

# **Algorithmique et programmation en Pascal**

**Cours avec 190 exercices corrigés**

FOR AUTHOR USE ONLY

**Éditions universitaires européennes**

# Table des matières

<b>Avant-propos .....</b>	<b>9</b>
<b>Chapitre 1 : Introduction.....</b>	<b>10</b>
1. Bref historique sur l'informatique .....	10
1.1. Introduction .....	10
1.2. Historique .....	10
1.3. Composantes de l'ordinateur .....	12
1.4. Notions de programme et de système d'exploitation .....	13
2. Introduction à l'algorithmique.....	14
2.1. Introduction .....	14
2.2. Notion d'algorithme.....	14
2.2.1. Définitions .....	14
2.2.2. Exemples.....	15
2.2.3. Langage de description d'algorithme .....	15
2.2.4. Caractéristiques d'un algorithme .....	16
2.2.5. Notions de programme et de langage de programmation .....	17
<b>Chapitre 2 : Les algorithmes séquentiels simples .....</b>	<b>19</b>
1. Principe général.....	19
2. Variables et constantes .....	19
3. Types .....	20
3.1. Le type entier .....	21
3.2. Le type réel .....	21
3.3. Le type booléen .....	21
3.4. Le type caractère.....	21
4. Opérations de base.....	22
4.1. L'affectation .....	22
4.2. Les entrées/sorties.....	23
5. Construction d'un algorithme simple .....	24
6. Représentation d'un algorithme par un organigramme.....	25
7. Traduction en langage Pascal .....	26
7.1. Exemple .....	26
7.2. Règles de base .....	27
7.3. Lecture .....	28
7.4. Ecriture .....	28
7.5. Manipulation des nombres.....	29
7.6. Manipulation des caractères.....	31
7.7. Manipulation des booléens .....	32
8. Exercices corrigés.....	32
8.1. Exercices.....	32
8.2. Corrigés .....	34
<b>Chapitre 3 : Les structures conditionnelles.....</b>	<b>44</b>
1. Introduction .....	44

2. Structure conditionnelle simple .....	44
3. Structure conditionnelle composée .....	45
4. Structure conditionnelle multiple .....	46
5. Le branchement .....	48
6. Exercices corrigés.....	48
6.1. Exercices.....	48
6.2. Corrigés .....	53
<b>Chapitre 4 : Les boucles.....</b>	<b>74</b>
1. Introduction .....	74
2. La boucle Tant que .....	74
3. La boucle Répéter.....	75
4. La boucle Pour.....	76
5. Les boucles imbriquées .....	78
6. Exercices corrigés.....	79
6.1. Exercices.....	79
6.2. Corrigés .....	82
<b>Chapitre 5 : Les tableaux et les chaînes de caractères.....</b>	<b>103</b>
1. Introduction .....	103
2. Le type tableau .....	103
2.1. Manipulation d'un tableau .....	104
2.2. Tri d'un tableau .....	104
2.3. Tableau à deux dimensions .....	105
3. Les chaînes de caractères.....	106
3.1. Opérations sur les chaînes de caractères .....	106
3.2. Déclaration d'une chaîne de caractères.....	107
3.3. Manipulation des chaînes de caractères .....	107
3.4. Edition d'une chaîne de caractères .....	108
3.5. Tableau de chaînes de caractères .....	109
4. Exercices corrigés.....	109
4.1. Exercices.....	109
4.2. Corrigés .....	122
<b>Chapitre 6 : Les sous-programmes : procédures et fonctions.....</b>	<b>162</b>
1. Introduction .....	162
2. Les sous-programmes .....	166
2.1. Les procédures.....	169
2.2. Les fonctions.....	170
3. Les variables locales et les variables globales .....	173
4. Le passage des paramètres.....	178
5. Construction d'un algorithme complexe.....	182
6. La récursivité (récursion).....	182
6.1. Définition.....	182
6.2. Exemples .....	182
6.2.1. La factorielle.....	182
6.2.2. Le PGCD .....	187

6.2.3. Tour de Hanoï .....	188
6.3. Transformation des boucles en procédures récursives .....	191
6.4. La récursion croisée (indirecte) .....	192
7. Exercices corrigés .....	193
7.1. Exercices .....	193
7.2. Corrigés .....	198
<b>Chapitre 7 : La complexité des algorithmes .....</b>	<b>215</b>
1. Introduction .....	215
2. Calcul de la complexité .....	215
2.1. Complexité dans le pire des cas (le cas le plus défavorable) .....	215
2.2. Complexité dans le meilleur des cas (le cas le plus favorable) .....	215
2.3. Complexité moyenne .....	216
3. Etude de cas .....	216
4. Ordres de grandeur (Classes de complexité) .....	216
5. Exercices corrigés .....	218
5.1. Exercices .....	218
5.2. Corrigés .....	221
<b>Chapitre 8 : Les types définis par l'utilisateur .....</b>	<b>224</b>
1. Introduction .....	224
2. Types simples définis par l'utilisateur .....	224
2.1. Le type énuméré .....	224
2.2. Le type intervalle .....	225
3. Types structurés définis par l'utilisateur .....	226
3.1. Le type ensemble .....	226
3.2. Le type enregistrement .....	227
4. Exercices corrigés .....	230
4.1. Exercices .....	230
4.2. Corrigés .....	232
<b>Chapitre 9 : Les fichiers .....</b>	<b>242</b>
1. Introduction .....	242
2. Les fichiers .....	242
3. Types de fichiers .....	242
3.1. Les fichiers texte .....	242
3.2. Les fichiers typés .....	243
3.3. Les fichiers non typés .....	244
4. Structure des enregistrements dans un fichier typé .....	244
5. L'organisation des enregistrements dans un fichier typé .....	245
5.1. Organisation séquentielle .....	246
5.2. Organisation relative .....	246
5.3. Organisation indexée .....	246
6. Les méthodes d'accès aux fichiers .....	246
6.1. Accès séquentiel .....	247
6.2. Accès direct .....	247
6.3. Accès indexé .....	247

7. Manipulation des fichiers .....	248
7.1. Assignment .....	248
7.2. Ouverture .....	249
7.3. Lecture et écriture .....	249
7.4. Accès aux enregistrements .....	249
7.5. Fermeture du fichier .....	250
8. Stratégies de traitement des fichiers .....	254
9. Exercices corrigés .....	255
9.1. Exercices .....	255
9.2. Corrigés .....	256
<b>Chapitre 10 : Les listes chaînées .....</b>	<b>267</b>
1. Introduction .....	267
2. Les pointeurs .....	267
3. Gestion dynamique de la mémoire .....	268
4. Les listes chaînées .....	271
5. Opérations sur les listes chaînées .....	273
5.1. Créer et remplir une liste .....	274
5.2. Ajouter un élément au début de la liste .....	278
5.3. Insérer un élément dans la liste .....	280
5.4. Supprimer la tête de la liste .....	283
5.5. Supprimer un élément de la liste .....	285
5.6. Afficher les éléments de la liste .....	288
6. Les listes doublement chaînées .....	288
6.1. Créer et remplir une liste doublement chaînée .....	289
6.2. Ajouter un élément au début de la liste doublement chaînée .....	293
6.3. Insérer un élément dans la liste doublement chaînée .....	296
6.4. Supprimer la tête de la liste doublement chaînée .....	301
6.5. Supprimer un élément de la liste doublement chaînée .....	302
6.6. Afficher les éléments de la liste doublement chaînée .....	304
7. Les listes chaînées particulières .....	306
7.1. Les piles .....	306
7.1.1. Primitives d'accès .....	307
7.1.2. Représentation d'une pile par une liste doublement chaînée .....	307
7.2. Les files .....	316
7.2.1. Accès à une file .....	317
7.2.2. Représentation d'une file par une liste doublement chaînée .....	317
8. Exercices corrigés .....	327
8.1. Exercices .....	327
8.2. Corrigés .....	330
<b>Chapitre 11 : Les arbres .....</b>	<b>352</b>
1. Introduction .....	352
2. Définitions .....	352
3. Arbre binaire .....	353
3.1. Définition .....	353
3.2. Passage d'un arbre n-aire à un arbre binaire .....	353

3.3. Représentation chaînée d'un arbre binaire .....	354
3.4. Parcours d'un arbre binaire .....	357
3.4.1. Parcours préfixé (appelé aussi préordre ou RGD).....	357
3.4.2. Parcours infixé (appelé aussi projectif, symétrique ou encore GRD) .....	359
3.4.3. Parcours postfixé (appelé aussi ordre terminal ou GDR) .....	360
3.5. Arbres binaires particuliers .....	361
3.5.1. Arbre binaire complet .....	361
3.5.2. Arbre dégénéré.....	361
3.5.3. Arbre binaire ordonné .....	361
4. Exercices corrigés.....	362
4.1. Exercices.....	362
4.2. Corrigés .....	364
<b>Chapitre 12 : Les graphes.....</b>	<b>377</b>
1. Introduction .....	377
2. Définitions.....	377
3. Représentation d'un graphe .....	377
4. Parcours d'un graphe.....	379
4.1. Parcours en largeur d'abord .....	380
4.2. Parcours en profondeur d'abord.....	380
5. Applications des parcours d'un graphe.....	380
5.1. Calcul des composantes fortement connexes.....	381
5.2. Calcul du plus court chemin : algorithme de Dijkstra.....	382
6. Exercices corrigés.....	384
6.1. Exercices.....	384
6.2. Corrigés .....	386
<b>Chapitre 13 : Les tables de hachage.....</b>	<b>395</b>
1. Table de hachage .....	395
2. Exemple d'utilisation.....	395
3. Fonction de hachage.....	395
4. Choix de la fonction de hachage.....	396
5. Résolution des collisions .....	397
5.1. Chaînage .....	397
5.2. Adressage ouvert .....	397
6. Domaine d'utilisation .....	398
7. Exercices corrigés.....	399
7.1. Exercices.....	399
7.2. Corrigés .....	399
<b>INDEX.....</b>	<b>407</b>

FOR AUTHOR USE ONLY

## **Avant-propos**

Ce livre est destiné à tous ceux qui veulent acquérir des bases solides en algorithmique et structures de données. Les algorithmes de ce livre ont été traduits en langage Pascal.

Ce livre permet un apprentissage autonome. Les exercices de chaque chapitre ont une difficulté progressive. Après avoir lu et compris le cours, l'étudiant est conseillé d'essayer de résoudre les exercices par lui-même avant de consulter la correction. L'étudiant ne doit pas oublier qu'un même problème peut être résolu par différents algorithmes.

L'auteur de ce livre sera très reconnaissant de recevoir toute remarque, suggestion ou correction.

FOR AUTHOR USE ONLY



## Chapitre 1 : Introduction

### 1. Bref historique sur l'informatique

#### 1.1. Introduction

L'informatique est le traitement automatique de l'information : le *traitement* est la création, l'ajout, la suppression, la modification, le stockage, etc. ; *automatique* veut dire effectué par machine ; *l'information* est la description ou l'interprétation d'une donnée (par exemple, l'expression '5 est un entier' est une information).

L'informatique est à la fois une technique très récente et une science très ancienne : il s'agit d'une technique très récente, car les premiers ordinateurs, dignes de ce nom, ne sont apparus qu'après la deuxième guerre mondiale. Ces machines ont permis la mise en œuvre des programmes, des systèmes d'exploitation, etc. Par contre, en tant que science, l'informatique est très ancienne, car elle est liée au développement des mathématiques. Ceci est depuis que l'homme a appris à compter, c'est-à-dire depuis qu'il a inventé les nombres et leurs codages, puis les opérations arithmétiques élémentaires, telles que l'addition et la multiplication.

#### 1.2. Historique

Il est difficile de faire commencer l'histoire des ordinateurs à une date bien précise. Nous citerons les principales innovations qui ont facilité ou automatisé le calcul : vers 2500 avant J-C, apparaissait déjà le boulier qui permettait d'effectuer des opérations arithmétiques élémentaires. Six siècles plus tard, une tablette babylonienne en argile (1900-1600 avant J-C) permettra à partir de deux côtés d'un triangle rectangle, de trouver le troisième.

Les premiers dispositifs mécaniques d'aide au calcul apparaissent en 1642, Blaise Pascal invente une machine permettant d'additionner et de soustraire. En 1671, le mathématicien allemand Gottfried Wilhem Leibniz conçoit une machine permettant les quatre opérations arithmétiques.

Au cours du XIX<sup>e</sup> siècle, quelques nouvelles machines capables d'effectuer les quatre opérations élémentaires apparaissent : l'arithmomètre (1820) de Thomas de Colmar conçu à partir des idées de Leibniz, la machine à curseur du suédois Odhner (1875), le comptomètre à clavier de l'américain Felt (1885). Jacquard, perfectionnant en 1801 une invention de Vaucanson (1745), crée la carte perforée destinée à commander des métiers à tisser.

L'anglais Charles Babbage propose en 1822 sa "machine différentielle" permettant d'élever un nombre à la puissance  $n$ . Dans sa "machine analytique", on trouve un organe d'introduction de données, un organe de sortie des résultats et un organe de contrôle et de calcul qui utilisait des dispositifs mécaniques. En 1833, une mémoire était réalisée par l'intermédiaire de roues dentées, tandis que les opérations à effectuer étaient introduites à l'aide de cartes perforées.

L'ingénieur américain Herman Hollerith développe, pour le recensement de 1890, une machine électromécanique plus élémentaire que celle de Babbage, capable de trier des cartes perforées et de les compter. Hollerith fonde en 1896 la "Tabulating Machine Company" reprise en 1911 par Thomas Watson qui crée en 1924 la firme IBM (International Business Machine).

La guerre 1939-1945, suite à l'effort de plusieurs pays, va donner l'impulsion décisive à la naissance des premiers ordinateurs dignes de ce nom :

- Les Z2 et Z3 de l'allemand Zuse prêts en 1939 et 1941.
- La série des "Automatic Sequence Controlled Computer Mark" conçus par Howard Aiken. Le Mark I fonctionnera en 1944.
- L'ENIAC (1943-1946), destiné initialement au calcul de tables d'artillerie, de Prosper Eckert et John Mauchly, utilisait des tubes à vide. Cette machine prend une place considérable ( $270 \text{ m}^3$  - 30 tonnes) et consomme 150 kw. L'ENIAC était une machine décimale dont les entrées-sorties et la mémoire auxiliaire étaient réalisées par des cartes perforées.

Dès 1944, des théoriciens de Princeton, tels que J. Von Neumann, A. Buks et H. Goldstine, se penchent sur les problèmes logiques posés par la réalisation des ordinateurs, et établissent les bases des futurs développements, notamment l'utilisation du binaire, les notions de programmes stockés en mémoire, d'ordinateurs à usages multiples, etc. L'EDSAC (Cambridge University 1949) et l'EDVAC (University of Pennsylvania 1950) utilisent des mémoires à ligne de retard à mercure (inventées par Eckert) qui permettent de stocker des programmes et des nombres sous forme digitale.

Vers 1951, apparaissent le premier UNIVAC (Eckert et Mauchly) utilisant des diodes à cristal et des bandes magnétiques comme mémoire de masse et l'IAS (Von Neumann) où une mémoire à tube de Williams (1947) permet l'accès parallèle à tous les bits. C'est une machine binaire où les instructions modifiables comportent une adresse.

Entre 1953 et 1960 apparaissent de nombreuses innovations (2<sup>ème</sup> génération d'ordinateurs) : les tores de ferrite utilisés comme mémoire

(d'après les travaux de J.W. Forrester en 1951 au MIT), les circuits imprimés, les disques magnétiques, les transistors (inventés en 1948), etc. La vitesse de traitement et la capacité de mémoire augmentent considérablement. Les "ordinateurs" (mot inventé en 1953 par M. Perret pour traduire "computer") acquièrent une plus grande sécurité de fonctionnement et une facilité d'emploi permettant d'effectuer des tâches de plus en plus vastes pour un nombre plus important d'utilisateurs. La production en série commence, et les coûts de production diminuent rapidement.

En 1963, les circuits imprimés sont remplacés par des circuits intégrés (3<sup>ème</sup> génération). Les équipements se miniaturisent, et on voit apparaître les mini- et micro-ordinateurs. La vitesse d'exécution diminue.

Les unités centrales réalisées sous de très faibles volumes, qui forment ce qu'on appelle les microprocesseurs, ont en effet été largement diffusées en 1976, lorsque fut mise au point la fabrication de masse des composants de très haute intégration (Very Large Scale Integration ou VLSI). Cette technologie, encore aujourd'hui en pleine évolution, a permis, par exemple, de construire les unités centrales des IBM 4300 avec des plaquettes de silicium de 20 millimètres carrés, ayant 704 circuits logiques.

### **1.3. Composantes de l'ordinateur**

Un ordinateur est donc une machine capable de traiter automatiquement l'information. Il est constitué de composantes matérielles (HARDWARE) et de composantes logicielles (SOFTWARE). Les composantes matérielles sont essentiellement des cartes électroniques, des circuits intégrés, des câbles électriques, des supports de mémoire (disque dur, RAM, ROM...), des dispositifs d'entrée/sortie (écran, clavier, imprimante...), etc. Les logiciels, qui pilotent le fonctionnement des composantes matérielles, sont des programmes stockés sous forme codée dans la mémoire de l'ordinateur.

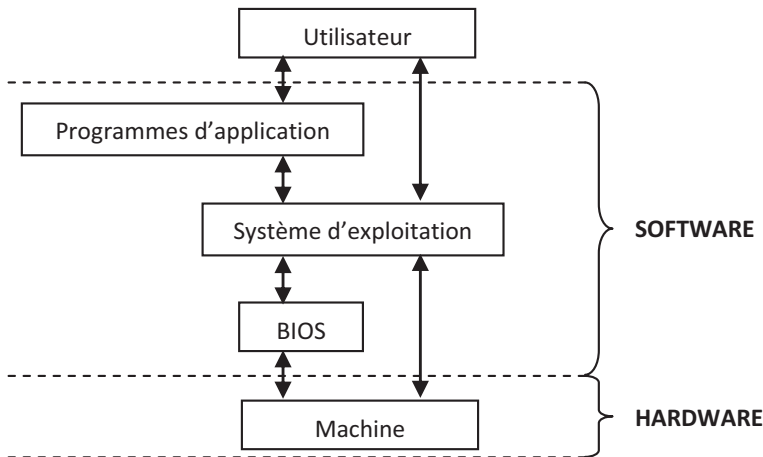
La forme actuelle d'un ordinateur la plus dominante est le PC (Personnel Computer). Un PC est constitué principalement de trois parties :

- A. Unité centrale : c'est la partie la plus importante dans un PC. Elle contient les éléments vitaux pour le bon fonctionnement d'un ordinateur, parmi lesquels on cite :
  - a. Le processeur : c'est le cerveau de la machine, constitué à son tour d'une UC (Unité de Commandes) qui a pour rôle le contrôle de la machine, et l'UAL (Unité Arithmétique et Logique) pour le calcul des expressions arithmétiques et logiques.

