



Cours ASD 1

Pour les étudiants de la 1ère année Licence LCS & IOT

Préparé par :

Dr. Naama AMDOUNI

PLAN DU COURS

Chapitre 1 : Introduction à l'Algorithmique

Chapitre 2 : Les Instructions Simples

Chapitre 3 : Les Structures Conditionnelles

Chapitre 4 : Les Structures Itératives

Chapitre 5 : Les Sous-Programmes

Chapitre 6 : Les Tableaux

Chapitre 7 : Les Algorithmes de Tri et de Recherche

Chapitre 8 : La Récursivité

Chapitre 1 : Introduction à l'Algorithmique

Objectifs du chapitre :

- Comprendre le cycle de résolution d'un problème informatique et le rôle de l'algorithme dans ce cycle.
- Connaître la structure générale d'un algorithme.

I- Cycle de vie d'un programme

Pour résoudre un problème en informatique, il faut passer par les étapes suivantes :

1. Définition et analyse du problème

Il s'agit de :

- Définir les données qu'on dispose et les objectifs qu'on souhaite atteindre
- Prévoir des réponses à tous les cas envisageables

Exemple : Si le problème est la résolution d'une équation de second degré $ax^2+bx+c=0$

- Les données sont a, b et c
- Les sorties sont x1 et x2
- Les cas : $a=0$ et $b \neq 0$, $a \neq 0$ et $b=0$, $a \neq 0$ et $b \neq 0$

2. Ecriture de l'algorithme

C'est la phase la plus difficile et la plus importante. Elle fournit la méthode et la démarche que l'ordinateur va suivre pour résoudre le problème posé.

Définition d'un algorithme :

Un algorithme est un ensemble d'étapes successives, finies et ordonnées et qui visent à résoudre un problème bien défini.

Un algorithme doit vérifier les propriétés (caractéristiques) suivantes :

- **Clair** : chaque instruction de l'algorithme ne doit pas présenter des ambiguïtés (càd elle ne doit pas être interprétable de plusieurs manières : elle ne doit avoir qu'un seul sens).
- **Déterministe** : pour un ensemble de données initiales, l'algorithme doit fournir le même résultat, quel que soit le moment d'exécution, et avec n'importe quelle machine.
- **Complet** : Il faut que l'algorithme considère tous les cas possibles et donne un résultat dans chaque cas.
- **Fini** : il doit se terminer quel que soit les données d'exécution, le moment et la machine.
- **Efficace** : l'algorithme doit effectuer le travail demandé avec le minimum de ressources (espace mémoire, temps d'exécution...).

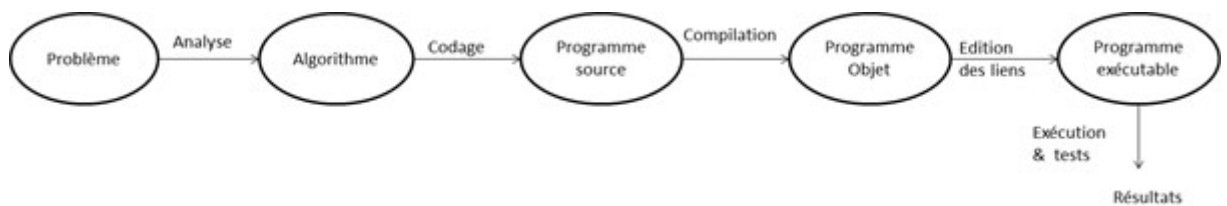
3. Codage de l'algorithme

Un algorithme doit être traduit en **langage de programmation**. Le programme résultant sera appelé **programme source** ou **code source**.

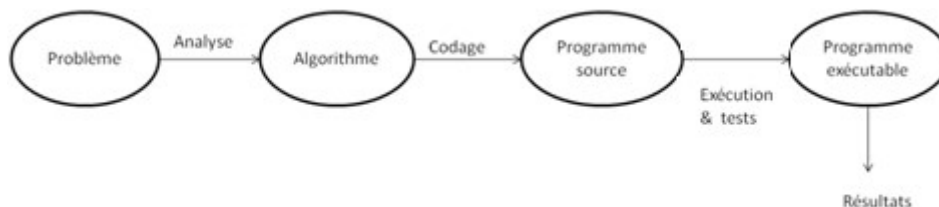
Un langage de programmation est défini par des règles d'écriture et de conception que doivent respecter les programmes. La difficulté, pour le programmeur, consiste à respecter ces règles imposées. On distingue plusieurs types de langages : langage de programmation procédurale (Pascal, C, ...), langage de programmation orientée objet (Java, C++, Python,) et langage de programmation événementielle (Delphi, VB, ...).

Il existe deux techniques principales pour traduire un code source en langage machine (pour être compris par la machine) :

- **La compilation** : c'est la traduction du programme source en programme objet. Elle est assurée par un programme spécialisé appelé **compilateur** qui traduit le programme source écrit dans un langage de haut niveau en un programme écrit dans un langage binaire de bas niveau directement compréhensible par la machine. Le programme issu de cette traduction est appelé **programme objet**. On notifie aussi que la compilation permet de détecter et de signaler les erreurs dues à un non-respect de la syntaxe du langage. A chaque fois que l'on apporte une modification au programme, il faut recompiler avant de voir le résultat. Un programme est très souvent composé de plusieurs parties qui devront être reliées entre elles pour former un **fichier exécutable**. Cette tâche est assurée par l'**éditeur des liens**. Pascal, C, C++, FORTRAN, ... sont des langages compilés.



- **L'interprétation** : c'est la traduction du code source ligne par ligne en langage machine par un logiciel qu'on appelle interpréteur. L'interpréteur va exécuter les lignes du code une par une, en décidant à chaque étape ce qu'il va faire ensuite. Perl, Basic, Javascript, Python,... sont des langages interprétés.



4. Exécution et tests du programme

Il s'agit de s'assurer que le programme donne un résultat correct dans tous les cas et pour toutes les éventualités.

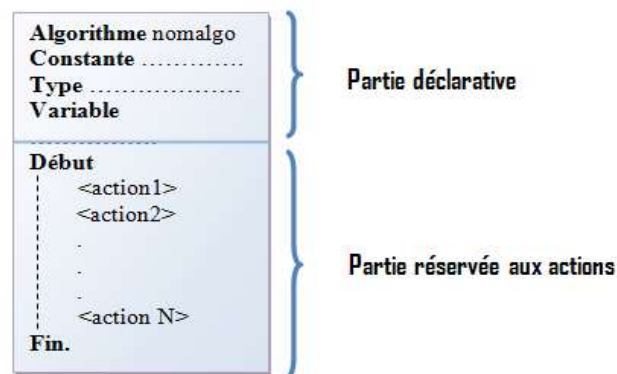
Il faut effectuer plusieurs jeux de tests correspondant aux différents cas et vérifier la validité des résultats.

II- Structure générale d'un algorithme

Un algorithme comporte généralement deux parties :

- **Partie déclarative** : elle contient l'entête définissant son nom et la déclaration des objets.
- **Partie corps de l'algorithme** : elle consiste en une séquence d'actions. Elle est délimitée par les deux mots-clés **Début** et **Fin.**

En outre, un algorithme est caractérisé par son nom qui devrait être significatif.



1- La déclaration des objets :

Un algorithme manipule des objets dont les caractéristiques doivent être bien spécifiées.

En effet, un objet est défini par :

- son nom ou son identificateur,
- sa nature : constante ou variable,
- son type : le domaine de définition de la variable (entier, réel...).

a- Les constantes

Une constante est une variable dont l'état reste inchangé tout le long d'un algorithme. Seule la consultation est autorisée. Une constante est définie par :

- son nom
- sa valeur

Syntaxe :

Const

Nom_const_1 = val_1

Nom_const_i = val_i

Exemple :

Const

Min = 10

Max = 200

PI=3.14

ANNEE=2020

b- Les variables

Une variable est un emplacement mémoire capable de contenir des valeurs de type défini au préalable. Elle peut être définie comme une boîte qui admet un nom, une taille, un contenu et une adresse.

- Le nom de la variable s'appelle identificateur de la variable.
- La taille dépend du type de la variable (exemple : 2 octets pour un entier, 1 octet pour un caractère, 4 octets pour un réel...)
- L'adresse désigne le numéro du 1^{er} octet occupé par cette variable en mémoire centrale
- Le contenu de la variable peut être changé indéfiniment au cours de l'exécution. Ce changement de valeur se fait par l'opération d'affectation.

Dans un algorithme, les variables sont déclarées comme suit :

Syntaxe :

Var

Liste des variables séparées par des virgules : type 1

Liste des variables séparées par des virgules : type 2

.

.

Liste des variables séparées par des virgules : type i

Dans un algorithme, on peut avoir 0 à plusieurs variables.

Exemple :

Var

X, Y : entier

A : réel

Remarques :

Tout identificateur (nom d'un algorithme, variable, constante) dans un algorithme doit respecter les règles suivantes :

- Il doit être significatif.
- Il doit être écrit au maximum sur 8 positions (Taille <= 8 caractères).
- Il doit commencer obligatoirement par une lettre.
- Il ne doit pas comporter le caractère « espace ».
- Il ne doit pas comporter de caractères spéciaux autres que le tiret de soulignement (_).
- Il doit être unique dans un algorithme.

Exemples d'identificateurs valides :

X, Algo1, TRUC, VAR1, VAR2, X1, Y, FICHE_ET ,...

