Exemple : calcul de la somme de **n** premiers nombres entiers positifs.

Somme(n) = 1 + 2 + 3 + … + (n-1) + n

Fonction Somme (n : entier) : entier ;
S: entier ;
Début
Si (n = 1) alors
S←1 ;
Sinon
S ← Somme (n-1) + n ;
Finsi
Retourner (S) ;
Fin.

b) La récursivité croisée :

- On appelle récursivité croisé le fait que **deux procédures** P1 et P2 s'appellent **mutuellement**, c.-à-d : lorsque **P1** s'exécute, elle fait appel à **P2**, et lorsque **P2** s'exécute, elle fait appel à **P1**.

Syntaxe :

Procédure P1
Début
…
Appel de P2 ;
…
Fin ;
Procédure P2
Début
…
Appel de P1 ;
…
Fin ;

Exemple :

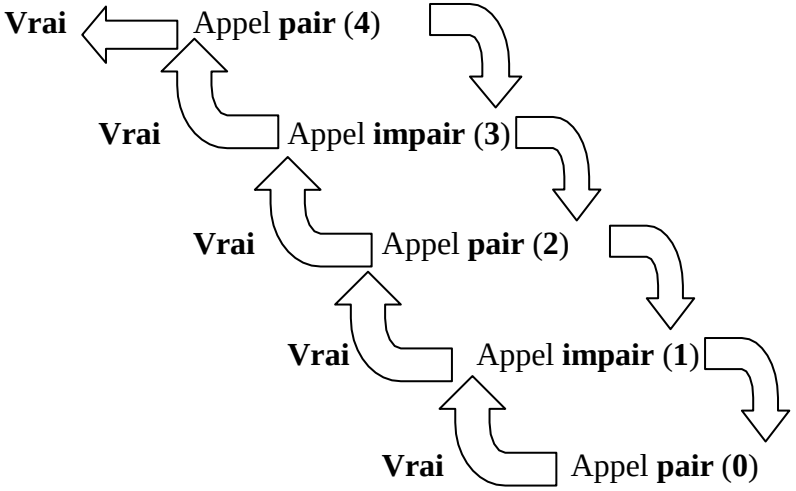
- Un nombre entier positif n peut être soit :

Pair → n = 2*k
Impair → n = 2*k+1

- Si on considère deux fonctions **Pair** (n) et **Impair** (n) à valeurs logiques (**booléen**) alors on aura :
  Si **Pair** (n) = vrai alors **Impair** (n) = faux
  Si **Impair** (n) = vrai alors **Pair** (n) = faux

Fonction Pair (n : entier): booléen ;	Fonction Impair (n : entier): booléen;
R: entier ;	R: entier ;
Début	**Début**
Si (n = 0) alors	**Si** (n = 0) alors
R ← vrai ;	R ← faux ;
Sinon	**Sinon**
R ← Impair (n-1) ;	R ← Pair (n-1) ;
Finsi	**Finsi**
Retourner (R) ;	Retourner (R) ;
Fin ;	**Fin.**

Application : on veut vérifier **pair (4) :**



8.3. Règles de conception :

a) Première règle :

- Chercher à décomposer le problème en plusieurs sous problèmes de même type mais de taille inférieure.

b) Deuxième règle :

- Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif (***Condition d'arrêt ou terminaison***). C'est le cas **trivial**, sinon risque de boucler infiniment.

Condition de terminaison :

- ✓ Les cas non récursifs d'un algorithme récursif sont appelés ***cas de base***.
- ✓ Les conditions que doivent satisfaire les données dans ces cas de base sont appelées **conditions de terminaison**.

Exemple :

Fonction fact (N : entier):Entier ;
R : entier ;
Début
Retourner (N * fact (N - 1)) ;
Fin.

Appel à fact(4)
4*fact(3) = ?
Apple à fact(3)
3*fact(2) = ?
Appel à fact(2)
2*fact(1) = ?
Appel à fact(1)
1*fact(0) = ?
Appel à fact(0)
0*fact(- 1) = ?
…

Pas de conditions
De terminaison ?!

Solution :

```
Fonction fact (N : entier):Entier ;
R : entier ;
Début
Si ( N = 0 ) alors
R ← 1;
Sinon
        R ← N * fact (N - 1);
Finsi
   Retourner (R) ;
Fin.
```

Cas de Base

Remarque :

- Puisqu'une fonction récursive s'appelle elle-même, il est impératif qu'on prévoie une condition d'arrêt à la récursivité, sinon le programme ne s'arrête jamais.
- On doit toujours tester en premier la condition d'arrêt, et ensuite, si la condition n'est pas vérifiée, lancer un appel récursif.

c) Troisième règle :

- Tout appel récursif doit se faire avec des données plus ‘’ proches ‘’ de données satisfaisant une condition de terminaison.

Avantages de la récursivité:

- Simplifier l’écriture des programmes, car les calculs à effectuer ne sont pas définis explicitement.
- Faciliter la tâche du programmeur qui n’aura plus à préciser le nombre de répétitions de la même action, ni à gérer les valeurs des différentes variables utilisées.