République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Centre Universitaire Belahdj Bouchaib - Ain Témouchent

Initiation à l'Algorithmique

Cours et exercices corrigés

1ère année tronc commun MI, ST et SM

Dr MEDEDJEL Mansour

Maître de conférences en Informatique

Département de Mathématiques et Informatique

Centre Universitaire Belhadj Bouchaib - Ain Temouchent

Préambule

Ce polycopié est destiné essentiellement aux étudiants de la 1ère année du tronc commun Mathématiques et Informatique (MI), ainsi qu'aux étudiants des autres domaines (ST et SM) désirant acquérir des bases solides en programmation sans connaissances préalables. Il s'agit d'un support pédagogique qui couvre une partie fondamentale et essentielle de l'algorithmique constituant ainsi un prérequis indispensable pour la programmation.

L'objectif de ce support est d'initier le lecteur à la résolution des problèmes par la programmation, commençant par l'analyse du problème, la recherche de la solution ou la méthode pour résoudre ce problème, l'écriture de cette solution sous forme d'un algorithme, et enfin la traduction de l'algorithme en programme exécutable par la machine en utilisant un langage de programmation tel que le langage C.

Il convient de noter que ce support de cours est un manuel d'accompagnement de l'étudiant et sa lecture ne constitue en aucun cas une dispense de la présence attentive aux séances de cours et de travaux dirigés ou pratiques.

Table des matières

Table des figures

Introduction générale

Dans ce support, le lecteur est initié à la notion d'algorithmique, ses concepts et ses fondements de base. L'accent est mis également sur les structures de données nécessaires au développement algorithmique tout en insistant sur le côté pratique à travers des exemples et des exercices corrigés à la fin du polycopié. Le côté programmation est aussi fort présent dans ce support à travers des exemples typiques de travaux pratiques. Le langage C est utilisé à cette fin pour ses caractéristiques plus proches aux algorithmes ce qui le rend un langage de programmation pédagogique convenable aux étudiants en phase d'initiation à la programmation.

Ce manuscrit est constitué également de trois parties :

- **Partie I – Cours**

  Cette partie couvre le côté théorique nécessaire à la compréhension du concept d'algorithmique et de programmation. Cette partie est composée également de huit chapitres.

  - **Chapitre 1** : est une initiation à la discipline d'algorithmique.
  - **Chapitre 2** : présente les instructions de base et fondamentales pour l'écriture des algorithmes.
  - **Chapitres 3 et 4** : traitent les instructions qui permettent de contrôler le flux d'instructions de base, il s'agit notamment des instructions conditionnelles (les alternatives) et des instructions d'itération (les boucles).
  - **Chapitres 5 et 6** : sont consacrés aux structures de données indispensables à la manipulation et au stockage des données dans la phase de traitement, à savoir les tableaux et les enregistrements (ou les structures).
  - **Chapitre 7** : présente une initiation à la modularité algorithmique à travers la notion des sous-programmes (fonctions et procédures). Les modes de passage de paramètres par valeur et par adresse sont également mis en évidence ainsi qu'un aperçu sur la notion de la récursivité.
  - **Chapitre 8** : introduit le lecteur à la notion des pointeurs avec des exemples d'application, ainsi qu'au mécanisme de gestion dynamique de la mémoire.

- **Partie II – Exercices corrigés**

  Cette partie est consacrée à la mise en œuvre des connaissances acquises dans les cours à travers un ensemble de travaux dirigés qui couvrent globalement tous les chapitres de la partie I.

- **Partie III – Travaux pratiques en C**

  Cette partie présente une initiation à la programmation à travers des exemples de travaux pratiques à réaliser en langage C avec quelques exemples de code source.

Partie I - Cours

Chapitre 1 - Introduction aux algorithmes

1. Contexte

Le terme **Informatique** est un néologisme proposé en 1962 par *Philippe Dreyfu*[1]*s* pour caractériser le traitement automatique de l'information, c'est une contraction de l'expression « information automatique ». Ce terme a été accepté par l'Académie française en avril 1966, et l'informatique devint alors officiellement la science du traitement automatique de l'information, où l'information est considérée comme le support des connaissances humaines et des communications dans les domaines techniques, économiques et sociaux [3].

2. Notions élémentaires

- **Informatique**

L'informatique est la science du traitement automatique de l'information. Elle traite de deux aspects complémentaires :

- les programmes ou logiciels (software) qui décrivent un traitement à réaliser,
- les machines ou le matériel (hardware) qui exécute ce traitement.

- **Hardware**

C'est l'ensemble des éléments physiques (microprocesseur, mémoire, disques durs, ...) utilisés pour traiter les informations.

- **Software**

C'est un programme (ou ensemble de programmes) décrivant un traitement d'informations à réaliser par un matériel informatique.

- **Algorithme**

La notion d'algorithme est à la base de toute la programmation informatique [8]. La définition la plus simple que l'on peut associer à cette notion est qu'un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre un problème ou effectuer une tâche. Le mot algorithme vient du nom latinisé du mathématicien perse *Al-Khawarizmi*, surnommé « le père de l'algèbre » [11].

[1] Informaticien Français

Exemple : Appel téléphonique

a. Ouvrir son téléphone,

b. Chercher/Composer le numéro du destinataire,

c. Appuyer sur le bouton « Appel ».

Ce mode d'emploi précise comment faire un appel téléphonique. Il est composé d'une suite ordonnée d'instructions (ouvrir, chercher/composez, appuyer) qui manipulent des données (téléphone, numéro, bouton) pour réaliser la tâche d'appel.

3. L'algorithmique

3.1. Définition

L'algorithmique est la science des algorithmes. Elle s'intéresse à l'art de construire des algorithmes ainsi qu'à déterminer leur validité, leur robustesse, leur réutilisabilité, leur complexité ou leur efficacité [3]. L'algorithmique permet ainsi de passer d'un problème à résoudre à un algorithme qui décrit la démarche de résolution du problème. Par conséquent, la programmation consiste à traduire un algorithme dans un langage « compréhensible » par l'ordinateur afin qu'il puisse être exécuté automatiquement.

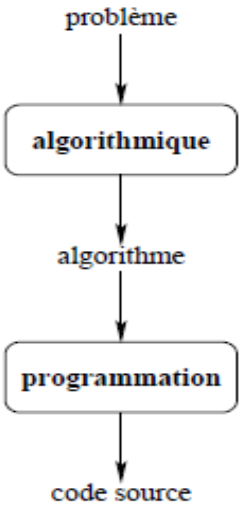


Figure 1. Etapes de développement.

La figure 1 ci-dessus illustre les deux phases nécessaires pour obtenir un code source :

- Phase d'algorithmique qui implique la recherche et l'écriture d'un algorithme ;
- Phase de programmation qui consiste à traduire l'algorithme obtenu en un programme à l'aide d'un langage de programmation (C, Java, Python,…).

Dans la première phase, on doit définir les données qu'on dispose et les objectifs qu'on souhaite atteindre, ainsi que prévoir des réponses à tous les cas possibles.

Exemple : résolution d'une équation de second degré $ax^2+bx+c=0$

→ Les données sont a, b et c

→ Les sorties sont x_1 et x_2

→ Les cas : a=0 et b≠0, a = 0 et b = 0, a ≠0 , ……

3.2. Principe général

Le traitement automatique de l'information consiste à exécuter des instructions (opérations élémentaires et complexes) sur des données d'entrée afin de générer d'autres informations appelées résultats ou données de sortie.

Figure 2. Principe du traitement automatisé.

Exemple : Calcul de la moyenne d'un étudiant

Supposons qu'on doit calculer la moyenne d'un étudiant pour un ensemble de matières. Donc, on doit :

i. Définir le nombre des matières concernées ainsi que les notes et les coefficients ;

ii. Réaliser les opérations suivantes :

- Multiplier chaque note d'une matière par son coefficient,
- Calculer la somme des résultats des multiplications,
- Diviser la somme obtenue par le total des coefficients,

iii. Afficher le la moyenne de l'étudiant (résultat final).

Remarque :

Lorsqu'on écrit un algorithme, les questions suivantes doivent être considérées :

- ✓ Quel est le résultat attendu ?
- ✓ Quelles sont les données nécessaires (informations requises) ?
- ✓ Comment faire (le traitement à réaliser) ?

4. Caractéristiques des algorithmes

4.1. Structure générale

Un algorithme se compose généralement de deux parties :

Partie déclarative : appelée aussi entête de l'algorithme, elle contient généralement les déclarations (des constantes, des variables, etc.).

Partie corps de l'algorithme : constituée d'une ou plusieurs séquences d'instructions faisant appel à des opérations de base à exécuter par l'ordinateur.

Syntaxe :

Algorithme nom_de_l'algorithme ;
    < Liste de variables/constantes > ; } **Partie déclarative (Entête)**
Début
    < Séquence d'instructions > ; } **Partie corps de l'algorithme**
Fin

4.2. Les variables et les constantes

L'élément unitaire de stockage de l'information est appelé **bit**. Un bit ne peut avoir que deux états distincts : 0 ou 1 (vrai ou faux dans la logique). Dans la mémoire de l'ordinateur, les données sont manipulées par groupes de 8 bits (octets), ou plus (mots de 16, 32, 64 bits,…). Une case mémoire est donc appelée **mot** et pour que l'unité centrale puisse stocker une information et la retrouver dans la mémoire, chaque mot est repéré par une **adresse** [4].

Dans la programmation, les adresses mémoire sont représentées par des noms. Le programmeur ne connait pas donc l'adresse d'une case mais plutôt son nom. Il y a donc deux façons de voir la mémoire centrale de l'ordinateur : côté programmeur et côté ordinateur tel qu'il est illustré, à titre d'exemple, dans le schéma suivant (figure 3).

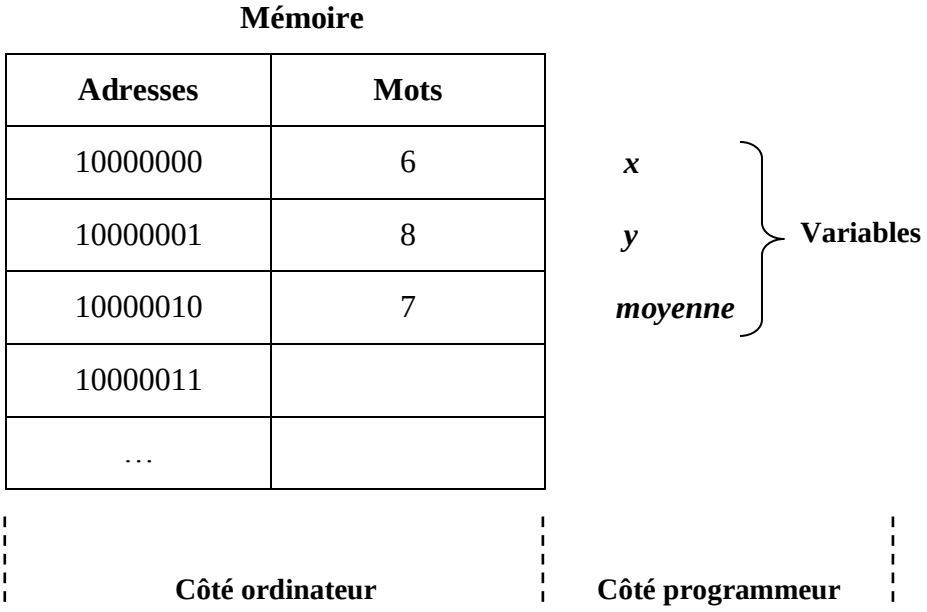


Figure 3. Organisation de la mémoire.

4.2.1. Les variables

Une variable est une case mémoire destiné à contenir des valeurs de type défini au préalable (nombres, caractères, chaînes de caractères,…). Elle possède un **nom**, un **type**, et un **contenu** qui peut être modifié au cours de l'exécution de l'algorithme.

Le mot clé est: **Var**.

4.2.2. Les constantes

La définition d'une constante est la même que celle d'une variable à la différence que sa valeur reste inchangée tout au long du déroulement (exécution) de l'algorithme.

Le mot clé est: **Const**.

Les variables et les constantes sont déclarées selon la syntaxe suivante :

Syntaxe :

Var nom_variable : type ;

Const nom_constante = valeur ;

Remarque :

Dans la partie déclarative, les variables et les constantes sont caractérisées essentiellement par :

- **Un identificateur** : est un nom attribué à la variable ou à la constante, qui peut être composé de lettres et de chiffres mais sans espaces.
- **Un type** : qui définit la nature et la taille de la variable.

Exemple :

Var x, y : entier;

Const alpha = 0,5 ;

Dans cet exemple, nous avons déclaré :

- Deux variables (x et y) de type entier, ce type est décrit dans la sous-section suivante.
- Une constante (alpha) égale à la valeur 0,5 à titre d'exemple.

4.3. Les types de base

Le type d'une variable définit l'ensemble des valeurs que peut prendre la variable, ainsi que l'ensemble des opérations que l'on peut appliquer sur cette variable. Il existe des types simples prédéfinis tels que : entier, réel, caractère et booléen.

4.3.1. Type entier

C'est un type numérique représentant l'ensemble des entiers relatifs, tels que: -9, 0, 31, …. Les opérations permises sur ce type sont : +, - , *, div (division entière) et mod (modulo ou reste de la division entière).

Le mot clé est : **entier.**

Exemple : Var x : **entier** ;

4.3.2. Type réel

C'est un type numérique aussi représentant l'ensemble des nombres réels, tels que : 0.25, -1.33, 2.5 e^{+10},… . Les opérations permises sur ce type sont : +, -, * et /.
Le mot clé est : **réel.**

Exemple : Var y : **réel** ;

4.3.3. Type caractère

Ce type représente tous les caractères alphanumériques tels que : ′a′, ′A′, ′3′, ′%′, ′ ′, …
Les opérations supportées par ce type sont : =, ≠, <, <=, >, >=.
Le mot clé est : **caractère.**

Exemple : Var a : **caractère** ;

4.3.4. Type booléen

Ce type est utilisé dans la logique pour représenter les deux valeurs : vrai et faux.
Les opérations prises en charge sont : NON, ET, OU.
Le mot clé est : **booléen.**

Exemple : Var b : **booléen** ;

4.3.5. Chaîne de caractères

Ce type représente les mots et les phrases tels que "Algorithmique", "Cours", etc. Le mot clé utilisé est : **chaîne**

Exemple : Var c : **chaîne** ;

Globalement, la partie déclarative d'un algorithme peut être représentée comme suit.

Exemple :

```
Var    x, y : entier ;
       z, w : réel ;
       lettre : caractère ;
       nom : chaîne ;
       Etat : booléen ;
Const  n = 100 ;
       arobase = '@' ;
       mot = "bonjour" ;
```

5. Conclusion

Ce chapitre constitue une initiation aux notions basiques de l'écriture des algorithmes. La syntaxe d'un algorithme, la notion de variable et de constante, ainsi que leurs types sont définis et expliqués à travers des exemples simples. Ceci constitue pour le lecteur un prérequis de base qui lui permettra de comprendre la notion d'instructions algorithmiques dans les prochains chapitres.

Chapitre 2 - Les instructions simples

1. Introduction

Un algorithme, par définition, est un ensemble d'instructions qui peuvent être simples ou complexes. Dans ce chapitre, on s'intéressera aux instructions simples notamment : les instructions d'affectation, de lecture et d'écriture.

2. L'instruction d'affectation

Cette instruction est élémentaire en algorithmique, elle permet d'assigner une valeur à une variable selon la syntaxe suivante :

variable ← expression ;

Une instruction d'affectation est exécutée comme suit :

- évaluation de l'expression située à droite de l'instruction, et
- affectation du résultat à la variable située à gauche de l'instruction.

L'expression peut être :

• une constante (c ← 10)

• une variable (v ← x)

• une expression arithmétique (e ← x + y)

• une expression logique (d ← a ou b)

Remarque :

- Une constante ne figure jamais à gauche d'une instruction d'affectation.
  Exemple d'instruction fausse : **Const** z = 1 ;
  z ← 2 ; ***« Faux »***
- Après une affectation, l'ancien contenu d'une variable est substitué (écrasé) par le nouveau contenu.
  Exemple : **Var** a : **entier** ;
  a ← 1 ;
  a ← 2 ;
  Après la deuxième affectation, la valeur de a est devenue 2 (la valeur 1 est écrasée).
- Une instruction d'affectation doit se faire entre deux types compatibles.
  Exemple : **Var** x, y : **entier** ; z : réel ; a, b : **caractère** ;

Instructions correctes	Instructions incorrectes
x ← y ; y ← x ; z ← x ; a ← b ; b ← a ;	x ← z ; x ← a ; b ← y ; a ← z ;

Les expressions arithmétiques ou logiques sont composées d'au moins deux termes reliés par un ou plusieurs opérateurs dont on peut distinguer :

a) Les opérateurs arithmétiques (par ordre de priorité) :

^ ou ** : Puissance

* , / , mod : Multiplication, Division et Modulo

+ , - : Addition et Soustraction

b) Les opérateurs logiques ou booléens :

NON : Non logique (négation)

ET : Et logique (conjonction)

OU : Ou logique (disjonction)

NON ET : négation de conjonction

NON OU : négation de disjonction

c) Les opérateurs de comparaison ou relationnels :

> , >= : supérieur et supérieur ou égal

< , <= : inférieur et inférieur ou égal

= , ≠ (ou < >) : égal et différent

Remarque :

Les expressions logiques peuvent être composées des opérateurs logiques et/ou relationnels. Par exemple, (A<20) ET (B>=10) est Vrai si A est inférieur à 20 et B est égal ou supérieur à 10, et faux sinon.

3. L'instruction de lecture

Cette instruction est très primordiale dans un algorithme. Elle permet de lire des valeurs en entrée (*input*) et les affecter aux variables stockées dans la mémoire. Les valeurs affectées sont souvent des données introduites à partir d'un périphérique d'entrée tel que le clavier.

Syntaxe :

Lire (var1, var2,…) ;

Exemple :

Lire(x) : lit et stocke une valeur donnée dans la case mémoire associée à x.

Lire(x, y) : lit et stocke deux valeurs, la première dans x et la deuxième dans y.

Illustration :

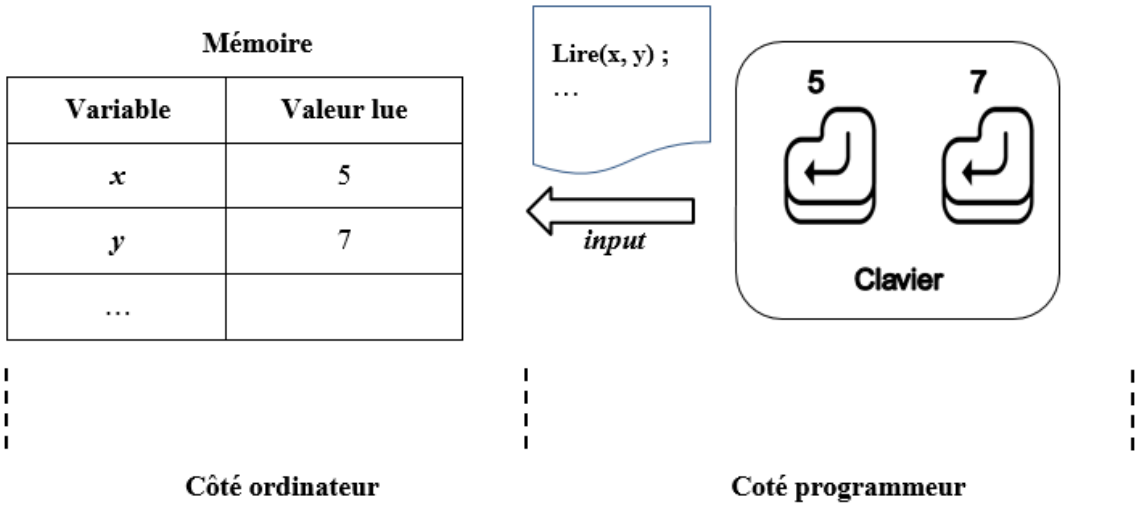


Figure 4. Opération de lecture.

4. L'instruction d'écriture

Cette instruction est aussi d'une grande importance dans les algorithmes. Elle permet d'écrire en sortie (*output*) les données résultant d'un traitement effectué par l'algorithme (valeur, texte, …) en les affichant par exemple sur un périphérique de sortie tel que l'écran.

Syntaxe :

Ecrire (var1, var2, expr1, expr2, …) ;

Remarque :

Dans le cas d'écriture d'une expression, c'est le résultat d'évaluation de cette expression qui est affiché et non pas l'expression elle-même. Par exemple :

Soient deux variables x et y tel que x=5 et y=7, l'instruction :

Ecrire (x+y) ;

Affiche en sortie le résultat d'addition de x et y (soit 12).