- b. La carte mère : avec laquelle toutes les autres composantes sont soit installées directement, soit connectées par des câbles ou des nappes.
  - c. La mémoire : permet le stockage de l'information. Il existe deux types de mémoire :
    - 1. La mémoire centrale : c'est là où les informations sont stockées pendant leur traitement. Il existe deux types de mémoires centrales.
      - La RAM (Random Access Memory) : elle stocke les programmes nécessaires pour le bon fonctionnement de la machine (système d'exploitation, antivirus, etc.), ainsi que ceux invoqués par l'utilisateur (Word, Excel, etc.). On peut dire que tout ce que l'on voit sur l'écran pendant le fonctionnement de la machine est stocké dans la RAM. On note que la RAM est une mémoire volatile, car elle perd son contenu lors de la coupure de l'alimentation électrique.
      - La ROM (Read Only Memory) : c'est une mémoire qui permet de lire uniquement, et non pas écrire (stocker) les informations. Elle contient un programme indispensable pour le démarrage de la machine appelé BOOT (Basic Input Output System).
    - 2. La mémoire auxiliaire : elle permet le stockage permanent des informations. Il existe plusieurs types de mémoires auxiliaires, parmi lesquels on cite : le disque dur, la clé USB, le CDROM, etc.
  - d. En plus, d'autres dispositifs, tels que le lecteur CDROM, les câbles, les nappes, etc.
- B. L'écran : il permet de sortir (afficher) les informations à l'utilisateur.
- C. Le clavier : il permet d'entrer les informations à la machine.
- Un PC peut être accompagné d'un ensemble de périphériques pour faciliter son utilisation et augmenter son efficacité. Il existe : la souris, l'imprimante, le scanner, le graveur, etc.

#### **1.4. Notions de programme et de système d'exploitation**

Actuellement, l'ordinateur ne peut être utilisé qu'au moyen d'un système d'exploitation. Un système d'exploitation est un ensemble de programmes capables de gérer l'ordinateur. Un programme est une suite d'instructions écrites en un langage de programmation pour accomplir une tâche. Le langage de programmation est un langage compréhensible par la machine, par exemple : Pascal, C, etc.



## 2. Introduction à l'algorithmique

### 2.1. Introduction

En fait, un ordinateur n'est qu'une machine capable d'exécuter automatiquement une série d'instructions simples qu'on lui a demandées de faire. L'intérêt de l'ordinateur est sa capacité de manipuler rapidement et sans erreur un grand nombre d'informations : mémoriser des quantités numériques ou alphabétiques, rechercher une quantité mémorisée, comparer ou classer des informations, etc.

Pour résoudre un problème à l'aide d'un ordinateur, il faut :

1. analyser ce problème : définir avec précision les résultats à obtenir, les informations dont on dispose, etc.
2. déterminer les méthodes de résolution : il s'agit de déterminer la suite d'opérations à effectuer pour obtenir, à partir des données, la solution au problème posé. Cette suite d'opérations constitue un algorithme. Parmi tous les algorithmes fournissant la solution, il faudra choisir le plus efficace.
3. formuler l'algorithme définitif : cette étape doit faciliter la résolution par ordinateur du problème en exprimant l'algorithme par un formalisme adéquat (langage de description d'algorithme (LDA), organigramme, arbre programmatique, etc.).
4. traduire l'algorithme dans un langage de programmation adapté.

### 2.2. Notion d'algorithme

#### 2.2.1. Définitions

Le mot algorithme provient du nom du célèbre mathématicien arabe de la première moitié du IX<sup>e</sup> siècle: Mohamed ibn Mousa Al Khawarizmi (780-850), auteur d'un ouvrage décrivant des méthodes de calculs

algébriques, ainsi que d'un autre introduisant les chiffres Arabes et le zéro. Son nom donna au moyen-âge le nom "algorithme" qui devint algorithme avec Lady Ada Lovelace, fille de lord Byron et assistante de Charles Babbage (1792-1871).

Un algorithme est une suite d'opérations élémentaires exécutées dans un ordre donné pour résoudre un problème ou accomplir une tâche. En tant que science, on parle de l'algorithmique.

### **2.2.2. Exemples**

L'algorithme de démarrage d'un moteur d'une voiture représenté par énumération des étapes est le suivant :

1. Ouvrir la porte.
2. Entrer et s'asseoir dans la voiture.
3. Fermer la porte.
4. Tourner la clé de contact.
5. Relâcher la clé.

L'exemple ci-dessus est un exemple simple qui ne contient aucun calcul. Dans un deuxième exemple, on va essayer de résoudre le problème de calcul de la somme de deux nombres. Pour cela, on doit comprendre la nature du problème posé, et préciser les données fournies ("entrées", ou "input" en anglais). Ensuite, on doit préciser les résultats que l'on désire obtenir ("sorties", ou "output" en anglais). Enfin, on va déterminer le processus de transformation des données en résultats. Maintenant, l'algorithme représenté par énumération des étapes peut être écrit comme suit :

1. Prendre connaissance des deux valeurs.
2. Additionner les deux valeurs.
3. Afficher le résultat.

Avant de traduire l'algorithme dans un langage de programmation, il doit être tout d'abord exprimé en un langage de description d'algorithme.

### **2.2.3. Langage de description d'algorithme**

Dans un langage de description d'algorithme, les actions sont généralement décrites par un symbole ou un verbe à l'infinitif choisi pour éviter les confusions. Ce langage est appelé pseudocode ou langage de description d'algorithme (LDA), ou encore langage algorithmique.

L'exemple précédent devient :

```
Algorithme Somme ;  
Variables  
    valeur1, valeur2, som : entier ;  
Début
```

```
Ecire('Entrez la première valeur : ');  
Lire(valeur1);  
Ecire('Entrez la deuxième valeur : ');  
Lire(valeur2);  
som ← valeur1 + valeur2;  
Ecire(som);  
Fin.
```

Un algorithme commence par le mot *Algorithme*, suivi de son nom et un point-virgule. Généralement, le nom de l'algorithme indique sa fonction. Le mot *Variables* précède la liste des variables manipulées dans l'algorithme et leurs types. Les variables du même type sont séparées par des virgules. Deux déclarations différentes sont séparées par un point-virgule. Les opérations de l'algorithme sont prises entre les mots *Début* et *Fin* indiquant le début et la fin de l'algorithme. Ces opérations sont séparées par des points-virgules. Le mot *Lire* permet la lecture à partir du clavier. Le mot *Ecire* permet l'affichage à l'écran. Le symbole  $\leftarrow$  correspond à l'opération d'affectation. Le symbole  $+$  est utilisé pour indiquer l'addition. Un algorithme se termine par un point. D'autres mots et symboles utilisés dans notre langage algorithmique seront découverts dans le reste de ce cours.

#### **2.2.4. Caractéristiques d'un algorithme**

Tout programme fourni à l'ordinateur n'est que la traduction d'un algorithme dans un langage de programmation, mis au point pour résoudre un problème donné. Pour obtenir un bon programme, il faut partir d'un bon algorithme. Il doit, entre autres, posséder les qualités suivantes :

- **Lisible** : clair et facile à comprendre par tous ceux qui le lisent.
- **De haut niveau** : indépendant du langage de programmation et du système d'exploitation.
- **Précis** : chaque élément de l'algorithme ne doit pas porter à confusion. Il est donc important de lever toute ambiguïté.
- **Concis** : être conçu de manière à limiter le nombre d'opérations à effectuer, et la place occupée en mémoire.
- **Structuré** : un algorithme doit être composé de différentes parties facilement identifiables.

Une des meilleures façons de rendre un algorithme clair et compréhensible est d'utiliser une programmation structurée n'utilisant qu'un petit nombre de structures indépendantes du langage de programmation utilisé. Une technique d'élaboration d'un bon algorithme est appelée méthode descendante (top down). Elle consiste à considérer

un problème dans son ensemble, à préciser les données fournies et les résultats à obtenir, puis à décomposer le problème en plusieurs sous-problèmes plus simples qui seront traités séparément, et éventuellement décomposer ces sous-problèmes de manière plus fine.

### **2.2.5. Notions de programme et de langage de programmation**

Un programme est obtenu après la traduction d'un algorithme dans un langage compréhensible par l'ordinateur (un langage de programmation).

Le langage de programmation est un moyen de communiquer avec l'ordinateur. On appelle langages de première génération les langages-machine ou "codes-machine" utilisés initialement (1945). Un programme en "code-machine" est une suite d'instructions élémentaires, composées uniquement de 0 et de 1, exprimant les opérations de base que la machine peut physiquement exécuter : instructions de calcul (addition...) ou de traitement ("et" logique...), instructions d'échange entre la mémoire principale et l'unité de calcul ou entre la mémoire principale et une mémoire externe, ou des instructions de test qui permettent, par exemple, de décider de la prochaine instruction à effectuer. Par exemple, le code machine de l'IBM 370 permettant de charger 293 dans le registre "3" est 01011000 0011 00000000000100100100 (=charger =3 =293). Ce type de code binaire est le seul que la machine puisse directement comprendre, et donc réellement l'exécuter. Tout programme écrit dans un langage évolué devra par conséquent être d'abord traduit en code-machine avant d'être exécuté.

La deuxième génération est caractérisée par l'apparition des langages d'assemblage (1950) où chaque instruction élémentaire est exprimée de façon symbolique. Un programme dit "assembleur" assure la traduction en code exécutable. L'instruction de l'exemple précédent s'écrira : constante X = 293 charger X R3.

La troisième génération débute en 1957 avec le 1<sup>ier</sup> langage dit évolué : FORTRAN (acronyme de mathematical FORMula TRANslating system). Apparaissent ensuite ALGOL (ALGORithmic Language en 1958), COBOL (COMmon Business Oriented Language en 1959), BASIC (Beginner's All-purpose Symbolic Instruction Code en 1965), Pascal (1968), etc. Les concepts employés par ces langages sont beaucoup plus riches et puissants que ceux des générations précédentes, et leur formulation se rapproche du langage mathématique. Il existe deux méthodes pour les rendre exécutables :

- La compilation : consiste à détecter les erreurs lexicales, syntaxiques et sémantiques, ensuite traduire le programme correct en code-



machine produit et optimisé pour l'exécuter par ordinateur. Le programme responsable de cette opération est appelé compilateur.

- L'interprétation : un programme (interpréteur) décode et effectue une à une, et au fur et à mesure les instructions du programme source.

La quatrième génération qui commence au début des années 80 devait mettre l'outil informatique à la portée de tous, en supprimant la nécessité de l'apprentissage d'un langage évolué. Ses concepts fondamentaux sont "convivialité" et "non-procéduralité" (il suffit de "dire" à la machine ce que l'on veut obtenir sans avoir à préciser comment le faire). Cet aspect a été rencontré avec les langages de type Visual qui prennent en charge l'élaboration de l'interface graphique.

La mise en œuvre de l'algorithme consiste à l'exprimer en un langage de programmation (Pascal, C...). Les algorithmes de ce cours sont traduits en langage Pascal. Le programme correspondant à l'algorithme précédent est le suivant :

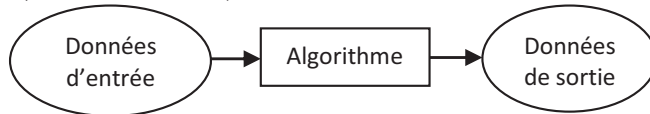
```
program Somme ;  
var  
    valeur1, valeur2, som : integer ;  
begin  
    write('Entrez la première valeur : ');  
    read(valeur1) ;  
    write('Entrez la deuxième valeur : ');  
    read(valeur2) ;  
    som := valeur1 + valeur2 ;  
    write(som) ;  
end.
```

Un programme Pascal commence par le mot `program` suivi de son nom et un point-virgule (;). Le mot `var` précède la liste des variables manipulées dans le programme et leurs types. Les instructions du programme sont séparées par des points-virgules (;). Elles sont prises entre les mots `begin` et `end` indiquant le début et la fin du programme. Le mot `read` permet la lecture à partir du clavier. Le mot `write` permet l'affichage à l'écran. Le symbole `:=` correspond à l'opération d'affectation. Le symbole `+` est utilisé pour indiquer l'addition. Un programme Pascal se termine par un point (.). D'autres mots et symboles utilisés dans le langage Pascal seront découverts dans le reste de ce cours.

## Chapitre 2 : Les algorithmes séquentiels simples

### 1. Principe général

Un algorithme peut être vu comme une boîte noire qui reçoit en entrée des données. Ces données vont subir à des opérations pour générer des résultats (données de sortie).



Donc un algorithme contient deux parties essentielles :

- Partie Données (Déclaration) : correspondent aux données d'entrée et de sortie (variables et constantes) .
- Partie Traitement (Processus de calcul, Code ou Suite d'opérations) : correspond à la partie traitement.

### 2. Variables et constantes

Les données sont les objets manipulés dans l'algorithme. Dans un algorithme, toute donnée utilisée doit être déclarée. Les données peuvent être des variables ou des constantes.

**Les variables :** comme son nom l'indique, une variable correspond à un objet dont la valeur peut varier au cours de déroulement de l'algorithme. Une variable est caractérisée par :

- Le nom : appelé aussi identificateur. Il doit être aussi explicite que possible, indiquant le rôle de la variable dans l'algorithme. La règle est de construire le nom en n'utilisant que des caractères alphabétiques, des chiffres et le symbole de soulignement '\_'. Un identificateur doit toujours commencer par une lettre, et ne doit jamais contenir des espaces.
- Le type : indique les valeurs qui peuvent être prises par une variable.
- La valeur : indique la grandeur prise par la variable à un moment donné.

Sur le plan technique, on peut voir la mémoire centrale comme un ensemble de cases numérotées. Le numéro est l'adresse de la case. Chaque case étant composée d'un ou plusieurs octets. Une variable correspond à une case mémoire tel que : le nom de la variable est l'adresse de la case mémoire ; le type indique le nombre d'octets qui composent la case ; la valeur représente le contenu de la case.

**Les constantes :** une constante est un cas particulier de la variable. Il s'agit d'une variable dont la valeur est inchangeable dans l'algorithme tout entier.

**Exemple :**

```
Algorithme calcul_surface ;  
Constantes  
  PI=3,14 ;  
Variables  
  rayon, surface : réel ;  
...
```

**En Pascal :**

```
program  
const  
  PI=3,14 ;  
var  
  rayon, surface : real ;  
...
```

**3. Types**

Les variables d'un algorithme contiennent les informations nécessaires à son déroulement. Chaque variable a un nom, une valeur et un type. Ce dernier correspond au genre ou la nature de l'information que l'on souhaite utiliser. Il existe des types simples et des types structurés.

**Les types simples :** sont des types dont les valeurs sont primitives, élémentaires non décomposables. Ces valeurs forment un ensemble ordonné permettant d'effectuer au moins les opérations d'affectation et de comparaison. A leurs tours, les types simples peuvent être classés en deux catégories :

1. Types numériques :
  - Entier : pour manipuler des entiers, par exemple : 12, -55, etc.
  - Réel : pour manipuler des nombres réels, par exemple : 12.5, -2.09, etc.
2. Types symboliques :
  - Booléen : pour manipuler des valeurs booléennes : VRAI et FAUX.
  - Caractère : pour manipuler des données alphanumériques, symboliques, ponctuation, etc. contenant un seul caractère, par exemple : 'a', '?', '3', ','; etc.

**Les types structurés :** c'est tout type dont la définition fait référence à d'autres types, c.-à-d. qu'ils sont définis à base de types simples. Ces types sont aussi dits types complexes, parmi lesquels on cite : le type tableau, chaîne de caractères, enregistrement, ensemble, etc. qui seront vus ultérieurement.

**La déclaration d'une variable :** ce n'est rien que l'association d'un nom avec un type, permettant de mémoriser une valeur de ce type. Chaque type donné peut être manipulé par un ensemble d'opérations.

**Remarque :**

Sur le plan technique, la déclaration d'une variable se traduit par une réservation d'un espace mémoire. L'identificateur de la variable représente l'adresse de l'espace réservé. La taille de l'espace réservé dépend du type de la variable.

### 3.1. Le type entier

Les opérateurs utilisés pour manipuler les entiers sont :

- Les opérateurs arithmétiques classiques : + (addition), - (soustraction), \* (produit).
- La division entière, notée  $\div$ , tel que  $n \div p$  donne la partie entière du quotient de la division entière de  $n$  par  $p$ .
- Le modulo, noté MOD, tel que  $n \text{ MOD } p$  donne le reste de la division entière de  $n$  par  $p$ .
- Les opérateurs de comparaison classiques : <, >, >=, <=, = et <> (différent).

### 3.2. Le type réel

Les opérateurs utilisés pour la manipulation des réels sont :

- Les opérateurs arithmétiques classiques : + (addition), - (soustraction), \* (produit), / (division)
- Les opérateurs de comparaison classiques : <, >, >=, <=, = et <>.

### 3.3. Le type booléen

Il s'agit du domaine dont les seules valeurs sont VRAI et FAUX. Les opérateurs utilisés pour la manipulation des booléens sont les connecteurs logiques : ET (pour le *et* logique), OU (pour le *ou* logique), NON (pour le *non* logique). Les équations logiques sont :

NON	
VRAI	FAUX
FAUX	VRAI

ET	VRAI	FAUX
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

OU	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

### 3.4. Le type caractère

Il s'agit du domaine constitué des caractères alphanumériques, symboliques, etc. Une variable de ce type ne peut contenir qu'un seul et unique caractère. Les opérateurs de manipulation des caractères sont les opérateurs de comparaison : >, <, =, etc.

Les types cités ci-dessus sont appelés "types élémentaires" ou "types de base" ou encore "types simples". On peut avoir besoin de regrouper plusieurs types de données ensemble pour avoir un objet complexe

adapté au problème posé. Ces types composés ou structurés seront vus ultérieurement.

#### 4. Opérations de base

Dans un algorithme, le traitement d'information est organisé en une suite d'opérations (appelées instructions pour un programme) ordonnées pour résoudre un problème. Les opérations d'un algorithme peuvent être classées en deux catégories : opérations de base et structures de contrôle. Les opérations de base permettent une exécution séquentielle dans un algorithme. Les structures de contrôle permettent le saut dans un algorithme. Les opérations de base sont l'affectation et les entrées/sorties.

##### 4.1. L'affectation

L'opération la plus importante en algorithmique est l'affectation (assignation) qui se note  $\leftarrow$ , et qui consiste à attribuer ou affecter à une variable, une valeur appartenant à son domaine de définition (type). La valeur affectée est souvent le résultat d'un calcul d'une expression arithmétique ou une expression logique.

**Les expressions arithmétiques :** elles servent à effectuer des calculs dont les résultats sont de type numérique. Ces expressions comportent :

- Des opérandes qui peuvent être des variables et/ou des constantes.
- Des opérateurs : + (addition), - (soustraction), \* (multiplication), / (division)...
- Des fonctions mathématiques :  $\ln(x)$ ,  $\sin(x)$ ,  $\arctg(x)$ , etc.

Ces expressions arithmétiques sont des formules mathématiques symbolisant des opérations sur des variables et/ou des constantes numériques. Les résultats d'évaluation de ces expressions sont la plus part du temps rangés dans des variables. On dit qu'il s'agit d'une affectation du résultat d'une expression à une variable.