Exemples d'identificateurs invalides :

3, VAR-1, X\$, X@Y, ...

c- Notion de type

Une variable ne peut prendre qu'un ensemble de valeurs connues à l'avance ; toute variable possède un domaine de définition. En informatique, ce domaine est appelé le **Type** de la variable qui définit :

- L'ensemble des valeurs que peut prendre la variable
- L'ensemble des opérations qu'on peut appliquer sur la variable

On distingue deux familles de types :

- **Les types simples** : ce sont des types qui sont supportés et reconnus par la majorité des langages de programmation. Lors de l'écriture de l'algorithme ou du programme, ce n'est pas la peine de les déclarer dans la partie déclarative réservée aux types (tels que les types : entier, réel, caractère ou booléen).
- **Les types composés ou complexes** : ce sont des types qui sont construits à partir des types simples mais qu'il faut les déclarer dans la partie réservée aux types ((tels que les types : tableaux, chaînes, enregistrements, ...).

Type entier

Le type entier comprend les valeurs numériques entières, positives ou négatives. Les opérations arithmétiques effectuées sur ces nombres sont:

L'addition (+), la soustraction (-), la division réelle (/), la division entière (Div), la multiplication (*), le reste de la division (mod) et l'opposé unaire (-).

Exemple :

X: entier

Annee, mois, jour: entier

Type réel

Le type réel comprend les valeurs décimales positives ou négatives. Les opérations appliquées sur le type entier sont valables pour le type réel.

On distingue deux formes de représentation : La forme usuelle « a.b » exemple : -4.6, 13.9 ou la forme scientifique a E b exemple : 345 = 3.45 E2 = 0.345 E3

Exemple :

X: réel

Type caractère

Une variable de type caractère accepte les 26 lettres latines majuscules et les 26 lettres minuscules, les 10 chiffres et les caractères spéciaux. Ces valeurs sont délimitées par 2 apostrophes. Les opérations permises : =, < >, <, <=, >, >=.

Exemple :

C: caractère

C1, C2: caractère

Type chaîne de caractères

Les chaînes de caractères sont délimitées par des guillemets. Exemples : "ali", "123", "vrai".

La déclaration d'une chaîne de caractères se fait suivant la syntaxe suivante:

<Nom_Variable_Chaine>: **chaîne** (longueur)

Exemple :

VAR

NOM : CHAINE(30) → Chaîne de 30 caractères maximum

ADRESSE : CHAINE → Chaîne de 128 caractères maximum

Type booléen

Il s'appelle aussi type logique et il accepte deux valeurs : vrai ou faux (true ou false).

Il y a 2 types d'opérateurs qui s'appliquent sur les variables de type booléen :

- Les opérateurs logiques
- Les opérateurs de comparaison

Les opérateurs logiques qui s'appliquent sur les variables de type booléen sont :

- NON pour la négation
- ET pour la conjonction
- OU pour la disjonction

Le tableau suivant résume le résultat de l'application de ces opérateurs sur deux variables Var1 et Var2.

Var1	Var2	NON Var1	Var1 ET Var2	Var1 OU Var2
Vrai	Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai
Faux	Vrai	Vrai	Faux	Vrai
Faux	Faux	Vrai	Faux	Faux

Les opérateurs de comparaison qui s'appliquent sur les variables de type booléen sont : <, >, <=, >=, =, <>

2- Corps de l'algorithme

La partie traitement d'un algorithme est composée d'une suite d'actions ou instructions permettant de générer le résultat final. On distingue quatre types d'actions :

- Les actions simples ou élémentaires tel que l'affectation, l'affichage à l'écran et la lecture du clavier.
- Les actions conditionnelles dans lesquelles une suite d'actions n'est exécutée que lorsqu'une certaine condition est réalisée.
- Les actions répétitives ou itératives permettant d'exécuter répétitivement un traitement et ceci est conditionné par une condition d'arrêt.
- Les actions composées qui peuvent être composées des trois actions précédentes.

Chapitre 2 : Les Instructions Simples

Objectifs du chapitre

- Comprendre les instructions d'affectation, de lecture et d'écriture en algorithmique.
- Connaître les expressions algorithmiques et maîtriser leurs évaluations en utilisant l'ordre de priorité des opérateurs.
- Construire des algorithmes simples s'appuyant sur des actions simples.

I- Instruction d'affectation

Une fois une variable de type donné est déclarée au début du programme, nous disposons d'un espace mémoire qui lui est alloué. Une opération d'affectation consiste à placer une valeur donnée dans cet espace mémoire.

Syntaxe

Identificateur var $\leftarrow$ *<expression>*

<expression> peut être :

- Une variable
- Une constante
- Une expression arithmétique
- Une expression logique

Remarques

- Une constante ne peut jamais figurer à gauche d'une affectation.
- Après une affectation, l'ancien contenu est perdu pour être remplacé par le nouveau contenu.
- Une action d'affectation doit se faire entre deux types compatibles.

1- Les expressions arithmétiques

<expression_arithmétique> *opérateur_arithmétique* *<expression_arithmétique>*

opérateur_arithmétique peut être : +, -, / ou *

Exemple :

$(Y/2) + x*3$

2- Les expressions logiques

Les expressions logiques admettent Vrai ou Faux comme résultat.

Elles peuvent utiliser des opérateurs relationnels (=, $\neq$, <, <=, >, >=) ou des opérateurs logiques (NON, ET, OU)

Exemple :

$(x < 6)$ ET $(Y = 20)$ donne vrai si $x < 6$ et $Y = 20$ et faux sinon

3- Ordre de priorité

Ordre décroissant	Opérateurs
1	Non, -(opposé)
2	^ (puissance)
3	*, Mod, Div
4	+, -
5	>, <, <=, >=
6	=, ≠
7	ET
8	OU

La table ci-dessus récapitule l'ordre de priorité des opérateurs. On note que les parenthèses peuvent changer cet ordre. En effet, les expressions entre parenthèses sont traitées en premier lieu. En cas d'absence de parenthèses, et lorsqu'on a des opérations ayant le même ordre de priorité, l'expression est traitée de gauche à droite opération par opération.

Exemple :

Donner pour chaque opération la valeur contenue dans chaque variable :

	A	B	C	X
A←1	1	--	--	--
B←2	1	2	--	--
C←3	1	2	3	--
X←A+B*C	1	2	3	7
X←(A+B)*C	1	2	3	9
X←A+B/C	1	2	3	1.67
X←(A+B)/C	1	2	3	1

II- Instruction de lecture ou d'entrée

La lecture est une opération d'entrée qui s'effectue à partir des périphériques d'entrée (clavier, souris, stylo optique,...) vers la mémoire centrale.

Elle permet à l'utilisateur d'introduire les données au programme càd elle permet d'affecter, à une variable, une donnée introduite à partir d'un périphérique d'entrée.

Syntaxe pour une seule variable

Lire(nom_var)

Syntaxe pour plusieurs variables

- Soit une lecture de toutes les variables ensemble :

Lire (nom_var1, nom_var2, ... ,nom_varN)

- Soit une lecture pour chaque variable :

Lire(nom_var1)

Lire(nom_var2)

.....

Lire(nom_varN)

Exemple

- *Lire(A)* : lit une valeur à partir du périphérique d'entrée et la range dans la case mémoire associée à A.
- *Lire(X,Y)* : lit deux valeurs la première pour X et la deuxième pour Y.

Remarques :

- Dès que le programme rencontre une instruction Lire, l'exécution s'interrompt, attendant la frappe d'une valeur au clavier, si la touche Entrée (Enter) a été frappée, l'exécution reprend.
- Il est interdit de lire une constante. Si $PI=3.14$ déclarée comme constante, *Lire(PI)* est impossible.
- Il est interdit de lire une expression arithmétique ou logique : ***Lire(a+b)* est impossible.**
- Il est interdit de lire un message : ***Lire('Bonjour')* est impossible.**

III- Instruction d'écriture ou de sortie

Une action d'écriture, parfois dite une primitive d'écriture est une opération qui permet à l'utilisateur d'afficher les résultats d'un programme sur le(s) périphérique(s) de sortie, tels que : écran, imprimante, traceur, ... Ce résultat peut être :

- Une chaîne de caractères délimitée par des apostrophes (' ')
- La valeur d'une ou plusieurs variables ou constantes dont les noms sont spécifiés
- La valeur d'une expression arithmétique et logique

Une instruction d'écriture peut être mixte, c'est à dire qu'elle regroupe des variables, des constantes, des expressions et des messages.

Syntaxes

- **Affichage d'une ou plusieurs variables**

Pour afficher le contenu d'une variable à l'écran, la syntaxe de l'instruction d'écriture doit se présenter comme suit :

Ecrire (nom_var)

Pour plusieurs variables, deux formats d'écriture peuvent être appliqués :

- Soit une écriture de toutes les variables ensemble :

Ecrire (*nom_var1*, *nom_var2*, ... , *nom_varN*)

- Soit une écriture pour chaque variable :

Ecrire (*nom_var1*)

Ecrire (*nom_var2*)

.....

Ecrire (*nom_varN*)

- **Affichage d'une constante**

Pour afficher la valeur d'une constante :

Ecrire(*valeur_constant*) ou **Ecrire**(*nom_constant*)

Exemples :

Ecrire(3.14)

Ecrire(PI)

- **Affichage d'une expression**

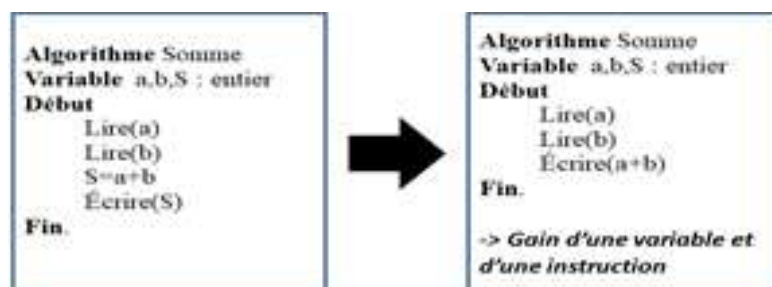
En vue d'optimiser le nombre de variables ainsi que le nombre d'instructions, on réalise directement une primitive d'écriture sur une expression qui est un ensemble de valeurs (variables, constantes,...), reliées par des opérateurs, et équivalent à une seule valeur.