Illustration :

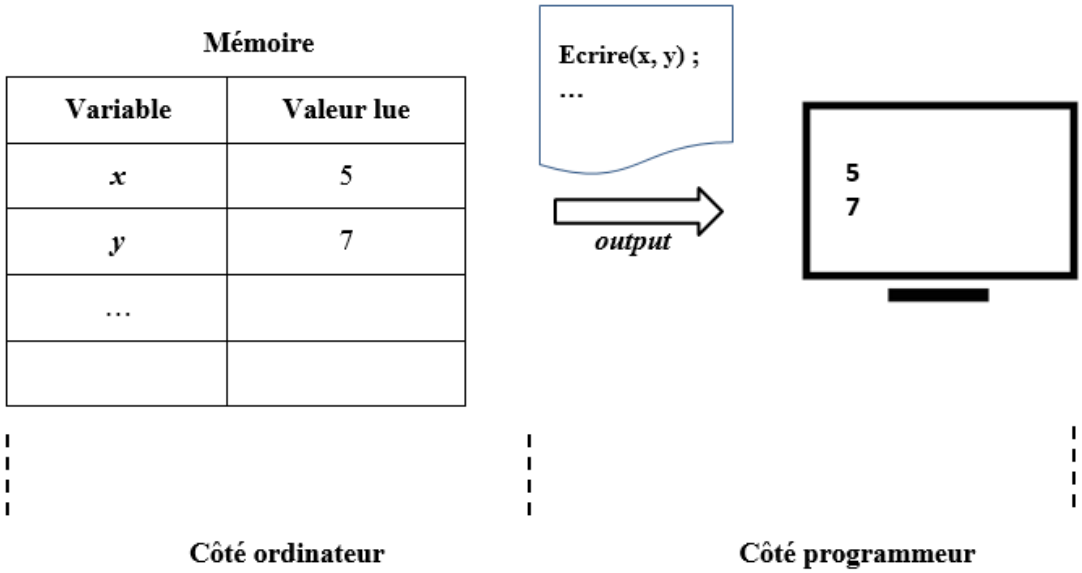


Figure 5. Opération d'écriture.

Exemple d'algorithme contenant les trois instructions précédentes :

```
Algorithme Moyenne_deux_réels
Var x, y, z : réel ;
Début
    Ecrire (″Donner la première valeur :″) ;
    Lire (x) ;
    Ecrire (″Donner la deuxième valeur :″) ;
    Lire (y) ;
    z ← (x + y)/2 ;
    Ecrire (″La moyenne est : ″, z) ;
    // On peut remplacer les deux dernières instructions par une seule :
    Ecrire (″La moyenne est : ″, (x + y)/2 ) ; // dans ce cas on a pas besoin de z
Fin
```

Dans cet algorithme, si l'utilisateur introduit 10 pour **x** et 20 pour **y** alors l'affichage sera :
La moyenne est : 15

5. Conclusion

Dans ce chapitre, nous avons présenté les instructions algorithmiques fondamentales à savoir l'affectation, la lecture et l'écriture. Ces trois simples instructions sont incontournables dans l'écriture d'un algorithme et constituent l'un des moyens les plus simples qui permettent au programmeur d'interagir avec son ordinateur à travers à travers des actions d'entrées/sorties.

Chapitre 3 - Les instructions conditionnelles (les alternatives)

1. Introduction

Les algorithmes comportent généralement deux types d'instructions :

- Les instructions simples : qui permettent la manipulation des variables telles que l'affectation, la lecture et l'écriture.
- Les instructions de contrôle : qui précisent l'enchainement chronologique des instructions simples. C'est en particulier le cas des instructions conditionnelles ou les tests.

2. Structure d'un test

Il existe deux formes de test : forme simple (ou réduite) et forme complète.

2.1. Forme simple

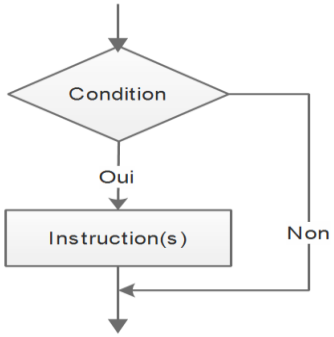
Dans cette forme, une action qui correspond à une ou plusieurs instructions, est exécuté si une condition est vérifiée. Sinon l'algorithme passe directement au bloc d'instruction qui suit immédiatement le bloc conditionnel.

Syntaxe :

Si (condition) **Alors**

instruction(s) ; // *action*

Finsi



Remarque :

La condition évaluée après l'instruction « Si » est une variable ou une expression booléenne qui, à un moment donné, est Vraie ou Fausse. par exemple : x=y ; x <= y ; ...

Exemple :

x ← 5 ; y ← 9 ;
Si (x = y) **Alors Ecrire** (″x est égale à y″) ;

Dans cet exemple, le message « x est égale à y » ne sera pas affiché puisque la condition (x = y) n'est pas vérifiée.

2.2. Forme complète

Cette forme permet de choisir entre deux actions selon qu'une condition est vérifiée ou non.

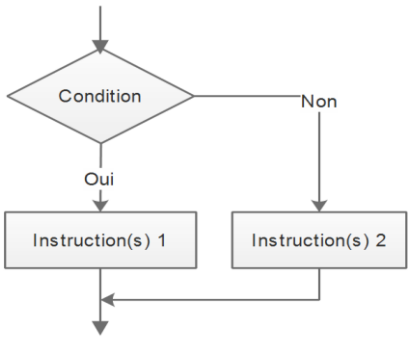
Syntaxe :

Si (condition) **Alors**

instruction(s) 1 ; // *action1*

Sinon instruction(s) 2 ; // *action2*

Finsi



Remarque :

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme d'une simple comparaison. Par exemple, la condition x ∈ [0, 1[s'exprime par la combinaison de deux conditions x >= 0 et x < 1 qui doivent être vérifiées en même temps.

Pour combiner ces deux conditions, on utilise les opérateurs logiques. Ainsi, la condition x ∈ [0, 1[pourra s'écrire sous la forme : (x >= 0) ET (x < 1). Cette dernière est appelée une condition composée ou complexe.

Exemple (sur la forme complète d'un test) :

x ← 5 ; y ← 9 ;

Si (x = y) **Alors Ecrire** (″x est égale à y ″) ;

Sinon Ecrire (″x est différente de y ″) ;

Avec cette forme, on peut traiter les deux cas possibles. Si la condition (x=y) est vérifiée, le premier message est affiché, si elle n'est pas vérifiée, le deuxième message est affiché.

3. Tests imbriqués

La forme **« Si … Alors…Sinon »** permet deux choix correspondants à deux traitements différents. Dans d'autres situations, on pourra avoir plus de deux cas ce qui rend cette alternative insuffisante pour traiter tous les cas possibles (voir exemple ci-dessous).

La forme complète permet de choisir entre plusieurs actions en imbriquant des formes simples selon la syntaxe ci-dessous.

Syntaxe :

Si (condition1) **Alors** instruction(s) 1 ;

Sinon Si (condition2) **Alors** instruction(s) 2 ;

Sinon Si (condition3) **Alors** instruction(s) 3 ;

```
                …
                Sinon  instruction(s) N ;
                FinSi
        FinSi
FinSi
```

Exemple : Etat de l'eau [10]

Dans les conditions normales de température et de pression, l'eau est sous forme de glace si la température est inférieure ou égale à 0° C, sous forme de liquide si la température est comprise entre 0° C et 100° C et sous forme de vapeur au-delà de 100° C. Ecrivons l'algorithme qui permet de vérifier l'état de l'eau selon sa température.

La solution pourrait être comme suit :

```
Algorithme Etat_Eau ;
Var t : réel ;
Début
    Ecrire ("Donner la température de l'eau :") ;
    Lire (t) ;
    Si (t <= 0) Alors  Ecrire ("Etat solide") ;
    FinSi
    Si (t > 0 ET t < 100) Alors Ecrire ("Etat liquide") ;
    Finsi
    Si (t >= 100) Alors Ecrire ("Etat gazeux") ;
    Finsi
Fin
```

Cet algorithme est correct mais il évalue les trois conditions qui portent sur la même variable et qui sont exclusives. En effet, si (t <= 0), alors on ne peut pas avoir (t>= 0 et t < 100) ni (t > 100). Il est donc inutile d'évaluer les deux dernières conditions si la première est vérifiée, ou d'évaluer la dernière condition si la deuxième est vérifiée. Pour éviter ce cas de figure, il sera préférable d'utiliser des tests imbriqués comme suit :

```
…
Début
    Ecrire ("Donner la température de l'eau:") ;
    Lire (t) ;
    Si (t <= 0) Alors Ecrire ("Etat solide") ;
    Sinon Si (t < 100) Alors Ecrire (" Etat liquide") ;
            Sinon Ecrire ("Etat gazeux") ;
            Finsi
    Finsi
Fin
```

L'organigramme correspondant est illustré dans la figure 6.

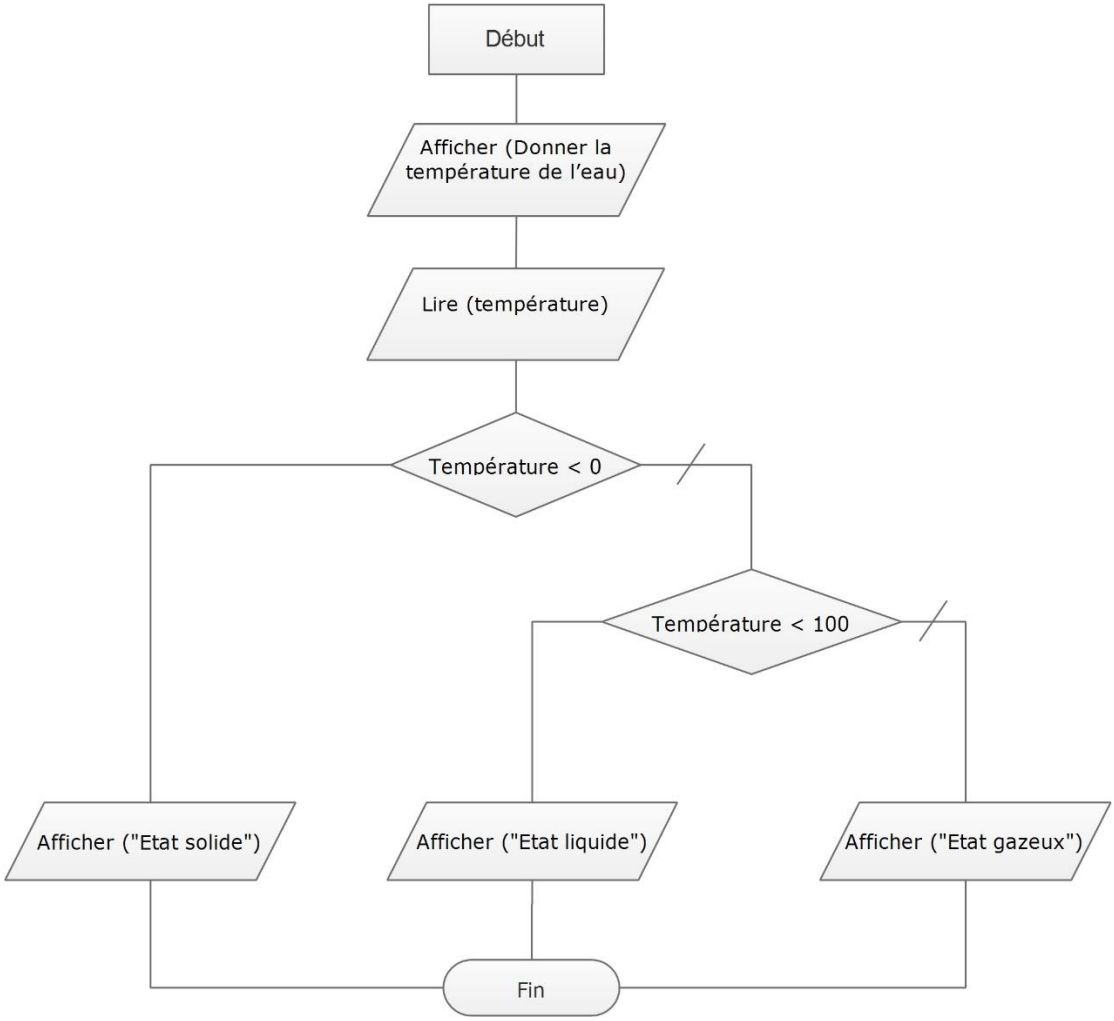


Figure 6. Organigramme « Etat de l'eau ».

Donc, l'utilisation de tests imbriqués permet de :

- Simplifier le (pseudo-) code : à travers l'imbrication nous n'avons utilisé que deux conditions simples au lieu de trois conditions dont une est composée.
  - ➔ un algorithme (ou programme) plus simple et plus lisible.
- Optimiser le temps d'exécution : dans le cas où la première condition est vérifiée, l'algorithme passe directement à la fin, sans tester le reste qui est forcément faux.
  - ➔ un algorithme (ou programme) plus performant à l'exécution.

Remarque :

Nous avons les équivalences suivantes :

$$\text{NON (A ET B)} \Leftrightarrow \text{NON A OU NON B}$$

$$\text{NON (A OU B)} \Leftrightarrow \text{NON A ET NON B}$$

Ainsi, toute structure de test avec l'opérateur logique ET peut être exprimée d'une manière équivalente avec l'opérateur logique OU et vice-versa. Par conséquent, les deux alternatives suivantes sont équivalentes [7].

Si A **ET** B **Alors**
    instruction(s) 1 ;
Sinon
    instruction(s) 2 ;
Finsi

⟺

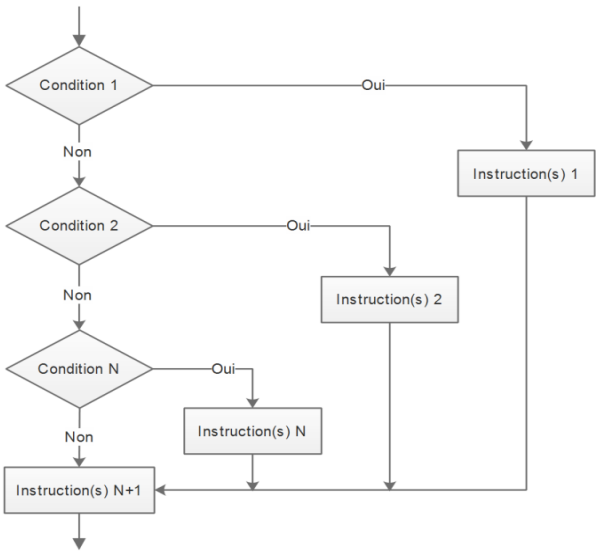
Si NON A **OU** NON B **Alors**
    instruction(s) 2 ;
Sinon
    instruction(s) 1 ;
Finsi

4. Les choix multiples

Il existe une autre variante d'instructions conditionnelles qui permet d'effectuer des actions différentes suivant les différentes valeurs que peut avoir une variable. Cette structure est décrite comme suit :

Syntaxe :

Selon (variable)
    valeur1 : instruction(s) 1 ;
    valeur2 : instruction(s) 2 ;
    …
    valeurN : instruction(s) N ;
    défaut : instruction(s) par défaut;
FinSelon ;



Remarque :

Dans la structure de test à choix multiples :

- L'action peut être une suite d'instructions ;
- La valeur est une constante de même type que la variable ;
- La partie « défaut » est exécutée si aucun des autres cas n'est vérifié ;
- L'exécution des différents cas (y compris le cas par défaut) est exclusive c'est-à-dire l'exécution d'un seul cas provoque la sortie de cette structure.

Exemple :

Dans ce qui suit, le nom du jour de la semaine correspondant est affiché selon la valeur de la variable « jour ».

```
jour ← 5 ;
Selon jour
        1 : Ecrire ("Dimanche") ;
        2 : Ecrire ("Lundi") ;
        3 : Ecrire ("Mardi") ;
        4 : Ecrire ("Mercredi") ;
        5 : Ecrire ("Jeudi") ;
        6 : Ecrire ("Vendredi") ;
        7 : Ecrire ("Samedi") ;
        Défaut : Ecrire ("Numéro de jour invalide.") ;
FinSelon
```

Donc, l'expression « Jeudi » est affichée dans ce cas.

5. Conclusion

Dans ce chapitre, nous avons présenté le principe de la condition, suivant lequel un algorithme peut effectuer une action ou prendre une décision. Ceci est mis en œuvre à travers les instructions conditionnelles ou tout simplement les tests avec leurs différentes formes vues précédemment.

Chapitre 4 - Les instructions itératives (les boucles)

1. Introduction

Considérons le même exemple qu'on a vu précédemment concernant le calcul de la moyenne générale d'un étudiant. Pour se faire, on doit :

- Lire toutes les notes (et leurs coefficients) de l'étudiant,
- Calculer la somme des notes,
- Diviser la somme obtenue sur le nombre (ou sur la somme des coefficients).

Si l'on veut maintenant calculer la moyenne d'un autre étudiant, les mêmes instructions doivent être répétées.

Pour N d'étudiants, il nous faudra donc répéter N fois la même séquence d'instructions.

Cet exemple soulève deux questions importantes :

1- Comment éviter d'écrire plusieurs fois la même séquence d'instructions ?

2- Combien de fois doit-on répéter l'exécution de la séquence d'instructions pour obtenir le résultat attendu ?

Pour répondre à ces questions, de nouvelles instructions de contrôle sont introduites. Il s'agit des instructions itératives (appelées aussi les boucles ou les itérations).

2. Définition

Une boucle (ou itération) est une instruction de contrôle qui permet de répéter plusieurs fois un ensemble d'instructions. Généralement, deux cas sont distingués :

- Le nombre de répétitions est connu.
- Le nombre des répétitions est inconnu ou variable.

3. L'instruction « Pour »

Lorsque le nombre de répétitions est déterminé (connu), l'utilisation de l'instruction « Pour » est privilégiée. Une structure de boucle avec l'instruction « Pour » s'arrête une fois que le nombre de répétitions est atteint. Cette structure possède un indice (compteur) de contrôle d'itérations caractérisé par :

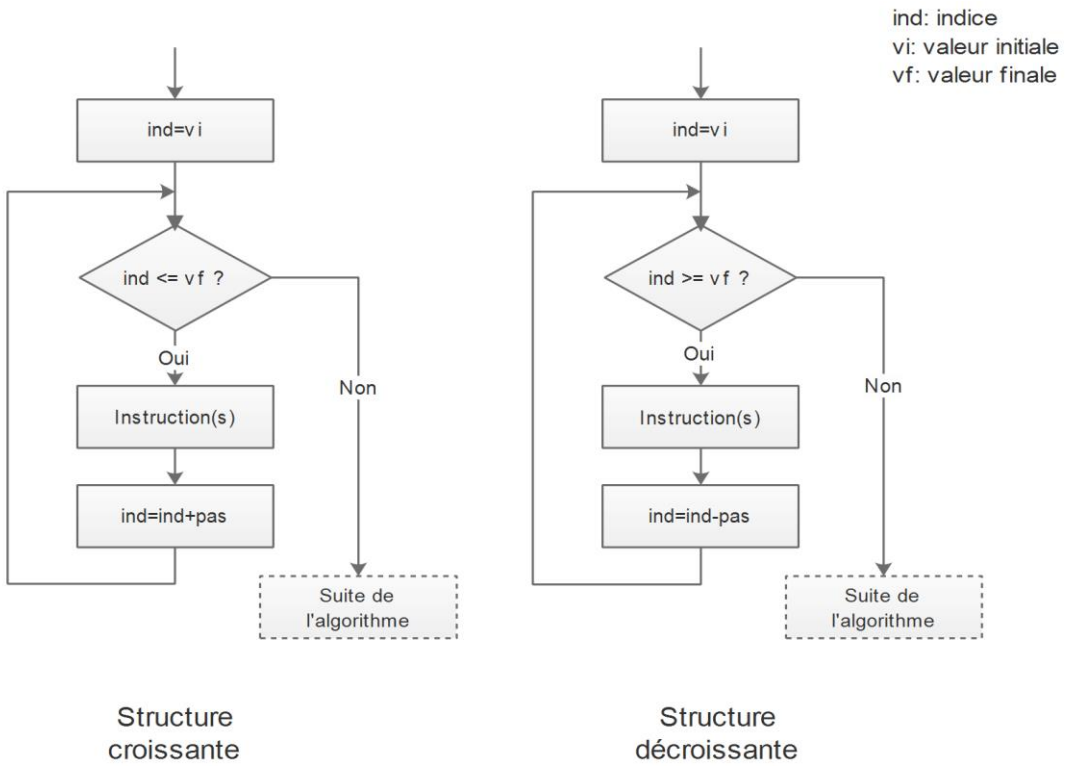
- une valeur initiale,
- une valeur finale,
- un pas de variation.

Syntaxe :

Pour indice **de** début **à** fin **Pas** valeur_du_pas

instruction(s) ;

FinPour



Cette structure est dite « croissante » lorsque la valeur initiale de l'indice est inférieure à sa valeur finale, le pas de variation est par conséquent positif**.** Autrement, elle est dite « décroissante ».

Exemple : un compteur croissant/décroissant

Les deux algorithmes suivants comptent de 1 à N et de N à 1 respectivement.

```
Algorithme compteur_croissant ;
Var  i : entier ;
Const N=100 ;
Début
   Pour i de 1 à N    /* par défaut le  pas = 1 */
         Ecrire (i);
   FinPour
Fin
```

Résultat d'exécution **:** 1,2, 3, … , 99, 100

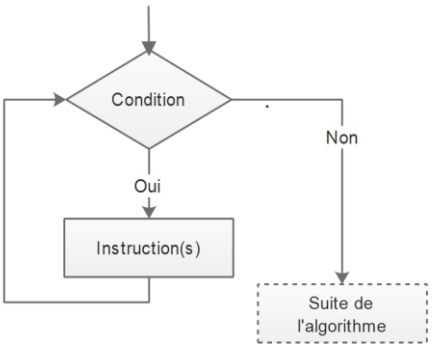
```
Algorithme compteur_decroissant ;
Var  i : entier ;
Const N=100 ;
Début
   Pour i de N à 1  /* par défaut le  pas = -1 */
         Ecrire (i);
   FinPour
Fin
```

Résultat d'exécution : 100, 99, 98, … , 2, 1

Remarque : Si la valeur du « pas » n'est pas précisée dans l'instruction « Pour », elle est par défaut égale à un (1).