##### Exemple :

```
Algorithme calculs ;  
Variables X, Y, Z : entier ;  
Début  
  X  $\leftarrow$  4 ;  
  Y  $\leftarrow$  X * 2 ;  
  Z  $\leftarrow$  Z - 6 ;  
Fin.
```

##### En Pascal :

```
program calculs ;  
var X, Y, Z : integer ;  
begin
```

```
X := 4 ;  
Y := X * 2 ;  
Z := Z - 6 ;
```

end.

L'exemple précédent est lu comme suit : la variable X reçoit la valeur 4, la variable Y reçoit la valeur de X multipliée par 2, et enfin la variable Z reçoit la valeur courante (actuelle) de Z moins 6.

### Remarques :

- La partie gauche d'une affectation contient une seule variable. Cependant, la partie droite peut contenir un ensemble de valeurs.
- Tandis que, dans la partie gauche on parle de variable, dans la partie droite on parle de valeurs. Ceci est valable même si on utilise le même identificateur dans les deux côtés, comme c'est le cas pour la variable X dans la troisième affectation.
- La partie droite et la partie gauche d'une affectation doivent être de même type, mais on peut avoir comme même une variable réel qui reçoit une valeur entière, ou bien elle reçoit une expression arithmétique comportant des valeurs entières.

Dans une expression arithmétique comportant plusieurs opérations, les règles de priorité entre opérations, par ordre de priorité décroissant, sont :

- Les fonctions mathématiques.
- La multiplication et la division.
- L'addition et la soustraction.

On pourra utiliser des parenthèses dans une expression arithmétique pour clarifier, ou pour changer l'ordre de priorité. Par exemple,  $Y \leftarrow X * (2 + Z)$  permet d'effectuer l'addition avant la multiplication.

**Les expressions logiques :** on peut aussi manipuler des expressions logiques ou booléennes pour effectuer des calculs dont les résultats sont de type booléen, en utilisant les opérateurs ET, OU et NON. Par exemple :  $H \leftarrow (10 > 5) \text{ ET } (2 < 3)$ , cette opération permet d'affecter la valeur VRAI à H.

Il faut différencier entre une affectation informatique et une égalité mathématique. L'affectation  $Z \leftarrow Z - 6$  ne s'agit pas de résoudre une équation avec une variable inconnue Z. D'ailleurs en math, c'est une équation sans solution, mais en informatique, c'est une opération correcte.

## 4.2. Les entrées/sorties

Les échanges d'informations entre l'utilisateur et la machine sont appelés "opérations d'entrée-sortie". Les opérations d'entrée-sortie sont :

- Lire(n) : qui récupère la valeur tapée au clavier, et l'affecte à l'espace mémoire désigné par la variable n.
- Ecrire(n) : qui récupère la valeur située à l'espace mémoire désigné par la variable n, et affiche cette valeur à l'écran. L'opération d'écriture permet aussi d'afficher une phrase sous forme de chaîne de caractères qui est une donnée alphanumérique.

**Remarque :** Le langage Pascal permet de lire et écrire une variable de type entier, réel, caractère ou chaîne de caractères. Une variable booléenne peut être seulement affichée.

**Une chaîne de caractères :** est une séquence de plusieurs caractères permettant de représenter des mots ou des phrases. Une chaîne de caractères doit être mise entre deux guillemets simples pour la distinguer d'un identificateur de variable, par exemple 'bonjour'.

Dans un algorithme de calcul de la surface d'un cercle, il faut saisir le rayon du cercle (entrée d'information ou lecture en utilisant l'opération Lire). Ensuite, on calcule la surface par la formule  $\pi * \text{rayon}^2$ . Enfin, l'affichage du résultat (sortie ou écriture en utilisant l'opération Ecrire). Il est fortement conseillé de précéder chaque opération de lecture par une opération d'écriture contenant un libellé indiquant à l'utilisateur ce qu'il doit taper (Voir l'exemple de la somme).

## 5. Construction d'un algorithme simple

Construire un algorithme consiste par convention à :

1. Lui donner un nom. Il est préférable que le nom de l'algorithme indique son travail.
2. Identifier les constantes : les nommer et les initialiser.
3. Identifier les variables : les nommer et indiquer leurs types. Il est préférable de choisir des identificateurs significatifs pour les constantes et les variables qui indiquent leurs rôles dans l'algorithme. Par exemple, pour une variable qui va contenir le maximum d'une liste, on choisit *max* comme identificateur.
4. Ecrire le corps de l'algorithme encadré par les mots Début et Fin qui indiquent le début et la fin de l'algorithme.

Le corps de l'algorithme est constitué d'une séquence d'opérations séparées par des points-virgules. On écrit en général une opération par ligne. On parle d'algorithme séquentiel. Le corps de l'algorithme peut contenir des opérations de lecture, d'écriture, d'affectation, et d'autres qui seront vues plus tard. Pour éclaircir l'algorithme, son corps peut contenir des commentaires mis entre (\* et \*), ou bien on peut faire des décalages de ligne pour avoir une structure en blocs facile à comprendre, et on dit qu'on *indente le texte de l'algorithme*.

**Exemple :**

L'algorithme de calcul de la surface d'un cercle représenté par énumération des étapes est le suivant :

1. Saisir le rayon du cercle (entrée d'information ou lecture).
2. Affecter à une variable nommée, par exemple, *surface*, le résultat de l'expression :  $\pi * (\text{rayon})^2$ .
3. Afficher le résultat (sortie ou écriture).

En utilisant un langage algorithmique, on obtient :

```



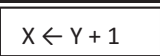

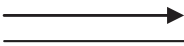
Algorithme Calcul_Surface ;
/* Algorithme de calcul de la surface d'un cercle */
Constantes
  PI = 3,14159 ;
Variables
  rayon, surface : réel ;
Début
  Ecrire('Entrez la valeur du rayon :') ;
  Lire(rayon) ;
  surface ← PI * rayon * rayon ;
  Ecrire(surface) ;
Fin.
```

Dans certains cas, on pourra utiliser des variables intermédiaires qui seront utilisées dans d'autres opérations pour faciliter les calculs.

**6. Représentation d'un algorithme par un organigramme**

La première forme pour représenter un algorithme est l'énumération de ses étapes. Ensuite, il va être écrit en un langage algorithmique. Un algorithme peut être représenté aussi par un organigramme facilitant sa compréhension.

Un organigramme est une représentation graphique de l'ossature de la solution d'un problème. Pour cela, on utilise des symboles géométriques.

	Début/Fin	Début ou fin de l'algorithme.
	Lire/Ecrire	Les opérations d'entrée/sortie.
	$X \leftarrow Y + 1$	Un traitement (une affectation par exemple).
	Oui / Condition / Non	Un test d'une condition pour une décision ou une sélection.
		Liaisons entre les différents points, et indiquent aussi l'ordonnancement des opérations.

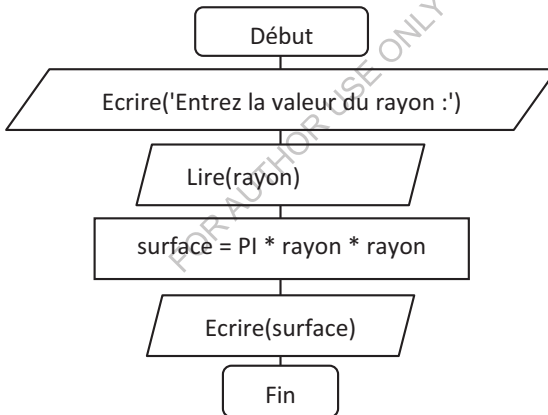


La représentation d'un algorithme par un organigramme possède plusieurs avantages :

1. Il est plus facile de dessiner un organigramme que d'écrire un programme directement.
2. Les organigrammes représentent une aide précieuse dans le développement d'un programme lui-même.
3. Les organigrammes sont plus faciles à comprendre que les programmes eux mêmes.
4. Les organigrammes sont indépendants du langage de programmation.

Pour la validation de notre organigramme, il faut tester si tous les cas possibles sont prévus. Il faut bien choisir les données pour la vérification. Ces données doivent être simples pour faire des calculs à la main. L'ensemble des données doit permettre de parcourir toutes les branches de l'organigramme.

L'algorithme de calcul de la surface peut être représenté par l'organigramme suivant :



Pour la vérification, on choisit rayon = 2, donc surface =  $3.14 \times 4 = 12.56$ .

## 7. Traduction en langage Pascal

### 7.1. Exemple

La traduction de l'algorithme de calcul de la surface en un programme Pascal est la suivante :

```
program Calcul_Surface (input,output);
const
  pi = 3.14159 ;
var
  rayon, surface : real ;
```

```
begin
  writeln('Entrez la valeur du rayon :');
  readln(rayon);
  surface := pi * sqr(rayon); { sqr c'est le carré }
  writeln(surface);
end.
```

## 7.2. Règles de base

A partir de l'exemple précédent, nous constatons que :

1. Un programme Pascal se compose de trois parties : un en-tête, caractérisé par le mot `program`. Une section déclarative, appelée aussi partie définition, introduite ici par le mot `var`, et sert à décrire les objets manipulés dans le corps du programme. Une section instructions ou corps du programme, délimitée par les mots `begin` et `end`. Le programme se termine par un point.
2. L'en tête (facultatif) sert à donner un nom au programme selon la forme : `program identificateur`. Les deux mots clés `input` et `output` placés entre parenthèses indiquent que le programme va avoir des entrées et des sorties.
3. Un identificateur en Pascal doit débuter par une lettre suivie d'un nombre quelconque de lettres, de chiffres ou de "\_" (caractère souligné). Les identificateurs ne peuvent contenir d'espacement (caractère "blanc") ou de caractères, tels que %, ?, \*, ., -..., mais peuvent être aussi longs que l'on veut.
4. Les variables doivent faire l'objet d'une déclaration de type de la forme : `var liste_des_variables : type`. On peut définir des constantes par le mot clé `const`.
5. Des points-virgules sont obligatoires pour séparer les trois parties, et pour séparer les instructions.
6. En Pascal, il existe deux types d'instructions : les instructions simples, telles que l'affectation, la lecture et l'écriture ; et les instructions de contrôle qui seront vues ultérieurement.
7. Les instructions de lecture et d'écriture se traduisent respectivement par `read` et `write` (ou `readln` et `writeln`) suivis d'une liste de variables ou d'expressions placées entre parenthèses et séparées par des virgules. L'ajout de `ln` après `write` (`writeln`) et `read` (`readln`) force le passage à la ligne lors de l'affichage suivant à l'écran.
8. L'affectation (l'assignation) se représente par `:=`.
9. Les opérateurs arithmétiques sont identiques à ceux du langage de description d'algorithme (LDA). En effet, outre les quatre opérations `+`, `-`, `*` et `/`, Pascal utilise deux opérateurs

supplémentaires : DIV fournissant la partie entière du quotient de deux nombres entiers, et MOD fournissant le reste de la division de deux nombres entiers. Ainsi,  $13 / 5$  fournit la valeur 2.6,  $13 \text{ DIV } 5$  fournit 2, et  $13 \text{ MOD } 5$  fournit 3.

10. Les mots `program`, `var`, `begin`, `end`, `div`, `mod`, `integer`, `read`, `write`, etc. ont un sens précis dans le langage : ce sont des mots réservés ou mots clés qui ne peuvent être choisis comme identificateurs par le programmeur.
11. Les mots du langage et les identificateurs doivent être séparés les uns des autres par un ou plusieurs blancs.
12. Il est à noter que le langage Pascal ne différencie pas entre la majuscule et la minuscule. `WRITELN` ou `writeln`, tous les deux sont corrects.
13. Un programme Pascal peut contenir éventuellement des commentaires mis entre `(*` et `*)`, ou entre `{` et `}`, pour expliquer le fonctionnement du programme aux autres programmeurs.

### 7.3. Lecture

Les lectures sont symbolisées par le mot `readln`. C'est la procédure `readln` qui transfère les nombres ou les chaînes de caractères du clavier vers la mémoire centrale. Ceux-ci doivent respecter la forme des constantes en Pascal, et doivent être séparés par un blanc au moins lors de la saisie au clavier. Le type de la constante donnée et celui de la variable d'accueil doivent se correspondre selon la règle d'assignation.

Pascal admet aussi la procédure `read` (qui a le même effet que `readln`, mais sans passage à la ligne). On note qu'il s'est révélé à l'usage, que celle-ci (`read`) était parfois source de problème, et il est préférable de l'éviter et d'utiliser `readln`.

### 7.4. Ecriture

Les écritures se font à l'aide de la procédure `write`. Les valeurs seront affichées à l'écran sur une seule ligne, parfois sans espacement entre elles. Par exemple, si une variable entière `X` contient la valeur 123, alors `write(X)` affiche 123. Pour améliorer la lisibilité, on peut utiliser la procédure `writeln` qui force le passage à la ligne suivante pour le prochain affichage.

Pour les formats d'édition qui précisent le nombre de caractères à utiliser pour afficher chacun des résultats, on a :

- `write(valeur_entière : n)` affiche la valeur entière sur `n` positions (insertion d'espacement à gauche du nombre s'il y a trop peu de chiffres, et ajustement automatique si `n` est insuffisant). Par exemple,

- si la variable entière X contient la valeur 123, alors `write(X:5)` affiche `^^123` (^ symbole d'espacement) et `write(X:2)` affiche 123.
- `write(valeur_réelle)` affiche le nombre en notation scientifique (`x.xxxxEx` précédé d'un espacement). Par exemple, si la variable réelle X contient la valeur 123.4567, alors `write(X)` affiche `^1.23456E+2` (^ symbole d'espacement).
  - `write(valeur_réelle : n)` affiche le nombre en notation scientifique sur n positions. Par exemple, si la variable réelle X contient la valeur 123.4567, alors `write(X:7)` affiche `^1.2E+2` (^ symbole d'espacement) et `write(x:2)` affiche 1.2E+2.
  - `write(valeur_réelle : n1 : n2)` affiche le nombre sur n1 positions avec n2 décimales (avec ajustement). Par exemple, si la variable réelle X contient la valeur 123.4567, alors `write(X:8:2)` affiche `^^123.46` (^ symbole d'espacement).
  - `write(chaine : n)` affiche la chaîne de caractères sur n positions (insertion d'espacement à gauche de la chaîne s'il y a trop peu de caractères, et ajustement automatique si n est insuffisant). Par exemple, si la variable X de type chaîne de caractères contient la valeur 'AZERTY', alors `write(X)` affiche AZERTY, `write(X:8)` affiche `^^AZERTY`, et `write(X:3)` affiche AZERTY (^ symbole d'espacement).

### 7.5. Manipulation des nombres

Si la mathématique distingue plusieurs types de nombres directement manipulables par les langages informatiques, Pascal n'en reconnaît que deux : les types entier et réel.

Le langage Pascal définit les entiers en 5 types pour mieux adapter le type aux valeurs que peut prendre une variable, et ce pour optimiser l'espace mémoire occupé.

Type	Bornes	Occupation en mémoire
SHORTINT	de -128 à +127	1 octet
BYTE	de 0 à 255	1 octet
INTEGER	de -32768 à +32767	2 octets
WORD	de 0 à 65535	2 octets
LONGINT	de -2147483648 à 2147483647	4 octets

Aussi, le langage Pascal définit les réels en 4 types pour mieux adapter le type aux valeurs que peut prendre une variable, et ce pour optimiser l'occupation de la mémoire.

Type	Bornes	Occupation en mémoire
SINGLE	$1,5 \cdot 10^{-45}$ à $3,4 \cdot 10^{38}$	4 octets
REAL	$2,9 \cdot 10^{-39}$ à $1,7 \cdot 10^{38}$	6 octets
DOUBLE	$5,0 \cdot 10^{-324}$ à $1,7 \cdot 10^{308}$	8 octets
EXTENDED	$3,4 \cdot 10^{-4932}$ à $1,1 \cdot 10^{4932}$	10 octets

Il est à noter que :

- Un nombre réel se trouvant entre les deux bornes peut prendre le signe positif (+) ou négatif (-).
- Dans une affectation, le type de l'expression doit correspondre au type de la variable de destination. Cette règle admet une seule exception : une variable réelle peut recevoir une valeur entière.
- Lors de l'évaluation des expressions arithmétiques, le langage Pascal respecte la même convention de priorité que l'arithmétique : les multiplications et les divisions (opérateurs \*, /, DIV et MOD) sont effectuées en premier lieu, puis les additions et les soustractions (opérateurs + et -) ; lorsqu'une expression contient plusieurs opérateurs de même priorité, les opérations sont effectuées de gauche à droite. Pour modifier cet ordre, il suffit d'introduire des parenthèses. Par exemple, `WRITELN(1/2*3)` n'affichera pas la valeur de 1/6, mais plutôt de 3/2, car la division se fera avant la multiplication. Les priorités données aux opérateurs, de la plus élevée à la plus basse, sont :

NOT

\* / DIV MOD AND

+ - OR

= < > <= >= IN

- Maintenant, pour le type des opérandes et du résultat, les opérateurs +, - et \* peuvent agir sur des opérandes réels ou entiers, et le résultat est réel, sauf si les deux opérandes sont des entiers. L'opérateur / peut agir sur des entiers et des réels, mais le résultat est toujours réel. Les opérateurs DIV et MOD ne peuvent être utilisés qu'avec des opérandes entiers, et fournissent un résultat entier.
- Le type d'une variable est défini dans la partie déclarative du programme. Pour les constantes numériques, c'est la forme d'écriture qui détermine le type d'une constante. Ainsi, 50 est une constante entière, tandis que 3.1416 et 50.0 sont des constantes réelles, car elles contiennent une partie fractionnaire.
- Les constantes entières ne peuvent contenir que des chiffres décimaux (0 à 9) précédés éventuellement d'un signe + ou -. Les constantes réelles doivent contenir en plus : soit une partie fractionnaire d'au moins un chiffre, séparée de la partie entière par un point, par exemple +1.2, -56, 0.01, 0.0 ; soit une partie exposant sous forme d'une constante entière, précédée par un E indiquant la puissance de 10 par laquelle il faut multiplier la valeur qui précède la lettre E, par exemple  $1E4$  vaut  $1 * 10^4 = 10000.0$ ,  $6E-2$  vaut  $6 * 10^{-2} =$

0.06 ; soit les deux, par exemple  $3.14E+4$  vaut  $3.14 * 10^4 = 31400.0$ . Dans ce dernier cas, et s'il n'y a qu'un seul chiffre non nul dans la partie entière, on parle de notation scientifique.