Ecrire (expression)

Exemples :

Ecrire(2*r*PI) avec r est un réel et PI est une constante

Ecrire (a ou b et c) avec a, b et c des booléens



- **Affichage d'un message**

L'utilisation des messages dans un programme ne fait que faciliter son utilisation. Un message est une suite de caractère ayant un sens et délimité par deux apostrophes.

Ecrire('message')

Exemple :

Ecrire('Bonjour') : cette opération affichera à l'écran le mot Bonjour.

Remarque :

Ecrire('Bonjour') : affichera à l'écran le mot Bonjour.

Ecrire(Bonjour) : affichera à l'écran le contenu de la variable Bonjour.

- **Affichage mixte**

Dans certains programmes, l'affichage doit être très précis et clair. Par exemple, on veut afficher la ligne suivante : Le Montant de la Facture est = 345 DT.

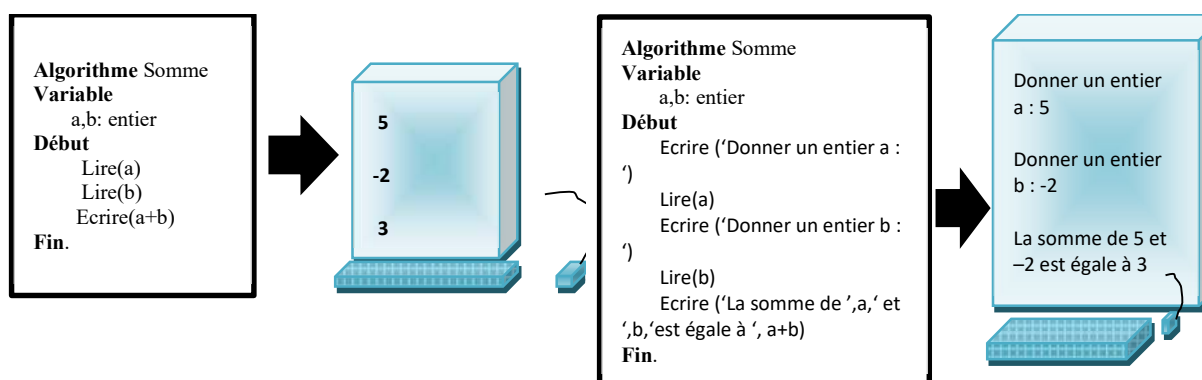
Pour obtenir un tel affichage, on doit présenter l'instruction d'écriture comme suit :

Ecrire ('Le Montant de la Facture est =', MONT_FACT, 'DT') où MONT_FACT est la variable qui contient ce montant. On parle dans ce cas d'écriture mixte dont la syntaxe est la suivante :

Ecrire (Liste d'expressions séparées par des virgules)

Exemple :

Considérons l'algorithme permettant de faire la somme de deux entiers. Avec le premier algorithme, l'affichage n'est pas clair surtout pour des utilisateurs qui n'ont pas d'idées sur le problème résolu. Il faut donc améliorer la présentation de l'affichage. Il suffit d'indiquer des messages. L'algorithme devient alors plus clair et l'utilisateur peut deviner facilement que ce programme réalise la somme de deux entiers.



Chapitre 3 : Les Structures Conditionnelles

Objectifs du chapitre :

- Comprendre l'utilité d'une structure conditionnelle.
- Connaître et manipuler les différentes structures conditionnelles disponibles en algorithmique.

I- Introduction

En programmation, on est souvent confronté à des situations où on a besoin de choisir entre 2 ou plusieurs traitements selon la réalisation ou non d'une certaine condition d'où la notion de traitement conditionnel. On distingue deux structures de traitement conditionnel à savoir :

- La structure conditionnelle simple qui consiste à évaluer une condition (expression logique à valeur vrai ou faux) et d'effectuer le traitement relatif à la valeur de vérité trouvée.
- La structure conditionnelle à choix multiple qui consiste à évaluer une expression qui n'est pas nécessairement à valeur booléenne (elle peut avoir plus de deux valeurs) et selon la valeur trouvée, effectue un traitement.

II- Structure conditionnelle simple

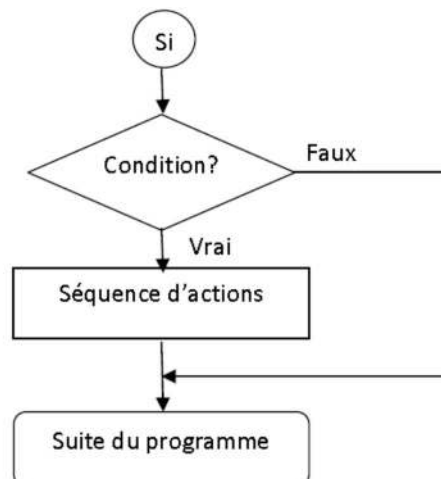
1. Forme simple : (Si Alors Fin si)

Syntaxe :

Si <condition> ***Alors***
 <action(s)>
Fin si

Dans cette forme, la condition est évaluée. Si elle vaut vrai alors c'est la séquence d'actions qui est exécutée sinon c'est l'action qui suit l'action conditionnelle dans l'algorithme qui est exécutée.

L'exécution de cette instruction se déroule selon l'organigramme suivant :



Exemple :

Ecrire un algorithme Verif_moyenne permettant de lire la moyenne d'un étudiant et d'afficher réussite si la moyenne est supérieure ou égale à 10.

```

ALGORITHME Verif_moyenne
Var
Moyenne: réel
Début
    Ecrire ("Saisir une moyenne:")
    Lire (moyenne)
    Si moyenne >= 10 Alors
        Ecrire ("réussite")
    Fin si
Fin
  
```

2. Forme composée : (Si ... alors.....sinon)**Syntaxe :**

```

Si <condition> Alors

    <action(s)1>

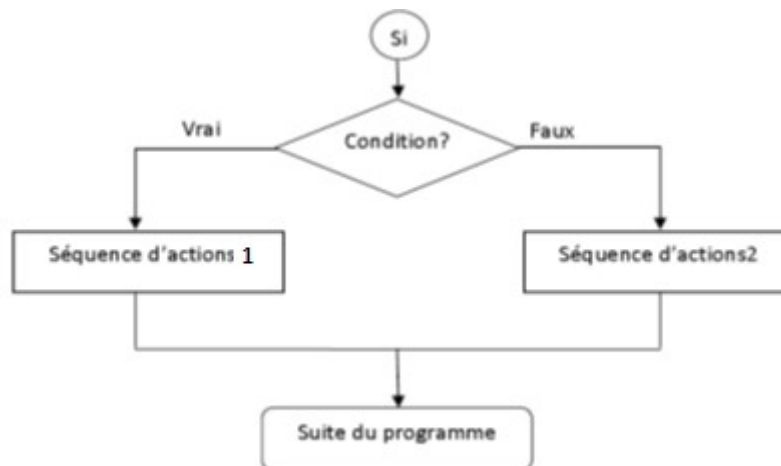
Sinon

    <action(s)2>

Fin si
  
```

Dans cette forme, la condition est évaluée. Si elle vaut vrai alors c'est la séquence d'actions 1 qui sera exécutée sinon c'est la séquence d'actions 2 qui sera exécutée.

L'exécution de cette instruction se déroule selon l'organigramme suivant :

**Exemple :**

Ecrire un algorithme Verif_moyenne permettant de lire la moyenne d'un étudiant et d'afficher réussite si la moyenne est supérieure ou égale à 10 et redoublement sinon.

```

ALGORITHME Verif_moyenne
Var
Moyenne: réel
Début
    Ecrire ("Saisir une moyenne:")
    Lire (moyenne)
    Si moyenne >= 10 Alors
        Ecrire ("réussite")
    Sinon
        Ecrire ("redoublement")
    Fin si
Fin

```

3. Forme imbriquée

Syntaxe

<pre> Si <condition1> Alors <action(s)1> Sinon Si <condition2> Alors <action(s)2> Sinon Si <condition N-1> Alors <action(s)N-1> Sinon <action(s)N> Fin si Fin si Fin si </pre>	<pre> Si <condition1> Alors <action(s)1> Sinon Si <condition2> Alors <action(s)2> Sinon Si <condition N-1> Alors <action(s)N-1> Sinon <action(s)N> Fin si </pre>
--	--

Si la condition 1 est vraie, alors la séquence d'actions 1 sera exécutée sinon on évalue la condition 2 si elle est vraie la séquence d'actions 2 sera exécutée. Enfin, si aucune des N-1 conditions est vraie alors on exécute la séquence d'actions N.