4. L'instruction « Tant que… faire »

Cette instruction permet de tester une condition et répéter le traitement associé tant que cette condition est vérifiée.



Syntaxe:

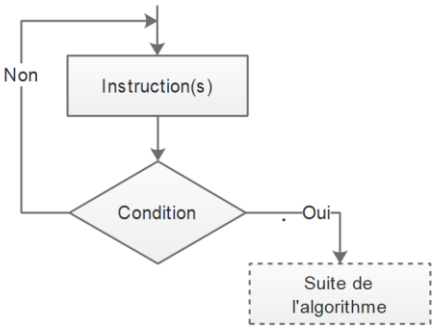
Tant que condition **faire**
    instruction(s) ;
FinTq

Exemple : Réécrivons l'algorithme précédent avec cette instruction.

```
Var i : entier ;
Début
    i ← 1 ;
    Tant que (i<=100) faire
        Ecrire (i) ;  i ← i+1 ;
    FinTq
Fin
```

6. L'instruction « Répéter… jusqu'à »

Dans cette instruction, un traitement est exécuté au moins une fois puis sa répétition se poursuit jusqu'à ce que la condition soit vérifiée.



Syntaxe:

Répéter
    instruction(s) ;
Jusqu'à (condition) ;

Exemple : Soit l'algorithme suivant :

```
Var n, p : entier ;
Début
  Répéter
      Ecrire ("Donner un nombre :") ; Lire (n) ;
      p ← n*n ; Ecrire (p);
  Jusqu'à (n=0)
  Ecrire ("Fin de l'algorithme") ;
Fin
```

Les instructions encadrées par les mots **répéter** et **jusqu'à** constituent le bloc de la boucle qu'il faut répéter jusqu'à ce que la condition **(n=0)** soit vérifiée. Donc le nombre de répétitions de cette boucle dépend des données fournies par l'utilisateur.

Question ?

Réécrire l'algorithme précédent avec « Tant que… faire » puis avec « Pour ».

Remarque :

Dans la boucle « Répéter… jusqu'à », la condition telle qu'elle est exprimée ci-dessus, constitue une **condition d'arrêt** de la boucle ; mais réellement, cela diffère selon le langage de programmation utilisé. Par exemple, en Pascal, la condition de cette boucle est une condition d'arrêt. Alors qu'en langage C, cette condition est exprimée en tant qu'une **condition de continuation**.

7. La notion du compteur

Un compteur est une variable associée à la boucle dont la valeur est incrémentée de un à chaque itération. Elle sert donc à compter le nombre d'itérations (répétitions) de la boucle.

La notion du compteur est associée particulièrement aux deux boucles : « **Répéter…jusqu'à** » et « **Tant que…faire** ». Par contre, dans la boucle « **Pour** », c'est l'indice qui joue le rôle du compteur.

L'utilisation du compteur dans les deux premières boucles est exprimée ainsi :

Bloc de la boucle

```
compt ← 0 ;
Répéter
    instruction(s) ;
    …
    compt ← compt +1 ;
Jusqu'à (condition) ;
```

```
compt ← 0 ;
Tant que (condition) faire
    instruction(s) ;
    …
    compt ← compt +1 ;
FinTant que ;
```

Remarque :

Il faut toujours initialiser le compteur avant de commencer le comptage. La variable « compt » (utilisée ci-dessus comme compteur), a été initialisée à zéro (0) avant le début de chaque boucle.

L'instruction « compt ← compt +1 » incrémente la valeur de « compt » de un (1). Elle peut être placée n'importe où à l'intérieur du bloc de la boucle.

Exemple :

```
i ← 0 ;
Répéter
    Ecrire (i);
    i ← i +1 ;
Jusqu'à (i=5) ;

Résultat d'exécution : 0, 1, 2, 3, 4
```

```
i ← 0 ;
Tant que (i<5) faire
    Ecrire (i);
    i ← i +1 ;
FinTant que ;

Résultat d'exécution : 0, 1, 2, 3, 4
```

8. La notion d'accumulation

Cette notion est fondamentale en programmation. Elle est utilisée notamment pour calculer la somme d'un ensemble de valeurs. L'instruction correspondante se présente ainsi :

variable ← variable + valeur ;

Cette instruction consiste à ajouter une valeur à une variable numérique, puis affecter le résultat dans la variable elle-même. En d'autres termes, la nouvelle valeur de variable égale à l'ancienne plus une certaine valeur [4].

Exemple : calcul de la somme de n valeurs données par l'utilisateur :

```
Var  i, n : entier ; som, val : réel ;
Début
    Écrire ("Donner le nombre de valeurs :") ; Lire (n) ;
    som ← 0 ;
    Pour i de 1 à n
            Écrire ("Enter une valeur :") ; Lire (val) ;
            som ← som + val ;
    Finpour
    Écrire ("La somme des valeurs est égale à :", som) ;
Fin
```

9. Les boucles imbriquées

Les boucles peuvent être imbriquées les unes dans les autres. Deux ou plusieurs boucles imbriquées peuvent être aussi les mêmes ou différentes.

Exemple :

```
Pour i de 1 à 2                          ┐
    Écrire ("i = ", i) ;                 │
    Pour j de 1 à 3          ┐           │ boucle 1
        Écrire ("j = ", j) ; │ boucle 2  │
    Finpour                  ┘           │
Finpour                                  ┘
```

Dans l'exemple ci-dessus, chaque itération de la boucle extérieure (boucle 1) exécute la boucle intérieure (boucle 2) jusqu'à la fin avant de passer à l'itération suivante, et ainsi de

suite jusqu'à la fin des deux boucles. Ainsi, le résultat d'exécution peut être représenté comme suit :

i = 1
j = 1
j = 2
j = 3
i = 2
j = 1
j = 2
j = 3

Remarque :

Des boucles peuvent être imbriquées ou successives. Cependant, elles ne peuvent jamais être croisées. Par exemple, l'algorithme suivant est faux puisqu'il comporte deux boucles croisées :

```
Var i, j : entier ;
Début
    i←1 ; j←1 ;
    Répéter
        Écrire i ;
        Répéter
            Écrire j ;
   i←i+1 ;
   Jusqu'à i>2
            j←j+1 ;
       Jusqu'à j>3
Fin
```

10. Conclusion

Ce chapitre a été consacré aux structures itératives ou boucles qui permettent de répéter l'exécution d'une séquence d'instructions plusieurs fois selon un nombre fixe ou certains critères dont l'utilisation a été explicitée à travers différents exemples. Ainsi, ces instructions sont d'une grande importance dans la manipulation de certaines structures de données telles que les tableaux que nous aborderons dans le prochain chapitre.

Chapitre 5 - Les tableaux

1. Introduction

Supposons que l'on a besoin de stocker et de manipuler les notes de 100 étudiants. On doit, par conséquent, déclarer 100 variables : n1, n2,…, n100. Vous pouvez remarquer que c'est un peu lourd de manipuler une centaine de variables (avec 100 fois de lecture/écriture…).
Imaginons maintenant le cas pour une promotion de 1000 étudiants, alors là devient notre cas un vrai problème.
En algorithmique (et en programmation), on peut regrouper toutes ces variables en une seule structure qui s'appelle **tableau**.
Un tableau est un ensemble de variables de même type ayant toutes le même nom.
Suite à cette définition, la question suivante se pose :

- Comment peut-on différencier entre des variables ayant le même nom ?

La réponse est dans la notion du tableau lui-même où chaque élément est repéré par un **indice**. Ce dernier est un numéro (généralement un entier) qui permet de différencier chaque élément du tableau des autres. Ainsi, les éléments du tableau ont tous le même nom, mais pas le même indice. Pour accéder à un élément d'un tableau, on utilise le nom du tableau suivi de l'indice de l'élément entre crochets [4].

Exemple

Soit le tableau T contenant les valeurs suivantes : 5, 10, 29, 3, 18 et 14 :
L'organisation du tableau T dans la mémoire peut être représentée comme suit :

T ⟹

indices :	i=0	i=1	i=2	i=3	i=4	i=5
valeurs :	T[0]=5	T[1]=10	T[2]=29	T[3]=3	T[4]=18	T[5]=14

2. Tableaux à une seule dimension

Dans ce type de tableaux, chaque élément est accessible (pour lecture ou modification) par un seul indice.

2.1. Déclaration

La syntaxe de déclaration d'un tableau à une seule dimension est la suivante :

Tableau nom_du_tableau [taille] : **type ;**

Exemple : Tableau Notes [100] : **réel ;**

Remarque :

L'indice d'un élément dans un tableau, peut être exprimé comme un nombre, mais aussi il peut être exprimé comme une variable ou une expression calculée [10].

La valeur de l'indice doit être toujours :

- **Supérieur ou égal à 0** : dans quelques langages, le premier élément d'un tableau porte l'indice 1 (comme en Pascal). Mais dans d'autres, comme c'est le cas en langage C, la numérotation des indices commence à zéro. Par exemple *Notes[1]* est le deuxième élément du tableau Notes.
- **de type entier** : quel que soit le langage, l'élément *Notes[1,…]* n'existe jamais.
- **Inférieur ou égal au nombre des éléments du tableau (moins 1 si l'on commence à zéro)**: En langage C, si un tableau T est déclaré comme ayant 10 éléments, la présence, dans une ligne du corps de l'algorithme, de l'expression *T[10]* déclenchera automatiquement une erreur.

2.2. Manipulation

Une fois déclaré, un tableau peut être manipulé comme un ensemble de variables simples. Les trois manipulations de base sont l'affectation, la lecture et l'écriture [4].

2.2.1. L'affectation

L'affectation d'une valeur v à un élément i d'un tableau T de type numérique, se fait par :

T[i] ← v ;

Par exemple, l'instruction : T[0] ← 5 ; affecte la valeur 5 au premier élément du tableau T.

Illustration :

T ⟹

indices :	i=0	i=1	i=2	i=3	i=4	i=5
valeurs :	5					

Supposons maintenant que l'on veut affecter la même valeur à tous les éléments du tableau T, on utilisera pour cela une boucle :

Pour i de 0 à n-1 // *n est la taille de T*
    T[i] ← 5 ;
FinPour

Cette boucle permet de parcourir tout le tableau T en affectant à chaque élément la valeur 5.

Illustration :

T ⟹

indices :	i=0	i=1	i=2	i=3	i=4	i=5
valeurs :	5	5	5	5	5	5

2.2.2. La lecture

Il est possible aussi d'affecter des valeurs aux éléments d'un tableau par une instruction de lecture.

Exemple :

Ecrire "Entrer une note :" ;

Lire T[0] ;

Dans cet exemple, la valeur saisie est affectée au premier (1er) élément du tableau T.

Illustration :

T ⟹	***indices :***	i=0	i=1	i=2	i=3	i=4	i=5
	valeurs :	0					

2.2.3. L'écriture

De même que la lecture, l'écriture de la valeur d'un élément du tableau s'écrira comme suit :
Ecrire T[i] ; Cette instruction permet d'afficher la valeur de l'élément i du tableau T.

Remarque :

Les éléments d'un tableau sont manipulés de la même façon que les variables simples. S'il s'agit d'un tableau de type numérique, les éléments peuvent être utilisés dans l'évaluation des expressions numériques du genre :

x ← (Notes [1] + Notes [2]) / 2 ;

Notes [0] ← Notes [0] + 1 ;

Exemple :

Soit *Notes* un tableau de valeurs réelles tel qu'il est illustré dans le schéma qui suit.

Illustration :

Notes ⟹	***indices :***	i=0	i=1	i=2	i=3	i=4	i=5
	valeurs :	**10**	**15**	**7**	**8.25**	**11.5**	**16**

L'exécution de l'instruction : x ← (Notes [1] + Notes [2]) / 2 ;

Est équivalente à l'opération **:** x **=** $\frac{\mathbf{15+7}}{\mathbf{2}}$ **= 11**

2.3. Application 1

Ecrivons un algorithme qui permet de lire des valeurs saisies par l'utilisateur dans un tableau de taille 20, et les afficher par la suite.

Algorithme :

```
Var i : entier ;
Tableau Tab [20] : réel ;
Début
    Pour i de 0 à 19
        Ecrire ("Donner une valeur :") ;
        Lire (Tab[i]) ;
    FinPour
    Pour i de 0 à 19
            Ecrire ("Valeur ", i, "=", Tab[i]) ;
    FinPour
Fin
```

L'exécution du programme correspondant à l'algorithme ci-dessus, permet d'initialiser le tableau Tab comme suit :

- Après la fin de la 1ère boucle (boucle de lecture) :

Tab ⇨

indices :	i=0	i=1	…	i=18	i=19
valeurs :	**5**	**34**	**…**	**14.5**	**60**

- Après la fin de la 2ème boucle (boucle d'affichage), on aura, sur écran par exemple, l'affichage suivant :

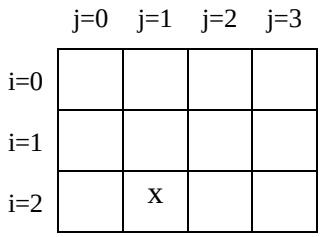
```
Valeur 0 = 5
Valeur 1 = 34
…
Valeur 18 = 14,5
Valeur 19 = 60
```

Remarque : les valeurs : 5, 34, 14.5 et 60 sont un exemple d'échantillon de valeurs saisies par l'utilisateur.

3. Tableaux à deux dimensions

Les tableaux à deux dimensions se présentent généralement sous forme d'un ensemble de lignes et de colonnes (Matrice). Par conséquent, chaque élément est repéré par deux indices.

Exemple : le tableau T ci-dessous possède 3 lignes et 4 colonnes.



i : indice des lignes
j : indice des colonnes

Donc, T[2][1] (i=2 et j=1) désigne l'élément de la 3ème ligne et la 2ème colonne (en commençant de zéro bien sûr).

3.1. Déclaration d'un tableau à deux dimensions

Syntaxe :

Tableau nom_tableau [taille1][taille2] : **type ;**

Exemple : Tableau Notes [10][20] : **réel ;**

Le tableau Notes est composé de 10 lignes et 20 colonnes. Ce tableau pourra contenir donc 10*20 soit 200 valeurs réelles.

Remarque :

L'utilité d'un tableau à deux dimensions réside dans la possibilité de déclarer un seul tableau au lieu de déclarer plusieurs tableaux identiques. En effet, le tableau de l'exemple précédent est équivalant à 10 tableaux simples de 20 éléments chacun. En d'autres termes, la déclaration :

Tableau Notes [10][20] : **réel ;**

remplace celle-ci :

Tableau Notes1 [20], Notes2 [20],…, Notes10 [20] : **réel ;**

3.2. Manipulation d'un tableau à deux dimensions

Un tableau à deux dimensions est manipulé de la même façon qu'un tableau simple (à une seule dimension) que ce soit pour l'affectation, la lecture ou l'écriture. Ces trois opérations sont illustrées dans la sous-section suivante.

3.3. Application 2

Reprenons l'algorithme de la sous-section 2.3 mais cette fois-ci pour un tableau de 5 lignes et 20 colonnes :

Var i, j : **Entier ;**
Tableau Tab [5][20] : **réel ;**

```
Début
    Pour i de 0 à 4
      Pour j de 0 à 19
            Ecrire ("Donner une valeur :") ;     Lire (Tab [i][j]) ;
      Finpour
    Finpour
    Pour i de 0 à 4
      Pour j de 0 à 19
            Ecrire  (Tab [i][j]) ;
      Finpour
    Finpour
Fin
```

4. Tableaux à n dimensions

Les tableaux à n dimensions (n>2), peuvent être utilisés pour diverses raisons telles que la création et le traitement des objets 3D par exemple qui nécessitent des tableaux de 3 dimensions au minimum. La déclaration de ce type de tableaux est comme suit :

Syntaxe :

Tableau nom_tableau [taille1][taille2] … [tailleN] : **type ;**

Exemple : Tableau T [10][20][50] : **réel ;** *// un tableau T à 3 dimensions*

La manipulation d'un tableau à plusieurs dimensions suit le même principe que celle des tableaux à deux dimensions. Ceci s'appuie sur l'utilisation des boucles imbriquées pour parcourir le tableau, de sorte qu'il y aura autant de boucles qu'il y a de dimensions.

5. La recherche dans un tableau

5.1. La notion du drapeau

Le drapeau (ou *flag* en Anglais) est une variable booléenne initialisée à Faux (drapeau baissé). Dès qu'un évènement attendu se produit, la variable change de valeur à Vrai (drapeau levé). Donc, la valeur finale du drapeau permet de savoir si un évènement a eu lieu ou pas. Cela devrait s'éclairer à l'aide d'un exemple extrêmement fréquent : la recherche de l'occurrence d'une valeur dans un tableau [10].

Exemple :

En supposant l'existence d'un tableau comportant N valeurs entières. On doit écrire un algorithme qui lit un nombre donné et informe l'utilisateur de la présence ou de l'absence de

ce nombre dans le tableau. La première étape consiste à écrire les instructions de lecture du nombre N et de parcours du tableau :

```
Tableau Tab[N] : Entier ;
Var val, i : Entier ;
Début
        Ecrire ("Entrer la valeur à rechercher :") ; Lire (val) ;
        Pour i de 0 à N-1
          …
        Finpour
Fin
```

Illustration *:*

On suppose que N=6 et les valeurs saisies sont celles figurant dans le schéma suivant :

Tab ⇨

indices :	i=0	i=1	i=2	i=3	i=4	i=5
valeurs :	**10**	**22**	**31**	**46**	**5**	**7**

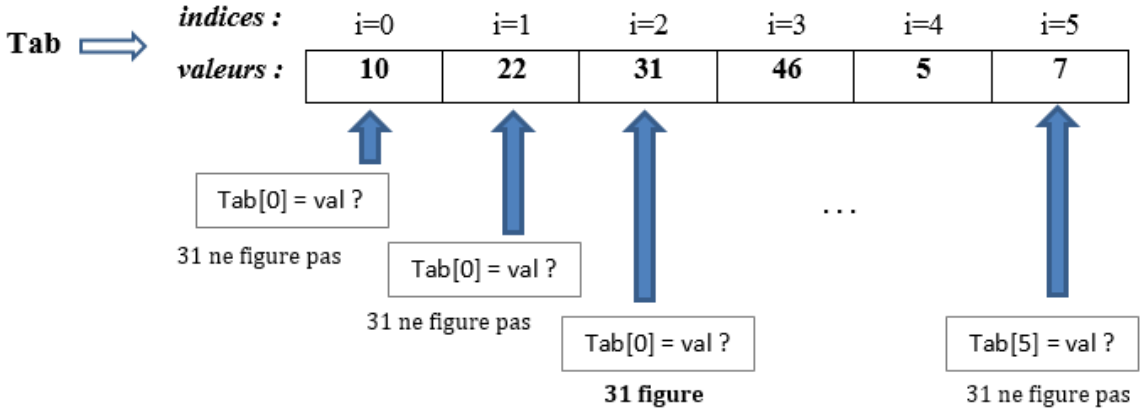
Revenons à l'algorithme, maintenant, il faut combler les points de la boucle (le bloc qui devra contenir les instructions de la recherche). Évidemment, il va falloir comparer « val » à chaque élément du tableau, s'il y a une égalité quelque part, alors « val » fait partie du tableau. Cela va se traduire, bien entendu, par l'instruction conditionnelle « Si … Alors … Sinon ».

```
…
Début
        Ecrire ("Entrez la valeur à rechercher ") ;   Lire (val) ;
        Pour i de 0 à N-1
                Si (val = Tab[i]) Alors
                        Ecrire (val, "figure") ;
                Sinon
                        Ecrire (val, "ne figure pas") ;
                Finsi
        Finpour
Fin
```

?

Illustration *:*

On suppose que la valeur à rechercher (val) est égale à 31 :



On peut constater qu'on a deux possibilités :

- ou bien la valeur « val » figure dans le tableau,
- ou bien elle n'y figure pas.

Mais dans tous les cas, l'algorithme ne doit produire qu'une seule réponse, quel que soit le nombre d'éléments du tableau. Or, l'algorithme ci-dessus affiche autant de messages qu'il y a de valeurs dans le tableau. Il y a donc une **erreur** quelque part.

En fait, on ne peut savoir si la valeur recherchée existe dans le tableau ou non que lorsque le parcours du tableau est entièrement accompli.

Pour pallier à cette erreur, on doit réécrire l'algorithme en plaçant le test après la boucle et en utilisant cette fois-ci une variable booléenne (drapeau) que l'on appelle « Existe ».

Cette variable doit être gérée comme suit :

- La valeur de départ de « Existe » doit être évidemment **Faux** (drapeau baissé).
- La valeur de la variable « Existe » doit devenir **Vrai** (drapeau levé), si un test dans la boucle est vérifié (lorsque la valeur de « val » est rencontrée dans le tableau). mais le test doit être asymétrique, c.à.d. qu'il ne comporte pas de "sinon".