- Les fonctions mathématiques sont détaillées dans le tableau suivant :

Notation mathématique	Fonction Pascal	Type de x	Type du résultat	Signification
$ x $	ABS(x)	Entier ou réel	Type de x	Valeur absolue de x
$x^2$	SQR(x)	Entier ou réel	Type de x	Carré de x
$x^{1/2}$	SQRT(x)	Entier ou réel	Réel	Racine carré de x
$\sin x$	SIN(x)	Entier ou réel	Réel	$\sin$ de x (x en radians)
$\cos x$	COS(x)	Entier ou réel	Réel	$\cos$ de x (x en radians)
$\arctg x$	ARCTAN(x)	Entier ou réel	Réel	Angle (en radians) dont la tangente vaut x
$e^x$	EXP(x)	Réel	Réel	Exponentielle de x
$\ln x$	LN(x)	Réel	Réel	Logarithme népérien de x
$[x]$	TRUNC(x)	Réel	Entier	Partie entière de x
$[x]$	INT(x)	Réel	Réel	Partie entière de x
arrondi de x	ROUND(x)	Réel	Entier	Entier le plus proche de x
décimal de x	FRAC(x)	Réel	Réel	Partie décimale de x

- Enfin, on notera l'absence des fonctions  $\lg x$  et  $x^y$  qui se traduiront, en employant les formules mathématiques adéquates, respectivement par  $\text{SIN}(x)/\text{COS}(x)$  et  $\text{EXP}(y * \text{LN}(x))$ .

## 7.6. Manipulation des caractères

En informatique, une variable de type caractère correspond à un seul symbole alphanumérique, et possède un code universel : le code ASCII (American Standard Code for Information Interchange). A titre d'exemple, les lettres majuscules de 'A' à 'Z' sont codées dans l'ordre par les codes 65 à 90.

Le type caractère est réservé aux variables contenant un seul caractère (lettre, symbole, ponctuation, etc.) qui s'écrivent sous la forme : 'A', 'B', '3', ' ', '!', etc., et il est possible d'en déterminer le successeur/prédécesseur/position dans la liste des codes ASCII. Ainsi, le successeur de 'B' est 'C', son prédécesseur est 'A', et son code ASCII est 66. Le type caractère est déclaré en utilisant le mot clé CHAR, par exemple `var C : CHAR.`

Les fonctions avec les quelles on peut manipuler des caractères sont détaillées dans le tableau suivant :

Fonction Pascal	Type du résultat	Signification
CHR(x)	caractère	Caractère correspondant au code ASCII spécifié entre parenthèses
ORD('A')	entier	Le code ASCII du caractère spécifié entre parenthèses
SUCC('A')	caractère	Le successeur du caractère spécifié entre parenthèses
PRED('B')	caractère	Le prédécesseur du caractère spécifié entre parenthèses

## 7.7. Manipulation des booléens

Les variables booléennes, déclarées en Pascal par le mot clé BOOLEAN, peuvent prendre soit la valeur TRUE (VRAI), soit la valeur FALSE (FAUX).

Sur les booléens, on peut effectuer les opérations suivantes : AND, OR et NOT. Ces opérations nécessitent des arguments booléens.

Les opérateurs de comparaison sont : <, >, >=, <=, = et <>. Ces opérateurs comparent tous les éléments de types simples (les deux arguments doivent être de même type, sauf les entiers et les réels qui peuvent être comparés entre eux), et renvoient une valeur booléenne. Les caractères sont comparés suivant l'ordre du code ASCII.

## 8. Exercices corrigés

### 8.1. Exercices

#### Exercice 1 :

Que vaut l'expression arithmétique suivante ?

$$5 + 2 / (3 - 1 / 1) * 6 - 9.$$

Que vaut l'expression logique ci-dessous ? Justifiez votre réponse.

$$((0 < 1) \text{ OU FAUX } ) \text{ ET } (\text{VRAI OU } (9 - 3 = 3)).$$

#### Exercice 2 :

Qu'affiche l'algorithme suivant ?

Algorithme calcul\_double ;

Variables

val, dbl : entier ;

Début

val  $\leftarrow$  231 ;

dbl  $\leftarrow$  val \* 2 ;

Ecrire(val) ;

Ecrire(dbl) ;

Fin.

**Exercice 3 :**

Ecrire un algorithme permettant de lire 2 nombres a et b, de calculer et d'afficher leur moyenne. Traduire l'algorithme en Pascal. Ensuite, représenter l'algorithme par un organigramme. Déroulez l'algorithme pour a=7, b=19.

**Exercice 4 :**

Ecrire un algorithme permettant de saisir trois nombres, de calculer la somme, le produit et la moyenne, puis de les afficher. Traduire l'algorithme en Pascal.

**Exercice 5 :**

Ecrire un algorithme qui demande un nombre à l'utilisateur, puis calcule et affiche le carré, le double et le triple de ce nombre. Traduire l'algorithme en Pascal.

**Exercice 6 :**

Ecrire un algorithme qui lit le prix unitaire HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant. Faire en sorte que des libellés apparaissent clairement. Traduire l'algorithme en Pascal.

**Exercice 7 :**

Ecrire un algorithme permettant de calculer le périmètre et la surface d'un rectangle. Traduire l'algorithme en Pascal.

**Exercice 8 :**

Ecrire un algorithme permettant de lire le rayon R d'une sphère, de calculer et d'afficher son aire =  $4 \pi R^2$ , et son volume =  $\frac{4}{3} \pi R^3$ . Traduire l'algorithme en Pascal.

**Exercice 9 :**

Ecrire un algorithme permettant de lire au clavier les longueurs des 3 côtés a, b et c d'un triangle. Calculer et afficher le périmètre et l'aire du triangle. Périmètre =  $p = a + b + c$ , et aire =  $\left(\frac{p}{2}-a\right)\left(\frac{p}{2}-b\right)\left(\frac{p}{2}-c\right)^{1/2}$ . Traduire l'algorithme en Pascal.

**Exercice 10 :**

Ecrire un algorithme permettant de lire au clavier le rayon R d'un cercle, et un angle a (en degré(s)). Calculer et afficher l'aire du secteur circulaire =  $\pi R^2 a / 360$ . Traduire l'algorithme en Pascal.

**Exercice 11 :**

Ecrire un algorithme permettant de saisir deux nombres, de les permuter puis de les afficher. Traduire l'algorithme en Pascal. Notons que la permutation de deux variables consiste à échanger (intervertir) leurs valeurs.



**Exercice 12 :**

Ecrire un algorithme qui exprime un nombre de secondes sous forme d'heures, minutes et secondes. La seule donnée est le nombre total de secondes que nous appellerons nsec. Les résultats consistent en 3 nombres h, m et s. Traduire l'algorithme en Pascal.

**8.2. Corrigés**

**Solution 1 :**

$5 + 2 / (3 - 1 / 1) * 6 - 9$  vaut 2.

$((0 < 1) \text{ OU FAUX }) \text{ ET } (\text{VRAI OU } (9 - 3 = 3))$  vaut VRAI.

Expression	$(0 < 1)$	$((0 < 1) \text{ OU FAUX})$	$(9 - 3 = 3)$	(VRAI OU $(9 - 3 = 3)$ )	$((0 < 1) \text{ OU FAUX })$ ET $(\text{VRAI OU } (9 - 3 = 3))$
Valeur	VRAI	VRAI	FAUX	VRAI	VRAI

**Solution 2 :**

Les valeurs suivantes seront affichées à l'écran :

231

462

**Solution 3 :**

Algorithme Moyenne ;

Variables

A,b : entier ;

Moy : réel ;

Début

Ecrire('Introduisez a :') ;

Lire(a) ;

Ecrire('Introduisez b :') ;

Lire(b) ;

$\text{moy} \leftarrow (a + b) / 2$  ;

Ecrire('Moyenne = ', moy) ;

Fin.

Le programme Pascal :

program Moyenne ;

var

A,b : integer ;

Moy : real ;

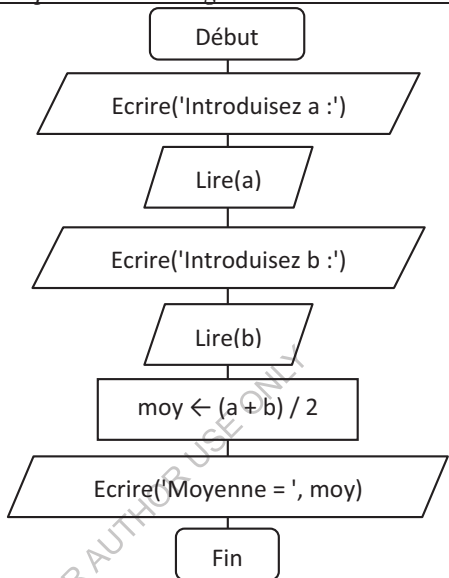
begin

writeln('Introduisez a :') ;

readln(a) ;

```
writeln('Introduisez b :');  
readln(b);  
moy := (a + b) / 2;  
writeln('Moyenne = ', moy);  
end.
```

L'organigramme représentant l'algorithme est le suivant :



Pour le déroulement de l'algorithme dans le cas de (a=7, b=19), on va noter les opérations comme suit :

Opération	Notation
Ecrire('Introduisez a :')	1
Lire(a)	2
Ecrire('Introduisez b :')	3
Lire(b)	4
$moy \leftarrow (a + b) / 2$	5
Ecrire('Moyenne = ', moy)	6

Le tableau suivant correspond au schéma d'évolution d'état des variables, opération par opération :

Variable Opération	a	b	moy	Affichage
1				Introduisez a :
2	7			
3	7			Introduisez b :

4	7	19		
5	7	19	13	
6	7	19	13	Moyenne = 13

**Remarque :** Quand on vous demande de dérouler l’algorithme ou de le simuler ou encore de donner sa trace (il s’agit de la même chose), vous devez décrire le changement des variables (sur le plan technique l’état de la mémoire) au cours d’exécution des opérations de l’algorithme dans un tableau appelé tableau de situation, comme déjà vu dans l’exemple précédent.

**Solution 4 :**

Algorithme calculs ;

Variables

somme, produit, moyenne, nb1, nb2, nb3 : réel ;

Début

Ecrire('Entrez vos trois nombres :') ;

Lire(nb1, nb2, nb3) ;

somme  $\leftarrow$  nb1 + nb2 + nb3 ;

produit  $\leftarrow$  nb1 \* nb2 \* nb3 ;

moyenne  $\leftarrow$  somme / 3 ;

Ecrire('La somme de ces trois nombres est : ', somme) ;

Ecrire('Le produit de ces trois nombres est : ', produit) ;

Ecrire('La moyenne de ces trois nombres est : ', moyenne) ;

Fin.

On peut ne pas utiliser les variables intermédiaires *somme*, *produit* et *moyenne*, et on met directement :

Ecrire('La somme de ces trois nombres est : ', nb1 + nb2 + nb3) ;

Ecrire('Le produit de ces trois nombres est : ', nb1 \* nb2 \* nb3) ;

Ecrire('La moyenne de ces trois nombres est : ', (nb1 + nb2 + nb3)/3) ;

C’est une question de style : dans le premier cas, on favorise la lisibilité de l’algorithme, dans le deuxième, on favorise l’économie de l’espace mémoire.

Le programme Pascal :

program calculs ;

var

somme, produit, moyenne, nb1, nb2, nb3 : real ;

begin

writeln('Entrez vos trois nombres :') ;

readln(nb1, nb2, nb3) ;

somme := nb1 + nb2 + nb3 ;

```
produit := nb1 * nb2 * nb3 ;
moyenne := somme / 3 ;
writeln('La somme de ces trois nombres est : ', somme) ;
writeln('Le produit de ces trois nombres est : ', produit) ;
writeln('La moyenne de ces trois nombres est : ', moyenne) ;
end.
```

**Solution 5 :**

Algorithme car\_dou\_trip ;

Variables

nb, carr, doub, trip : entier ;

Début

Ecrire('Entrez un nombre :') ;

Lire(nb) ;

carr  $\leftarrow$  nb \* nb ;

doub  $\leftarrow$  2 \* nb ;

trip  $\leftarrow$  3 \* nb ;

Ecrire('Son carré est : ', carr) ;

Ecrire('Son double est : ', doub) ;

Ecrire('Son triple est : ', trip) ;

Fin.

Le programme Pascal :

```
program car_dou_trip ;
```

```
var
```

```
nb, carr, doub, trip : integer ;
```

```
begin
```

```
writeln('Entrez un nombre :') ;
```

```
readln(nb) ;
```

```
carr := nb * nb ;
```

```
doub := 2 * nb ;
```

```
trip := 3 * nb ;
```

```
writeln('Son carré est : ', carr) ;
```

```
writeln('Son double est : ', doub) ;
```

```
writeln('Son triple est : ', trip) ;
```

```
end.
```

**Solution 6 :**

Algorithme facture ;

Variables

pu, tva, ttc : réel ;

nb : entier ;

Début

Ecrire('Entrez le prix unitaire hors taxes :') ;

Lire(pu) ;

Ecrire('Entrez le nombre d"articles :') ;

Lire(nb) ;

Ecrire('Entrez le taux de TVA :') ;

Lire(tva) ;

$ttc \leftarrow nb * pu + ((nb * pu) * tva)$  ;

Ecrire('Le prix toutes taxes est : ', ttc) ;

Fin.

Le programme Pascal :

program facture ;

var

pu, tva, ttc : real ;

nb : integer ;

begin

writeln('Entrez le prix unitaire hors taxes :') ;

readln(pu) ;

writeln('Entrez le nombre d"articles :') ;

readln(nb) ;

writeln('Entrez le taux de TVA :') ;

readln(tva) ;

$ttc := nb * pu + ((nb * pu) * tva)$  ;

writeln('Le prix toutes taxes est : ', ttc) ;

end.

**Solution 7 :**

Algorithme rectangle ;

Variables

longueur, largeur, périmètre, surface : réel ;

Début

Ecrire('Entrez la longueur et la largeur du rectangle :') ;

Lire(longueur, largeur) ;

$Périmètre \leftarrow 2 * (longueur + largeur)$  ;

$surface \leftarrow longueur * largeur$  ;

Ecrire('Le périmètre du rectangle est : ', périmètre) ;

Ecrire('La surface du rectangle est : ', surface) ;

Fin.

Le programme Pascal :

program rectangle ;

```
var
    longueur, largeur, perimetre, surface : real ;
begin
    writeln('Entrez la longueur et la largeur du rectangle :)') ;
    readln(longueur, largeur) ;
    perimetre := 2 * (longueur + largeur) ;
    surface := longueur * largeur ;
    writeln('Le périmètre du rectangle est : ', perimetre) ;
    writeln('La surface du rectangle est : ', surface) ;
end.
```

**Solution 8 :**

Algorithme sphère ;

Constantes

pi = 3.1416 ;

Variables

R, Aire, Vol : réel ;

Début

Ecrire('Introduisez la valeur du rayon :)') ;

Lire(R) ;

Aire  $\leftarrow 4 * \pi * R * R$  ;

Vol  $\leftarrow 4/3 * \pi * R * R * R$  ;

Ecrire('Aire = ', Aire) ;

Ecrire('Volume = ', Vol) ;

Fin.

Le programme Pascal :

```
program sphere ;
```

```
const
```

```
    pi = 3.1416 ;
```

```
var
```

```
    R, Aire, Vol : real ;
```

```
begin
```

```
    writeln('Introduisez la valeur du rayon :)') ;
```

```
    readln(R) ;
```

```
    Aire := 4 * pi * R * R ;
```

```
    Vol := 4/3 * pi * R * R * R ;
```

```
    writeln('Aire = ', Aire) ;
```

```
    writeln('Volume = ', Vol) ;
```

```
end.
```

### Solution 9 :

Algorithme triangle ;

Variables

a, b, c, Perim, Aire, p : réel ;

Début

Ecrire('Introduisez la longueur du côté a :') ;

Lire(a) ;

Ecrire('Introduisez la longueur du côté b :') ;

Lire(b) ;

Ecrire('Introduisez la longueur du côté c :') ;

Lire(c) ;

$\text{Perim} \leftarrow a + b + c$  ;

$p \leftarrow \text{Perim} / 2$  ;

$\text{Aire} \leftarrow (p * (p-a)*(p-b)*(p-c))^{1/2}$  ;

Ecrire('Périmètre = ', Perim) ;

Ecrire('Aire = ', Aire) ;

Fin.

Le programme Pascal :

program triangle ;

var

a, b, c, Perim, Aire, p : real ;

begin

writeln('Introduisez la longueur du côté a :') ;

readln(a) ;

writeln('Introduisez la longueur du côté b :') ;

readln(b) ;

writeln('Introduisez la longueur du côté c :') ;

readln(c) ;

Perim := a + b + c ;

p := Perim / 2 ;

Aire := SQRT(p \* (p-a)\*(p-b)\*(p-c)) ;

writeln('Périmètre = ', Perim) ;

writeln('Aire = ', Aire) ;

end.

### Solution 10 :

Algorithme secteur ;

Constantes

pi = 3.1416 ;

Variables

```
R, Angle, Aire : réel ;
Début
  Ecrire('Introduisez le rayon :') ;
  Lire(R) ;
  Ecrire('Introduisez l'angle (en degré) :') ;
  Lire(Angle) ;
  Aire ←  $\pi * R * R * \text{Angle} / 360$  ;
  Ecrire('Aire = ', Aire) ;
Fin.
```

Le programme Pascal :

```
program secteur ;
const
  pi = 3.1416 ;
var
  R, Angle, Aire : real ;
begin
  writeln('Introduisez le rayon :') ;
  readln(R) ;
  writeln('Introduisez l'angle (en degré) :') ;
  readln(Angle) ;
  Aire :=  $\pi * R * R * \text{Angle} / 360$  ;
  writeln('Aire = ', Aire) ;
end.
```

**Solution 11 :**

Permutation en utilisant une variable intermédiaire :

```
Algorithme Permuter1 ;
Variables
  nb1, nb2, nb3 : réel ;
Début
  Ecrire('Entrez deux nombres :') ;
  Lire(nb1, nb2) ;
  nb3 ← nb1 ;
  nb1 ← nb2 ;
  nb2 ← nb3 ;
  Ecrire('Voici les deux nombres permutés :') ;
  Ecrire('nb1 = ', nb1) ;
  Ecrire('nb2 = ', nb2) ;
Fin.
```



Le programme Pascal :

```
program permuter1 ;
var
  nb1, nb2, nb3 : real ;
begin
  writeln('Entrez deux nombres :');
  readln(nb1, nb2) ;
  nb3 := nb1 ;
  nb1 := nb2 ;
  nb2 := nb3 ;
  writeln('Voici les deux nombres permutés :') ;
  writeln('nb1 = ', nb1) ;
  writeln('nb2 = ', nb2) ;
end.
```

Permutation sans variable intermédiaire :

```
Algorithme Permuter2 ;
Variables
  nb1, nb2 : réel ;
Début
  Ecrire('Entrez deux nombres :') ;
  Lire(nb1, nb2) ;
  nb1  $\leftarrow$  nb1 + nb2 ;
  nb2  $\leftarrow$  nb1 - nb2 ;
  nb1  $\leftarrow$  nb1 - nb2 ;
  Ecrire('Voici les deux nombres permutés :') ;
  Ecrire('nb1 = ', nb1) ;
  Ecrire('nb2 = ', nb2) ;
Fin.
```

Le programme Pascal :

```
program permuter1 ;
var
  nb1, nb2 : real ;
begin
  writeln('Entrez deux nombres :');
  readln(nb1, nb2) ;
  nb1 := nb1 + nb2 ;
  nb2 := nb1 - nb2 ;
  nb1 := nb1 - nb2 ;
  writeln('Voici les deux nombres permutés :') ;
```

```
writeln('nb1 = ', nb1) ;  
writeln('nb2 = ', nb2) ;  
end.
```

**Solution 12 :**

Algorithme conversion ;

Variables

nsec, h, m, s : entier ;

Début

Ecrire('Introduisez le nombre de secondes :') ;

Lire(nsec) ;

$s \leftarrow \text{nsec} \bmod 60$  ;

$m \leftarrow (\text{nsec} \% 3600) \div 60$  ;

$h \leftarrow \text{nsec} \div 3600$  ;

Ecrire(nsec, ' seconde(s) valent : ', h, ' heure(s), ', m, ' minute(s) et ', s, ' seconde(s)') ;

Fin.