Exemple :

Ecrire un algorithme Verif_moyenne permettant de lire la moyenne d'un étudiant et d'afficher :

- Redoublement si la moyenne est entre 0 et 10
- Mention passable si la moyenne est entre 10 et 12
- Mention assez bien si la moyenne est entre 12 et 14
- Mention bien si la moyenne est entre 14 et 16
- Mention très bien si la moyenne est entre 16 et 20
- Un message indiquant d'entrer une moyenne entre 0 et 20

```

ALGORITHME Verif_moyenne
Var
Moyenne:réel
Début
    Ecrire ("Saisir une moyenne:")
    Lire (moyenne)
    Si (moyenne >=0) et (moyenne <10) alors
        Ecrire ("redoublement")
    Sinon
        Si (moyenne >=10) et (moyenne <12) alors
            Ecrire ("Mention passable")
        sinon
            si (moyenne >=12) et (moyenne <14) alors
                Ecrire ("Mention Assez bien")
            sinon
                si (moyenne >=14) et (moyenne <16) alors
                    Ecrire ("Mention Bien")
                sinon
                    si (moyenne >=16) et (moyenne <=20) alors
                        Ecrire ("Mention Très bien")
                    Sinon
                        Ecrire ("la moyenne doit être entre 0 et 20")
                    Fin si
                Fin si
            Fin si
        Fin si
    Fin si
Fin

```

III- Structure conditionnelle à choix multiple

Syntaxe

```

Selon <sélecteur> faire
    <liste de valeurs1> : <traitement 1>
    <liste de valeurs2> : <traitement 2>
    .....
    .....
    <liste de valeursN> : <traitement N>
Sinon
    <traitement N+1>
Fin selon

```

- Le sélecteur est un identificateur
- <traitement i> est une séquence d'actions.
- <liste de valeurs i> peut être une constante ou un intervalle de constantes de même type que le sélecteur.
- La partie sinon est facultative. Elle est exécutée si aucune des valeurs n'est égale au sélecteur.

Exemple :

Ecrire un algorithme qui lit la valeur d'une température de l'eau et d'afficher son état : liquide ($1 \leq \text{température} \leq 100$), solide ($-100 \leq \text{température} \leq 0$) ou gaz ($101 \leq \text{température} \leq 200$).

```
Algorithme EtatLiqu  
Var  
t : entier  
Début  
    Lire(t)  
    Selon (t) faire  
        -100 .. 0 : Ecrire('Etat Solide')  
        1..100 : Ecrire('Etat Liquide')  
        101..200 : Ecrire('Etat Gaz')  
    Sinon Ecrire('Etat Inexistant')  
    FinSelon  
Fin
```

Chapitre 4 : Les Structures Itératives

Objectifs du chapitre :

- Comprendre l'utilité d'une structure itérative.
- Connaître et manipuler les différentes structures itératives disponibles en algorithmique.
- Savoir faire la comparaison entre les différentes structures itératives.

I- Introduction :

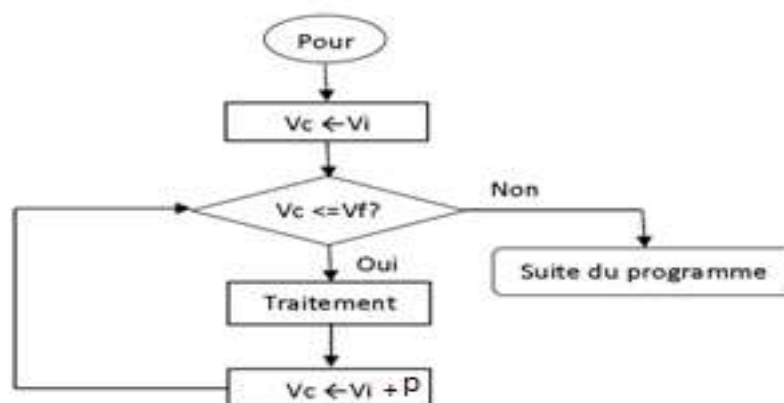
Les structures itératives (ou répétitives) permettent d'exécuter plusieurs fois de suite une ou plusieurs instructions, d'où leur appellation équivalente de boucle. Il en existe trois distinctes.

II- La structure « Pour faire.....Finpour »

Syntaxe :

Pour $\langle Vc \rangle$ de $\langle Vi \rangle$ à $\langle Vf \rangle$ pas= $\langle p \rangle$ ***faire***
 $\langle \text{Traitement} \rangle$
Finpour

- $\langle Vc \rangle$: compteur de type entier ;
- $\langle Vi \rangle$ et $\langle Vf \rangle$: valeur initiale et valeur finale du compteur ;
- $\langle \text{Traitement} \rangle$: action ou séquence d'actions à répéter ($vf-vi+1$) fois ;
- La boucle *Pour* est utilisée lorsque le nombre d'itération est connu à l'avance ;
- $\langle Vc \rangle$ reçoit une valeur initiale $\langle Vi \rangle$ pour la première fois, $\langle Vi \rangle$ ne doit pas être modifiée par une action de traitement à l'intérieur de la boucle ;
- Si on n'attribue pas une valeur de $\langle p \rangle$ alors, par défaut c'est la valeur 1.
- $\langle Vc \rangle$ est incrémenté automatiquement par $\langle p \rangle$ à chaque exécution du corps de la boucle *Pour*. Cette valeur d'incrément est appelée le pas de la boucle ;
- L'exécution de la boucle s'arrête lorsque $\langle Vc \rangle$ atteint $\langle Vf \rangle$.



Exemple:

```
Pour j de 1 à 5 faire
    Ecrire (j * 2)
Fin pour
```

Exécution : j = 1 2 3 4 5 6

Résultat : 2 4 6 8 10

Remarques:

- Une boucle peut être exécutée une ou plusieurs fois.
- Si le pas est différent de 1, il faut ajouter l'option (pas = constante)

Exemple :

```
Pour j de 5 à 1 (pas = -1) faire
    Ecrire (j * 2)
Fin pour
```

Exécution : j = 5 4 3 2 1 0

Résultat : 10 8 6 4 2

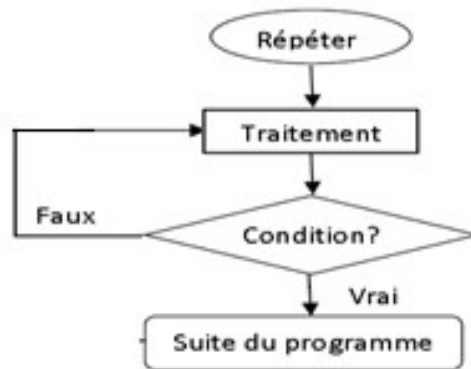
Application 1 :

Ecrire un algorithme Somme permettant de faire la somme des 10 premières valeurs entières à partir de la valeur 1 en utilisant la boucle Pour.

III- La structure « Répéter Jusqu'à »**Syntaxe**

```
Répéter
    <Traitement>
Jusqu'à (<Condition_arrêt>)
```

- <Condition_arrêt>: condition d'arrêt et de sortie de la boucle
- <Traitement>: action ou ensemble d'actions à exécuter tant que la condition n'est pas vérifiée, dès qu'elle soit vérifiée, l'exécution du traitement s'arrête.
- Le nombre de répétition n'est pas connu à l'avance.
- Le traitement est exécuté au moins une fois quelque soit le résultat de la condition.
- La condition doit être initialisée avant le début de la boucle et doit être modifiée à l'intérieur de la boucle.

**Exemple 1:**

```

j ← 1
Répéter
    Ecrire (j *2)
    j ← j +1
jusqu'à (j > 5)
  
```

Exécution : j = 1 2 3 4 5 6

Résultat : 2 4 6 8 10

Exemple 2:

Ecrire un algorithme permettant de lire une valeur entière strictement supérieure à 0.

```

ALGORITHME Lire_Valeur_Sup_0
Var
  val: entier
Début
  Répéter
    Ecrire ("Donner une valeur supérieure à 0:")
    Lire (val)
  Jusqu'à val >0
Fin
  
```

Remarque:

L'instruction répéter est exécutée avant d'évaluer la condition donc elle est exécutée au moins une fois.

Application 2 :

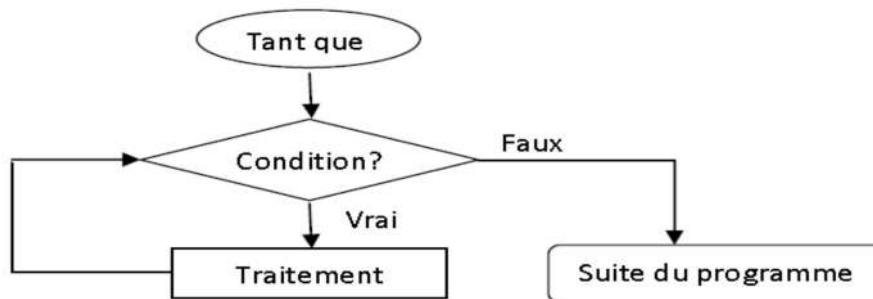
Ecrire un algorithme Somme permettant de faire la somme des 10 premières valeurs entières à partir de la valeur 1 en utilisant la boucle répéter - jusqu'à.

IV- La structure « Tantque..... Faire..... Fintantque »

Syntaxe

Tantque (*Condition*) *faire*
 <*Traitement*>
Fintantque

- <Condition> : condition d'exécution de la boucle.
- <Traitement> : action ou ensemble d'actions à exécuter tant que la condition est vérifiée.
- Le traitement est exécuté tant que la condition est vérifiée sinon on sort de la boucle.
- Si la condition n'est pas vraie dès le début, la boucle ne sera jamais exécutée (0 fois).
- Le nombre de répétition n'est pas connu à l'avance.
- La condition doit être initialisée avant la boucle et modifiée à l'intérieur de la boucle.