Donc l'algorithme correct devrait être comme suit :

```
Tableau Tab[N] : entier ;
Var val, i : entier ;
      existe : booléen ;
Début
        Ecrire ("Entrez la valeur à rechercher ") ;
        Lire (val) ;
        existe ← faux ;
        Pour i de 0 à N-1
                Si (val = Tab[i])  Alors   existe ← vrai;   Finsi
        Finpour
```

Si (existe) **Alors** **Ecrire** (val, "fait partie du tableau") ;

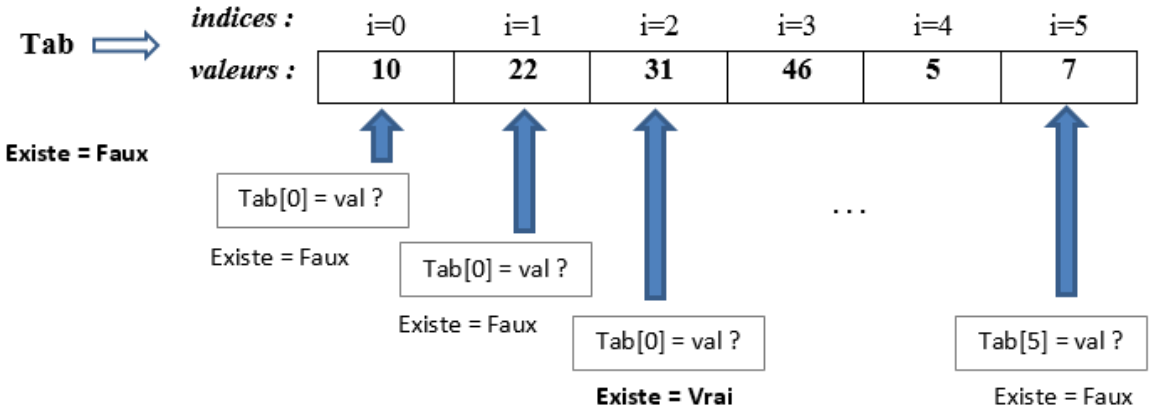
Sinon **Ecrire** (val, "ne fait pas partie du tableau") ;

Finsi

Fin

Illustration *:*

En utilisant un drapeau (la variable « Existe ») :



➢ *Question* ?

Réécrire le même algorithme mais cette fois-ci, la boucle de recherche doit être arrêtée dès que la valeur du drapeau change.

6. Le tri d'un tableau

Qu'est-ce qu'un tri ?

Le tri est une opération consistant à ordonner un ensemble d'éléments suivant une relation d'ordre prédéfinie. Le problème du tri est un grand classique en algorithmique. Trier un tableau numérique c'est donc ranger ses éléments en ordre croissant ou décroissant. Il existe plusieurs algorithmes de tri, parmi lesquels le tri par sélection, tri par insertion, tri à bulles, tri par fusion, etc. Nous illustrons dans ce qui suit deux types de tri, à savoir le tri par sélection et le tri par insertion.

6.1. Tri par sélection

Cette technique est parmi les plus simples, elle consiste à sélectionner, pour une place donnée, l'élément qui doit y être positionné. Par exemple pour trier un tableau en ordre croissant, on met en première position le plus petit élément du tableau et on passe à la position suivante pour mettre le plus petit élément parmi les éléments restants et ainsi de suite jusqu'au dernier [2].

Exemple :

Soit à trier, en ordre croissant, le tableau suivant :

25	10	13	31	22	4	**2**	18

Nous commençons par la recherche de la plus petite valeur et sa position. Une fois identifiée (dans ce cas, c'est le nombre 2 en 7ème position), nous l'échangeons avec le 1er élément (le nombre 25). Le tableau devient ainsi :

2	10	13	31	22	**4**	**25**	18

Nous recommençons la recherche, mais cette fois, à partir du 2ème élément (puisque le 1er est à sa position correcte). Le plus petit élément se trouve en 6ème position (le nombre 4). Nous échangeons donc le 2ème élément avec le 6ème élément :

2	**4**	13	31	22	**10**	25	18

Nous recommençons la recherche à partir du 3ème élément (puisque les deux premiers sont maintenant bien placés), Le plus petit élément se trouve aussi en 6ème position (10), en l'échangeant avec le 3ème, ça donnera:

2	4	**10**	31	22	**13**	25	18

Nous recommençons maintenant à partir du 4ème élément et de la même façon nous procédons jusqu'à l'avant dernier :

2	4	10	**13**	22	**31**	25	**18**

2	4	10	13	**18**	31	25	**22**

2	4	10	13	18	**22**	**25**	**31**

2	4	10	13	18	22	25	31

Algorithmiquement, nous pouvons décrire ce processus de la manière suivante :

- Boucle principale : prenant comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.
- Boucle secondaire : à partir de ce point de départ mouvant, nous recherchons jusqu'à la fin du tableau le plus petit élément. Une fois trouvé, nous l'échangeons avec le point de départ.

Donc, cela s'écrit :

```
/* boucle principale : le point de départ se décale à chaque tour */
Pour i de 0 à 6
    /* on considère provisoirement que T(i) est le plus petit élément */
    posmin ← i ; // posmin est la position du minimum initialisée par i
    /* on examine tous les éléments suivants */
    Pour j de (i + 1) à 7
        Si (T(j) < T(posmin)) Alors   posmin ← j ;   Finsi
    Finpour
    /* on sait maintenant où est le plus petit élément. Il ne reste plus qu'à effectuer la permutation */
    temp ← T(posmin) ;
    T(posmin) ← T(i) ;
    T(i) ← temp ;
    /* On a placé correctement l'élément numéro i, on passe à présent au suivant */
FinPour
```

6.2. Tri par insertion

Soit à trier un tableau d'éléments en ordre croissant.

Le principe de ce type de tri repose à chaque itération sur trois phases [2] :

a) On prend le premier élément dans la partie non encore triée du tableau (la clé).

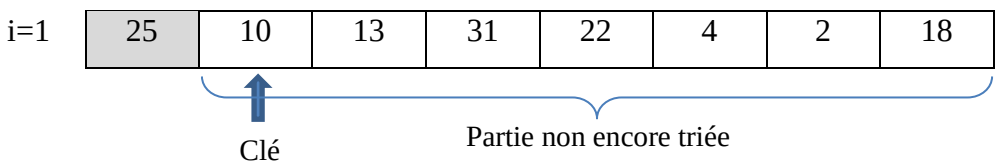
b) On cherche la place de la clé dans la partie déjà triée du tableau, en commençant par la droite de cette partie.

c) Une fois cette place trouvée, on y insère la clé après qu'on ait décalé vers la droite tous les éléments de la partie triée dont la valeur est plus grande ou égale à la valeur de la clé.

Il faut noter qu'initialement, la partie triée est constituée seulement du premier élément du tableau, autrement dit, le processus du tri commence à partir du deuxième élément.

Exemple :

Soit à trier, en ordre croissant, le même tableau précédent en appliquant le tri par insertion :



On décale le 1er élément de la partie triée vers la droite puisque sa valeur est supérieure à la clé. Cette dernière est déplacée à la 1ère position :

i=2

10	25	13	31	22	4	2	18

On recommence le processus avec une nouvelle clé. Le 1er élément à droite de la partie triée (25) est décalé vers la droite puisque sa valeur est supérieure à la clé. Le 2ème élément ne sera pas décalé puisqu'il est inférieur à la clé. Par conséquent, la clé est insérée dans la 2ème position du tableau :

i=3

10	13	25	31	22	4	2	18

On ne déplace pas cette clé (31) puisque sa valeur est supérieure à celles des éléments qui la précèdent.

i=4

10	13	25	31	22	4	2	18

On décale les deux premiers éléments (31 et 25) vers la droite et la clé est insérée à la 3ème position :

i=5

10	13	22	25	31	4	2	18

On décale tous les éléments de la partie triée vers la droite puisque leurs valeurs sont supérieures à celle de la clé. Cette dernière est déplacée à la 1ère position :

i=6

4	10	13	22	25	31	2	18

La même opération est répétée pour cette clé (2) :

i=7

2	4	10	13	22	25	31	18

Les trois éléments (22,25 et 31) sont décalés vers la droite et la clé est déplacée vers la 5ème position :

2	4	10	13	18	22	25	31

Nous obtenons donc un tableau qui est trié en ordre croissant.

L'algorithme correspondant à ce type de tri est présenté dans ce qui suit :

```
Algorithme Tri_Insertion ;
    Var i, j, n, clé : Entier;  // n est la taille du tableauT
    T : Tableau d'entiers ;
Début
    Pour i de 1 à n-1  // on commence par le 2ème élément (début de la partie non triée)
        clé ← T[i] ;
        j ← i - 1 ;  // indice du 1er élément à droite de la partie triée
        Tant que ((j >= 0) ET (clé < T[j])) Faire
            T[j +1] ← T[j];   // Décalage
            j ← j - 1;
        FinTant que
        T[j +1] ← clé;   // Insertion de la clé
    FinPour
Fin
```

6.3. Comparaison

Dans l'algorithme de tri par sélection, nous avons dans tous les cas, la boucle interne est exécuté pour i=1, 2, 3 jusqu'à i=(n-1) par conséquent, nous avons (n-1) + (n-2) + (n-3) + … + 1 étant *n (n-1) / 2* exécutions. Par exemple, pour un tableau de 100 éléments, la boucle est exécutée 4950 fois dans tous les cas.

Dans l'algorithme de tri par insertion, nous avons dans le pire des cas un tableau trié à l'envers (en ordre décroissant dans ce cas), et la boucle interne est exécuté (n-1) + (n-2) + (n-3) + … + 1 fois, étant *n (n-1) / 2* exécutions au maximum. Au meilleur des cas, le tableau est trié en ordre voulu (croissant dans ce cas) et la boucle interne ne s'exécutera jamais. En moyenne, le nombre d'exécutions est *n (n-1) / 4*. Par exemple, pour un tableau de 100 éléments, la boucle est exécutée 4950 fois au maximum et 2475 en moyenne.

7. Conclusion

Dans ce chapitre, nous avons vu comment mémoriser et manipuler un ensemble de valeurs représentées par le même nom et identifiées par des numéros à travers la notion du tableau. En fait, un tableau n'est pas un type de données mais plutôt une liste d'éléments d'un type donné. Le problème du tri qui est un grand classique de l'algorithmique, a été abordé par la suite à travers deux algorithmes parmi les plus simples, à savoir le tri par sélection et le tri par insertion. Une comparaison entre les deux algorithmes a été donnée à la fin de ce chapitre.

Chapitre 6 - Les enregistrements (structures)

1. Introduction

Nous avons vu dans le chapitre précédent, que les tableaux nous permettent de stocker plusieurs éléments de même type, tel que stocker les notes des étudiants dans un tableau de type réel. Supposons maintenant qu'en plus des notes des étudiants, nous voulons stocker aussi leurs informations (nom, prénom, matricule, …). Il nous faut dans ce cas de figure, une autre structure qui permet de stocker des données de différents types. Donc, une nouvelle structure appelée enregistrement est plus adaptée dans ce cas.

2. Définition

Un enregistrement (ou structure) permet de regrouper un ensemble de données de différents types sous le même nom (un seul objet). Il est défini par un ensemble d'éléments appelés champs. Ces derniers sont des données élémentaires ou composées qui peuvent être de types différents.

3. Déclaration et manipulation

La syntaxe de déclaration d'une structure est la suivante :

Structure nom_structure
champ1 **: type1 ;**
champ2 **: type2 ;**
...
champN **: typeN ;**
FinStructure ;

Tel que « nom_structure » est le nom d'enregistrement défini par l'utilisateur. « champ1, champ2, …, champN » sont les variables membres de la structure déclarée.

Exemple :

```
Structure Etudiant
   num : entier ;
   nom : chaîne ;
   moyenne : réel;
FinStructure ;
```

Par la suite, des variables de type Etudiant peuvent être déclarées comme suit :

Var x,y : **Structure** Etudiant ; */* Deux variables structures x et y de type Etudiant */*

La manipulation d'une variable structure se fait par champ (membre) et l'accès à une information contenue dans un champ se fait en précisant le nom de la variable structure suivie du champ concerné. Les deux séparés par un point.

Exemple :

```
x.num ←123 ;          // affecte le nombre 123 au champ num
Lire (x.nom) ;        // lire le champ nom
x.moyenne ←13.5 ;     // affecte le nombre 13.5 au champ moyenne
```

4. Tableau de structures

Il est possible de déclarer un tableau dont les éléments sont des structures avec la syntaxe suivante :

Tableau nom_tableau [taille] **: Structure** nom_structure **;**

Exemple : Tableau T[10] : **Structure** Etudiant ;

Donc, T est un tableau de 10 éléments de type structure (« Etudiant » dans ce cas).

L'accès à un champ d'élément i d'un tableau de structure se fait comme suit :

nom_tableau [i].champ

Par exemple, les instructions :

```
T[0].nom ← "Mohamed" ;
T[0].num ← 1 ; et
T[0].moyenne ← 12.5 ;
```

affectent respectivement la valeur entière « 1 », la chaîne de caractères « Mohamed » et la valeur réelle « 12.5 » aux champs « num », « nom » et « moyenne » du 1er élément du tableau T.

Illustration :

T ⇨

indices :	i=0	i=1	…	i=9
Eléments :	num : **1** nom : **Mohamed** moyenne : **12.5**	num : nom : moyenne :	…	num : nom : moyenne :

De même, les instructions :

```
Ecrire (T[0].num) ;
Ecrire (T[0].nom) ;
Ecrire (T[0]. moyenne) ;
```

Affichent les contenues des trois champs du 1er élément du tableau T.

5. Structure membre d'une autre structure

Une structure peut figurer parmi les champs d'une autre structure. Dans ce cas, elle doit être déclarée avant la structure qui la contient [6].

Exemple :

```
Structure Date
        jour, mois, annee : entier ;
Finstructure ;
Structure Compte
        Ncpt: entier;
        nom: chaine ;
        DtOuverture : Date ;
Finstructure ;
```

Où « DtOuverture » est une variable structure de type « Date », champ de la structure « Compte ». Donc, la structure « Date » est déclarée obligatoirement avant la structure « Compte ».

Lorsqu'un champ d'une structure est lui-même une structure, l'accès à ce champ se fait comme suit :

variable_structure.variable_sous_structure.champ

Par exemple : c.DtOuverture.annee ← 2019 ; affecte la valeur 2019 au champ année de l'enregistrement « DtOuverture » qui est lui-même un champ de l'enregistrement « c ». Tel que « c » est une variable de type « Compte ».

Dans le cas d'un tableau, l'accès se fait comme suit :

nom_tableau[indice].variable_sous_structure.champ

Par exemple : T[i].DtOuverture.annee ← 2019 ;

Où T[i] fait référence au $i^{ème}$ élément du tableau T de type « Compte ».

6. Conclusion

Nous avons vu dans le cinquième chapitre que les tableaux sont très pratiques, cependant ils ne permettent pas de répondre à tous les besoins de stockage tels que le regroupement de plusieurs données de types différents en un seul objet. A cet effet, la notion de structure (ou d'enregistrement) abordée dans le présent chapitre permet de pallier ce problème en créant un nouveau type permettant le stockage des données de types différents ou non. Un enregistrement qui est composé de plusieurs champs où chaque champ correspond à une donnée, constitue la brique de base pour les structures de données telles que les listes, les piles et les files qui ne font pas l'objet d'étude dans ce polycopié.

Chapitre 7 - Les fonctions et les procédures

1. Introduction

La fiabilité, la lisibilité et la réutilisabilité des programmes, reposent sur l'utilisation des sous-programmes. Ces derniers permettent :

- La réduction de la taille des programmes : il est possible de déterminer les blocs analogues, les substituer par un sous-programme, ensuite l'appeler dans des points déterminés au niveau du programme principal.
- L'organisation du code : le problème initial peut être découpé en sous-problèmes (modules) où chacun sera résolu par un sous-programme [1].

Exemple : Codage en base b [3]

Un entier positif en base b est représenté par une suite de chiffres $(c_n\, c_{n-1} \ldots c_1 c_0)_b$ où les c_i sont des chiffres de la base b ($0 \leq c_i < b$).

L'équivalent décimal de ce nombre est :

$$c_n b^n + c_{n-1} b^{n-1} + \ldots + c_1 b^1 + c_0 b^0 = \sum_{i=0}^{n} c_i\, b^i$$

On suppose que le nombre x est représenté par un tableau de chiffres (code) en base b ; par exemple si b = 2 et code = [1,0,1], alors en base 10 le nombre entier x correspondant vaudra $1\times2^2 + 0\times2^1 + 1\times2^0 = 4+0+1 = 5$. Etant donné code et b, l'algorithme qui permet de calculer x en base 10 est le suivant :

```
x ← 0 ;
Pour i de 0 à L-1  // L est la longueur du code
x ← x + code[i]*b^(L-1-i);
```

Supposons que l'on veut calculer successivement la valeur décimale x des nombres $(123)_5$ et $(123)_8$, on devra donc recopier deux fois l'algorithme ci-dessus.

```
b ← 5 ;
code ← [1,2,3] ;
x ← 0 ;
Pour i de 0 à L-1
        x ← x + code[i]*b^(l-1-i);
```

```
b ← 8 ;
code ← [1,2,3] ;
x ← 0 ;
Pour i de 0 à L-1
        x ← x + code[i]*b^(l-1-i);
```

Dans la pratique, il n'est pas souhaitable d'écrire deux fois le même programme, d'autant plus, si celui-ci nécessite de très nombreuses lignes de code source. Pour améliorer la réutilisabilité de l'algorithme la solution est d'encapsuler le code à répéter au sein d'un sous-programme.

2. La notion de sous-programme

Un sous-programme est une portion de code analogue à un programme, destiné à réaliser une certaine tâche à l'intérieur d'un autre programme. Il est identifié par un nom unique et un bloc d'instructions qui peut être exécuté plusieurs fois par des appels. Un appel est une instruction qui fait partie d'un autre programme ou sous-programme appelé le (sous-) programme appelant.

Pour résumer, un sous-programme est utilisé pour deux raisons essentielles :

- Lorsqu'une tâche est répétée plusieurs fois : on écrit un sous-programme pour cette tâche et l'on appelle à chaque endroit où l'on en a besoin c.à.d. on évite de réécrire le même code à plusieurs endroits dans le même programme.
- Pour réaliser la structuration d'un problème en sous-problèmes : on divise le problème en sous-problèmes pour mieux le contrôler (diviser pour régner).

Il existe deux types de sous-programmes : les **fonctions** et les **procédures**. Cependant, avant de détailler ces deux concepts, il sera utile de définir quelques notions utiles.

2.1. La portée d'une variable

Dans cette sous-section, nous illustrons une notion appelée **portée d'une variable**. L'idée principale est qu'il est possible d'avoir deux variables différentes avec le même nom toutefois qu'elles ont des portées différentes. Le cas le plus connus que l'on peut citer, est qu'une variable définie au niveau du programme principal (celui qui résout le problème initial) est appelée **variable globale** et sa portée est totale, par conséquent, tout sous-programme du programme principal peut utiliser cette variable. Alors qu'une variable définie au sein d'un sous-programme est appelée **variable locale** dont la portée est limitée au sous-programme dans lequel elle est déclarée [7].

Remarque :

Si l'identificateur (le nom) d'une variable locale est identique à celui d'une variable globale, cette dernière est localement masquée. Autrement dit, la variable globale devient inaccessible dans le sous-programme contenant la variable locale de même nom.

Exemple :

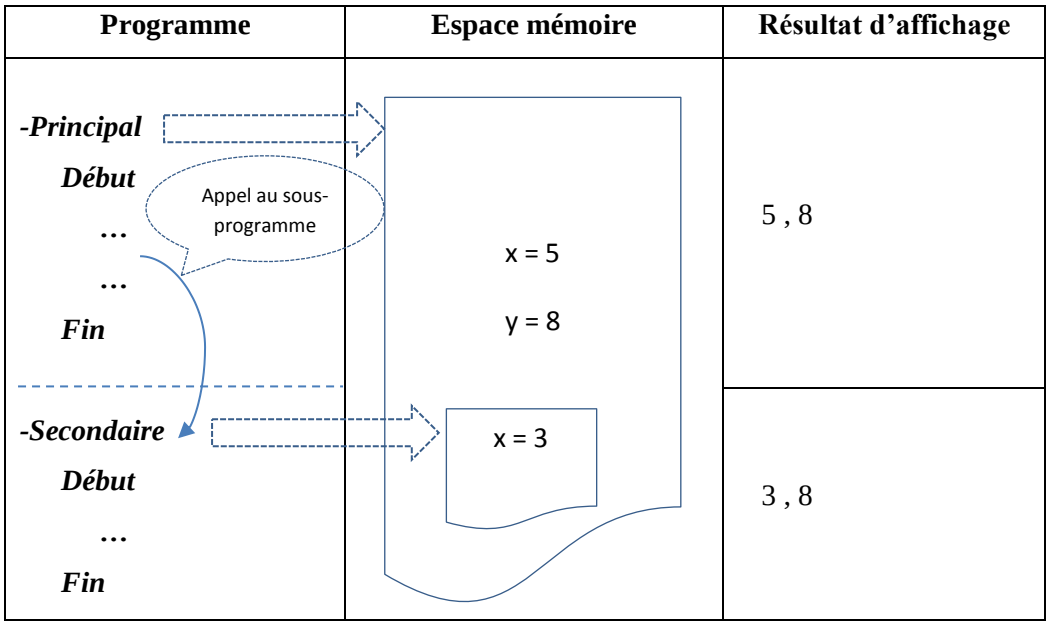
Supposons qu'une partie de notre programme sera le sous-programme suivant :

```
Algorithme Secondaire ;
Var x : entier ; // variable locale
Début
        x ← 3 ;
        Ecrire (x, y) ;
Fin
```

Supposons maintenant que nous appelons ce sous-programme « Secondaire » depuis une autre partie du programme « Principal » qui utilise également deux variables globales :

```
Algorithme Principal ;
Var  x, y : entier ; // variables globales
Début
        x ← 5 ; y ← 8 ;
        … ;  // Appel au sous-programme Secondaire
        Ecrire (x, y) ;
Fin
```

L'exécution de ce programme peut être illustrée comme suit :



Dans cet exemple, la variable « x », ayant la valeur 5 dans le programme principal, est une variable globale. Une autre variable qui porte le même nom « x » est utilisée au niveau du programme secondaire ayant comme valeur 3. Cet variable locale a masqué la variable globale « x » au niveau du programme secondaire, par conséquent, l'affichage de « x » au niveau de ce sous-programme correspond à la valeur 3. Par contre, la variable globale y est quant à elle accessible, ce qui justifie l'affichage de la valeur 8 au niveau du sous-programme.