Le programme Pascal :

program conversion ;

var

nsec, h, m, s : integer;

begin

writeln('Introduisez le nombre de secondes :') ;

readln(nsec) ;

$s := \text{nsec} \bmod 60$ ;

$m := (\text{nsec} \bmod 3600) \div 60$ ;

$h := \text{nsec} \div 3600$  ;

writeln(nsec, ' seconde(s) valent : ', h, ' heure(s), ', m, ' minute(s) et ', s, ' seconde(s)') ;

end.

## Chapitre 3 : Les structures conditionnelles

### 1. Introduction

Les algorithmes vus précédemment sont dits linéaires. Ils sont simples et exécutés séquentiellement. Les ruptures des séquences peuvent être appelées par des structures de contrôle classées en deux catégories : les structures conditionnelles (simples, composées et multiples) et les boucles.

Les structures conditionnelles sont appelées aussi structures alternatives, structures de choix ou tout simplement les tests.

### 2. Structure conditionnelle simple

La structure conditionnelle simple se présente sous la forme suivante :

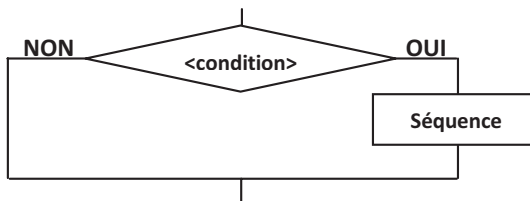
Si <condition> Alors <Séquence>

Si la condition est vérifiée (VRAI), alors la séquence d'opérations s'exécute.

La condition s'exprime sous forme d'une expression logique. Cette expression logique peut contenir des opérateurs de comparaison (=, <>, <, >, >=, <=), et donne une valeur booléenne (VRAI ou FAUX). L'expression peut être simple (condition simple) ou composée (plusieurs conditions composées avec des opérateurs logiques ET, OU et NON). On note que dans le cas de conditions composées, les parenthèses jouent un rôle fondamental. Aussi, on peut remplacer une expression logique par une autre équivalente. Par exemple, l'expression (A ET B) est équivalente à (NON A OU NON B).

La séquence peut contenir une ou un ensemble d'opérations. Si la séquence contient plusieurs opérations, alors elles sont séparées par des points-virgules et mises entre début et fin. Si la séquence contient une seule opération, alors les mots début et fin ne sont pas obligatoires.

La structure conditionnelle simple peut être représentée dans un organigramme comme suit :



Voyons les exemples suivants :

- Si (X > 10) Alors Ecrire(X) ;

Dans cet exemple, on affiche la valeur de X si elle est supérieure à 10.

- Si  $(X > 10)$  Alors début Ecrire(X) ;  $Y \leftarrow X$  ; fin ;

Dans cet exemple, on affiche la valeur de X et on affecte la valeur de X à Y, si X est supérieure à 10. Les deux opérations d'écriture et d'affectation sont mises obligatoirement entre début et fin.

- Si  $(X > 10)$  ET  $(X < 15)$  Alors Ecrire(X) ;

Dans cet exemple, on affiche la valeur de X si elle est prise entre 10 et 15. La condition est une expression logique composée. Cet expression logique est utilisée en informatique pour représenter la formule mathématique  $(10 < X < 15)$ .

- Si  $(X > 10)$  Alors Si  $(X < 15)$  Alors Ecrire(X) ;

Cet exemple est équivalent à l'exemple précédent, sauf que cette fois-ci, on utilise des tests imbriqués.

- Si  $(X < 10)$  Alors Si  $(X > 15)$  Alors Ecrire(x) ;

Dans cet exemple, la valeur de X n'est jamais affichée car il n'y aucun cas qui satisfait la condition.

En Pascal, la structure conditionnelle simple s'écrit sous la forme :

if <condition> then <Séquence>

Si la séquence contient plus d'une instruction, alors le begin et end sont obligatoires pour délimiter la suite d'instructions.

Voyons les exemples précédents exprimés en langage Pascal :

- if  $(X > 10)$  then write(X) ;
- if  $(X > 10)$  then begin write(X) ;  $Y := X$  ; end ;
- if  $(X > 10)$  AND  $(X < 15)$  then write(X) ;
- if  $(X > 10)$  then if  $(X < 15)$  then write(X) ;
- if  $(X < 10)$  then if  $(X > 15)$  then write(X) ;

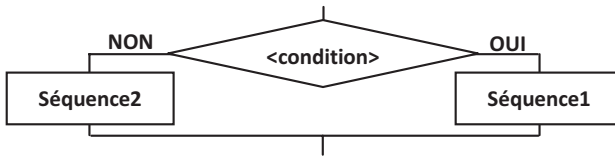
### 3. Structure conditionnelle composée

La structure conditionnelle composée se présente sous la forme suivante :

Si <condition> Alors <Séquence1> Sinon <Séquence2>

Si la condition est vérifiée (VRAI), alors les opérations de la séquence1 sont exécutées. Dans le cas contraire, ce sont les opérations de la séquence2 qui vont être exécutées. Les mots début et fin sont utilisés pour délimiter une séquence de plusieurs opérations.

La structure conditionnelle composée peut être représentée dans un organigramme comme suit :



Voyons l'exemple suivant :

Si  $(X > 10)$  Alors Ecrire(X) Sinon Ecrire('valeur non acceptée') ;

Dans cet exemple, la valeur de X est affichée si elle est supérieure à 10, sinon on affiche un message d'erreur.

En Pascal, ça s'écrit comme suit :

if  $(x > 10)$  then writeln(x) else writeln('valeur non acceptée') ;

Si nous avons une suite d'instructions dans une séquence, alors on aurait dû utiliser les mots begin et end.

### Remarques :

- En Pascal, le else n'est jamais précédé par un point-virgule (;).
- Le else se rapporte toujours au if...then... le plus proche. Pour casser ce rapport, il est possible d'utiliser le begin et end. Par exemple, dans le cas de :

```

if (x > 10) then begin if (x < 20) then writeln(x) ; end
                    else write('valeur non acceptée') ;
  
```

le else suit le premier if et non pas le deuxième.

- Les deux règles précédentes sont aussi appliquées dans le LDA utilisé dans ce cours, c.-à-d. le Sinon n'est jamais précédé par un point-virgule, et le Sinon suit le Si le plus proche.

## 4. Structure conditionnelle multiple

La structure conditionnelle multiple, appelée aussi l'alternative classifiée ou le choix multiple, peut comparer une expression à toute une série de valeurs, et exécuter une séquence d'opérations parmi plusieurs, en fonction de la valeur effective de l'expression. Une séquence par défaut peut être prévue dans le cas où l'expression n'est égale à aucune des valeurs énumérées.

Chaque séquence est étiquetée par une valeur. Pour que cette séquence soit choisie, il faut que sa valeur soit équivalente à l'expression. La structure alternative multiple se présente comme suit :

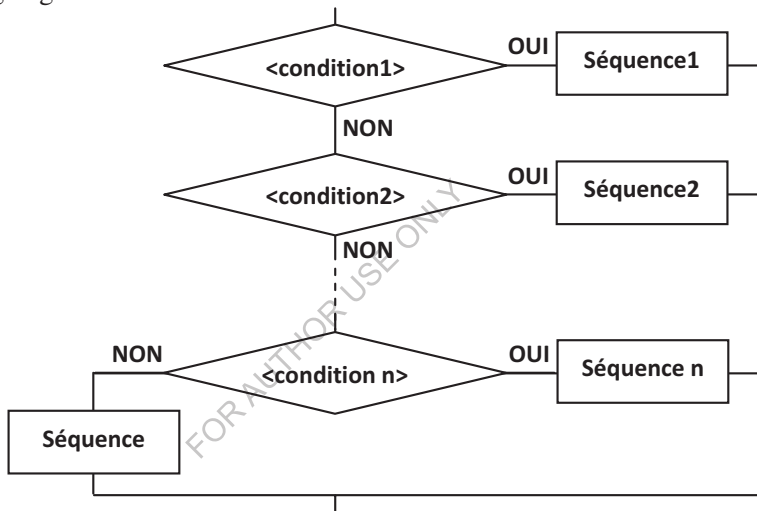
```

Cas <expression> de
  Valeur 1 : <Séquence1>
  Valeur 2 : <Séquence2>
  ...
  Valeur n : <Séquence n>
  Sinon <Séquence par défaut>
fin ;
  
```

Il s'agit donc d'une généralisation de la structure conditionnelle simple Si. Elle permet d'exécuter une parmi plusieurs actions en s'orientant selon les différentes valeurs que peut prendre une expression. Par conséquent, elle est équivalente à :

Si (expression = Valeur1) Alors <Séquence1>  
 Sinon Si (expression = Valeur2) Alors <Séquence2>  
 ...  
 Sinon Si (expression = Valeur n) Alors <Séquence n>  
 Sinon <Séquence par défaut>

La structure conditionnelle multiple peut être représentée dans un organigramme comme suit :



Dans l'exemple suivant, la valeur de X est affichée en lettres si elle est égale à 1 ou 2, sinon on affiche un message d'erreur :

Cas (X) de  
 1 : Ecrire('Un') ;  
 2 : Ecrire('Deux')  
 Sinon Ecrire('Valeur sup à deux ou inf à un') ;  
 fin ;

En Pascal, ça s'écrit sous la forme :

```

case (x) of
  1 : writeln('Un');
  2 : writeln('Deux')
else writeln('Valeur sup à deux ou inf à un') ;
end;
```

### Remarques :

- Dans l'instruction de choix multiple, l'ordre de représentation ne change rien.
- L'expression et les valeurs à choisir doivent être de même type.
- Si une séquence est composée d'un ensemble d'instructions, elles doivent être prises entre begin et end.

## 5. Le branchement

On ne peut pas quitter ce chapitre sans noter qu'il est possible d'effectuer un saut direct vers une opération en utilisant une opération de branchement de la forme aller à <étiquette>. Les étiquettes sont des adresses qui doivent être déclarées dans la partie déclaration de l'algorithme.

### Exemple :

```
Algorithme afficher_nbr_positif ;  
  Etiquettes 10 ;  
  Variables  
    I : entier ;  
Début  
  Lire(I) ;  
  Si (I<0) Alors aller à 10 ;  
  Ecrire(I) ;  
10 : Fin.
```

En Pascal, ça s'écrit comme suit :

```
program afficher_nbr_positif ;  
  label 10 ;  
  var  
    I : integer ;  
begin  
  readln(I);  
  if (I<0) then goto 10 ;  
  writeln(I) ;  
10 : end.
```

On note qu'il est déconseillé d'utiliser l'instruction de branchement, et cela pour réduire la complexité des programmes en termes de temps.

## 6. Exercices corrigés

### 6.1. Exercices

#### Exercice 1 :

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on laisse de côté le cas où le

nombre vaut zéro). Traduire l'algorithme en Pascal. Ensuite, représenter l'algorithme par un organigramme.

**Exercice 2 :**

Ecrire un algorithme permettant d'afficher la valeur absolue d'un nombre réel. Traduire l'algorithme en Pascal.

**Exercice 3 :**

Ecrire un algorithme permettant de comparer deux nombres réels. Traduire l'algorithme en Pascal.

**Exercice 4 :**

Ecrire un algorithme qui permet d'afficher le résultat d'un étudiant (accepté ou rejeté) à un module, sachant que ce module est évalué par une note d'oral de coefficient 1, et une note d'écrit de coefficient 2. La moyenne obtenue doit être supérieure ou égale à 10 pour valider le module. Comme entrées, on a la note d'oral et la note d'écrit, ensuite on calcule la moyenne, et on affiche le résultat. Traduire l'algorithme en Pascal.

**Exercice 5 :**

Ecrire un algorithme permettant de déterminer le plus grand de trois nombres réels lus à partir du clavier. Traduire l'algorithme en Pascal.

**Exercice 6 :**

Ecrire un algorithme qui demande deux nombres à l'utilisateur, et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul). Attention ! On ne doit pas calculer le produit des deux nombres. Traduire l'algorithme en Pascal.

**Exercice 7 :**

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on inclut cette fois-ci le traitement du cas où le nombre vaut zéro). Traduire l'algorithme en Pascal.

**Exercice 8 :**

Ecrire un algorithme qui demande deux nombres à l'utilisateur, et l'informe ensuite si le produit est négatif ou positif (on inclut cette fois-ci le traitement du cas où le produit peut être nul). Attention ! On ne doit pas calculer le produit. Traduire l'algorithme en Pascal.

**Exercice 9 :**

Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :

- Poussin, de 6 à 7 ans.
- Pupille, de 8 à 9 ans.
- Minime, de 10 à 11 ans.
- Cadet, après 12 ans.



Traduire l'algorithme en Pascal.

**Exercice 10 :**

Ecrire un algorithme permettant de résoudre l'équation du 1<sup>ier</sup> degré :  $ax+b=0$ . Traduire l'algorithme en Pascal.

**Exercice 11 :**

Ecrire un algorithme qui permet de calculer les racines de l'équation du second degré suivante :  $ax^2 + bx + c = 0$ , sachant que :

- Si  $b^2-4ac > 0$ , l'équation possède deux racines  $x_1$  et  $x_2$  telles que :  $x_1 = (-b - \sqrt{b^2 - 4ac})/2a$  ;  $x_2 = (-b + \sqrt{b^2 - 4ac})/2a$ .
- Si  $b^2-4ac = 0$  alors l'équation possède une racine  $x = -b/2a$ .
- Si  $b^2-4ac < 0$  alors l'équation ne possède pas de racine.

Dans chaque cas, l'algorithme devra afficher le message correspondant, et la ou les racines éventuelles. Traduire l'algorithme en Pascal.

**Exercice 12 :**

Ecrire un algorithme qui demande d'entrer un nombre entre 1 et 7, et donne le nom du jour correspondant (samedi, dimanche...) en utilisant une structure conditionnelle multiple. Traduire l'algorithme en Pascal.

**Exercice 13 :**

Ecrire un algorithme permettant à partir d'un menu affiché, d'effectuer la somme, le produit ou la moyenne de trois nombres. Nous appelons menu, l'association d'un numéro séquentiel aux différents choix proposés par un algorithme. Traduire l'algorithme en Pascal.

**Exercice 14 :**

Ecrire un algorithme permettant de donner le résultat d'un jeu dont les règles sont très simples : deux joueurs A et B se cachent la main droite derrière le dos. Chacun choisit de tendre un certain nombre de doigts (de 0 à 5), toujours derrière le dos. Les deux joueurs se montrent la main droite en même temps. Si la somme des nombres de doigts montrés est paire, le premier joueur a gagné, sinon c'est le second. Les étapes de l'algorithme sont les suivantes :

- Prendre connaissance du nombre de doigts de A.
- Prendre connaissance du nombre de doigts de B.
- Calculer la somme de ces deux nombres.
- Si la somme est paire, A est le gagnant.
- Si la somme est impaire, B est le gagnant.

Pour déterminer si un nombre est pair ou impair, il suffit de calculer le reste de la division par 2 (modulo 2) : il vaut 0 dans le premier cas et 1 dans le second. Traduire ensuite l'algorithme en Pascal.

**Exercice 15 :**

Ecrire un algorithme permettant de calculer le salaire d'un employé payé à l'heure à partir de son salaire horaire et du nombre d'heures de travail.

Le salaire est calculé en multipliant le salaire horaire par le nombre d'heures de travail pour les premières 160 heures. Il va y avoir une réduction de 25% pour le reste des heures (qui dépasse 160 h) et 50% pour le reste des heures (qui dépasse 200 h). Traduire l'algorithme en Pascal.

**Exercice 16 :**

Ecrire un algorithme permettant de déterminer si l'année A est bissextile. On doit savoir que si A n'est pas divisible par 4, l'année n'est pas bissextile. Si A est divisible par 4, l'année est bissextile, sauf si A est divisible par 100 et non pas par 400. Traduire l'algorithme en Pascal. Testez votre programme pour les années 111, 1984, 1900 et 800.

**Exercice 17 :**

Ecrire un algorithme permettant d'effectuer la transformation des coordonnées cartésiennes (x,y) en coordonnées polaires (r,t). Cette transformation se fait par les formules :

- $r^2 = x^2 + y^2$
- Si  $x = 0$  alors
  - $t = \pi/2$  si  $y > 0$ .
  - $t = -\pi/2$  si  $y < 0$ .
  - t n'existe pas si  $y = 0$ .
- Sinon  $t = \arctg(y/x)$  auquel il faut ajouter  $\pi$  si  $x < 0$ .

Traduire ensuite l'algorithme en Pascal.

**Exercice 18 :**

Ecrire un algorithme permettant de lire une date (j/m/a) et déterminer le jour de la semaine. L'algorithme doit être valable pour les dates postérieures à 1582. Essayez de suivre les indications suivantes :

- Pour janvier et février, il faut augmenter m de 12 et diminuer a de 1.
- Calculer s qui vaut la partie entière de a divisé par 100.
  - $JD = 1720996,5 - s + s \div 4 + [365,25*a] + [30,6001*(M+1)] + j$
  - $JD = JD - [JD/7]*7$
  - $JS = [JD] \text{ MOD } 7$
- Si  $JS = 0$ , le jour est mardi,
- Si  $JS = 1$ , le jour est mercredi,
- ....
- Si  $JS = 6$ , le jour est lundi.

Traduire ensuite l'algorithme en Pascal.

**Exercice 19 :**

Ecrire un algorithme permettant le calcul du sinus d'un angle exprimé en radians. Les étapes de cet algorithme sont les suivantes :

- On commence par lire un angle Rad exprimé en radians.
- Mettre +1 dans Sign.

- Enlever un nombre entier de tours grâce à :
  - $Rad = Rad - 2 * \pi * [Rad / (2 * \pi)]$ , si  $rad > 0$ .
  - $Rad = Rad + 2 * \pi * [-Rad / (2 * \pi)] + 2 * \pi$ , si  $rad \leq 0$ .
- Si  $Rad > \pi$ , le ramener entre 0 et  $\pi$  en soustrayant  $\pi$  et s'en souvenir en mettant -1 dans Sign.
- Si  $Rad > \pi/2$  alors  $Rad = \pi - Rad$ .
- Si  $Rad > \pi/4$  alors
  - $Rad = \pi/2 - Rad$ .
  - Sinus vaut  $1 - Rad * Rad / 2 + Rad * Rad * Rad * Rad / 24$ .
- Si  $Rad \leq \pi/4$ , Sinus vaut  $Rad - Rad * Rad * Rad / 6 + Rad * Rad * Rad * Rad * Rad / 120$ .
- Multiplier Sinus par Sign et afficher la réponse.