Exemple:

```

j ← 1
tantque (j ≤ 5) faire
    Ecrire (j * 2)
    j ← j + 1
Fintantque
  
```

Exécution : j = 1 2 3 4 5 6

Résultat : 2 4 6 8 10

Application 3 :

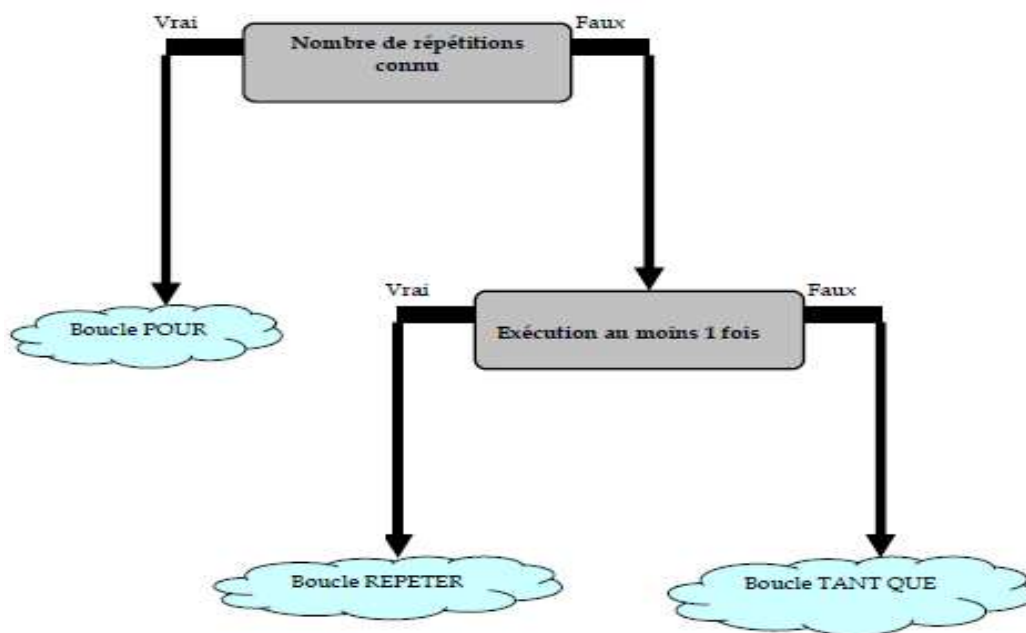
Ecrire un algorithme Somme permettant de faire la somme des 10 premières valeurs entières à partir de la valeur 1 en utilisant la boucle Tant que.

V- Comparaison des structures itératives

Pour choisir lesquelles des formes utiliser, il est conseillé de suivre la démarche suivante:

- Si on connaît le nombre de répétitions alors il est préférable d'utiliser **POUR**
- Sinon si on veut exécuter un traitement au moins une fois alors on utilise **REPETER**
- Sinon, on utilise **TANT QUE**

La figure suivante résume la manière de choisir une structure itérative:



Chapitre 5 : Les Sous-Programmes

Objectifs du chapitre :

- Comprendre la notion de division de problème.
- Connaitre les fonctions et les procédures ainsi que la différence entre les deux.
- Appliquer le concept de programmation structurée lors de l'écriture des algorithmes.

I- Introduction :

Le but de l'utilisation des sous-programmes :

- Décomposition des problèmes en modules (sous problèmes de taille réduite) :
 - ✓ Dans un programme plusieurs séquences d'instructions sont appelées plusieurs fois et depuis divers points du programme. Il serait donc plus intéressant d'isoler ces séquences dans un sous programme qui peut être appelé depuis n'importe quel point du programme.
 - ✓ L'approche modulaire réduit énormément le nombre d'instructions redondantes (qui se répètent) moyennant l'ajout d'une séquence d'appel pour le module à différents endroits du programme. D'où la réduction de la taille du programme (code).
 - ✓ La lisibilité qui facilite notablement la compréhension du programme
- Réutilisation du sous programme

En résumé, le programme sera plus lisible et plus facile à maintenir (à modifier éventuellement par la suite).

On distingue deux formes de sous programmes : Les procédures et les fonctions.

II- Les procédures :

Une procédure est un sous-programme qui effectue un traitement (suite d'instructions) qui permet la résolution d'un problème précis et qui peut avoir zéro ou plusieurs résultats.

1- Déclaration de la procédure :

La déclaration d'une procédure se fait dans la partie déclarative de l'algorithme comme suit :

Procédure nom_procedure (param1: type1, param2: type2,..., paramN: TypeN)

Partie Déclarative

Début

Corps de la procédure

Fin

2- Appel de la procédure :

L'appel de la procédure se fait dans la partie corps de l'algorithme comme suit :

nom_procedure (Liste paramètre(s) effectif(s))

Remarques :

- (param1, param2, ..., paramN) sont appelés les *paramètres formels*: ce sont des variables qui permettent à la procédure de communiquer avec l'extérieur.
- Il faut que les deux listes de paramètres formels et effectifs aient le même nombre et ordre de paramètres et que les paramètres formels et effectifs correspondants soient compatibles.
- On peut trouver certaines procédures sans paramètres ; on déclarera alors la procédure de la manière suivante : **Procédure nom_procedure ()**.

Exemple :

Écrire une procédure permettant d'afficher la somme de deux entiers.

Algorithme Principal

Var

t :réel

x, y,z : entier

Procédure Somme(a,b:entier)

Début

Ecrire(a, « + », b, « = », a+b)

Fin

Début

Ecrire("Saisir 3 entiers")

Lire(x,y,z)

Ecrire("Saisir un réel")

Lire(t)

Somme(x,y) //ok

Somme(x,y,z) // interdit car le nombre de paramètres formels est différent du nombre de paramètres effectifs

Somme(x,t) // interdit car les paramètres formels et effectifs ne sont pas compatibles

Fin

Exemple :

Procédure sans paramètres :

```

Procédure message()
Var
    mess : chaîne
Début
    Lire(mess)
    Écrire(mess)
Fin

```

III- Les fonctions :

Une fonction est un sous-programme contenant un certain nombre d'instructions, qui retourne un **résultat unique** de **type simple affecté à son nom**.

1- Déclaration :

La déclaration d'une fonction se fait dans la partie déclarative de l'algorithme comme suit.

```

Fonction nom_fonction(liste des paramètres formels) : type du résultat retourné
    Partie Déclarative
Début
    Corps fonction
Fin

```

Exemple :

Ecrire une fonction Max qui permet de retourner le max entre 2 entiers.

```

Fonction Max(x, y : entier) : entier
Début
    Si x >= y Alors
        Max ← x
    Sinon
        Max ← y
    Finsi
Fin

```

Remarque :

Le corps de la fonction doit contenir au moins une instruction de retour de la valeur de la fonction comme suit :

nom_fonction ← <expression>

<expression> doit être de même type que la fonction.

Remarque :

On peut trouver certaines fonctions sans paramètres ; on déclarera alors la fonction de la manière suivante : **Fonction** *nom_fonction*() : *type du résultat retourné*

Exemple :

Écrire une fonction sans paramètres qui retourne le message 'Bonjour'

```
Fonction Message() : chaine
Début
    Message ← 'Bonjour'
Fin
```

2- Appel de la fonction :

Pour appeler la fonction, il suffit de suivre le prototype, c'ad donner son nom et la liste de ses *paramètres effectifs* entre parenthèses. On doit utiliser une variable pour stocker le résultat retourné par la fonction.

Exemple :

```
Algorithme Maximum
Var
    a,b,m : réel
Fonction Max(x, y : réel) : réel
Début
    Si x >= y Alors
        Max ← x
    Sinon
        Max ← y
    Finsi
Fin
Début
    Ecrire("Saisir deux réels")
    Lire(a,b)
    m ← Max(a,b)
    Ecrire(" le maximum est : ",m)
Fin
```

Remarques :

- La dernière des valeurs affectées constitue le résultat de l'évaluation de l'appel de la fonction.
- C'est l'algorithme principal qui appelle ou invoque des sous-programmes (procédure ou fonction). D'où, il est appelé l'**appelant** et le sous-programme est nommé l'**appelé**.
- un sous-programme (procédure ou fonction) peut appeler un autre sous-programme. Par la suite, le sous-programme qui appelle est l'**appelant** et celui qui a subit l'appel est l'**appelé**.

IV- Passage des paramètres

1- Passage de paramètres par valeur :

Exemple :

On considère l'algorithme suivant :

```
Algorithme passage_valeur
Var
  x : entier
Procédure Incrémenter (y : entier)
Début
  Ecrire(y)
  y ← y+1
  Ecrire(y)
Fin
Début
  x ← 0
  Ecrire(x)
  Incrémenter(x)
  Ecrire(x)
Fin
```

Résultat :

Le x reste intact=0 car on a utilisé le mode de passage de paramètres par valeur (Les identificateurs des paramètres formels ne sont précédés d'aucun mot clé). Donc, les paramètres effectifs sont lus à l'appel de la procédure puis leurs valeurs sont affectées à des variables temporaires locales à la procédure et au retour du sous-programme appelé chaque paramètre effectif garde sa valeur initiale de l'appel.

Remarque :

Le Passage de paramètres par valeur est le mode de communication le plus simple qui ne permet le passage de valeurs que dans un seul sens :

Programme appelant



Programme appelé

Donc pour incrémenter x, la solution est d'utiliser le *Passage de paramètres par variable (ou par adresse)*.

2- Passage de paramètres par variable :

Dans ce mode de passage les paramètres formels sont précédés du mot clé **VAR** dans leur déclaration.