2.2. Les paramètres

Les paramètres d'un sous-programme sont un ensemble de variables locales (**paramètres formels**) associées à un ensemble de variables ou constantes du (sous-) programme appelant (**paramètres effectifs**).

Remarque :

- Un paramètre est une variable locale, donc il admet un type.
- L'appel d'un sous-programme possédant un ou plusieurs paramètres, implique une association entre ces paramètres et les paramètres effectifs du programme (ou sous-programme) appelant, respectivement de même type.

Par exemple, si le sous-programme *Racine* permet de calculer la racine carrée d'un réel :

- Ce sous-programme admet un seul paramètre de type réel positif.
- Le programme appelant *Racine* doit fournir le réel positif dont il veut calculer la racine carrée, cela peut être une variable (*Racine* (y)) ou une constante (*Racine* (9)).

2.3. Le passage de paramètres

L'association entre les paramètres effectifs et les paramètres formels est appelé passage de paramètres. Il existe deux types de passage de paramètres :

- Le passage par valeur.
- Le passage par référence (ou adresse).

Dans le premier type, la valeur du paramètre effectif est affectée (copiée) au paramètre formel correspondant. Sachant que les paramètres formels ne sont que des variables locales de la fonction. Ce type de passage sera illustré dans la section qui suit.

Dans le second type, c'est l'adresse du paramètre effectif qui est affectée au paramètre formel correspondant. Ceci rend possible au sous-programme d'accéder directement à l'adresse de la variable concernée (le paramètre effectif) et par conséquent la modifier si nécessaire. Ce type de passage nécessite d'utiliser la notion des pointeurs [5] [9].

Remarque :

Une illustration du passage de paramètre par valeur est donnée dans l'exemple 3 de la section 4. Le passage de paramètre par adresse sera illustré dans le chapitre suivant.

3. Les fonctions

Une fonction est un sous-programme qui admet un nom, un type et des paramètres. Une fonction admet un type et retourne toujours un résultat.

3.1. Définition d'une fonction

On définit une fonction comme suit :

Fonction *nom_fonction* (*paramètres : types*) **:** *type de la valeur retournée*
Var *var1, var2,…: types; // variables_locales*
Début
instructions de la fonction ;
Retourner *(résultat) ; // (au moins une fois)*
Fin

Où :

- Les paramètres sont en nombre fixe (n≥0)
- Le type de la valeur retournée est le type de la fonction.
- La valeur de retour (ou le résultat) est spécifiée par l'instruction ***Retourner*.**

3.2. Appel d'une fonction

On fait appel à une fonction comme suit :

var ← nom_fonction (paramètres) ;

Où les paramètres d'appel peuvent être des variables, des constantes ou même des résultats d'une autre fonction.

Remarque :

A la définition ou à l'appel d'une fonction, les parenthèses sont toujours présentes même lorsqu'il n'y a pas de paramètres.

Exemple 1 :

Soit l'algorithme A qui calcule la valeur absolue d'un entier en utilisant une fonction :

```
Algorithme A ;
Var  a, b : entier ;
Fonction Abs (n : entier) : entier /* Définition de la fonction */
Var  valabs : entier
Début
        Si (n >= 0) alors valabs ← n ;
        Sinon valabs ← -n ;
        FinSi
    Retourner valabs ;
Fin

Début /* Algorithme principal */
        Ecrire (" Donner une valeur ") ;
        Lire (a) ;
        b ← Abs(a) ;
        Ecrire (b) ;
Fin
```

Passage de paramètre par valeur : la valeur de la variable **a** est copiée dans la variable **n.**

Lors de l'exécution de la fonction Abs(), il y a une association entre le paramètre effectif **a** et le paramètre formel **n** d'où la valeur de **a** est copiée dans **n.** Ce type d'association s'appelle **passage de paramètre par valeur.**

4. Les procédures

Une procédure est un sous-programme qui admet également un nom et des paramètres mais ne retournant aucun résultat.

4.1. Définition d'une procédure

On définit une procédure comme suit :

```
Procédure nom_procédure  (paramètres : types)
Var var1, var2,…: types; // variables_locales
Début
        instructions de la procédure ;
Fin
```

4.2. Appel d'une procédure

L'appel d'une procédure se fait comme suit :

nom_procédure (paramètres) ;

Exemple 2 :

Soit l'algorithme B calculant la valeur absolue d'un entier mais cette fois-ci en utilisant une procédure :

```
Algorithme B ;
Var a : entier ;
Procédure Abs (n : entier) /* Définition de la procédure */
        Début
           Si (n >= 0) alors
              Ecrire n ;
           Sinon
              Ecrire -n ;
           FinSi
        Fin
Début /* Algorithme principal */
        Ecrire (" Donner une valeur ") ;
        Lire (a) ;
        Abs(a) ;
Fin
```

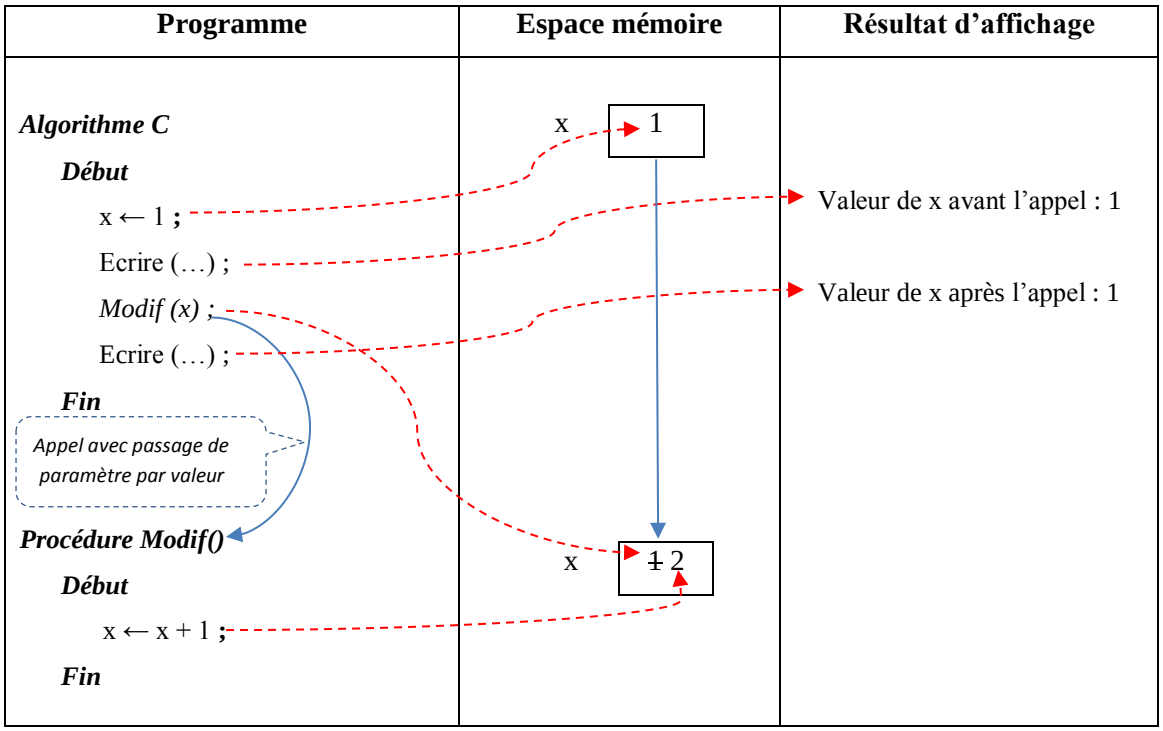
Exemple 3 :

Soit un algorithme utilisant une procédure comme suit :

```
Algorithme C ;
Var x : entier ;
Procédure Modif (x : entier)
Début
    x ← x + 1 ;  /* le x local est modifié, pas le x du programme principal */
Fin

Début /* Algorithme principal */
    x ← 1 ;
    Ecrire ("Valeur de x avant l'appel :", x) ;
    Modif (x) ;
    Ecrire ("Valeur de x après l'appel :", x) ;
Fin
```

L'exécution de l'algorithme C peut être illustrée comme suit :



Lorsqu'on passe un paramètre à une fonction (ou à une procédure), cette dernière ne peut pas modifier la variable. La variable est automatiquement recopiée et la fonction travaille sur une copie de la variable. La modification de la copie n'entraîne pas une modification de la variable originale. C'est ce qu'on appelle le passage de paramètre par valeur [3].

5. Fonctions et procédures récursives

La récursivité est une méthode de description d'algorithmes qui permet à une fonction (ou procédure) de s'appeler elle-même directement ou indirectement.

5.1. Exemple illustratif

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

Définition non récursive : N ! = N * N-1 * …* 2 * 1

Définition récursive : N ! = N * (N – 1) ! et 0 ! = 1

Par conséquent, deux solutions sont possibles pour le calcul de la factorielle.

a) Solution itérative :

```
Fonction FACT ( n : entier ) : entier
Var  i, F: entier ;
Début
        Si (n = 0) alors  F ← 1 ;
        Sinon
                F ← 1 ;
                Pour i de 2 à n
                        F ← F * i;
                Finpour
                Retourner F;
        Finsi
Fin
```

b) Solution récursive :

```
Fonction FACT ( n : entier ) : entier
        Début
                Si (n=0) alors Retourner 1 ;
                Sinon Retourner n * fact (n-1) ; // Appel récursif de la fonction FACT
                Finsi
        Fin
```

5.2. Interprétation

Une procédure ou une fonction récursive doit comporter une condition d'arrêt (n=0 dans l'exemple étudié ci-dessus). Cette condition empêche des appels récursifs sans arrêt. Généralement, 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. Dans le cas contraire, la fonction ou la procédure continue à exécuter les appels récursifs.

D'autre part, le paramètre de l'appel récursif doit converger toujours vers la condition d'arrêt. Un processus récursif remplace en quelque sorte une boucle, ainsi tout processus récursif peut être également formulé en tant qu'un processus itératif.

5.3. Mécanisme de fonctionnement

Considérons, à titre d'exemple, le calcul de la factorielle de 4 en appliquant la fonction récursive FACT(). Pour calculer FACT(4), il faut calculer FACT(3). Pour calculer FACT(3), il faut calculer FACT(2). Pour calculer FACT(2), il faut calculer FACT(1). Pour calculer FACT(1), il faut connaître FACT(0), ce dernier vaut 1. Ensuite, on revient à rebours pour terminer le calcul pour 1, 2, 3 puis 4.

Il y a donc autant d'occurrences de paramètre n et d'appel récursif que de niveaux de récursivité. A chaque niveau, un nouvel environnement, comprenant les paramètres et les variables locales de la fonction, est créé. Cette gestion des variables est invisible à l'utilisateur et effectuée automatiquement par le système si le langage admet la récursivité [5]. Une illustration de ce mécanisme est donnée dans la figure 6.

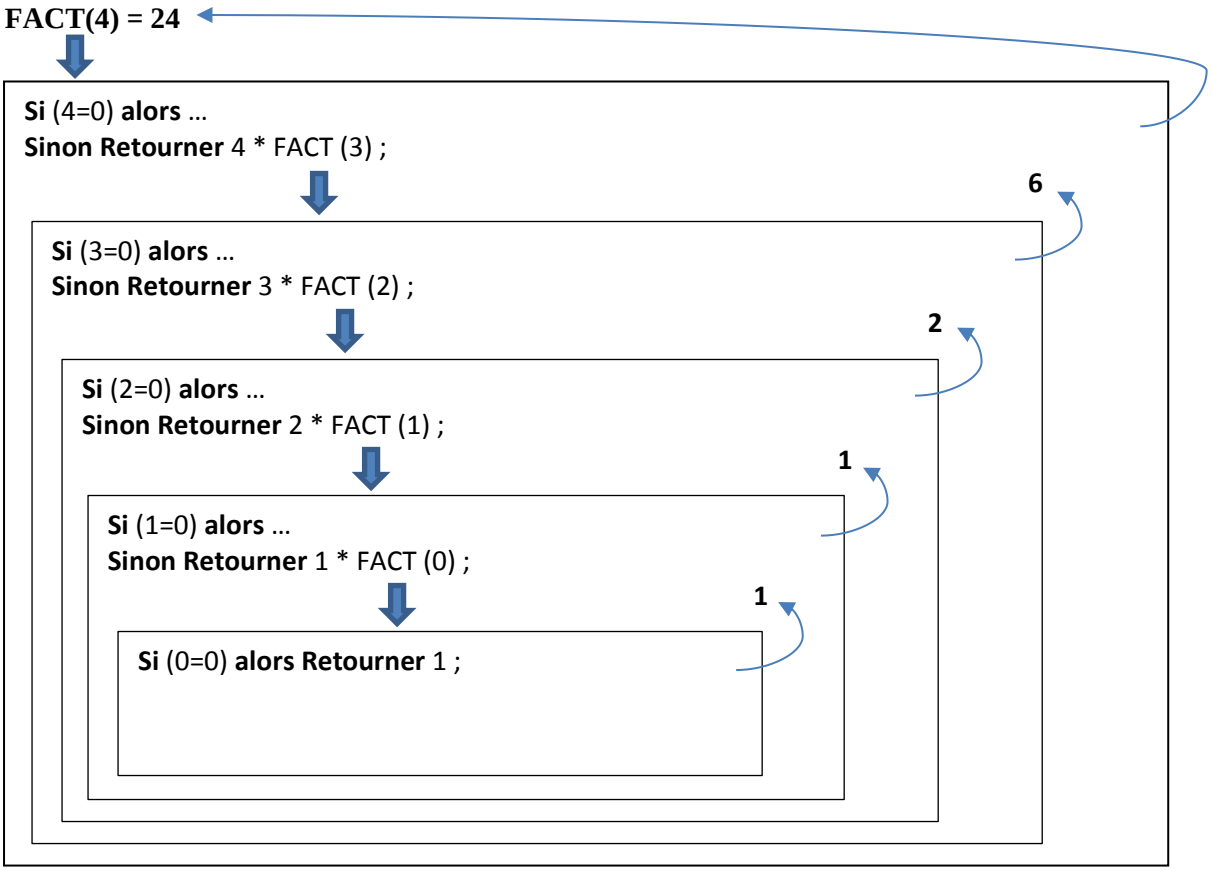


Figure 7. Calcul de la factorielle par récursivité.

6. Conclusion

La notion du sous-programme est très importante en programmation. Elle résulte de la décomposition du programme initial en de plus petites unités ou parties réutilisables qui sont ensuite appelées le moment opportun par le programme principal. Un sous-programme évite la répétition inutile de code et permet de clarifier le programme. En algorithmique, un sous-programme correspond à une fonction ou à une procédure dont la description et le mécanisme de fonctionnement sont présentés plus haut dans ce chapitre. Aussi, une initiation à la notion de récursivité, une des outils puissants de la programmation, a été donnée à la fin de ce chapitre.

Le prochain chapitre traite l'un des concepts utiles et indispensables au passage de paramètres par adresse (ou référence), c'est le concept de pointeur.

Chapitre 8 - Les pointeurs

1. Introduction

Lorsqu'une variable est déclarée et ce, quel que soit le langage de programmation, le compilateur réserve, à une adresse donnée en mémoire, l'espace nécessaire au contenu de cette variable. Donc, toute variable possède :

- Un identificateur (nom),
- Une valeur (donnée),
- Une adresse en mémoire.

Une donnée peut s'étaler sur plusieurs octets, donc occuper une plage d'adresses (par exemple 2 octets pour un entier, 4 ou 8 octets pour un réel, plus encore pour une chaîne).

Exemple :

Var n : **entier** ; *// déclaration d'un entier (sur 2 octets)*
n ← 5 ; *// affectation de la valeur 5 à n*

Dans cet exemple, le nom de la variable c'est n, la valeur c'est 5 et l'adresse c'est son emplacement dans la mémoire.

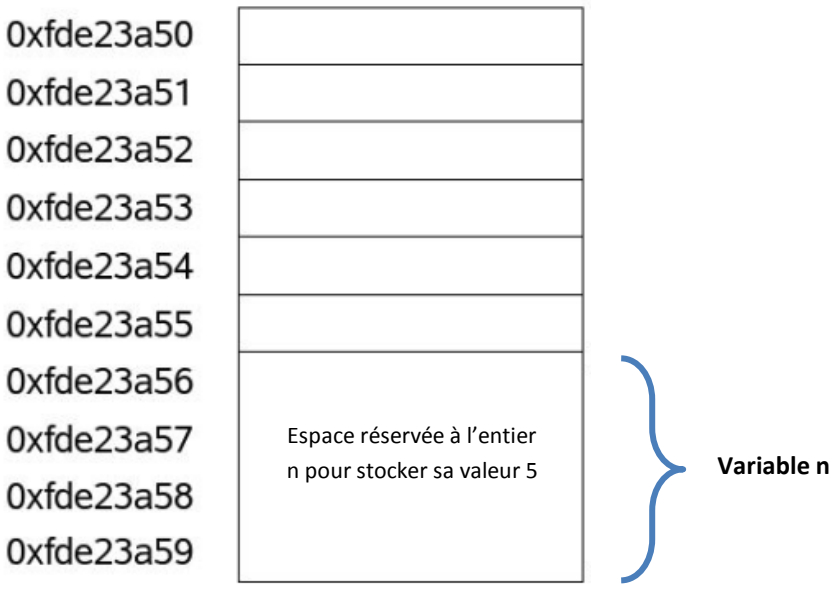


Figure 8. Représentation d'une variable en mémoire [10].

On peut donc accéder à une variable de deux façons :

- par son identificateur,
- par l'adresse mémoire à partir de laquelle elle est stockée (pointeur).

2. Notion de pointeur

2.1. Définition

Un pointeur est une variable qui contient l'adresse d'une autre variable.
Le pointeur pointe sur une autre variable dont il contient l'adresse mémoire, cette dernière étant dite variable pointée. Si l'on affiche le contenu d'un pointeur, on obtient une adresse qui est celle de la variable pointée, tandis que si l'on affiche le contenu de la variable pointée, on obtient la valeur associée à cette dernière.

Un pointeur est une variable. De ce fait, elle doit être déclarée, dispose elle-même de sa propre adresse en mémoire, et se voit définir un type. Le type d'un pointeur ne décrit pas ce qu'il contient (c'est une adresse, donc en principe d'une longueur de 32 ou 64 bits selon les architectures) mais le type de la variable qu'il pointe. Un pointeur sur une variable de type réel devrait donc être déclaré avec un type réel [10].

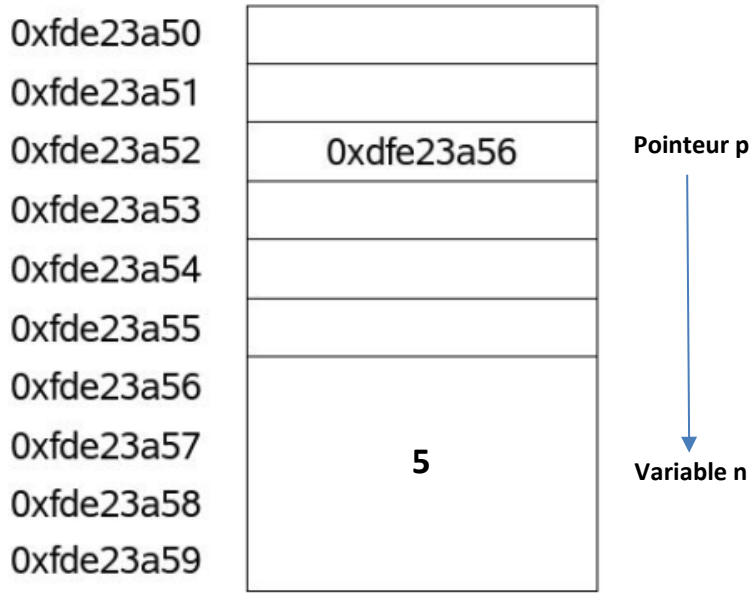


Figure 9. Le pointeur et la variable pointée en mémoire [10].

Dans ce qui suit, nous décrivons comment déclarer et manipuler un pointeur avec quelques cas d'applications par la suite.

2.2. Opérations sur les pointeurs

2.2.1. Déclaration

En algorithmique, un pointeur est déclaré comme suit :

nom_pointeur : **pointeur sur type ;** ou

nom_pointeur : ***type ;** *// où type est le type de l'élément pointé.*

Dans ce qui suit, nous utilisons la deuxième syntaxe qui est plus proche du langage C.

Par exemple :

p : ***entier ;** *// p est un pointeur vers un entier (il contient l'adresse d'un entier)*

2.2.2. Initialisation

Lorsqu'un pointeur ne pointe aucune variable, il faut l'initialiser avec la constante symbolique **NIL.** Un pointeur non initialisé pointe n'importe quoi dans la mémoire.

Par exemple :

p : ***entier ;**

p ← NIL ; *// p est un pointeur qui ne pointe rien*

2.2.3. Accès aux données

L'accès aux données se fait en utilisant les deux opérateurs suivants :

* : opérateur unaire qui permet de déréférencer un pointeur (accéder directement à la valeur de l'objet pointé).