Traduire ensuite l'algorithme en Pascal.

### Exercice 20 :

Ecrire un algorithme permettant de lire le prix unitaire (*pu*) d'un produit et la quantité commandée (*qtcom*), ensuite de calculer le prix de livraison (*pl*), le taux de la réduction (*tr*), et enfin de calculer le prix à payer (*pap*) sachant que :

- La livraison est gratuite, si le prix des produits (*tot*) est supérieur à 500 DA. Dans le cas contraire, le prix de la livraison est de 2% du *tot*.
- La réduction est de 5%, si *tot* est compris entre 200 et 1000 dinars et de 10% au-delà.

Traduire l'algorithme en Pascal. Donnez une description de l'algorithme par un organigramme

### Exercice 21 :

Ecrire un algorithme qui calcule le prix d'un billet de cinéma selon une grille tarifaire qui comprend plusieurs classes. On accorde un escompte sur ce prix aux conditions suivantes :

- Si le client est âgé de 15 ans ou moins, ou âgé de 60 ans ou plus :
  - Si le billet est vendu un jour de semaine (Samedi à Mercredi), on accorde un escompte de 25%.
  - Sinon, on accorde un escompte de 10%.
- Si le billet est vendu à un client qui n'entre pas dans cette catégorie d'âge :
  - Si le billet est vendu un lundi ou un jeudi, on accorde un escompte de 15%.
  - Sinon, on n'accorde aucun escompte.

Pour calculer le prix du billet, l'algorithme demande d'entrer le jour sous forme d'un numéro (0 : samedi, 1 : dimanche, etc.), l'année en cours, l'année de naissance du client et le tarif de base. On veut afficher

l'âge du client, le montant de l'escompte accordé, ainsi que le prix du billet. Traduire ensuite l'algorithme en Pascal.

### Exercice 22 :

Pour simuler les ventes, un magasin de chaussures offre un taux de réduction ( $tr$ ) sur achat dans les conditions suivantes :

- Si le nombre de paires ( $nb\_paire$ ) achetées est supérieur à 2 :
  - Si le montant d'achat total ( $mnt\_achat$ ) est inférieur ou égal à 1000 DA, le taux de réduction est de 10%.
  - Si le montant d'achat est supérieur à 1000 DA, le taux de réduction est de 20%.
- Si le nombre de paires achetées est inférieur ou égal à 2 :
  - Si le montant d'achat est inférieur à 500 DA, il n'y a pas de réduction.
  - Si le montant d'achat est supérieur ou égal à 500 DA, le taux de réduction est de 5%.

Ecrire un algorithme, ensuite le programme Pascal, qui détermine le montant d'achat total ( $mnt\_achat$ ), montant de la réduction accordée ( $tr$ ), ainsi que le prix net total à payer ( $pr\_net$ ). En entrée, on doit avoir le nombre de paires achetées ( $nb\_paire$ ) et le prix unitaire de chaque paire ( $pu$ ).

### 6.2. Corrigés

#### Solution 1 :

Algorithme positif\_négatif :

Variables

n : entier ;

Début

Ecrire('Entrez un nombre :) ;

Lire(n) ;

Si (n >= 0) Alors Ecrire('Ce nombre est positif.')

Sinon Ecrire('Ce nombre est négatif.') ;

Fin.

Le programme Pascal :

program positif\_negatif ;

var

n : integer ;

begin

writeln('Entrez un nombre :) ;

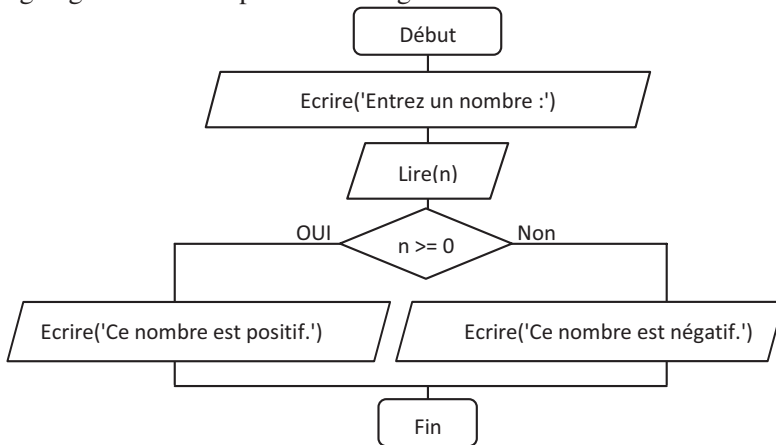
readln(n) ;

if (n >= 0) then writeln('Ce nombre est positif.')

else writeln('Ce nombre est négatif.') ;

end.

L'organigramme correspondant à l'algorithme :



### Solution 2 :

Algorithme valeur\_absolue ;

Variables

x, val\_abs : réel ;

Début

Ecrire('Entrez un nombre :) ;

Lire(x) ;

val\_abs ← x ;

Si (val\_abs < 0) Alors val\_abs ← -val\_abs ;

Ecrire('La valeur absolue de ', x, ' est ', val\_abs) ;

Fin.

Le programme Pascal :

program valeur\_absolue ;

var x, val\_abs : real ;

begin

writeln('Entrez un nombre :) ;

readln(x) ;

val\_abs := x ;

if (val\_abs < 0) then val\_abs := - val\_abs;

writeln('La valeur absolue de ', x, ' est ', val\_abs) ;

end.

### Solution 3 :

Algorithme comparaison ;

Variables

x, y : réel ;

Début

```
Ecrire('Entrez deux nombres x et y :') ;  
Lire(x, y) ;  
Si (x = y) Alors Ecrire(x, ' = ', y)  
    Sinon Si (x > y) Alors Ecrire(x, ' est supérieur à ', y)  
        Sinon Ecrire(y, ' est supérieur à ', x) ;
```

Fin.

Le programme Pascal :

```
program comparaison ;  
var x, y : real ;  
begin  
    writeln('Entrez deux nombres x et y :');  
    readln(x, y) ;  
    if (x = y) then writeln(x, ' = ', y)  
        else if (x > y) then writeln(x, ' est supérieur à ', y)  
            else writeln(y, ' est supérieur à ', x) ;  
end.
```

#### **Solution 4 :**

Algorithme évaluation ;

Variables

ne, no, moy : réel ;

Début

```
Ecrire('Entrez la note de l'examen écrit :) ;  
Lire(ne) ;  
Ecrire('Entrez la note de l'examen oral :) ;  
Lire(no) ;  
moy  $\leftarrow$  (2 * ne + no)/3 ;  
Si (moy >= 10) Alors Ecrire('accepté')  
    Sinon Ecrire('rejeté') ;
```

Fin.

Le programme Pascal :

```
program evaluation ;  
var  
    ne, no, moy : real ;  
begin  
    writeln('Entrez la note de l'examen écrit :) ;  
    readln(ne) ;  
    writeln('Entrez la note de l'examen oral :) ;  
    readln(no) ;
```

```
moy := (2 * ne + no)/3;
if (moy >= 10) then writeln('accepté')
  else writeln('rejeté') ;
end.
```

**Solution 5 :**

Algorithme plus\_grand ;

Variables

x, y, z, pg : réel ;

Début

Ecrire('Entrez trois nombres :') ;

Lire(x, y, z) ;

Si ( x>= y) et ( x>= z) Alors pg ← x

    Sinon Si (y >= z) Alors pg ← y

        Sinon pg ← z ;

Ecrire('Le plus grand des trois nombres est : ', pg) ;

Fin.

Le programme Pascal :

program plus\_grand ;

var

x, y, z, pg : real ;

begin

writeln('Entrez trois nombres :') ;

readln(x, y, z) ;

if ( x>=y ) AND (x>=z) then pg := x

    else if (y>=z) then pg := y

        else pg := z ;

writeln('Le plus grand des trois nombres est : ', pg) ;

end.

**Solution 6 :**

Algorithme résultat\_produit ;

Variables

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 de ', m, ' et ', n, ' est positif.')

    Sinon Ecrire('Le produit de ', m, ' et ', n, ' est négatif.')

Fin.

Le programme Pascal :

```
program resultat_produit ;
var
  m, n : integer ;
begin
  writeln('Entrez deux nombres :)') ;
  readln(m, n) ;
  if ((m > 0) and (n > 0)) or ((m < 0) and (n < 0)) then
    writeln('Le produit de ', m, ' et ', n, ' est positif.')
  else writeln('Le produit de ', m, ' et ', n, ' est négatif.') ;
end.
```

**Solution 7 :**

Algorithme positif\_négatif ;

Variables

n : entier ;

Début

Ecrire('Entrez un nombre :)') ;

Lire(n) ;

Si (n < 0) Alors Ecrire('Ce nombre est négatif.')

    Sinon Si (n = 0) Alors Ecrire('Ce nombre est nul.')

        Sinon Ecrire('Ce nombre est positif.') ;

Fin.

Le programme Pascal :

```
program positif_negatif ;
```

```
var
```

```
  n : integer ;
```

```
begin
```

```
  writeln('Entrez un nombre :)') ;
```

```
  readln(n) ;
```

```
  if (n < 0) then writeln('Ce nombre est négatif.')
```

```
    else if (n = 0) then writeln('Ce nombre est nul.')
```

```
    else writeln('Ce nombre est positif.') ;
```

```
end.
```

**Solution 8 :**

Algorithme résultat\_produit ;

Variables

m, n : entier ;

Début

Ecrire('Entrez deux nombres :)') ;



```

Lire(m, n) ;
Si (m = 0) OU (n = 0) Alors Ecrire('Le produit de ', m, ' et ', n, ' est nul.')
Sinon Si ((m < 0) ET (n < 0)) OU ((m > 0) ET (n > 0)) Alors
    Ecrire('Le produit de ', m, ' et ', n, ' est positif.')
Sinon Ecrire('Le produit de ', m, ' et ', n, ' est négatif.') ;
Fin.

```

Le programme Pascal :

```

program resultat_produit ;
var
    m, n : integer ;
begin
    writeln('Entrez deux nombres :)') ;
    readln(m, n) ;
    if ((m = 0) OR (n = 0)) then writeln('Le produit de ', m, ' et ', n, ' est nul.')
    else if ((m < 0) and (n < 0)) or ((m > 0) and (n > 0)) then
        writeln('Le produit de ', m, ' et ', n, ' est positif.')
    else writeln('Le produit de ', m, ' et ', n, ' est négatif.') ;
end.

```

**Solution 9 :**

Algorithme enfant ;

Variable

age : entier ;

Début

Ecrire('Entrez l'âge de l'enfant :)') ;

Lire(age) ;

Si (age >= 12) Alors Ecrire('Catégorie Cadet.')

Sinon Si (age >= 10) Alors Ecrire('Catégorie Minime.')

Sinon Si (age >= 8) Alors Ecrire('Catégorie Pupil.')

Sinon Si (age >= 6) Alors Ecrire('Catégorie Poussin.') ;

Fin.

Le programme Pascal :

program enfant ;

var

age : integer ;

begin

writeln('Entrez l'âge de l'enfant :)') ;

readln(age) ;

if (age >= 12) then writeln('Catégorie Cadet.')

else if (age >= 10) then writeln('Catégorie Minime.')

```
        else if (age >= 8) then writeln('Catégorie Pupille.')
```

```
        else if (age >= 6) then writeln('Catégorie Poussin.');
```

```
end.
```

**Solution 10 :**

Algorithme Eq\_1er\_degre ;

Variables

a, b, x : réel ;

Début

Ecrire('Résolution de  $ax + b = 0$ ') ;

Ecrire('Introduisez a :') ;

Lire(a) ;

Ecrire('Introduisez b :') ;

Lire(b) ;

Si (a=0) Alors Si (b=0) Alors Ecrire('Equation indéterminée.')

        Sinon Ecrire('Equation impossible.')

        Sinon début

$x \leftarrow -b/a$  ;

            Ecrire('x = ',x) ;

        fin ;

Fin.

Le programme Pascal :

program Eq\_1er\_degre;

var a, b, x : real;

begin

writeln('Résolution de  $ax + b = 0$ ');

writeln('Introduisez a :');

readln(a);

writeln('Introduisez b :');

readln(b);

if (a=0) then if (b=0) then writeln('Equation indéterminée.')

        else writeln('Equation impossible.')

    else begin

        x:= -b/a;

        writeln('x = ',x:5:2);

    end;

end.

**Solution 11 :**

Algorithme Eq\_2eme\_degre ;

Variables

```

a, b, c, d, x1, x2, x : réel ;
Début
  Ecrire('Entrer les valeurs a, b et c de l"équation :');
  Lire(a, b, c);
  d ← b * b - 4 * a * c;
  Si (d>0) Alors début
    x1 ← (-b - d1/2) / (2*a);
    x2 ← (-b + d1/2) / (2*a);
    Ecrire('L"équation possède deux racines :');
    Ecrire('x1 = ', x1, ' x2 = ', x2);
  fin
  Sinon Si (d=0) Alors début
    x ← -b / (2*a);
    Ecrire('L"équation possède une racine :');
    Ecrire('x = ', x);
  fin
  Sinon Ecrire('L"équation ne possède pas de racines.');
```

Fin.

Le programme Pascal :

```

program Eq_2eme_degre;
var
  a, b, c, d, x1, x2, x : real;
begin
  writeln('Entrer les valeurs a, b et c de l"équation :');
  readln(a, b, c);
  d := sqr(b) - 4 * a * c;
  if (d > 0) then begin
    x1:=(-b - sqrt(d)) / (2*a);
    x2:=(-b + sqrt(d)) / (2*a);
    writeln('L"équation possède deux racines :');
    writeln('x1 = ', x1, ' x2 = ', x2);
  end
  else if (d=0) then begin
    x := -b / (2*a);
    writeln('L"équation possède une racine :');
    writeln('x = ', x);
  end
  else writeln('L"équation ne possède pas de racines.');
```

end.

**Solution 12 :**

Algorithme jour ;

Variables

no : entier ;

Début

Ecrire('Entrer le num de jour : 1.Samedi, 2.Dimanche ... :') ;

Lire(no) ;

Cas (no) de

1 : Ecrire('Samedi') ;

2 : Ecrire('Dimanche') ;

3 : Ecrire('Lundi') ;

4 : Ecrire('Mardi') ;

5 : Ecrire('Mercredi') ;

6 : Ecrire('Jeudi') ;

7 : Ecrire('Vendredi')

Sinon Ecrire('Numéro non accepté.') ;

fin ;

Fin.

Le programme Pascal :

program jour ;

var

no : integer;

begin

writeln('Entrer le num de jour : 1.Samedi, 2.Dimanche ... :');

readln(no) ;

case no of

1 : writeln('Samedi') ;

2 : writeln('Dimanche') ;

3 : writeln('Lundi') ;

4 : writeln('Mardi') ;

5 : writeln('Mercredi') ;

6 : writeln('Jeudi') ;

7 : writeln('Vendredi')

else writeln('Numéro non accepté.');

end;

end.

**Solution 13 :**

Algorithme menu ;

Variables

```

nb1, nb2, nb3 : réel ;
Choix : entier ;
Début
Ecrire('Entrez trois nombres :') ;
Lire(nb1, nb2, nb3) ;
{ Affichage du menu et saisie du choix }
Ecrire('1-pour la multiplication') ;
Ecrire('2-pour la somme') ;
Ecrire('3-pour la moyenne') ;
Ecrire('Votre choix : ') ;
Lire(choix) ;
Cas (choix) de
  1 : Ecrire('Le produit des trois nombres est : ', nb1 * nb2 * nb3) ;
  2 : Ecrire('La somme des trois nombres est : ', nb1 + nb2 + nb3) ;
  3 : Ecrire('La moyenne des trois nombres est : ', (nb1 + nb2 + nb3)/3)
  Sinon Ecrire('Choix incorrect.') ;
fin ;
Fin.

```

Le programme Pascal :

```

program menu ;
var nb1, nb2, nb3 : real ; choix : integer ;
begin
  writeln('Entrez trois nombres :');
  readln(nb1, nb2, nb3);
  { Affichage du menu et saisie du choix }
  writeln('1. pour la multiplication') ;
  writeln('2. pour la somme') ;
  writeln('3. pour la moyenne') ;
  writeln('Votre choix : ') ;
  readln(choix) ;
  case choix of
    1 : writeln('Le produit des trois nombres est : ', nb1 * nb2 * nb3) ;
    2 : writeln('La somme des trois nombres est : ', nb1 + nb2 + nb3) ;
    3 : writeln('La moyenne des trois nombres est : ', (nb1 + nb2 +
nb3)/3)
    else writeln('Choix incorrect.' ) ;
  end ;
end.

```

**Solution 14 :**

Algorithme Jeu ;

Variables

na, nb, reste : entier ;

Début

Ecrire('Introduisez le nombre de doigts montrés par le joueur A :');

Lire(na) ;

Ecrire('Introduisez le nombre de doigts montrés par le joueur B :');

Lire(nb) ;

Si (na <= 5) ET (nb <= 5) ET (na >= 0) ET (nb >= 0) Alors début

reste ← (na + nb) MOD 2;

Si (reste = 0) Alors Ecrire('Le joueur A a gagné.')

Sinon Ecrire('Le joueur B a gagné.');

Ecrire('Bravo pour le gagnant!')

fin

Sinon Ecrire('Le nombre de doigts doit être entre 0 et 5.');

Fin.

Le programme Pascal :

program Jeu;

var

na, nb, reste : integer;

begin

writeln('Introduisez le nombre de doigts montrés par le joueur A :');

readln(na);

writeln('Introduisez le nombre de doigts montrés par le joueur B :');

readln(nb);

if (na <= 5) AND (nb <= 5) AND (na >= 0) AND (nb >= 0) then begin

reste := (na + nb) MOD 2;

if (reste=0) then writeln('Le joueur A a gagné.')

else writeln('Le joueur B a gagné.');

writeln('Bravo pour le gagnant!');

end

else writeln('Le nombre de doigts doit être entre 0 et 5.');

end.

**Solution 15 :**

Algorithme calcul\_salaire ;

Variables

sh, salaire : réel ;

nbn : entier ;

Début

```
Ecrire('Entrer le nombre d"heures :') ;  
Lire(nbh) ;  
Ecrire('Entrer le salaire horaire :') ;  
Lire(sh) ;  
Si (nbh < 160) Alors salaire  $\leftarrow$  sh * nbh  
Sinon Si (nbh < 200) Alors salaire  $\leftarrow$  nbh * sh - (nbh - 160) * 0,25 * sh  
Sinon salaire  $\leftarrow$  nbh * sh - 40 * sh * 0,25 - (nbh - 200) * sh * 0,5 ;  
Ecrire('Le salaire = ', salaire);
```

Fin.