Les paramètres effectifs sont liés aux paramètres formels. Toute modification de la valeur du paramètre formel entraîne automatiquement la modification du paramètre effectif qui lui correspond. Il y a communication de valeurs dans les deux sens:



Au retour du sous-programme appelé chaque paramètre effectif contient la valeur du paramètre formel qui lui correspond après l'exécution de la procédure.

Exemple :

```

Algorithme Passage_variable
Var
  x : entier
Procédure Incrémente (VAR y : entier)
Début
  Ecrire(y)
  y ← y+1
  Ecrire(y)
Fin
Début
  x ← 0
  Ecrire(x)
  Incrémente(x)
  Ecrire(x)
Fin
  
```

V- La Portée des Variables :

On peut manipuler 2 types de variables dans un module (procédure ou fonction) : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "champ de définition", leur "durée de vie").

Une **variable locale** n'est connue qu'à l'intérieur du module où elle a été définie. Elle est créée à l'appel du module et détruite à la fin de son exécution.

Une variable globale est connue par l'ensemble des modules et le programme principal. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme.

Une variable globale est précédée par le terme **Globale** lors de sa déclaration.

Remarque :

- Par défaut, une variable est définie comme locale.
- Une même variable peut apparaître localement dans deux sous-programmes différents.

Chapitre 6 : Les Tableaux

Objectifs du chapitre :

- Connaître le type tableau et son utilité pour résoudre des problèmes en algorithmique.
- Manipuler les différentes techniques et opérations de base sur les tableaux.
- Construire des algorithmes s'appuyant sur le type tableau à une et à deux dimensions.

I- Les tableaux à une dimension

1- Définition

Un tableau *T* est une structure de données formée de *N* éléments de *même type*. Nous pouvons accéder à ces éléments par des *indices*.

Les types des éléments sont quelconques (entier, réel, caractère, booléen,...) sauf le type fichier. Le tableau nous permet de manipuler plusieurs valeurs en utilisant un seul nom de variable.

Remarques :

- On dit encore que *T* est *un vecteur* de dimension *N*.
- Le nombre de composantes *N* est alors la dimension du tableau qui doit être connu à l'avance.



2- Déclaration

<i>Type</i>	<i>Nomtableau : Tableau [borne_inf.. borne_sup] de type_élément</i>
<i>Var</i>	<i>Nomvar : Nomtableau</i>

Exemple :

On veut déclarer un tableau *T* qui contient les moyennes de 20 élèves :

Type	Tab_Moy : Tableau [1..20] de réel
Var	T : Tab_Moy

3- Accès aux composantes d'un tableau

Considérons un tableau *T* de dimension *N* :

- L'accès au premier élément du tableau se fait par **T[1]**
- L'accès au dernier élément du tableau se fait par **T[N]**
- L'accès à l'*i*^{ème} élément, $1 < i < N$, du tableau se fait par **T[i]**

Exemple :

Nom : T	100	200	300	400	500
Indice :	1	2	3	4	5
Contenu	T[1]	T[2]	T[3]	T[4]	T[5]

4- Remplissage d'un tableau

Ecrire un algorithme qui permet de remplir un tableau de N=5 entiers.

```

Algorithme Remplissage
CONST
    N=5
TYPE
    Tab :Tableau [1..N] de entier
VAR
    T : Tab
    i : entier
Début
    Pour i de 1 à N Faire
        Ecrire ("Saisir T [ " , i , " ] :")
        Lire(T[i])
    Finpour
Fin
  
```

5- Affichage du contenu d'un tableau

```

Algorithme Affichage
CONST
    N=5
TYPE
    Tab : Tableau [1..N] d'entier
VAR
    T : Tab
    i : entier
Début
    Pour i de 1 à N Faire
        Ecrire (T[i])
    Finpour
Fin
  
```

II- Les tableaux à deux dimensions

1- Définition

Un tableau à deux dimensions M, appelé également matrice, est une structure de données permettant d'organiser des informations de même type (qui peut être simple ou structuré) en lignes et en colonnes. Il est caractérisé donc par son nombre de lignes et son nombre de colonnes.

On appelle **L** le **nombre de lignes** du tableau et **C** le **nombre de colonnes** du tableau.

Un tableau M à deux dimensions contient **L*C** composantes.

2- Déclaration

Type
Matrice : Tableau [1..Nbr_Lig, 1..Nbr_Col] de Type_élément_Matrice

Var
M : Matrice

Exemple :

```
CONST
    NL = 30      (* Nombre de Lignes *)
    NC = 20      (* Nombre de colonnes *)

TYPE
    MAT : Tableau [1.. NL, 1.. NC] d'entier

VAR
    M1 : MAT
```

3- Accès aux composantes d'une matrice

Considérons un tableau M de NL lignes et NC colonnes :

- L'accès à un élément de la matrice ne peut se faire qu'avec deux indices. Les indices du tableau varient de 1 à NL, respectivement de 1 à NC.
- La composante de la $i^{ème}$ ligne et $j^{ème}$ colonne est notée : M[i,j].

Syntaxe générale: <Nom du tableau>[<ligne> ,<colonne>]

Exemple :

Considérons une matrice A de 3 lignes et 4 colonnes :

	1	2	3	4
1	A[1 ,1]	A[1,2]	A[1,3]	A[1 ,4]
2	A[2 ,1]	A[2 ,2]	A[2 ,3]	A[2 ,4]
3	A[3 ,1]	A[3 ,2]	A[3 ,3]	A[3 ,4]

4- Remplissage d'une matrice

```
Algorithme Remplissage_Mat
CONST
    NL = 30
    NC = 20
TYPE
    Mat :Tableau [1.. NL, 1..NC] d'entier
VAR
    M : Mat
    i ,j : entier
Début
    Pour i de 1 à NL Faire
        Pour j de 1 à NC Faire
            Ecrire ("Saisir M [", i , ", ", j , "] :")
            Lire (M [i, j])
        Fin pour
    Fin pour
Fin
```

5- Affichage d'une matrice

```
Algorithme Affichage_Mat
CONST
    NL = 30
    NC = 20
TYPE
    MAT : Tableau [1.. NL, 1..NC] d'entier
VAR
    M : MAT
    i,j : entier
Début
    Pour i de 1 à NL Faire
        Pour j de 1 à NC Faire
            Ecrire ( M[i, j])
        Finpour
    Finpour
Fin
```

Application :

Soient M1 et M2 deux matrices d'entiers à L lignes et C colonnes ($0 < L, C \leq 50$).

- Ecrire une procédure qui permet de saisir les dimensions d'une matrice.
- Ecrire une procédure qui permet de remplir une matrice M.
- Ecrire une procédure qui calcule les éléments de la matrice $M3 = M1 + M2$.
- Ecrire une procédure qui affiche la matrice M3.
- Ecrire l'algorithme principal.

Rappel:

$$\begin{array}{c}
 / \quad \underline{M1} \quad \backslash \\
 | \ a \ b \ c \ d \ | \\
 | \ e \ f \ g \ h \ | \\
 | \ i \ j \ k \ l \ | \\
 \backslash \quad \quad \quad /
 \end{array}
 +
 \begin{array}{c}
 / \quad \underline{M2} \quad \backslash \\
 | \ a' \ b' \ c' \ d' \ | \\
 | \ e' \ f' \ g' \ h' \ | \\
 | \ i' \ j' \ k' \ l' \ | \\
 \backslash \quad \quad \quad /
 \end{array}
 =
 \begin{array}{c}
 / \quad \underline{M3} \quad \backslash \\
 | \ a+a' \ b+b' \ c+c' \ d+d' \ | \\
 | \ e+e' \ f+f' \ g+g' \ h+h' \ | \\
 | \ i+i' \ j+j' \ k+k' \ l+l' \ | \\
 \backslash \quad \quad \quad /
 \end{array}$$

Chapitre 7 : Les Algorithmes de Tri et de Recherche

Objectif du chapitre :

- Comprendre et manipuler les différents algorithmes de tri et de recherche.

I- Algorithmes de tri

On désigne par tri, l'opération qui consiste à ordonner une suite de valeurs suivant une relation d'ordre (ordre croissant et décroissant). Il y a plusieurs façons ou techniques pour effectuer un tri.

1- Tri par sélection (par minimum)

Le principe de cette méthode est simple. Elle consiste à :

- Chercher l'indice du plus petit élément du tableau $T[1..N]$ et permuter l'élément correspondant avec l'élément d'indice 1;
- Chercher l'indice du plus petit élément du tableau $T[2..N]$ et permuter l'élément correspondant avec l'élément d'indice 2 ;
-
- Chercher l'indice du plus petit élément du tableau $T[N-1..N]$ et permuter l'élément correspondant avec l'élément d'indice $N-1$;

L'exemple suivant montre le fonctionnement de cet algorithme :

Tableau initial

<u>60</u>	50	20	40	10	30
-----------	----	----	----	----	----

Après la 1 ^{ère} itération	10	<u>50</u>	20	40	60	30
-------------------------------------	----	-----------	----	----	----	----

Après la 2 ^{ème} itération	10	20	<u>50</u>	40	60	30
-------------------------------------	----	----	-----------	----	----	----

Après la 3 ^{ème} itération	10	20	30	<u>40</u>	60	50
-------------------------------------	----	----	----	-----------	----	----

Après la 4 ^{ème} itération	10	20	30	40	<u>60</u>	50
-------------------------------------	----	----	----	----	-----------	----

Après la 5 ^{ème} itération	10	20	30	40	50	60
-------------------------------------	----	----	----	----	----	----

```

Procédure Tri_Selection (VAR T : Tab, N : entier)
Var
    i, j, aux, indmin : entier
Début
    Pour i de 1 à N-1 faire
        indmin ← i
        Pour j de i+1 à N faire
            Si ( T[j] < T[indmin] ) alors
                indmin ← j
        Finsi
    FinPour
    Si ( i <> indmin ) alors
        aux ← T[i]
        T[i] ← T[indmin]
        T[indmin] ← aux
    Finsi
FinPour
Fin

```

2- Tri à bulles

Le principe du tri à bulle consiste à répéter le traitement suivant :
 Parcourir les éléments du tableau T de 1 à (N-1); si $T[i] > T[i+1]$, alors on les permute.
 Le programme s'arrête lorsqu'aucune permutation n'est réalisable.