& : opérateur unaire permettant d'obtenir l'adresse d'une variable.

Il convient de noter que ces opérateurs sont les mêmes utilisés en langage C, d'autres opérateurs peuvent être rencontrés par le lecteur tels que ^ et @ en Pascal à titre d'exemple.

Exemple :

Algorithme Exemple_pointeur ;
Var x : **entier ;**
p1, p2 : ***entier ;**
Début
x ← 3 **;**
p1 ← &x **;** *// p1 contient l'adresse de x*
p2 ← NIL ; *// p2 ne contient aucune adresse*
Ecrire ("Le contenu de la variable pointé par p1 est :", *p1) **;**
***p1 ← 5 ;** *// modification de x à travers p1*
Ecrire ("x=", x, "*p1=", *p1) **;**
p2 ← p1 ; *// affectation de l'adresse contenue dans p1 à p2*
Ecrire ("Le contenu de la variable pointé par p2 est :", *p2) **;**
Fin

L'exécution de cet algorithme donne :

Le contenu de la variable pointé par p1 est : 3
x=5 , *p1=5
Le contenu de la variable pointé par p2 est : 5

Remarque :

Avant toute utilisation, un pointeur doit être initialisé :

- par la valeur générique NIL, par exemple **p ← NIL** ;
- par l'affectation de l'adresse d'une autre variable, par exemple **p ← &v**;
- par **allocation dynamique** d'un nouvel espace-mémoire.

3. Allocation dynamique

Nous avons vu dans la section précédente qu'un pointeur reçoit l'adresse d'une variable qui existe déjà par affectation. Il est aussi possible de réserver un emplacement mémoire pour une donnée pointée directement. Dans ce cas, on peut créer un pointeur sur un entier par exemple, et réserver un espace mémoire (qui contiendra cet entier) sur lequel la variable pointeur pointera. C'est le principe de l'allocation dynamique de mémoire.

On peut employer la syntaxe suivante :

pointeur ← **nouveau type ;**

Le type doit bien entendu être celui de la valeur qui sera contenue à l'emplacement mémoire alloué. Après cette instruction, le pointeur reçoit l'adresse mémoire de la zone réservée. En cas d'échec (plus de mémoire disponible par exemple) il reçoit la valeur NIL.

Exemple :

Dans l'algorithme qui suit, un pointeur « p » sur un entier est déclaré. Pour placer une valeur entière dans la zone mémoire pointée, il faut d'abord réserver l'emplacement nécessaire. Puis on accède à l'élément pointé et on y place un entier.

```
Algorithme allouer ;
Var p : *entier
Début
    p ← nouveau Entier ; // allocation d'un espace mémoire dont l'adresse est affectée à p
    *p ← 12345 ;  // affectation d'un entier
    Ecrire ("Le contenu de p est :", *p) ;
Fin
```

Quand une zone mémoire est allouée dynamiquement, elle reste occupée tout le temps de l'existence du pointeur. Sans rien d'autre, la mémoire est récupérée uniquement à la sortie du programme. Il est aussi facile de libérer un espace alloué de la mémoire dès que le ou les pointeurs ne sont plus utiles. Pour ceci on applique la syntaxe suivante :

Libérer pointeur **;**

Exemple :

Dans l'algorithme qui suit, un pointeur « p » sur un entier est déclaré. Pour placer une valeur entière dans la zone mémoire pointée, il faut d'abord réserver l'emplacement nécessaire. Puis on accède à l'élément pointé et on y place un entier.

```
Algorithme libérer ;
Var p : *entier
Début
    p ← nouveau Entier ;
    *p ← 12345 ;
    Ecrire ("Le contenu de p est :", *p) ;
    Libérer p ; // libérer l'espace mémoire pointé par p
    p ← NIL ;  // réinitialiser p
Fin
```

Quand on libère un pointeur, on libère la zone mémoire sur laquelle il pointait, cette zone redevient disponible pour toute autre utilisation. Après chaque libération, il est préférable de réinitialiser le pointeur par la valeur NIL, et de penser à tester le pointeur avant de l'utiliser. Dans le cas où l'adresse de la zone mémoire libérée est conservée dans un autre pointeur, il faut faire attention au fait que ce pointeur pointe sur une zone éventuellement réaffectée à autre chose. Y accéder risque de fournir une valeur arbitraire, y écrire risque d'occasionner des problèmes, voire des plantages [10].

4. Application des pointeurs

Les applications des pointeurs sont nombreuses, à titre d'exemple en langage C :

- Une fonction ne peut pas retourner plus d'une valeur, un tableau ou un enregistrement. En utilisant un pointeur, ceci est possible en retournant l'adresse de l'ensemble ou de la structure des données à travers un pointeur.
- Les tableaux se manipulent très facilement via des pointeurs, puisqu'il est possible de faire des calculs sur les adresses : +1 va au contenu de l'adresse suivante, et ainsi de suite.
- Les pointeurs offrent la possibilité d'utiliser des structures de données plus complexes telles que listes chainées, les arbres, etc.
- Le passage de paramètres par adresse n'est possible aussi que par l'utilisation des pointeurs où un sous-programme peut modifier le contenu d'une case mémoire à l'adresse passée via un pointeur (voir l'exemple suivant).

Exemple :

Nous reprenons dans ce qui suit le même exemple vu dans le chapitre précédent concernant le passage de paramètres en utilisant le deuxième mode : passage par adresse.

```
Algorithme D ;
Var x : entier ;
Procédure Modif (px : *entier) // Procédure utilisant un pointeur comme paramètre local
Début
    *px ← *px + 1 ;   // le contenu pointé par px est modifié
Fin
Début /* Algorithme principal */
    x ← 1 ;
    Ecrire ("Valeur de x avant l'appel :", x) ;
    Modif (&x) ;
    Ecrire ("Valeur de x après l'appel :", x) ;
Fin
```

Passage de paramètre par adresse : l'adresse de la variable **x** est affectée au pointeur **px.**

L'exécution de l'algorithme D peut être illustrée comme suit :

Programme	Espace mémoire	Résultat d'affichage
Algorithme D ***Début*** x ← 1 ; Ecrire (…) ; *Modif (&x) ;* Ecrire (…) ; ***Fin*** *Appel avec passage de paramètre par adresse* ***Procédure Modif()*** ***Début*** *px ←*px + 1 ; ***Fin***	x → ~~1~~ 2 px → (&x)	Valeur de x avant l'appel : 1 Valeur de x après l'appel : 2

Lorsqu'on passe l'adresse d'une variable à une fonction (ou à une procédure), cette dernière peut modifier directement cette variable à travers le pointeur contenant son adresse. Donc l'accès au contenu pointé (*px) est équivalent à l'accès à la variable (x) dans ce cas. La modification se fait directement sur la variable originale. C'est le passage de paramètre par adresse.

5. Conclusion

Dans ce dernier chapitre de la partie « cours », nous avons présenté la notion des pointeurs avec des exemples sur leur utilisation. Le lecteur est initié aussi au mécanisme de gestion dynamique de mémoire à travers les opérations d'allocation et de libération. A la fin de ce chapitre, un survol sur les principales applications des pointeurs notamment en langage C a été présenté dans lequel le mode de passage de paramètres par adresse a été pris comme exemple d'application.

Partie II - Exercices corrigés

Série 1 : Initiation aux algorithmes

Exercice 1

Soit l'algorithme suivant :

```
Algorithme A ;
Var a, b, c : entier ;
Début
        a←5 ;
        b←a+1 ;
        c←a+b ;
Fin
```

a) Expliquer chaque ligne, mot et symbole dans cet algorithme.

b) Sur la base de cette explication, ajouter des commentaires à cet algorithme.

c) A quoi servent ces commentaires ? Est-ce qu'ils sont obligatoires ?

d) Dérouler l'algorithme et donner les valeurs des variables **a, b** et **c**.

Exercice 2

Soit l'algorithme suivant :

```
Algorithme B ;
Var a, b, c, d : entier ;
Début
        Ecrire (″Donner a et b″) ;
        Lire (a, b) ;
        c←a+b ;
        d←a*b ;
        Ecrire (c, d) ;
Fin
```

a) Que fait cet algorithme ?

b) Dérouler l'algorithme pour a=5 et b=6.

c) Même question pour a=6 et b=5.

Exercice 3

Soit l'algorithme suivant :

```
Algorithme C ;
Var x,y :entier ;
Début
        Ecrire (Donner x, y) ;
        Lire (x,y) ;
        x ← y+1 ;
        z ← z+x ;
        Ecrire ('x,z') ;
Fin
```

a) Corriger l'algorithme s'il y a des erreurs.

b) Dérouler l'algorithme pour x=9 et y=3.

Exercice 4

Soit l'algorithme suivant :

```
Algorithme D ;
Var a, b, c : entier ;
Début
        Ecrire (donner a et b) ;
        Lire ('a, b') ;
        c←a*a ;
        d←b*b ;
        Ecrire (c, d) ;
Fin
```

a) L'algorithme contient-il des erreurs ?

b) Dérouler l'algorithme pour a=5 et b=7 et déduire qu'est-ce qu'il fait.

Exercice 5

Corriger l'algorithme suivant, s'il le faut, et donner son résultat :

```
Algorithme A ;
Var x, y, z, s : entier ;
Début
        x ← 10 ;
        y ← 15 ;
```

```
        z ← 20 ;
        m ← (x + y + z) / 2 ;
        Ecrire (m) ;
        Ecrire (x + y + z / 2) ;
Fin
```

Exercice 6

a) Quel résultat produira-t-il le déroulement de l'algorithme suivant ?

```
Algorithme B ;
Var val, double, triple : entier ;
Début
        val ← 1000 ;
        double ← val * 2 ;
        triple ← val * 3 ;
        Ecrire val ;
        Ecrire double ;
        Ecrire triple ;
Fin
```

b) Proposer une simplification de cet algorithme en produisant le même résultat.

Exercice 7

Supposons que l'on veut afficher à l'écran de l'utilisateur le message « Bonjour à tous ».

a) Essayer d'écrire l'algorithme correspondant.

b) Quelle est la sortie (output) de cet algorithme ? et l'entrée (input) ?

c) Quelles sont les instructions à utiliser pour manipuler les entrées (et les sorties) d'un algorithme ?

d) Reformuler votre algorithme pour qu'il y ait une entrée et une sortie.

Série 2 : Instructions algorithmiques de base

Exercice 1

a. Ecrire un algorithme qui permet de lire un nombre (donné par l'utilisateur), puis il calcule et affiche son carré.

b. Même question pour calculer et afficher le cube, ensuite l'inverse de ce nombre.

Exercice 2

a. Ecrire un algorithme qui permet de lire les notes de trois matières ensuite il calcule et affiche leur moyenne.

b. Modifier l'algorithme dans le cas où les matières ont des coefficients qui doivent être donnés avec les notes.

Exercice 3

Ecrire un algorithme qui permet de lire deux variables numériques a et b et de les afficher avant et après leur permutation.

Par exemple, avant : a=5 et b=7, après : a=7 et b=5.

Exercice 4

Proposer un algorithme qui réalise la permutation de deux variables numériques sans avoir utiliser une troisième variable.

Série 3 : Les instructions conditionnelles

Exercice 1

Ecrire un algorithme qui permet d'afficher la valeur absolue d'un nombre donné.

Exercice 2

Ecrire un algorithme qui permet de déterminer si un entier donné est pair ou impair.

Exercice 3

Ecrire un algorithme qui demande trois lettres à l'utilisateur, et l'informe ensuite si leur ordre de lecture et le même que l'ordre alphabétique.

Exercice 4

a) Écrire un algorithme qui lit trois variables au clavier et affiche le maximum des trois.

b) Même question pour plus de trois variables.

Exercice 5

a) Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif mais sans le calculer. (On laisse de côté le cas où le produit est nul).

b) Même question en incluant cette fois-ci le cas où le produit peut être nul.

Exercice 6

Ecrire un algorithme qui permet de lire un numéro du jour de la semaine (numéro entre 1 et 7) et d'afficher le nom du jour correspondant. Par exemple, le dimanche correspond au numéro 1.

Série 4 : Les instructions itératives

Exercice 1

Donner les affichages produits par l'exécution des algorithmes suivants :

Algorithme 1 :

```
Var i : entier ;
Début
  Pour i←2 à 8
     Écrire ("Bonjour") ;
     Écrire (i) ;
  Fin pour
  Écrire ("fin") ;
Fin
```

Algorithme 2 :

```
Var encore : booléen ;
Début
    encore ← Vrai ;
    Répéter
       Écrire ("Bonjour") ;
    Jusqu'à (encore)
    Écrire ("fin") ;
Fin
```

Algorithme 3 :

```
Var encore : booléen ;
Début
    encore ← Faux ;
    Tant que (encore)  faire
       Écrire ("Salut") ;
    Fin tant que
    Écrire ("fin") ;
Fin
```

Exercice 2

Ecrire l'algorithme qui affiche la somme des prix d'une suite d'articles saisie par l'utilisateur et se terminant par zéro (Justifier le choix de la boucle à utiliser).

Exercice 3

a) Ecrire l'algorithme qui demande un entier, ensuite il affiche les dix entiers suivants. Par exemple, si l'on entre le nombre 10, l'algorithme affichera les nombres 11, 12,…, 20, 21.

b) Modifier l'algorithme pour qu'il affiche les dix nombres pairs suivants.

Exercice 4

a) Ecrire l'algorithme qui demande un nombre entier n, ensuite il affiche la somme des entiers positifs jusqu'à n.
Par exemple, si n=5, l'algorithme affiche : 1 + 2 + 3 + 4 + 5 = 15

b) Réécrire le même algorithme mais cette fois ci pour le calcul d'un produit.
Par exemple, pour n=5, l'algorithme affiche : 1 x 2 x 3 x 4 x 5 = 120

Remarque. On souhaite afficher uniquement le résultat sans la décomposition du calcul.

Exercice 5

Ecrire un algorithme qui permet de calculer la factorielle d'un entier (qui doit être positif).
Où n! = n x (n-1) x (n-2) x … x 1, si $n \geq 1$ et n! = 1 si n = 0.

Exercice 6

a) Ecrire un algorithme qui demande successivement dix nombres à l'utilisateur, ensuite il affiche le plus petit parmi eux (le minimum).

Exemple :

Entrer le nombre numéro 1 : 12
Entrer le nombre numéro 2 : 3
... (on suppose que les autres nombres sont ≥ 3)
Entrer le nombre numéro 10 : 6

Résultat :
Le minimum est : 3

b) Modifier l'algorithme pour qu'il affiche, en plus, la position de ce nombre.
Pour l'exemple précédent, le minimum se trouve à la position : 2

c) Réécrire l'algorithme précédent dans le cas où l'on ne connaît pas à l'avance combien de nombres l'utilisateur souhaite saisir, mais la saisie des nombres s'arrête lorsque l'utilisateur entre un zéro.

Exercice 7

Ecrire un algorithme qui demande un nombre puis vérifier si ce nombre est premier ou non

Exercice 8

a) Supposons que le code pin d'un utilisateur est 5454. Ecrire un algorithme qui demande à cet utilisateur de saisir son code pin jusqu'à ce que la réponse convienne.

b) Ajouter une condition qui annule la saisie après trois tentatives erronées.

Série 5 : Les tableaux et les structures

Exercice 1

a) Ecrire un algorithme qui permet de lire 10 valeurs données par l'utilisateur en les stockant dans un tableau, ensuite l'algorithme doit afficher seulement les valeurs impaires.

b) Modifier l'algorithme (a) pour que le nombre de valeurs soit donné par l'utilisateur.

c) Réécrire l'algorithme (b) en divisant cette fois-ci le tableau initial en deux tableaux, l'un contenant les valeurs paires et l'autre contenant les valeurs impaires, et en les affichant par la suite.

Exercice 2

a) Soit un tableau de 10 éléments réels, écrire un algorithme qui permet de lire ce tableau et rechercher le maximum des éléments ainsi que sa position, ensuite l'afficher.

b) Même question pour un tableau de 10 lignes et 5 colonnes.

Exercice 3

Ecrire un algorithme qui permet de rechercher une valeur numérique saisie par l'utilisateur dans une matrice de taille N*M (N et M sont données).

Exercice 4

Ecrire l'algorithme qui permet de compter le nombre d'occurrences d'un élément donné dans une matrice de taille N*M.

Exercice 5

Définir une structure Rationnel permettant de coder un nombre rationnel, avec numérateur et dénominateur. Ecrire ensuite l'algorithme qui permet la saisie, l'affichage, la multiplication et l'addition de deux rationnels. Pour l'addition, afin de simplifier, on ne cherchera pas nécessairement le plus petit dénominateur commun.

Exercice 6

Soit une structure Compte qui code un compte bancaire défini par un numéro, nom et prénom et date d'ouverture. Donner la définition des structures nécessaires.
Ecrire ensuite l'algorithme qui permet de saisir un tableau de N comptes et de l'afficher.

Série 6 : Les fonctions et les procédures

Exercice 1

Supposons qu'on veut écrire un sous-programme MoySom qui prend en paramètres trois entiers a, b et c, et qui affiche leur somme et renvoie leur moyenne.

Quelle est la déclaration correspondante ?

- **Fonction** MoySom (a : **entier**, b: **entier**, c: **entier**) : **entier** ;
- **Fonction** MoySom (a : **entier**, b: **entier**, c: **entier**) : **réel** ;
- **Procédure** MoySom (a : **entier**, b: **entier**, c: **entier**, som: **entier**, moy: **réel**) ;
- **Fonction** MoySom (a : **entier**, b: **entier**, c: **entier**) : **entier, réel** ;

Justifier la réponse choisie**.**

Exercice 2

a) Écrire deux fonctions Min et Max qui retournent le minimum et le maximum de deux nombres réels donnés comme paramètres.

b) Écrire deux autres fonctions Min_4 et Max_4 utilisant les fonctions Min et Max pour retourner le minimum et le maximum de quatre nombres réels passés en paramètres.

c) Incorporer ces fonctions dans un algorithme complet.

Remarque : On suppose que tous les nombres donnés sont différents.

Exercice 3

Ecrire un algorithme qui permet de lire une liste de N étudiants ayant comme informations : numéro, nom, prénoms et moyenne du bac. L'algorithme doit afficher cette liste triée par moyenne en ordre décroissant.

Remarque : utiliser une fonction ou une procédure pour le tri.

Exercice 4

Un magasin de vente des composants électroniques vend quatre types de produits :

- Des cartes mères (code 1) ;
- Des processeurs (code 2) ;
- Des barrettes de mémoire (code 3) ;
- Des cartes graphiques (code 4).

Chaque produit possède une référence (qui est un nombre entier), un prix en DA et une quantité disponible.

a) Définir une structure Produit qui code un produit.

b) Ecrire une fonction qui permet la saisie et l'affichage des données d'un produit.

c) Ecrire une fonction qui permet à un utilisateur de saisir une commande d'un produit. L'utilisateur saisit la quantité commandée et les données du produit. L'ordinateur affiche toutes les données de la commande, y compris le prix.

Exercice de réflexion (sans correction)

Exercice A :

On se propose de réaliser une calculatrice qui permet de faire les opérations arithmétiques de base (+, -, * et /) de deux nombres a et b, ainsi que de savoir :

- si la somme *a + b* est paire ;
- si le produit *a*b* est pair ;
- le signe de la somme *a + b* ;
- le signe du produit *a*b*.

Ecrire l'algorithme correspondant.

Exercice B :

On se propose de faire la rotation (décalage des positions) des éléments d'un tableau.

1. Proposer un algorithme pour ce problème sachant que le degré de rotation doit être défini par l'utilisateur. Exemple pour un degré de rotation égal à 2 :

   Avant rotation : 1,3,9,4,5,2 ➔ Après rotation : 5,2,1,3,9,4

2. Modifier l'algorithme pour que la rotation puisse être faite dans les deux sens (avant-arrière).

Exercice C :

Une société désire organiser un concours de recrutement en deux spécialités S1 et S2.

Le concours se porte sur trois (03) matières : M1, M2 et M3.

Un candidat est retenu si les conditions suivantes sont toutes remplies :

- Il doit se classer parmi les dix (10) premiers,
- Il ne doit pas avoir eu un zéro dans l'une des trois matières,
- Sa spécialité doit correspondre à l'une des deux spécialités concernées.

La note de classement est la moyenne des notes des trois matières.

Ecrire l'algorithme qui permet de lire les informations nécessaires, de classer tous les candidats par ordre de mérite, ensuite afficher la liste des candidats retenus pour le recrutement.

Exercice D :

Une entreprise veut gérer la liste de ses employés en les stockant dans un tableau.
Sachant que chaque employé possède comme information : un matricule, un nom et prénom, une date de naissance, une situation familiale et une adresse. Cette dernière est définie par un numéro, rue, ville et code postale.

1. Donner la fonction ou la procédure qui permet de :

- rechercher un employé par son matricule et afficher ses informations.
- afficher les nom et prénoms des employés qui habitent une même ville donnée.