Le programme Pascal :

```
program calcul_salaire ;
```

```
var
```

```
sh, salaire : real ;
```

```
nbh : integer ;
```

```
begin
```

```
writeln('Entrer le nombre d"heures :') ;
```

```
readln(nbh) ;
```

```
writeln('Entrer le salaire horaire :') ;
```

```
readln(sh) ;
```

```
if (nbh < 160) then salaire := sh * nbh
```

```
else if (nbh < 200) then salaire := nbh * sh - (nbh - 160) * 0.25 * sh
```

```
else salaire := nbh * sh - 40 * sh * 0.25 - (nbh - 200) * sh * 0.5;
```

```
writeln('Le salaire = ', salaire);
```

```
end.
```

### **Solution 16 :**

Algorithme bissextile ;

Variables

A : entier ;

Début

```
Ecrire('Introduisez l"année :') ;
```

```
Lire(A) ;
```

```
Si NON (A % 4 = 0) Alors Ecrire('L"année ', A, ' n"est pas bissextile.')
```

```
Sinon Si (A % 100 = 0) ET NON (A % 400 = 0) Alors
```

```
Ecrire('L"année ', A, ' n"est pas bissextile.')
```

```
Sinon Ecrire('L"année ', A, ' est bissextile.');
```

Fin.

Le programme Pascal :

```
program bissextile ;
```

```

var
  A : integer ;
begin
  writeln('Introduisez l'année :) ;
  readln(A) ;
  if NOT (A mod 4 = 0) then writeln('L'année ', A, ' n'est pas bissextile.')
  else if (A mod 100 = 0) and not (A mod 400 = 0) then
    writeln('L'année ', A, ' n'est pas bissextile.')
  else writeln('L'année ', A, ' est bissextile.') ;
end.

```

Après test on a obtenu : 111 est non bissextile, 1984 est bissextile, 1900 n'est pas bissextile, et 800 est bissextile.

### **Solution 17 :**

Algorithme Cartesien\_Polaire ;

Constantes

Pi=3.14 ;

Variables

x, y, r, t : réel ;

Début

Ecrire('Introduisez l'abscisse x :) ;

Lire(x) ;

Ecrire('Introduisez l'ordonnée y :) ;

Lire(y) ;

$r \leftarrow (x^2 + y^2)^{1/2}$  ;

Si (x=0) Alors Si (y>0) Alors Ecrire('r=', r, ' et t=', pi/2)

Sinon Si (y<0) Alors Ecrire('r=', r, ' et t=', -pi/2)

Sinon Ecrire('r=', r, ' et t n'existe pas.') ;

Sinon début

$t \leftarrow \arctg(y/x)$  ;

Si (x<0) Alors  $t \leftarrow t + \pi$  ;

Ecrire('r=', r, ' et t=', t) ;

fin ;

Fin.

Le programme Pascal :

program Cartesien\_Polaire ;

const

pi = 3.14;

var



```

x, y, r, t : real ;
begin
  writeln('Introduisez l'abscisse x :') ;
  readln(x) ;
  writeln('Introduisez l'ordonnée y :') ;
  readln(y) ;
  r := SQRT(x*x+y*y);
  if (x=0) then if (y>0) then writeln('r=', r:5:2 , ' et t=', pi/2:5:2)
                else if (y<0) then writeln('r=', r:5:2 , ' et t=', -pi/2:5:2)
                else writeln('r=', r:5:2 , ' et t n'existe pas.')
  else begin
    t := ARCTAN(y/x) ;
    if (x<0) then t := t + pi ;
    writeln('r=', r , ' et t=', t) ;
  end ;
end.

```

### **Solution 18 :**

Algorithme Calendrier ;

Variables

j, m, a, s, JS : entier ;

JD: réel ;

Dat : chaîne de caractères;

Début

Ecrire('Calendrier perpétuel.');

Ecrire('Introduisez le jour (1-31) :') ;

Lire(j) ;

Ecrire('Introduisez le mois (1-12) :') ;

Lire(m) ;

Ecrire('Introduisez l'année (xxxx) :') ;

Lire(a) ;

Si (m < 3) Alors début

m ← m + 12

a ← a - 1 ;

fin ;

s ← a ÷ 100 ;

JD ← 1720996,5 - s + s ÷ 4 + [365,25\*a] + [30,6001\*(M+1)] + j ;

JD ← JD - [JD/7]\*7 ;

JS ← [JD] mod 7 ;

Cas (JS) de

```

0 : Dat ← 'mardi' ;
1 : Dat ← 'mercredi' ;
2 : Dat ← 'jeudi' ;
3 : Dat ← 'vendredi' ;
4 : Dat ← 'samedi' ;
5 : Dat ← 'dimanche' ;
6 : Dat ← 'lundi' ;
fin ;
Ecrire('Le ', j, '/', m, '/', a, 'est un ', Dat) ;
Fin.

```

Le programme Pascal :

```

program Calendrier ;
var
  a, m, j, s, JS : WORD;
  JD : real;
  Dat : string;
begin
  writeln('Calendrier perpétuel. ');
  writeln('Introduisez le jour (1-31) : ');
  readln(j);
  writeln('Introduisez le mois (1-12) : ');
  readln(m);
  writeln('Introduisez l'année (xxxx) : ');
  readln(a);
  if (m<3) then begin
    m:=m+12;
    a := a - 1 ;
  end;
  s := a DIV 100 ;
  JD := 1720996.5 - s + s DIV 4 + TRUNC(365.25*a) +
  TRUNC(30.6001*(M+1)) + j ;
  JD := JD-TRUNC(JD/7)*7;
  JS := TRUNC(JD) MOD 7;
  case (JS) of
    0: Dat:='mardi';
    1: Dat:='mercredi';
    2: Dat:='jeudi';
    3: Dat:='vendredi';
    4: Dat:='samedi';
  end;
end;

```

```

5: Dat:='dimanche';
6: Dat:='lundi';
end ;
writeln('Le ',j,'/',m,'/',a,' est un ',Dat);
end.

```

### **Solution 19 :**

Algorithme Calcul\_Sinus ;

Constantes

pi = 3.1416 ;

Variables

x, Rad, Sinus : réel ;

Sign : entier

Début

Ecrire('Calcul du sinus d'un angle.');

Ecrire('Introduisez l'angle (en radians) :');

Lire(x) ;

Rad  $\leftarrow$  x ;

Sign  $\leftarrow$  +1 ;

Si (Rad > 0) Alors Rad  $\leftarrow$  Rad - 2\*pi \* [Rad / (2\*pi)]

    Sinon Rad  $\leftarrow$  Rad + 2\*pi \* [-Rad / (2\*pi)] + 2\*pi

Si (Rad > pi) Alors début

    Rad  $\leftarrow$  Rad - pi ;

    Sign  $\leftarrow$  -1 ;

fin ;

Si (Rad > pi/2) Alors Rad  $\leftarrow$  pi - Rad ;

Si (Rad > pi/4) Alors début

    Rad  $\leftarrow$  pi/2 - Rad ;

    Sinus  $\leftarrow$  1 - Rad\*Rad/2 + Rad\*Rad\*Rad\*Rad/24 ;

fin

    Sinon Sinus  $\leftarrow$  Rad - Rad\*Rad\*Rad/6 + Rad\*Rad\*Rad\*Rad\*Rad/120 ;

Sinus  $\leftarrow$  Sinus \* Sign ;

Ecrire('sin(',x,')=' ,Sinus) ;

Fin.

Le programme Pascal :

program Calcul\_Sinus;

const

pi=3.1416;

var

x, Rad, Sinus : real;

```

Sign : shortint;
begin
  writeln('Calcul du sinus d'un angle. ');
  writeln('Introduisez l'angle (en radians): ');
  readln(x);
  Rad := x;
  Sign := +1;
  if (Rad > 0) then Rad := Rad - 2*pi * TRUNC(Rad / (2*pi))
    else Rad := Rad + 2*pi * TRUNC(-Rad / (2*pi)) + (2*pi);
  if (Rad > pi) then begin
    Rad := Rad - pi;
    Sign := -1 ;
  end;
  if (Rad > pi/2) then Rad := pi - Rad;
  if (Rad > pi/4) then begin
    Rad := pi/2 - Rad;
    Sinus := 1 - Rad*Rad/2 + Rad*Rad*Rad*Rad/24 ;
  end
  else Sinus := Rad - Rad*Rad*Rad/6 + Rad*Rad*Rad*Rad*Rad/120;
  Sinus := Sinus * Sign;
  writeln('sin(',x,')=',Sinus) ;
  readln
end.

```

### **Solution 20 :**

Algorithme facture ;

Variables

Pu, qtcom, tr, pl, tot, pap : réel ;

Début

Ecrire('Entrez le prix unitaire et la quantité commandée :')

Lire(pu, qtcom) ;

{ Calcul du total net }

$\text{tot} \leftarrow \text{pu} * \text{qtcom}$  ;

{ Calcul du prix de livraison }

Si (tot>500) Alors pl  $\leftarrow$  0

    Sinon pl  $\leftarrow$  tot \* 0,02 ;

{ Calcul de la réduction }

Si (tot>1000) Alors tr  $\leftarrow$  tot \* 0,10

    Sinon Si (tot>=200) Alors tr  $\leftarrow$  tot \* 0,05

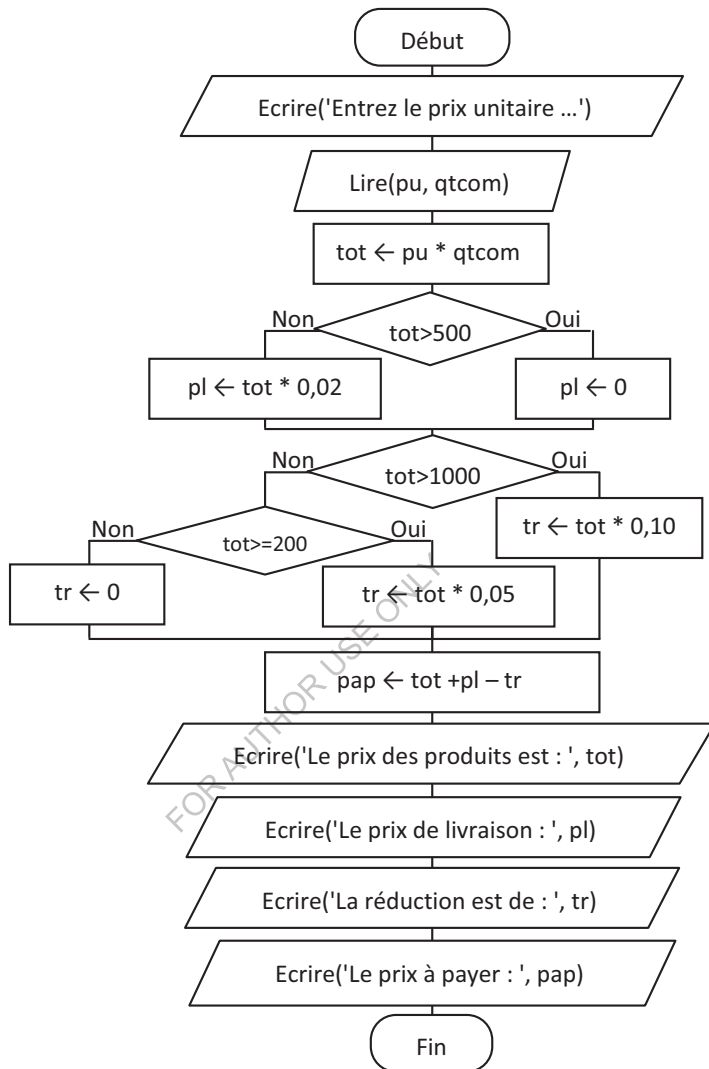
        Sinon tr  $\leftarrow$  0 ;

```
{ Calcul du prix à payer }
pap ← tot + pl – tr ;
{ Édition de la facture }
Ecrire('Le prix des produits est : ', tot) ;
Ecrire('Le prix de livraison : ', pl) ;
Ecrire('La réduction est de : ', tr) ;
Ecrire('Le prix à payer : ', pap) ;
Fin.
```

Le programme Pascal :

```
program facture ;
var
  Pu, qtcom, tr, pl, tot, pap : real ;
begin
  writeln('Entrez le prix unitaire et la quantité commandée :') ;
  readln(pu, qtcom) ;
  { Calcul du total net }
  tot := pu * qtcom ;
  { Calcul du prix de livraison }
  if (tot>500) then pl := 0
    else pl := tot * 0.02;
  { Calcul de la réduction }
  if (tot>1000) then tr := tot * 0.10
    else if (tot>=200) then tr := tot * 0.05
      else tr := 0;
  { Calcul du prix à payer }
  pap := tot + pl - tr ;
  { Edition de la facture }
  writeln('Le prix des produits est : ', tot) ;
  writeln('Le prix de livraison : ', pl) ;
  writeln('La réduction est de : ', tr) ;
  writeln('Le prix à payer : ', pap) ;
end.
```

L'organigramme :



**Solution 21 :**

Algorithme prix\_billet ;

Variables

jour, annee\_cours, annee\_naiss, age : entier ;

prix\_base, escompte, prix : réel ;

Début

Ecrire('Entrez le Num du jour (0.Samedi, 1.Dimanche,...) :) ;

Lire(jour) ;

```

Ecrire('Entrer l'année en cours :') ;
Lire(annee_cours) ;
Ecrire('Entrer l'année de naissance :') ;
Lire(annee_naiss) ;
age ← annee_cours – annee_naiss ;
Ecrire('Entrer le tarif de base :') ;
Lire(prix_base) ;
Si ((age<=15) OU (age>=65)) Alors
    Si ((jour >= 0) ET (jour <= 4)) Alors escompte ← 0.25 * prix_base
    Sinon escompte ← 0.10 * prix_base
    Sinon Si ((jour = 2) OU (jour = 5)) Alors escompte ← 0.15 * prix_base
    Sinon escompte ← 0 ;
prix ← prix_base – escompte ;
Ecrire('Age : ', age) ;
Ecrire('Escompte : ', escompte) ;
Ecrire('Prix : ', prix) ;

```

Fin.

Le programme Pascal :

```

program prix_billet ;
var
    jour, annee_cours, annee_naiss, age : integer ;
    prix_base, escompte, prix : real ;
begin
    writeln('Entrer le Num du jour (0.Samedi, 1.Dimanche,...) :') ;
    readln(jour) ;
    writeln('Entrer l'année en cours :') ;
    readln(annee_cours) ;
    writeln('Entrer l'année de naissance :') ;
    readln(annee_naiss) ;
    age := annee_cours - annee_naiss ;
    writeln('Entrer le tarif de base :');
    readln(prix_base) ;
    if ((age <= 15) OR (age >= 65)) then
        if ((jour >= 0) AND (jour <= 4)) then escompte := 0.25 * prix_base
        else escompte := 0.10 * prix_base
        else if ((jour = 2) or (jour = 5)) then escompte := 0.15 * prix_base
        else escompte := 0 ;
    prix := prix_base - escompte;
    writeln('Age : ', age) ;

```

```
writeln('Escompte : ', escompte);  
writeln('Prix : ', prix);  
end.
```

**Solution 22 :**

Algorithme ventes ;

Variables nb\_paire, pu, mnt\_achat, tr, pr\_net : réel ;

Début

Ecrire('Entrez le nombre de paires achetées ainsi que le prix unitaire :') ;

Lire(nb\_paire, pu) ;

mnt\_achat ← nb\_paire \* pu ;

Si (nb\_paire > 2) Alors

    Si (mnt\_achat ≤ 1000) Alors tr ← mnt\_achat \* 0,10

    Sinon tr ← mnt\_achat \* 0,20

Sinon Si (mnt\_achat < 500) Alors tr ← 0

    Sinon tr ← mnt\_achat \* 0,05 ;

pr\_net ← mnt\_achat – tr ;

Ecrire('Montant d'achat total : ', mnt\_achat) ;

Ecrire('Montant de la réduction : ', tr) ;

Ecrire('Prix net à payer : ', pr\_net) ;

Fin.

Le programme Pascal :

program ventes ;

var nb\_paire, pu, mnt\_achat, tr, pr\_net : real ;

begin

writeln('Entrez le nombre de paires achetées ainsi que le prix unitaire :') ;

readln(nb\_paire, pu) ;

mnt\_achat := nb\_paire \* pu ;

if (nb\_paire > 2) then

    if (mnt\_achat ≤ 1000) then tr := mnt\_achat \* 0.10

    else tr := mnt\_achat \* 0.20

else if (mnt\_achat < 500) then tr := 0

    else tr := mnt\_achat \* 0.05 ;

pr\_net := mnt\_achat - tr ;

writeln('Montant d'achat total : ', mnt\_achat:3:2) ;

writeln('Montant de la réduction : ', tr:3:2) ;

writeln('Prix net à payer : ', pr\_net:3:2) ;

end.



## Chapitre 4 : Les boucles

### 1. Introduction

La structure alternative est insuffisante pour exprimer des algorithmes dont la longueur peut varier selon les circonstances. Lorsqu'on est amené dans un algorithme à répéter une opération ou une séquence d'opérations plusieurs fois, il peut être plus commode d'utiliser des structures répétitives (structures itératives) appelées les boucles.

Dans une boucle, le nombre de répétitions peut être connu, fixé à l'avance, comme il peut dépendre d'une condition permettant l'arrêt et la sortie de cette boucle.

Il existe trois types de boucles fondamentaux : la boucle Tant que, la boucle Répéter et la boucle Pour.

### 2. La boucle Tant que

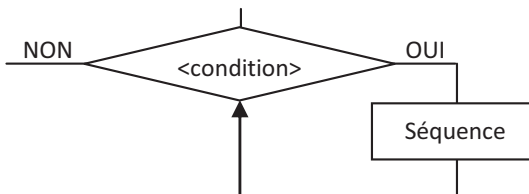
Cette structure permet la répétition d'une séquence d'opérations tant que la condition est satisfaite (VRAI). Quand la condition devient fausse, la boucle est achevée. Le nombre de répétition de la séquence d'opérations n'est pas connu d'avance. La condition est testée avant la première exécution de la séquence définie. Si la condition est fausse dès le départ, la séquence d'opérations n'est jamais exécutée.

Format général : Tant que <Condition> Faire <Séquence>

Souvent, la condition correspond à une expression logique qui doit être évaluée (il faut donc veiller à sa valeur lors de l'entrée dans la boucle) : si sa valeur est VRAI, le corps de la boucle est exécuté, puis l'expression logique est réévaluée (il faut donc qu'elle puisse changer de valeur pour sortir de la boucle et éviter le cas de la boucle infinie), et quand elle aura la valeur FAUX, la boucle est achevée.

Il est à noter qu'il est préférable d'exprimer l'expression logique sous la forme NON (condition d'arrêt). Il est en effet plus simple de déterminer les raisons d'arrêter le processus répétitif que celles de continuer.