Remarque :

On est sûr d'obtenir un tableau trié au bout de N-1 parcours mais parfois le processus s'arrête un peu plus tôt. L'exemple suivant montre le fonctionnement de cet algorithme :

Tableau initial :

60	50	20	40	10	30
----	----	----	----	----	----

1 ^{ère} itération	i=1	50	60	20	40	10	30
	i=2	50	20	60	40	10	30
	i=3	50	20	40	60	10	30
	i=4	50	20	40	10	60	30
	i=5	50	20	40	10	30	60

2 ^{ème} itération	i=1	20	50	40	10	30	60
	i=2	20	40	50	10	30	60
	i=3	20	40	10	50	30	60
	i=4	20	40	10	30	50	60

3 ^{ème} itération	i=1	20	40	10	30	50	60
	i=2	20	10	40	30	50	60
	i=3	20	10	30	40	50	60

4 ^{ème} itération	i=1	10	20	30	40	50	60
	i=2	10	20	30	40	50	60

5 ^{ème} itération	i=1	10	20	30	40	50	60
----------------------------	-----	----	----	----	----	----	----

Procédure Tri_Bulles (VAR T : Tab , N : entier) Var i, aux : entier permute: booléen Début Répéter permute $\leftarrow$ Faux Pour i de 1 à (N-1) Faire Si(T[i] > T[i+1]) Alors /*on les permute*/ aux $\leftarrow$ T[i] T[i] $\leftarrow$ T[i+1] T[i+1] $\leftarrow$ aux permute $\leftarrow$ Vrai Fin Si Fin Pour Jusqu'à (permute = Faux) Fin	Procédure Tri_Bulles_opt (VAR T : Tab, N : entier) Var i, aux : entier permute: booléen Début Répéter permute $\leftarrow$ Faux Pour i de 1 à (N-1) Faire Si(T[i] > T[i+1]) Alors /*on les permute*/ aux $\leftarrow$ T[i] T[i] $\leftarrow$ T[i+1] T[i+1] $\leftarrow$ aux permute $\leftarrow$ Vrai Fin Si Fin Pour N $\leftarrow$ N-1 Jusqu'à ((permute = Faux) OU (N = 1)) Fin
---	---

3- Tri par insertion

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le $i^{\text{ème}}$ élément, les éléments qui le précèdent sont déjà triés. L'objectif d'une étape est d'insérer le $i^{\text{ème}}$ élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux éléments qui le précèdent, puis décaler à droite les éléments afin de pouvoir effectuer l'insertion.

L'exemple suivant montre le fonctionnement de cet algorithme :

Tableau initial :

60	50	20	40	10	30
----	----	----	----	----	----

1^{ère} itération :

50	60	20	40	10	30
----	----	----	----	----	----

2^{ème} itération :

20	50	60	40	10	30
----	----	----	----	----	----

3^{ème} itération :

20	40	50	60	10	30
----	----	----	----	----	----

4^{ème} itération :

10	20	40	50	60	30
----	----	----	----	----	----

5^{ème} itération :

10	20	30	40	50	60
----	----	----	----	----	----

Procédure Tri_Insertion(VAR T: Tab, N: entier) Var i,j,aux,pos: Entier Début Pour i de 2 à N Faire pos $\leftarrow$ i -1 Tant que (pos >=1) ET (T[pos] > T[i]) Faire pos $\leftarrow$ pos - 1 Fin Tant que pos $\leftarrow$ pos + 1 aux $\leftarrow$ T[i] Pour j de (i-1) à pos [pas = -1] Faire /*décalage à droite*/ T[j+1] $\leftarrow$ T[j] Fin Pour T[pos] $\leftarrow$ aux Fin Pour Fin
--

II- Algorithmes de recherche

1- La recherche séquentielle

Le principe de l'algorithme de recherche séquentielle consiste à parcourir un tableau d'éléments dans l'ordre de ses indices jusqu'à ce qu'un élément recherché soit trouvé ou bien que la fin du tableau soit atteinte et le rang (indice) de l'élément est alors retourné.

La fonction ci-dessous est une fonction qui prend en entrée, un tableau T d'entiers non triés de N éléments et un entier *ValRech* qui est la valeur recherchée.

Si la valeur est trouvée, la fonction renvoie l'indice de l'élément recherché.

Lorsque la valeur recherchée n'est pas trouvée dans le tableau, la fonction renvoie -1.

```

Fonction Recherche_Sequentielle(T : Tab, N: entier, ValRech : entier) : entier
Var
    i,pos: entier
Début
    pos ← -1
    i ← 1
    Tant que (i ≤ N ET pos = -1 ) Faire
        Si T[i]=ValRech Alors
            pos ← i
        Sinon
            i ← i+1
        FinSi
    FinTantque
    Recherche_Sequentielle ← pos
Fin
  
```

2- La recherche dichotomique

L'algorithme de recherche dichotomique est un algorithme efficace et rapide pour la recherche d'un élément dans un **tableau trié**.

Le principe de cet algorithme consiste à diviser le tableau en deux parties et à comparer la valeur recherchée avec l'élément situé au milieu du tableau (l'élément médian). Trois cas peuvent se produire:

- **Si ValRech = T[milieu]** alors ValRech est trouvée et la recherche est terminée.
- **Si ValRech < T[milieu]** alors on va chercher ValRech dans la partie gauche du tableau T.
- **Si ValRech > T[milieu]** alors on va chercher ValRech dans la partie droite du tableau T.

On poursuit la recherche tant que T[milieu] est différent de ValRech est tant que la dimension du sous tableau reste valide.

Remarque :

Pour utiliser la recherche dichotomique, il faut tout d'abord trier le tableau en utilisant l'un des algorithmes de tri.

```

Fonction Rech_Dichotomique ( T : tab, N, ValRech :entier) : entier
Var
    i, pos,milieu, inf, sup : entier
Début
    pos ← -1
  
```

```

inf ← 1
sup ← N
Tant que ( inf ≤ sup et pos = -1 ) Faire
    milieu ← (inf + sup) DIV 2
    Si (T[milieu] = ValRech ) Alors
        pos ← milieu
    Sinon
        Si (ValRech < T[milieu] ) Alors
            sup ← milieu - 1
        Sinon
            inf ← milieu + 1
    Finsi
Finsi
FinTantque
Rech_Dichotomique ← pos
Fin

```

Exemple :

Le tableau ci-dessous montre le fonctionnement de la recherche dichotomique. La valeur recherchée est 28.

1	2	3	4	5	6	7	8
1	5	7	14	28	31	35	42
1	5	7	14	28	31	35	42
				28	31	35	42
				28			

La première ligne indique les indices.

La deuxième ligne indique la position initiale avant le démarrage de l'algorithme. Dans la ligne suivante, l'algorithme compare le nombre 14 qu'est l'élément médian à la valeur 28 ($14 < 28$). Dans ce cas l'algorithme continue la recherche entre les indices $\text{inf} = \text{milieu} + 1 = 5$ et $\text{sup} = 8$.

Lors de l'itération suivante, l'algorithme compare l'élément $T[\text{milieu}] = T[6] = 31$ qu'est l'élément médian à la valeur 28 ($28 < 31$). Dans ce cas l'algorithme continue la recherche entre les indices $\text{inf} = 5$ et $\text{sup} = \text{milieu} - 1 = 5$.

Lors de l'itération suivante, la valeur recherchée est trouvée ($T[\text{milieu}] = T[5] = 28$), la fonction s'interrompt et renvoie l'indice de cet élément.

Chapitre 8 : La Récursivité

I- Définition :

Une des caractéristiques les plus importantes de programmation est la possibilité pour un sous-programme (une **procédure** ou une **fonction**) de s'appeler lui-même. On parle de **récursivité**. Il faut prévoir toujours une condition d'arrêt pour stopper les appels récursifs sinon on obtient un traitement similaire à une boucle infinie.

La récursivité est particulièrement utile pour minimiser la complexité d'un problème informatique. Elle est **très économique pour le programmeur** ; elle permet de faire les choses correctement, en très peu d'instructions.

Syntaxe :

```

Fonction Récursive (param : type_param) : typer_retour
Var
    Résultat : type_retour
Début
    Si (condition_arret) Alors
        Résultat ← val
    Sinon
        Résultat ← Récursive (Nouveau_param)
    FinSi
    Récursive ← Résultat
Fin
  
```

II- Etude d'exemples

1) Exemple 1

On peut définir la factorielle d'un nombre N non négatif de 2 manières :

- Définition non récursive (**itérative**): $N! = N * N-1 * \dots * 2 * 1$
- Définition **récursive** : $N! = N * (N-1)!$ et $0! = 1$

Solution itérative	Solution récursive
Fonction Fact (N : entier) : entier VAR i: entier DEBUT Si (N = 0) alors Fact ← 1 Sinon Fact ← 1 Pour i de N à 1 (pas = -1) faire Fact ← Fact* i FinPour Finsi FIN	Fonction FACT (N : entier) : entier DEBUT Si (N = 0) alors FACT ← 1 Sinon FACT ← N * FACT (N-1) Finsi FIN

Interprétation :

Chaque **procédure** ou **fonction récursive** doit comporter une condition d'arrêt (dans notre exemple $N=0$). Cette condition empêche des appels récursifs sans fin (similaire à une boucle infinie). Habituellement, la condition d'arrêt se présente sous la forme d'une instruction **Si..... Alors.....Sinon** qui permet de stopper la récurrence si la condition d'arrêt est satisfaite. Par contre, tant que la condition d'arrêt n'est pas remplie, la procédure (ou la fonction) s'appelle au bon endroit.