2. Ecrire l'algorithme qui permet de créer la liste des employés et de faire appel à ces fonctions (ou procédures).

Corrigé série 1

Solution 1 :

a) Algorithme A ;

```
Var a, b, c : entier ; // déclaration des variables
Début
    a← 5 ;    // affectation de la valeur 5 à la variable a
    b← a+1 ; // affectation du résultat de a+1 à la variable b
    c← a+b ; // affectation de la somme de a et b à la variable c
Fin
```

b) Les commentaires dans un algorithme (ou programme) sont ajoutés à titre indicatif pour le rendre plus lisible et compréhensible mais ils ne sont pas obligatoires.

c) Le résultat du déroulement de cet algorithme est comme suit :

a=5
b=5+1=6
c=6+5=11

Solution 2 :

a) L'algorithme calcule la somme et le produit de deux entiers

b) C=5+6=11

c) D=5×6=30

Solution 3 :

a) Algorithme C ;

```
Var x,y :entier ;
Const z =10 ;
Début
    Ecrire ("donner x, y") ;
    Lire (x,y) ;
    x ←  y+1 ;
    z ← z+x ;  (z est une constante)
    Ecrire ('x,z') ;
Fin
```

b) Déroulement de l'algorithme pour x=9 et y=3

x=3+1=4

Affichage : x=4, z=10

Solution 4 :

a)

```
Algorithme D ;
Var a, b, c : entier ;
Début
        Ecrire ("donner a et b") ;
        Lire (a, b) ;
        c← a ;
        a← b ;
        b← c ;
        Ecrire (a, b) ;
Fin
```

b) Déroulement de l'algorithme pour a=5 et b=7

c=5
a=7
b=5
Affichage : a=7 et b=5

Donc, l'algorithme réalise la permutation de a et b.

Solution 5 :

s : variable déclarée sans être utilisée

m : doit être de type réel

Affichage : 22.5 ; 35

Solution 6 :

a) val=1000 ; double=2000 ; triple=3000

b) Simplification **:**

```
Var val : entier ;
Début
  val ← 1000 ;
  Ecrire (val, val*2, val*3) ;
Fin
```

Solution 7 :

a) Algorithme Un_Bonjour ;

Début

Ecrire ("Bonjour à tous") ;

Fin

Cet algorithme n'a qu'une sortie, le message « Bonjour à tous ».

b) Pour manipuler les entrées et les sorties d'un algorithme, on utilise respectivement les deux instructions : Lire() et Ecrire().

c) Algorithme Un_ReBonjour ;

Var mge : chaine ;

Début

Ecrire ("Veuillez saisir un message :") ;

Lire (mge) ;

Ecrire (mge) ;

Fin

Corrigé série 2

Solution 1 :

a) Algorithme carré ;

```
Var  nb, carr : réel ;
Début
  Lire (nb) ;
  carr ← nb * nb ; // ou carr ← nb^2
  Ecrire (carr) ;
Fin
```

b) Algorithme cube ;

```
Var  nb, cub : réel ;
Début
  Lire (nb) ;
  cub ← cub ^ 3
  Ecrire (cub) ;
Fin
```

Algorithme inverse ;

```
Var  nb, inv : réel ;
Début
  Lire (nb) ;
  inv← 1/ nb ; // tel que nb ≠ 0
  Ecrire (inv) ;
Fin
```

Solution 2 :

a. Algorithme moyenneA ;

```
Var n1, n2, n3, moy : réel ;
Début
  Ecrire ("Donner trois notes :") ; // ce message est optionnel
  Lire (n1, n2, n3) ;
  moy ← (n1+n2+n3) / 3 ;
  Ecrire (moy) ;
Fin
```

b. Algorithme moyenneB ;

```
Var n1, n2, n3, moy : réel ; c1, c2, c3 : entier ;
Début
  Lire (n1, n2,n 3) ;
  Lire (c1, c2, c3) ;
  moy ← (n1*c1+n2*c2+n3*c3) / (c1+c2+c3)  ;
  Ecrire (moy) ;
Fin
```

Solution 3 :

```
Algorithme permutation ;
Var a,b,c : réel ;
Début
   Ecrire (″Entrer la valeur de a :″) ;
   Lire (a) ;
   Ecrire (″Entrer la valeur de b :″) ;
   Lire (b) ;
   Ecrire (″a=″,a) ;
   Ecrire (″b=″,b) ;
   c ← a ;
   a ← b ;
   b ← c ;
   Ecrire (″a=″,a) ;
   Ecrire (″b=″,b) ;
Fin
```

La permutation est réalisée à travers ces trois affectations. La variable « c » est une variable temporaire qui doit être de même type que a et b.

Solution 4 :

```
Algorithme permutation2 ;
Var  x, y : réel ;
Début
  Lire (x,y) ;
  x ← x + y ;
  y ← x - y ;
  x ← x - y  ;
  Ecrire (x,y) ;
Fin
```

Corrigé série 3

Solution 1 : (L'écriture des entêtes d'algorithmes est omise dans quelques solutions)

```
Var n : réel ;
Début
  Ecrire ("Entrez un nombre : ") ;
  Lire (n) ;
  Si (n < 0) Alors   n← -n ;
  FinSi
  Ecrire ("la valeur absolue est", n) ;
Fin
```

Solution 2 :

```
Var  n : entier ;
Début
  Ecrire ("Entrez un nombre : ") ;
  Lire (n) ;
  Si (n mod 2 = 0) Alors   Ecrire ("Ce nombre est pair") ;
  Sinon  Ecrire ("Ce nombre est impair") ;
  FinSi
Fin
```

Solution 3 :

```
Var a, b, c : caractère ;
Début
  Ecrire ("Entrez successivement trois lettres : ") ;
  Lire (a, b, c) ;
  Si (a < b ET b < c) Alors   Ecrire ("Les lettres sont classées alphabétiquement") ;
  Sinon   Ecrire ("Les lettres ne sont pas classées") ;
  FinSi
Fin
```

Solution 4 :

a)

```
Var a, b, c, max ;
Début
  Ecrire ("Entrez trois nombres réels : ") ;
  Lire (a, b, c) ;
```

```
Si (a < b) Alors max ← b;
Sinon max ← a;
FinSi
Si (max < c) Alors max ← c; FinSi
Ecrire ("Le maximum des trois nombres est :", max);
Fin
```

Autre variante :

```
...
Lire (a, b, c) ;
max ← a ;
Si (max<b) Alors  max←b ;   FinSi
Si (max<c) Alors  max←c ;   FinSi
Ecrire ("Le maximum des trois nombres est : ", max) ;
Fin
```

b) Même principe pour 4 et 5 variables (juste pour illustrer l'utilité de la variable max).

Solution 5 :

a)

```
Var  m, n : entier ;
Début
 Ecrire ("Entrez deux nombres : ") ;
 Lire (m, n) ;
 Si ((m > 0 ET n > 0) OU (m < 0 ET n < 0)) Alors   Ecrire ("Le produit est positif") ;
 Sinon   Ecrire ("Le produit est négatif") ;
 FinSi
Fin
```

b)

```
Var  m, n : entier ;
Début
 Ecrire ("Entrez deux nombres : ") ;   Lire (m, n) ;
 Si (m = 0 OU n = 0) Alors    Ecrire ("Le produit est nul") ;
 Sinon  Si ((m < 0 ET n < 0) OU (m > 0 ET n > 0)) Alors
                    Ecrire ("Le produit est positif") ;
          Sinon Ecrire ("Le produit est négatif") ;
         FinSi
 FinSi
Fin
```

Solution 6 :

```
Var  j : entier ;
Début
  Ecrire ("Donner le numéro du jour :") ;   Lire (j) ;
  Selon j
        1 : Ecrire ("Dimanche") ;
        2 : Ecrire ("Lundi") ;
        3 : Ecrire ("Mardi") ;
        4 : Ecrire ("Mercredi") ;
        5 : Ecrire ("Jeudi") ;
        6 : Ecrire ("Vendredi") ;
        7 : Ecrire ("Samedi") ;
        Défaut : Ecrire ("Donner un numéro de jour valide (entre 1 et 7).") ;
  FinSelon
Fin
```

Corrigé série 4

Solution 1 :

Algorithme 1 :

Bonjour
2
Bonjour
3
Bonjour
4
Bonjour
5
Bonjour
6
Bonjour
7
Bonjour
8
Fin

Algorithme 2 :

Bonjour

Fin

Algorithme 3 :

Fin

Solution 2 :

```
Var p, s : réel ;
Début
    s ← 0 ;
    Répéter
        Ecrire ("Entrer le prix de l'article (0 si fin):");
        Lire (p) ;
        s ← s + p ;
    Jusqu'à (p = 0)
    Ecrire (" La somme des prix des articles est ", s) ;
Fin
```

Puisque le nombre d'articles n'est pas connu à l'avance ainsi qu'il faut faire au moins une saisie pour terminer, la boucle « Répéter » est appliquée dans ce cas.

Solution 3 :

a)
```
Var N, i : Entier ;
Début
  Ecrire ("Entrer un entier : ") ;  Lire (N) ;
  Ecrire ("Les 10 nombres suivants sont : ")
  Pour i de (N + 1) à (N + 10)
    Ecrire (i) ;      /* Si (i mod 2 = 0)  Ecrire i ;  */
  FinPour
Fin
```

b) Pour le deuxième cas, il suffit de remplacer l'instruction : « Ecrire (i) » par « Si (i mod 2 = 0) Alors Ecrire i » dans la 6ème ligne.

Solution 4 : (les instructions correspondantes à la question b sont mises en commentaires)

```
Var N, i, som : Entier ;
Début
  Ecrire ("Donner un entier : ") ;
  Lire (N) ;  som ← 0 ; // prod← 1 ;
  Pour i de 1 à N
    som ← som + i ; // prod← prod * i ;
  FinPour
  Ecrire ("La somme = ", som ); // Ecrire ("Le produit = ", prod );
Fin
```

Solution 5 :

```
Var N, i, F : entier ;
Début
  Ecrire ("Entrer un entier positif : ") ;  Lire (N) ;
  Si (N<0) Alors   Ecrire ("Erreur !") ;
  Sinon
     F ← 1 ;
     Si (N>1) Alors
```

Autre possibilité :

```
Sinon
    F ← N ;
    Tant que (N>1) faire
      N ← N-1 ;
      F ← F * N ;
    FinTq
 Fsi
```

```
        Pour i de 2 à N
          F ← F * i ;
        FinPour
      Fsi
      Ecrire ("La factorielle est ", F) ;
  Fsi
Fin
```

Solution 6 :

a) et b)

```
Var N, i, min, pmin : Entier ;
Début
  min ← 0 ; // juste pour initialiser la var
  Pour i de 1 à 10
    Ecrire ("Entrer le nombre numéro ", i)  ;    Lire (N) ;
    Si (i = 1 ou N < min) Alors    min ← N ;      pmin ← i ;
    FinSi
  FinPour
  Ecrire ("Le minimum est : ", min );
  Ecrire ("Il a été saisi en position :", pmin) ;
Fin
```

c/

```
Var N, i, min, pmin : Entier ;
Début
   i ← 1 ;   min ← 0 ;
  Répéter
    Ecrire ("Entrer le nombre numéro ", i)  ;     Lire (N) ;
    Si (i = 1 ou N < min) Alors
      min ← N ;  pmin ← i ;
    FinSi
    i ← i + 1 ;
  Jusqu'à (N=0)
  Ecrire ("Le minimum est ", min),
  Ecrire ("Il a été saisi en position numéro ", pmin),
Fin
```

Solution 7 :

```
Algorithme nombre_premier
Var i, N : entier ;
    x : booleen;
Début
        Ecrire ("entrer N");  Lire(N);
        x ← faux;
        i ← 2;
        Tant que (i<N et x = faux) Faire
                Si (N mod i = 0) alors
                        Ecrire ("le nombre n'est pas premier") ;
                        x← vrai ;
                        i ← i+1;
                Finsi
        Fin Tq
        Si (x=faux) alors Ecrire ("le nombre est premier") ;
Fin
```

Solution 8 :

a) et b)

```
        Algorithme codePin ;
        Var codePin, monCode, tentative: Entier ;
        Début
                codePin ← 5454 ;
                tentative  ← 3 ;
                Répéter
                        Ecrire ("Entrez le code PIN :") ;  Lire (monCode) ;
                        tentative ← tentative – 1 ;
                        Si (monCode ≠ codePin) Alors Ecrire ("Code incorrect") ; FinSi
                Jusqu'à (monCode = codePin ou tentative=0)
                Si (tentative=0) Alors
                        Ecrire ("Vous ne pouvez plus saisir de code") ;
                Sinon
                        Ecrire ("Bienvenue") ;
                FinSi
        Fin
```

Corrigé série 5

Solution 1 :

Algorithme a ;

```
Var i : entier ;
Tableau T[10] : entier ;
Début
        Pour i de 0 à 9
                Ecrire ("Donner un entier : ") ; Lire T[i] ;
        FinPour
        Pour i de 0 à 9
                Si (T[i] mod 2 <> 0) Alors Ecrire T[i] ;
        FinPour
        Fin
```

Algorithme b ;

```
Var  i, n: entier ;
Tableau T[n] : entier ;
Début
        Ecrire ("Donner le nombre de valeurs à saisir ") ; Lire (n) ;
        Pour i de 0 à n-1
                Ecrire ("Donner un entier : ") ; Lire T[i] ;
        FinPour
        Pour i de 0 à n-1
                Si T[i] mod 2 <>0 Alors Ecrire T[i] ;
        FinPour
Fin
```

Algorithme_c ;

```
Var i, n, j, k: entier ;
Tableau T[n], Ti[n], Tp[n] : entier ; // on suppose par défaut que les trois tableaux ont la
                                        même taille
Début
        Ecrire ("Donner le nombre de valeurs à saisir ") ; Lire (n) ;
        Pour i de 0 à n-1
                Ecrire ("Donner un entier : ") ;  Lire T[i] ;
        FinPour
```

```
        j←0 ; k←0 ;
        Pour i de 0 à n-1
                Si (T[i] mod 2 <>0) Alors Ti[j] ← T[i] ; j←j+1 ;
                Sinon Tp[k] ← T[i] ; k←k+1 ;
        FinPour
        Pour i de 0 à j-1
                Ecrire Ti[i] ;
        FinPour
        Pour i de 0 à k-1
                Ecrire Tp[i] ;
        FinPour
    Fin
```

Solution 2:

a)

```
Algorithme Max_tableau1D
Var i, pos: entier ;  max : réel ;
Tableau A [10]: réel ;
Début
        Pour i de 0 à 9
                Ecrire ("Entrer un nombre:");  Lire (A[i]) ;
                Si (i = 0 OU max< A[i]) Alors
                        max ← A[i]; pos ← i ;
                Finsi
        Finpour
        Ecrire ("Le maximum du tableau est :" , max);
        Ecrire ("Il se trouve à la position", pos+1);
Fin
```

b)

```
Algorithme Max_tableau2D
Var i,j, pos: entier ;  max  : réel ;
Tableau A[5] [10]: réel ;
Début
        Pour i de 0 à 4
                Pour j de 0 à 9
                        Ecrire ("Entrer un nombre:");
```

```
                    Lire (A[i][j]) ;
                    Si (i = 0 ET j = 0 OU max< A[i] [j]) Alors
                        max ← A[i] [j] ; lin ← i ; col← j ;
                    Finsi
                Finpour
            Finpour
            Ecrire ("Le maximum du tableau est :" , max);
            Ecrire ("Il est positionné à la ligne", lin+1, "et la colonne", col+1);
Fin
```

Solution 3 :

```
Algorithme Recherche_tableau2D
Var i, j, n, m: entier ; v : réel ; drap  : booléen ;
Tableau A[n][m]: réel ;
Début
        Ecrire ("Donner le nombre de lignes:"); Lire(n) ;
        Ecrire ("Donner le nombre de colonnes:"); Lire(m) ;
        Pour i de 0 à n-1
            Pour j de 0 à m-1
                Ecrire ("Entrer un nombre:");  Lire (A[i][j]) ;
            Finpour
        Finpour
        Ecrire ("Donner la valeur recherchée:"); Lire(v) ;
        drap← Faux ; i←0 ; j←0 ;
        Tan tque (i < n ET drap=Faux) faire
            Tant que (j < m ET drap=Faux) faire
                Si (A[i][j]=v) Alors drap ← Vrai ;
                Finsi
            FinTantque
        FinTantque
        Si (drap=Vrai) Alors
            Ecrire ("La valeur recherchée existe dans le tableau");
        Sinon
            Ecrire ("La valeur recherchée n'existe pas dans le tableau");
        Finsi
Fin
```

Solution 4 :

```
Algorithme Occurence_tableau2D
Var i, j, n, m, compt: entier ; v : réel ;
Tableau A[n][m]: réel ;
Début
        Ecrire ("Donner le nombre de lignes:"); Lire(n) ;
        Ecrire ("Donner le nombre de colonnes:"); Lire(m) ;
        Pour i de 0 à n-1
                Pour j de 0 à m-1
                        Ecrire ("Entrer un nombre:");
                        Lire (A[i][j]) ;
                Finpour
        Finpour
        Ecrire ("Donner une valeur:"); Lire(v) ;
        compt← 0 ; i←0 ; j←0 ;
        Pour i de 0 à n-1
                Pour j de 0 à m-1
                        Si (A[i][j]=v) Alors compt← compt+1 ;
                        Finsi
                Finpour
        Finpour
        Ecrire ("Le nombre d'occurrences de la valeur donnée est :", compt);
Fin
```

Solution 5 :

```
Algorithme Calcul_rationnel ;
Structure Rationnel  // Définition de la structure d'un nombre rationnel
    numerateur, denominateur: entier;
FinStructure ;
Var   p, q, r : Structure Rationnel ;
Début
  /* Saisie */
  Ecrire ("Entrez le numérateur et le dénominateur du premier nombre : ");
  Lire (p.numerateur, p.denominateur);

  Ecrire ("Entrez le numérateur et le dénominateur du deuxième nombre : ");
  Lire (q.numerateur, q.denominateur);
```

```
/* Affichage */
Ecrire (p.numerateur, p.denominateur);
Ecrire (q.numerateur, q.denominateur);
/* Multiplication*/
r.numerateur ← p.numerateur * q.numerateur ;
r.denominateur ← p.denominateur * q.denominateur ;
Ecrire (r.numerateur, r.denominateur);
/* Addition*/
r.numerateur ← p.numerateur * q.denominateur + q.numerateur * p.denominateur;
r.denominateur ← p.denominateur * q.denominateur;
Ecrire (r.numerateur, r.denominateur);
Fin
```

Solution 6 :

```
Algorithme Tableau_comptes ;
Structure Date
        jour, mois, annee : entier ;
FinStructure ;
Structure Compte
        num: entier;
        nom, pnom: chaine ;
        dtOuvert : Date ;
FinStructure ;
Var  i , n : entier ;
Tableau C[n] : Structure Compte ;
Début
        Ecrire ("Nombre des comptes : ") ;
        Lire (n) ;
        Pour i de 0 à n-1
                Ecrire ("No du compte: ") ;
                Lire(C[i].num) ;
                Ecrire ("Nom et prénom du client : ") ;
                Lire(C[i].nom, C[i].pnom) ;
                Ecrire ("Date de naissance du client : ") ;
                Lire(C[i].dtOuvert.jour, C[i].dtOuvert.mois, C[i].dtOuvert.annee) ;
        Finpour
```

```
        Pour i de 0 à n-1
            Ecrire ("N°:" C[i].num, "Nom et prénom:", C[i].nom,
            "Date de naissance:", C[i].dtOuvert.jour, C[i].dtOuvert.mois,
            C[i].dtOuvert.annee) ;
        Finpour
Fin
```

Corrigé série 6

Solution 1 :

La déclaration qui correspond à ce sous-programme est : b

Fonction MoySom (a : **entier**, b: **entier**, c: **entier**) : **réel** ;

Cette fonction prend trois paramètres entiers et elle est de type réel puisqu'elle va envoyer une valeur réelle (la moyenne).

Solution 2 :

a.

```
Fonction Min (n1, n2 : réel) : réel
Début
   Si (n1 < n2) Alors Retourner n1 ;
   Sinon  Retourner n2 ;
   FinSi
Fin
Fonction Max (n1, n2 : réel) : réel
Début
   Si (n1 < n2) Alors Retourner n2 ;
   Sinon  Retourner n1 ;
   FinSi
Fin
```

b.

```
Fonction Min_4 (n1, n2, n3, n4 : réel) : réel
Début
   Si (Min (n1, n2) < Min (n3, n4)) Alors Retourner Min (n1, n2) ;
   Sinon  Min (n3, n4) ;
   FinSi
Fin
Fonction Max_4 (n1, n2, n3, n4 : réel) : réel
Début
   Si (Max (n1, n2) > Max (n3, n4)) Alors Retourner Max (n1, n2) ;
   Sinon  Max (n3, n4) ;
   FinSi
Fin
```

c.

```
Algorithme Max_Min
Var a, b, c, d, min, max : réel;
... /* Ici on écrit les fonctions définies ci-dessus */
Début /* Algorithme principal */
        Ecrire (" Donner deux valeurs : ") ; Lire (a,b) ;
        min ← Min (a,b) ; // Appel de la fonction Min
        max ← Max (a,b) ; // Appel de la fonction Max
        Ecrire (" Le minimum et le maximum des deux valeurs données sont
        respectivement: min, max") ;
        /* Cas de quatre nombres */
        Ecrire (" Donner quatre valeurs : ") ;
        Lire (a,b,c,d) ;
        min ← Min_4 (a,b) ; // Appel de la fonction Min
        max ← Max_4 (a,b) ; // Appel de la fonction Max
        Ecrire (" Le minimum et le maximum des valeurs données sont respectivement:
        min, max") ;
Fin
```

Solution 3 :

```
Algorithme Tri_Liste ;
/* Déclaration de l'enregistrement Etudiant */
Structure Etudiant
        num: entier;
        nom, pnom: chaine ;
        moy : réel ;
FinStructure ;
/* Déclaration de la liste des étudiants et des autres variables nécessaires */
Var  i , N : entier ;
Tableau Liste [N] : Structure Etudiant ;
/*Déclaration du prototype de la procédure de tri, la définition est laissée à la fin*/
Procédure Tri (Tableau T : Structure Etudiant, N : entier) ;
/* programme principal */
Début
        Ecrire ("Donner le nombre des étudiants : ") ;
        Lire (N) ;
        Pour i de 0 à N-1
```

```
                Ecrire ("Numéro d'étudiant: ") ; Lire (Liste[i].num) ;
                Ecrire ("Nom et prénom : ") ; Lire (Liste[i].nom, Liste[i].pnom) ;
                Ecrire ("La moyenne du bac : ") ; Lire (Liste[i].moy) ;
        Finpour
        Tri(Liste,N) ; // Appel de la procédure Tri
        Ecrire ("La liste des étudiants triés par moyenne est:") ;
        Pour i de 0 à N-1
                Ecrire ("N:"Liste[i].num,"Nom et prénom: ", Liste[i].nom,
                Liste[i].pnom,"Moyenne:", Liste[i].moy) ;
        Finpour
Fin
```

/ On applique dans ce qui suit l'algorithme de tri par sélection vu au cours selon un ordre décroissant sur la moyenne */*

```
Procédure Tri (Tableau T : Structure Etudiant, N : entier)
Var i, posmax : entier ; temp : Structure Etudiant ;
Début
        Pour i de 0 à N-2
                posmax ← i ;
                Pour j de i + 1 à N-1
                    Si (T(j) > T(posmax)) Alors  posmax ← j ;  Finsi
                Finpour
                Si (posmax ≠ i) Alors
                    temp ← T(posmax) ;
                    T(posmax) ← T(i) ;
                    T(i) ← temp ;
                Finsi
        Finpour
Fin
```

Solution 4 :

```
Algorithme Gestion_vente ;
Structure Produit   // Définition de la structure Produit
    ref, quantite : entier;
    type : chaine;
    prix : réel ;
FinStructure ;
Var   p : Structure Produit ;
```

/ Définition des sous-programmes */*

```
Fonction Saisie () : Structure Produit
```

```
 Var p : Structure Produit ;
 Début
    Ecrire ("Entrez le type du produit : ");
    Lire (p.type);
    Ecrire ("Entrez la référence : ");
    Lire (p.ref);
    Ecrire ("Entrez le prix : ");
    Lire (p.prix);
    Ecrire ("Entrez la quantité : ");
    Lire (p.quantite);
    Retourner(p) ;
 Fin
Procédure Affichage (p : Structure Produit )
 Début
    Ecrire ("Type :", p.type);
    Ecrire ("Référence : ", p.ref);
    Ecrire ("Prix : ", p.prix, "DA");
    Ecrire ("Quantité : ", p.quantite);
 Fin
Procédure Commande_prod ()
 Var qte : entier ;
      p : Structure Produit ;
 Début
    p ← Saisie() ;
    Ecrire ("Entrez la quantité commandée : ");
    Lire (qte);
    Ecrire ("Récapitulatif de la commande :");
    Affichage(p);
    Ecrire ("Valeur de la commande : ", p.prix * qte, "DA");
 Fin
 Début  /* Programme principal */
    Commande_prod () ;
 Fin
```

Partie III – Travaux pratiques en C

TP 1

1. Objectif

Initiation au langage C.

2. Présentation du langage C

Le langage C a été conçu en 1972 par Dennis Richie[2] et Ken Thompson[1], afin de développer le système d'exploitation UNIX. Ensuite en 1978, Brian Kernighan et Denis Ritchie, publient la définition classique du C dans le livre *The C Programming language*[3].

Le C est un langage compilé (par opposition au langage interprété). Cela signifie qu'un programme C est décrit par un fichier texte, appelé fichier source. Ce fichier n'étant pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé compilateur.

- **Les composants élémentaires du C**

Un programme en langage C est constitué des groupes de composants suivants :

– les identificateurs,

– les mots-clés,

– les opérateurs,

– les signes de ponctuation.

On peut ajouter à ces groupes les commentaires, qui ne sont pas considérés dans la compilation.

- **Structure d'un programme C**

Un programme C se présente sous la forme suivante :

```
[directives au préprocesseur]
main()
{
        déclarations de variables internes
        instructions    /* commentaire */
 }
```

[2] Chercheurs aux *Bell Laboratories*.

[3] Kernighan (B.W.) et Richie (D.M.), The C programming language. Prentice Hall, 1988, 2nd edition.

- Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de directives. Les différentes directives au préprocesseur sont introduites par le caractère #, telles que:
  - l'incorporation de fichiers source ou fichiers de bibliothèque (**#include**),
  - la définition de constantes symboliques (**#define**),

  Par exemple, ***# include <fichier.h>*** où *fichier.h* est le nom d'un fichier entête.
- La fonction principale ***main*** peut avoir des paramètres formels. On supposera dans un premier temps que la fonction main n'a pas de valeur de retour. Ceci est toléré par le compilateur mais produit souvent un message d'avertissement.
- Une instruction est une expression suivie d'un point-virgule. Plusieurs instructions peuvent être rassemblées par des accolades **{** et **}** pour former un bloc (ou instruction composée) qui est syntaxiquement équivalent à une instruction. Par exemple :

```
if (x != 0)
{
        z = y / x;
        t = y % x;
}
```

- Ainsi, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

➢ **Syntaxe du langage C**

- **Quelques mots-clés :**

  const , int , char , float , double , signed , unsigned , short , long , if , else , switch , case , for , while , do , break , continue , struct , typedef , void , goto , return, ...

- **Les types de base :**

  char : codé en 1 octet (8 bits)
  int : codé en 2 octets (16 bits)
  float : codé en 4 octets
  double : codé en 8 octets

- **Les opérateurs arithmétiques:**

  + : Addition
  - : Soustraction
  ***** : Multiplication
  / : division
  % : modulo (reste de la division entière)

- **Les opérateurs relationnels (de comparaison):**

  = = : égal à
  != : différent de
  < , **<=** , **>=** , **>** : inférieur, inférieur ou égal, supérieur ou égal , supérieur

- **Les opérateurs logiques :**

  ! : NON logique (unaire)

  && : ET logique (binaire)

  || : OU logique (binaire)

Remarque : Il existe d'autres types d'opérateurs en C (pour plus de détails, voir par exemple : http://www.commentcamarche.net/contents/115-langage-c-les-operateurs).

- **Les fonctions d'entrée et de sortie :**
  - La fonction ***scanf()*** est une fonction de lecture (entrée).

    Syntaxe : ***scanf*** ("code format", &variable1, …, &variableN) ;

    Le symbole **&** est obligatoire dans la fonction ***scanf*** devant toute variable sauf pour les variables de type chaîne de caractères.

  - La fonction ***printf()*** est une fonction d'affichage (sortie).

    Syntaxe : ***printf*** ("code format", variable1, …, variableN) ;

Le « **code format** » est un ensemble de codes associés aux types des variables lues ou écrites :

Type	Code format	Interprétation
int	%d	ce code est remplacé par un entier
float	%f	ce code est remplacé par un réel
char	%c	ce code est remplacé par un caractère
char	%s	ce code est remplacé par une chaîne de caractères

Remarque :

Les deux fonctions ***scanf()*** et ***printf()*** nécessitent l'utilisation du fichier entête <**stdio.h**>.

3. Travail demandé

Ecrire un programme C qui emploie ces différentes notions. Essayer de le compiler et l'exécuter.

TP 2

1. Objectif

Initiation à la résolution des problèmes et à l'implémentation des algorithmes.

2. Travail demandé

Soit la facture suivante réalisée sous Excel :

Désignation	Prix unitaire	Quantité	Prix total
Art 1	1250,00	2	2500,00
Art 2	230,00	5	1150,00
Art 3	450,50	4	1802,00
Art 4	880,33	2	1760,66
Art 5	190,00	10	1900,00
		Montant total HT	**9112,66 DA**
		Taxe (19%)	**1731,40 DA**
		Montant total TTC	**10844,06 DA**

A)

1- Ecrire l'algorithme qui permet de réaliser cette facture.

C'est-à-dire, on doit lire l'article, son prix unitaire et sa quantité, ensuite on calcule son prix total. Enfin, on doit afficher le montant total de l'ensemble des articles (sans et avec la taxe).

2- Traduire cet algorithme en programme C et tester-le sur votre machine.

B)

1- Modifier le programme pour une facture qui peut contenir n articles (n est donné).

2- Modifier, une autre fois, le programme de sorte que l'utilisateur ait la possibilité de limiter le montant total de sa facture (un seuil donné qui ne doit pas être dépassé).

C) Supposons que l'on veut faire une réduction de 10% sur toute facture arrêtée entre 20000 et 30000 DA, et de 15% au-delà de 30000DA. Ajouter la modification nécessaire à votre programme.

TP 3

1. Objectif

Initiation à la résolution des problèmes mathématiques.

2. Exercice 1

a) Ecrire un programme C qui permet de résoudre une équation du second degré :

$$ax^2 + bx + c = 0 \text{ ; solution } x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

b) Utiliser votre programme pour remplir le tableau ci-dessous :

a	36	60	18	3	300
b	-30	20	18	6	65
c	6	5	0	3	-15
x1					
x2					

3. Exercice 2

a) Ecrire un programme C pour calculer la somme suivante :

$$S = 1 + 2 + 3 + \ldots + N$$

N est un entier donné par l'utilisateur.

b) Exécuter votre programme pour remplir le tableau ci-dessous :

N	100	150	700	800	1000
S					

c) Modifier votre programme pour calculer la somme suivante :

$$S' = 1 + 3 + 5 + \ldots + (2k + 1)$$

Tel que (2k+1) est un nombre impair.

d) Si vous avez utilisé un pas égal à 2, réécrire le programme pour calculer la même somme mais cette fois avec un pas de boucle égal à 1 (for (… ;… ; **i++**)).

e) Exécuter votre programme pour remplir le tableau ci-dessous :

N	5	20	300	900	1000
S					

TP 4

1. Objectif

Manipulation des tableaux (vecteurs et matrices).

2. Exercice

Ecrire un programme C qui permet de calculer :

a) la somme de deux vecteurs,

b) la somme de deux matrices,

c) le produit de deux matrices.

Tester vos programmes avec les exemples suivants:

1) $\begin{matrix}4\\8\\5\end{matrix} + \begin{matrix}4\\3\\2\end{matrix} = \begin{matrix}8\\11\\7\end{matrix}$

2) $\begin{matrix}1&2\\4&0\\6&1\end{matrix} + \begin{matrix}1&0\\6&9\\2&3\end{matrix} = \begin{matrix}2&2\\10&9\\8&4\end{matrix}$

3) $\begin{matrix}1&2\\4&0\\6&1\end{matrix} \text{ X } \begin{matrix}1&4&6\\2&5&7\end{matrix} = \begin{matrix}5&14&20\\4&16&24\\8&29&43\end{matrix}$

Remarque :

Dans les trois cas, le programme doit lire la taille du tableau et vérifier quelques conditions nécessaires telles que :

- Avoir la même taille pour la somme.
- Le nombre de colonnes du premier tableau doit être identique au nombre de lignes du deuxième tableau, dans le cas du produit.

Corrigé type

Cas c)

```
main()
{
        int  x, y, z, i,  j, k ;
        printf("Nombre de lignes de M1 :") ;
        scanf("%d",&x) ;
        printf("Nombre de colonnes de M1 :") ;
        scanf("%d",&y) ;
        printf("Nombre de colonnes de M2:") ;
        scanf("%d",&z) ;
        float M1[x][y] , M2[y][z] , M3[x][z] ;
        //Lecture de M1
        for(i=0 ;i<x ;i++)
            for(j=0 ;j<y ;j++)
                scanf("%f",&M1[i][j]) ;

        //Lecture de M2
        for(i=0 ;i<y ;i++)
            for(j=0 ;j<z ;j++)
                scanf("%f",&M2[i][j]) ;
        //Calcul du produit matriciel
        for(i=0 ;i<x ;i++)
             for(j=0 ;j<z ;j++)
                {
                    M3[i][j] = 0 ;
                 for(k=0 ;k<y ;k++)
                     M3 [i][j] = M3 [i][j] + M1[i][k] * M2[k][j]  ;
                }
        //Affichage du résultat
        for(i=0 ;i<x ;i++)
            {  for(j=0 ;j<z ;j++)
                    printf("%f", M3[i][j]) ;
               printf("\n") ;
            }
}
```

TP 5

1. Objectif

Utilisation des fonctions

2. Exercice 1

Écrire un sous-programme qui détermine si un nombre entier positif est un nombre déficient.

Écrire le programme appelant (fonction principale).

Remarque : on rappelle qu'un nombre est déficient s'il est strictement supérieur à la somme de ses diviseurs stricts (c.-à-d. sauf lui-même).

Par exemple :

$8 > 1 + 2 + 4$ est déficient,

$6 = 1 + 2 + 3$ n'est pas déficient,

$12 < 1 + 2 + 3 + 4 + 6$ n'est pas déficient.

3. Exercice 2

Écrire un sous-programme qui reçoit en paramètre deux valeurs représentant le numérateur et le dénominateur d'une fraction et puis affiche une fraction équivalente simplifiée.

Par exemple, si les valeurs données initialement sont 15 et 25, le sous-programme affichera les messages suivants:

La fraction initiale est : 15/25

La fraction simplifiée est : 3/5

Corrigé type

Ex1 :

```
int Deficient (int N) // la fonction Deficient permet de vérifier si un nombre est déficient ou pas
{
        int i, som =0 ;
        for (i=N-1 ; i>=1 ; i--)
                if (N%i == 0) som = som+1 ;
        if (N > som) return 1 ;
        else return 0 ;
}
main()// la fonction principale
{
        int N ;
        printf("Entrez un entier N :") ; scanf("%d",&N) ;
        if(Deficient(N))
                printf("%d est deficient \n", N) ;
        else printf("%d n'est pas deficient \n", N) ;
}
```

Ex2 :

```
/* Définition de la fonction de simplification */
void Simplifier (int numerateur , int denominateur )
{
        printf ("\n La fraction initiale : %d/%d \n", numerateur , denominateur );

        int diviseur = 2;
        while ((numerateur >= diviseur)&&(denominateur >= diviseur))
        {
                if ((numerateur % diviseur == 0) && (denominateur % diviseur == 0))
                {
                        numerateur = numerateur / diviseur ;
                        denominateur = denominateur / diviseur ;
                }
                else  diviseur ++;
        }
        printf ("\n La fraction simplifiee : %d/%d \n", numerateur, denominateur );
 }
```

```
/* La fonction principale */

main ()
{
        int  N1, N2;
        printf (" Entrer 2 entiers, numerateur et denominateur :");
        scanf ("%d %d", &N1,&N2 );
        Simplifier( N1, N2 );

}
```

TP 6

1. Objectif

Manipulation des structures à travers les fonctions.

2. Exercice

Une société de menuiserie gère un stock de panneaux de bois. Chaque panneau possède une largeur, une longueur et une épaisseur en millimètres, ainsi que le type de bois qui peut être *pin* (code 0), *chêne* (code 1) ou *hêtre* (code 2).

a) Définir une structure Panneau contenant toutes les informations relatives à un panneau de bois.

b) Écrire des fonctions de saisie et d'affichage d'un panneau de bois.

c) Écrire une fonction qui calcule le volume en mètres cube d' un panneau.

Corrigé type

a)

```
typedef struct bois
{
 float largeur, longueur, epaisseur;
 char essence;
} Panneau;
```

b)

```
Panneau Saisie()
{
  Panneau p;
  printf("Entrez la largeur, la longueur et l'épaisseur : ");
  scanf("%f %f %f", &p.largeur, &p.longueur, &p.epaisseur);
  printf("Entrez l'essence de bois : ");
  scanf("%c", &p.essence);
  return p;
}

void Affichage(Panneau p)
{
  printf("Panneau en ");
  switch (p.essence)
  {
        case '0' : printf("pin\n"); break;
        case '1' : printf("chêne\n"); break;
        case '2' : printf("hêtre\n"); break;
        default: printf("inconnue\n");
  }
  printf("largeur = %f ; longueur = %f ; epaisseur = %f\n", p.largeur, p.longueur, p.epaisseur);
}
```

c)

```
float Volume(Panneau p)
{
```

```
    return (p.largeur * p.longueur * p.epaisseur) / 1e9;
}
```

TP 7

1. Objectif

Applications de la récursivité.

2. Exercice 1

Écrire une fonction récursive pour calculer la somme :

$$Un = 1 + 2^2 + 3^2 + \cdots + n^2$$

Ecrire la fonction principale.

3. Exercice 2

Les coefficients binomiaux C_n^k pour $k \leq n$ sont donnés par la formule de Pascal :

$$C_n^0 = 1 \; ; \; C_n^n = 1$$
$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

a) Proposer une fonction récursive pour calculer les coefficients binomiaux.

b) Proposer une fonction itérative pour réaliser le même calcul.

Corrigé type

Ex1 :

```
int Somme(int n)
{
  if (n <= 0)   return 0;
  else  return (n * n + Somme(n - 1));
}

/* fonction principale */

main ()
{
        int  n;
        printf (" Entrez un entier :");
        scanf ("%d ", &n);
        printf("La somme = %d", Somme(n));
}
```

Ex2 :

a)

```
int Coeffbinom(int k, int n)
{
   if (k == 0 || k == n) return 1;
   else  return Coeffbinom(k - 1, n - 1) + Coeffbinom(k, n - 1);
}
```

b)

```
/*Cette fonction calcule tous les coefficients binomiaux en les mettant dans tab*/
void Coeffbinom(int k, int n, int **tab)
{
    int i, j;
    for (i = 0; i <= n; i++)
    {
      tab[i][0] = 1;
      tab[i][i] = 1;
    }
    for (i = 2; i <= n; i++)
       for (j = 1; j < i; j++)
           tab[i][j] = tab[i - 1][j - 1] + tab[i - 1][j];
}
```

Conclusion générale

Ce polycopié constitue un manuel d'initiation à l'algorithmique et à la programmation à travers un ensemble de cours et d'exercices corrigés destinés essentiellement aux étudiants en début de parcours (Domaine MI, ainsi que ST et SM).

Dans la première partie qui représente le volet théorique, les concepts et les fondements préliminaires des algorithmes ont été abordés. Dans le premier chapitre, le lecteur est initié à la notion de variables et de constantes, ainsi qu'à la syntaxe générale d'écriture d'un algorithme à travers des exemples de base. Au deuxième chapitre, l'accent a été mis sur les trois instructions incontournables dans l'écriture d'un algorithme, à savoir l'affectation, la lecture et l'écriture. Ces instructions qui sont à la base des entrées/sorties de l'ordinateur sont illustrées à travers des exemples simplifiés. Le troisième et le quatrième chapitre, ont été consacrés respectivement aux instructions conditionnelles et itératives. Ces instructions permettent de tester, de décider et de répéter certaines actions par l'algorithme sans intervention de l'utilisateur. Dans les chapitres cinq et six, les deux structures de base de mémorisation de données, à savoir les tableaux et les enregistrements, ont été présentées avec des exemples sur leur manipulation et leur utilisation. Le problème de tri a été évoqué aussi à travers la description des deux algorithmes : tri par sélection et tri par insertion. Dans le chapitre sept, la notion de sous-programme (fonctions et procédures) a été présentée avec des illustrations sur le principe de fonctionnement et les modes de passage de paramètres associés. La notion de récursivité a été abordée aussi. Dans le dernier chapitre, le lecteur a été initié à l'utilisation d'un nouveau type de variable à travers la notion des pointeurs et leur utilité dans la gestion de la mémoire et la manipulation des différentes structures de données.

La deuxième et la troisième partie de ce polycopié ont comme objectif de consolider les connaissances acquises dans la première partie à travers un ensemble d'exercices (avec ou sans correction) sous formes de travaux dirigés dans la deuxième partie, et travaux pratiques dans la troisième partie qui constitue elle-même une initiation à la programmation avec le langage C.

Références bibliographiques

[1] D. BOUCHIHA « Initiation à l'Algorithmique et à la Programmation en Pascal » Errachad, 2019.

[2] T. CORMEN, C. LEISERSON, R. RIVEST et C. STEIN « Introduction à l'algorithmique : Cours et exercices corrigés » 2ème éd. Dunod, 2002.

[3] P. DAMPHOUSSE « Petite introduction à l'algorithmique : à la découverte des mathématiques du pas à pas » Ellipses, 2005.

[4] F. DAOUDI « Introduction à l'algorithmique » l'Abeille, 2008.

[5] M. DIVAY « Algorithmes et structures de données génériques : Cours et exercices corrigés en langage C » 2ème éd. Dunod, 2004.

[6] S. GRAÏNE « Le langage C avec exercices corrigés » l'Abeille, 2009.

[7] C. KHICHANE « Le leader de l'Algorithmique - Cours et exercices avec corrections » El Maarifa, 2004.

[8] D.E. KNUTH « The art of computer programming: Volume 1: Fundamental Algorithms » 3rd Ed. Addison-Wesley, 1997.

[9] R. MALGOUYRES, R. ZROUR et F. FESCHET « Initiation à l'algorithmique et à la programmation en C : Cours avec 129 exercices corrigés » 2ème éd. Dunod, 2014.

[10] S. ROHAUT « Algorithmique Techniques fondamentales de programmation » ENI, 2007.

[11] H. ZEMANEK « Al-khorezmi his background, his personality his work and his influence ». In : Algorithms in Modern Mathematics and Computer Science. Springer, Berlin, Heidelberg, 1981. p. 1-81.