On remarque que le processus récursif remplace en quelque sorte une boucle.

On remarque aussi que la version récursive appelle successivement la fonction **FACT**, une fois dans le programme principal et $(N-1)$ fois depuis elle-même, de façon totalement transparente pour l'utilisateur. Le **seul résultat visible est la réponse finale**.

Mécanisme de fonctionnement de la récursivité

L'évaluation récursive d'une factorielle donne lieu au traitement des appels de la fonction dans l'ordre suivant:

Soit $N=3$: $(3!=3*2*1=6)$

FACT (3)

3*FACT (2)

2* FACT (1)

1* FACT (0)

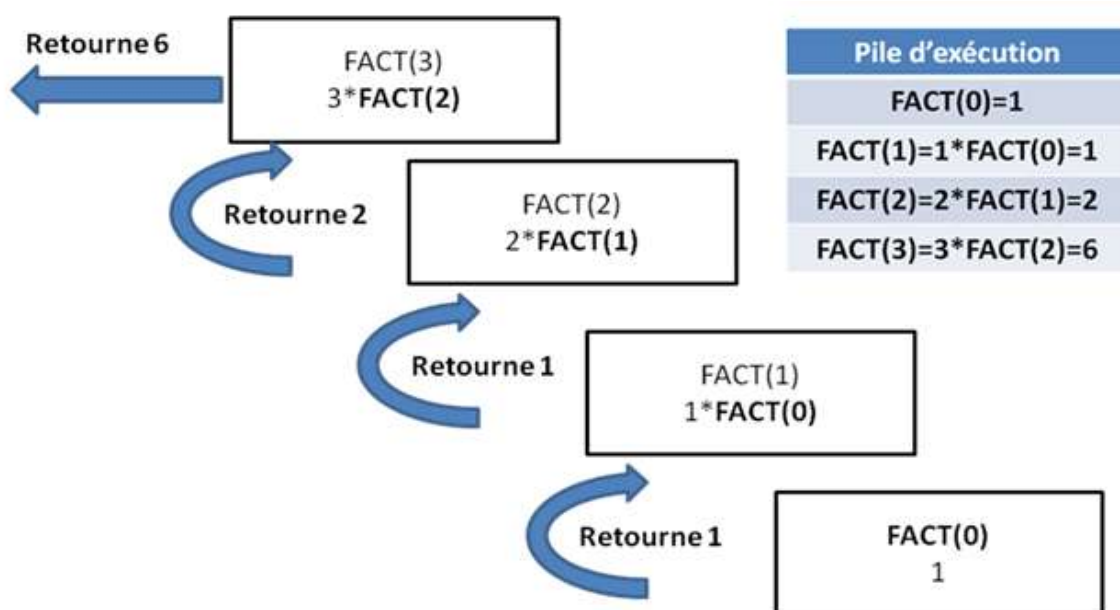
1

Lors de l'exécution d'un sous-programme récursif, les appels successifs de la fonction (ou procédure) récursive ne sont pas exécutés immédiatement. Ils sont placés dans **une pile** jusqu'à ce que la condition d'arrêt du processus récursif soit atteinte.

Les appels de fonction sont alors exécutés (résultat renvoyé) en ordre inverse, au fur et à mesure qu'ils sont retirés de la pile.

Remarque :

Une pile est une structure de données de type **LIFO** (Last In, First Out) ou « **Dernier Entré, Premier Sorti** »



2) Exemple 2 :

On calcule la somme des N premiers entiers positifs de 2 manières :

- Solution non récursive (**itérative**): $\text{Somme} = N + (N-1) + (N-2) + \dots + 1$
- Solution récursive de cette somme serait:
 - ✓ $\text{Somme}(N) = N + \text{Somme}(N-1)$
 - ✓ $\text{Somme}(1) = 1$

Solution itérative	Solution récursive
Fonction Somme (N : entier) : entier VAR i: entier DEBUT Si (N = 1) alors Somme $\leftarrow$ 1 Sinon Somme $\leftarrow$ 0 Pour i de N à 1 (pas = -1) faire Somme $\leftarrow$ Somme + i FinPour Finsi FIN	Fonction Somme (N : entier) : entier Début Si (N = 1) Alors Somme $\leftarrow$ 1 Sinon Somme $\leftarrow$ N+ Somme (N-1) FinSi Fin

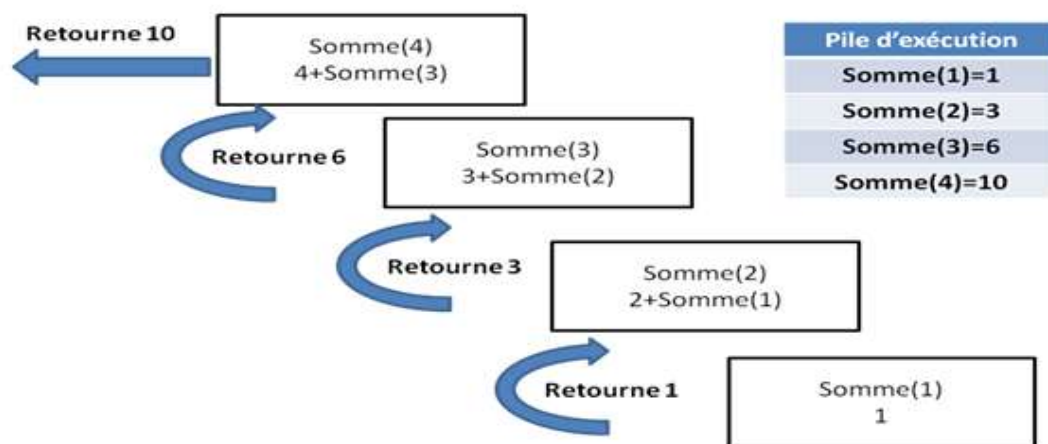
Interprétation

Pour la solution récursive, il est clair que sans le test « Si (N = 1) » cette fonction ne s'arrêterait jamais.

Mécanisme de fonctionnement de la récursivité

Soit $N=4$, supposons qu'on appelle la fonction **Somme (4)**.

Puisque $4 \neq 1$, cette fonction va s'appeler elle-même avec la valeur 3 (Somme (3)). Ce nouvel appel suivra le même processus et appellera Somme (2) qui lui-même appellera Somme (1) qui est la condition d'arrêt. Les appels de fonction sont alors exécutés en ordre inverse, au fur et à mesure qu'ils sont retirés de la pile. Somme (1) retourne 1, Somme(2) retourne 3(=2 + 1), Somme (3) retourne 6 (=3 + 3). Cette dernière valeur sera ajoutée à la valeur 4 et l'appel de Somme (4) retournera la valeur finale 10 (=4 + 6).



Remarque :

Tout problème formulé d'une façon récursive peut également être formulé en façon itérative!

III- Applications

- 1) Ecrire une fonction permettant la recherche dichotomique d'une manière récursive d'un élément x dans un tableau trié T de dimension N . Si x est trouvé son indice sera retourné, -1, sinon.

Version itérative	Version Récursive
Fonction Rech_Dico (T : tab, N, x :entier) : entier VAR milieu,inf,sup : entier Début Rech_Dico $\leftarrow$ -1 inf $\leftarrow$ 1 sup $\leftarrow$ N Tant que (inf $\leq$ sup et Rech_Dico = -1) Faire milieu $\leftarrow$ (inf + sup) DIV 2 Si (T[milieu] = x) Alors Rech_Dico $\leftarrow$ milieu SinonSi (x < T[milieu]) Alors sup $\leftarrow$ milieu - 1 Sinon inf $\leftarrow$ milieu + 1 Finsi FinTantque Fin	Fonction Rech_Dico(T : tab, x,inf,sup :entier) : entier VAR milieu : entier Début Si inf > sup Alors Rech_Dico $\leftarrow$ -1 Sinon milieu $\leftarrow$ (inf + sup) DIV 2 Si (T[milieu] = x) Alors Rech_Dico $\leftarrow$ milieu SinonSi (x < T[milieu]) Alors Rech_Dico $\leftarrow$ Rech_Dico(T,x,inf,milieu-1) Sinon Rech_Dico $\leftarrow$ Rech_Dico(T,x,milieu+1,sup) FinSi FinSi Fin

2) Ecrire une fonction qui permet de rechercher et retourner la position de la première occurrence d'un élément x dans un tableau T de dimension N d'une manière récursive.

Version itérative	Version récursive
Fonction Recherche_Sequentielle(T : Tab, x,N: entier) : entier Var i,pos: entier Début pos ← -1 i ← 1 Tant que (i ≤ N et pos = -1) Faire Si T[i]=x Alors pos ← i Sinon i ← i+1 FinSi FinTantque Recherche_Sequentielle ← pos Fin	Fonction RechSeq_rec(T:tab,x,N,d:entier):entier Début Si d>N Alors RechSeq_rec ← -1 SinonSi (T[d]=x) Alors RechSeq_rec ← d Sinon RechSeq_rec ← RechSeq_rec (T,x,N,d+1)) FinSi Fin