La boucle Tant que peut être représentée dans un organigramme comme suit :



Voyons l'exemple suivant qui permet d'afficher les valeurs de 1 à 10 :

$i \leftarrow 1$  ;

```
Tant que (i <= 10) Faire
  début
    Ecrire(i) ;
    i ← i + 1 ;
  fin ;
```

En Pascal, la boucle Tant que s'exprime comme suit:

```
i := 1 ;
while (i <= 10) do
  begin
    write(i);
    i := i + 1 ;
  end ;
```

Comme vous remarquez, la séquence d'instructions est entourée comme d'habitude par un begin et un end qui peuvent être omis lorsque la séquence se réduit à une seule instruction.

Voyons maintenant des exemples en Pascal sur des boucles infinies :

```
while (TRUE) do write(X);
while (10 > 3) do write(X);
```

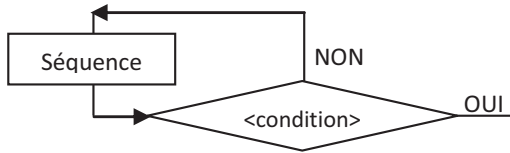
Ces deux boucles permettent d'afficher la valeur de X indéfiniment, sans jamais quitter le programme. Attention ! La boucle infinie est considérée comme une erreur logique qui n'est pas déclarée par le compilateur, et c'est le programmeur qui doit la détecter.

### 3. La boucle Répéter

Cette structure permet la répétition d'une ou plusieurs opérations jusqu'à ce qu'une condition soit vérifiée. Le nombre de répétition de la séquence d'opérations n'est pas connu d'avance. Dans cette structure, contrairement à la structure Tant que, lorsque la condition est vérifiée, le traitement s'arrête. De plus, la condition n'est testée qu'après une première exécution de la séquence définie. Les opérations sont donc exécutées au moins une fois, même si la condition est vérifiée dès le départ.

Format général : Répéter <Séquence> Jusqu'à <Condition>

L'expression logique est évaluée après l'exécution du corps de la boucle: si sa valeur est FAUX, le corps de la boucle est exécuté à nouveau, puis l'expression logique est réévaluée (il faut donc qu'elle puisse changer de valeur pour sortir de la boucle et éviter le cas de la boucle infinie), et quand elle aura la valeur VRAI, la boucle est achevée. La boucle Répéter peut être représentée dans un organigramme comme suit :



Reprenons l'exemple permettant d'afficher les valeurs de 1 à 10 :

```

i ← 1 ;
Répéter
  Ecrire(i) ;
  i ← i + 1 ;
Jusqu'à (i > 10) ;
  
```

En Pascal, la boucle Répéter s'exprime comme suit:

```

i := 1 ;
repeat
  write(i) ;
  i := i + 1 ;
until (i > 10)
  
```

En comparant la boucle Répéter avec la boucle Tant que, on peut trouver plusieurs points de différence, parmi lesquels on cite :

- Pour la boucle Répéter, la séquence d'opérations n'est pas obligatoirement entourée par début et fin, même s'il existe plusieurs opérations.
- Pour la boucle Tant que, l'expression logique exprime les raisons de continuer, et dans la boucle Répéter, l'expression logique exprime les raisons d'arrêter ; donc le test effectué pour la boucle Tant que est l'inverse de celui utilisé pour la boucle Répéter. C'est-à-dire que si la condition est VRAI, cela implique l'exécution de la séquence pour la boucle Tant que, et l'arrêt d'exécution pour la boucle Répéter.
- La séquence de la boucle Répéter est exécutée au moins une fois. Par contre, la séquence de la boucle Tant que peut ne jamais s'exécuter.

#### 4. La boucle Pour

Cette structure permet de répéter l'exécution d'une séquence d'opérations pour toutes les valeurs d'une variable, dite variable de contrôle, à partir d'une valeur initiale à une valeur finale. Le nombre de répétition de la séquence d'opérations est connu à l'avance.

Format général :

```

Pour <Compteur> ← <Valeur initiale> à <Valeur finale> pas de <Incrément>
Faire <Séquence>
  
```

La séquence est une ou un ensemble d'opérations qui doivent être exécutées plusieurs fois. On précise entre les deux mots Pour et Faire comment seront contrôlées les répétitions. On y définit une variable appelée variable de contrôle (Compteur), ainsi que les valeurs que prendra cette variable : une première valeur (Valeur initiale) indiquée après  $\leftarrow$ , et une dernière valeur (Valeur finale) indiquée après le à. La variable de contrôle est initialisée à la première valeur. Avant chaque exécution du corps de la boucle, la valeur de la variable de contrôle est comparée à la valeur finale. Si la variable de contrôle ne dépasse pas cette valeur, on exécute le corps de la boucle, sinon la boucle est achevée. Après chaque exécution du corps de la boucle, la variable de contrôle est augmentée d'une unité (Incrément ou Pas de progression).

La variable de contrôle (Compteur) est de type entier. La valeur initiale et la valeur finale sont des constantes ou des variables de type entier. Ces deux valeurs permettent de calculer le nombre de répétition de la séquence :  $\text{nbr} = \text{valeur finale} - \text{valeur initiale} + 1$  (dans le cas où le Pas est égal à 1). L'incrément est la valeur d'augmentation progressive du compteur. La valeur par défaut du Pas de progression est de 1 (dans le cas où on ne le précise pas).

Si la valeur initiale est égale à la valeur finale, la séquence définie est exécutée une seule fois. Par contre, la séquence n'est jamais exécutée dans deux cas : Si la valeur initiale est supérieure à la valeur finale alors que la valeur de l'incrément est positive, ou bien si la valeur initiale est inférieure à la valeur finale alors que la valeur de l'incrément est négative.

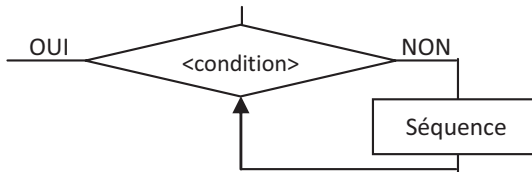
#### Remarques :

- Dans une boucle Pour, le compteur (variable de contrôle) peut être utilisé, mais il ne doit jamais être modifié à l'intérieur de la boucle.
- Par convention, lorsque l'incrément est égal à 1, il n'est pas indiqué.
- La première valeur, la dernière valeur et l'incrément peuvent être des expressions numériques.

La structure Pour est une simplification de la structure suivante :

```
<Compteur>  $\leftarrow$  <Valeur initiale> ;  
Tant que <Compteur> <= <Valeur finale> Faire début  
  <Séquence> ;  
  <Compteur>  $\leftarrow$  <Compteur> + <Incrément> ;  
fin ;
```

La boucle Pour peut être représentée dans un organigramme comme suit :



Gardons le même exemple permettant d'afficher les valeurs de 1 à 10 :

Pour  $i \leftarrow 1$  à 10 Faire Ecrire( $i$ ) ;

En Pascal, la boucle for s'exprime comme suit :

for  $i := 1$  to 10 do write( $i$ ) ;

La boucle for  $i := 10$  downto 1 do write( $i$ ) ; affiche les valeurs de 10 à 1.

### Remarques :

- La notion de Pas de progression en Pascal, par rapport à d'autres langages, n'existe pas explicitement. Cette progression est définie implicitement par les expressions to (pour un Pas de 1) et downto (pour un Pas de -1). Donc, il est impossible de traduire en Pascal les boucles avec des incréments différents de 1 et -1. On note aussi que dans les boucles while et for, si la séquence contient juste une seule instruction, les mots clés begin et end, qui délimitent la séquence, ne sont pas obligatoires.
- L'instruction for est redondante, car elle peut être exprimée par les boucles while ou repeat, mais il est recommandé d'utiliser la boucle for chaque fois que c'est possible.
- Pascal permet d'utiliser les formes suivantes :
  - for car := 'a' to 'z' do ... ;
  - for car := 'z' downto 'a' do ... ;
  - for bl := false to true do ... ;
  - for bl := true downto false do ... ;
 avec car de type char, et bl de type boolean.
- Le compteur ne doit pas être modifié à l'intérieur de la boucle for, mais la modification de la valeur initiale et la valeur finale n'a aucun effet sur le nombre de répétition de la séquence d'instructions.

## 5. Les boucles imbriquées

Les boucles peuvent être imbriquées les unes dans les autres. Une boucle Tant que peut contenir une autre boucle Tant que, une autre boucle Répéter, ou une autre boucle Pour, et vice versa. De plus, une boucle peut contenir une autre boucle, qui elle-même peut contenir une autre boucle, et ainsi de suite.

L'algorithme suivant permet d'afficher toutes les tables de multiplication de 1 jusqu'à 10.

Algorithme boucles\_imbriquées ;

Variables

i, j : entier ;

Début

Pour i  $\leftarrow$  1 à 10 Faire

Pour j  $\leftarrow$  1 à 10 Faire Ecrire(i, '\*', j, '=', i\*j) ;

Fin.

Lorsque i = 1, la deuxième boucle s'exécute pour j = 1 à 10. On obtient ainsi la table de multiplication de 1. Le compteur i passe de 1 à 2, et on exécute à nouveau la deuxième boucle pour j = 1 à 10. On obtient ainsi la table de multiplication de 2. Et ainsi de suite, jusqu'à i = 10.

## 6. Exercices corrigés

### 6.1. Exercices

#### Exercice 1 :

Ecrire trois algorithmes permettant de calculer la somme de la suite des nombres 1, 2, 3, ..., n (n est un entier qui doit être positif lu à partir du clavier). Chaque algorithme doit utiliser l'une des boucles : Tant que, Répéter et Pour. Traduire les algorithmes en Pascal. Représentez chaque algorithme par l'organigramme correspondant.

#### Exercice 2 :

Ecrire un algorithme permettant de calculer la factorielle d'un nombre entier n strictement positif, sachant que :  $n! = 1*2*3*\dots*n$ , et que  $0! = 1$ . Par convention, n! se lit "n factorielle". La factorielle d'un nombre négatif n'existe pas. Il est donc nécessaire que l'algorithme vérifie si le nombre donné n'est pas négatif. Traduire l'algorithme en Pascal.

#### Exercice 3 :

Ecrire un algorithme permettant de calculer la  $n^{\text{ième}}$  puissance entière d'un entier x par multiplications successives du nombre par lui-même. Ici, le nombre de répétition (n) de l'instruction de multiplication est connu. Traduire l'algorithme en Pascal.

#### Exercice 4 :

Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3, jusqu'à ce que la réponse convienne. Traduire l'algorithme en Pascal.

#### Exercice 5 :

Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus

grand ! » si le nombre est inférieur à 10. Traduire l'algorithme en Pascal.

**Exercice 6 :**

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur donne le nombre 17, l'algorithme affichera les nombres de 18 à 27. Traduire l'algorithme en Pascal.

**Exercice 7 :**

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur donne le nombre 7) :

Table de 7 :

$$7 \times 1 = 7$$

$$7 \times 2 = 14$$

...

$$7 \times 10 = 70$$

Traduire l'algorithme en Pascal.

**Exercice 8 :**

Etant donnés deux nombres entiers  $m$  et  $n$  positifs ou nuls, on demande d'en calculer le PGCD (le plus grand diviseur commun). L'algorithme d'Euclide permet de résoudre ce problème en prenant d'abord le reste de la division de  $m$  par  $n$ , puis le reste de la division de  $n$  par ce premier reste, etc., jusqu'à ce qu'on trouve un reste nul. Le dernier diviseur utilisé est le PGCD de  $m$  et  $n$ .

Pour  $m=1386$  et  $n=140$ , on a successivement :  $1386 = 140 * 9 + 126$ ,  $140 = 126 * 1 + 14$ ,  $126 = 14 * 9 + 0$ . Donc, le PGCD de 1386 et 126 est bien 14. Remarquons que par définition, si l'un des nombres est nul, l'autre nombre est le PGCD. Notons aussi que l'ordre de  $m$  et  $n$  n'a pas d'importance.

Ecrire un algorithme permettant de déterminer le PGCD de  $m$  et  $n$ . Traduire l'algorithme en Pascal. Déroulez l'algorithme pour  $m = 24$  et  $n = 18$ .

**Exercice 9 :**

Ecrire un algorithme permettant de saisir une série de nombres réels positifs. On termine la saisie par un nombre négatif qui ne sera pas tenu en compte lors des calculs. Puis, on propose indéfiniment (en boucle) à l'utilisateur, par l'intermédiaire d'une sorte de menu à choix multiple, d'afficher la valeur minimale, la valeur maximale, la somme ou la moyenne des nombres entrés, ou encore de quitter le programme. Traduire l'algorithme en Pascal.

**Exercice 10 :**

Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dit ensuite quel était le plus grand parmi ces 20 nombres. Voici un exemple d'exécution :

Entrez le nombre numéro 1 : 12

Entrez le nombre numéro 2 : 14

...

Entrez le nombre numéro 20 : 6

Le plus grand de ces nombre est : 14

Traduire l'algorithme en Pascal.

Modifiez ensuite l'algorithme pour qu'il affiche en plus, en quelle position avait été saisi ce nombre. Traduire l'algorithme en Pascal.

**Exercice 11 :**

Réécrire l'algorithme et le programme précédents, mais cette fois-ci, on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un zéro. Le zéro lui aussi est tenu en compte lors des calculs.

**Exercice 12 :**

Ecrire un algorithme permettant de lire la suite des prix (en dinars entiers positifs) des achats d'un client. La suite se termine par zéro. Calculer la somme qu'il doit, lire la somme qu'il paye, et simuler la remise de la monnaie en affichant les textes « 10 DA », « 5 DA » et « 1 DA » autant de fois qu'il y a de coupures de chaque sorte à rendre. Traduire l'algorithme en Pascal.

**Exercice 13 :**

Ecrire un algorithme qui calcule itérativement le  $n^{\text{ième}}$  terme de la suite de Fibonacci définie comme suit : Si  $n = 0$  alors  $F_n = 0$ . Si  $n = 1$  alors  $F_n = 1$ . Si  $n > 1$  alors  $F_n = F_{n-1} + F_{n-2}$ . Traduire l'algorithme en Pascal.

**Exercice 14 :**

Ecrire un algorithme permettant de calculer la somme des chiffres d'un nombre entier positif fourni par l'utilisateur. Par exemple, la somme des chiffres du nombre 1945 est égale à 19. Traduire l'algorithme en Pascal.

**Exercice 15 :**

Qu'affiche le programme Pascal suivant :

```
program affichage ;
```

```
label 200;
```

```
var
```

```
  i : boolean ;
```

```
  j : char;
```

```
  k : integer;
```



```
begin
  for i := FALSE to TRUE do begin
    if NOT i then while i do goto 200
    else if i AND (5 < 6) then writeln('bon courage')
    else for j := 'd' to 'b' do writeln('bonjour');
  for k := 3 to 5 do
    if FALSE OR i then
      begin if k = 3 then writeln('merci'); end
    else writeln(k-2);
  200:writeln('***');
end ;
end.
```

## 6.2. Corrigés

### Solution 1 :

L'algorithme avec la boucle Tant que :

Algorithme somme\_suite1 ;

Variables

i, n, somme : entier ;

Début

Ecrire('Donnez un nombre :)') ;

Lire(n) ;

Si (n > 0) Alors début

somme ← 0 ;

i ← 1 ;

Tant que (i ≤ n) Faire début

somme ← somme + i ;

i ← i + 1 ;

fin ;

Ecrire('La somme de la suite = ', somme) ;

fin

Sinon Ecrire('La valeur du nombre doit être positive.') ;

Fin.

L'algorithme avec la boucle Répéter :

Algorithme somme\_suite2 ;

Variables

i, n, somme : entier ;

Début

Ecrire('Donnez un nombre :)') ;

Lire(n) ;

```
Si (n > 0) Alors début
    somme ← 0 ;
    i ← 1 ;
    Répéter
        somme ← somme + i ;
        i ← i + 1 ;
    Jusqu'à (i > n) ;
    Ecrire('La somme de la suite = ', somme) ;
Fin
Sinon Ecrire('La valeur du nombre doit être positive.') ;
Fin.
```

L'algorithme avec la boucle Pour :

Algorithme somme\_suite3 ;

Variables

i, n, somme : entier ;

Début

Ecrire('Donnez un nombre :)') ;

Lire(n) ;

Si (n > 0) Alors début

somme ← 0 ;

Pour i ← 1 à n Faire somme ← somme + i ;

Ecrire('La somme de la suite = ', somme) ;

fin

Sinon Ecrire('La valeur du nombre doit être positive.') ;

Fin.

Les programmes Pascal correspondant aux algorithmes précédents sont les suivants :

Boucle While :

program somme\_suite1 ;

var

i, n, somme : integer ;

begin

writeln('Donnez un nombre :)') ;

readln(n) ;

if (n > 0) then begin

somme := 0;

i := 1 ;

while (i <= n) do begin

somme := somme + i ;

```
    i := i + 1 ;  
end ;  
writeln('La somme de la suite = ', somme) ;  
end  
else writeln('La valeur du nombre doit être positive.') ;  
end.
```

*Boucle Repeat :*

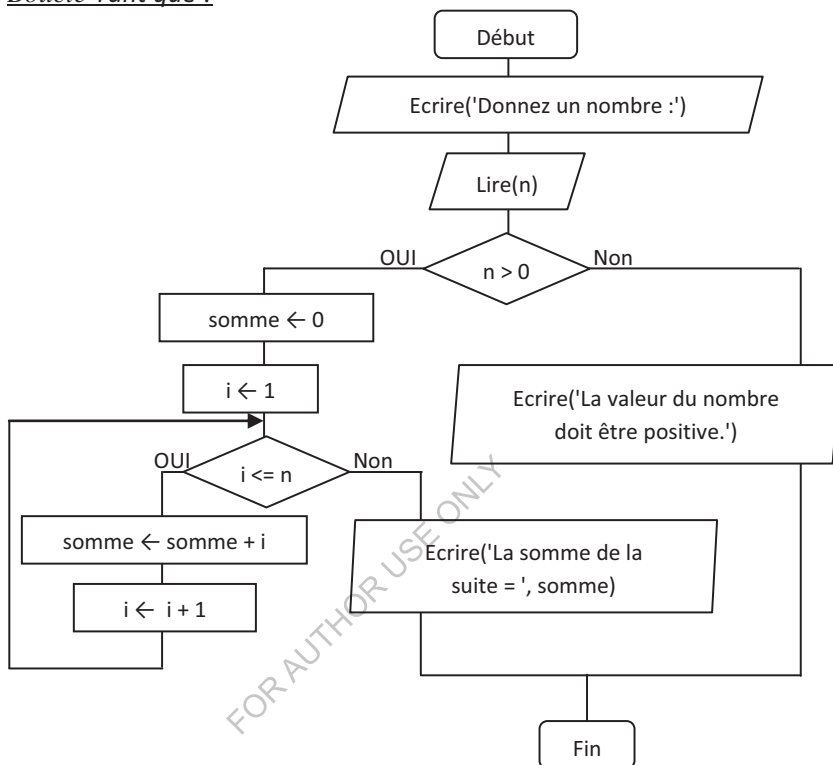
```
program somme_suite2;  
var  
    i, n, somme : integer ;  
begin  
    writeln('Donnez un nombre :') ;  
    readln(n) ;  
    if (n > 0) then begin  
        somme := 0;  
        i := 1 ;  
        repeat  
            somme := somme + i ;  
            i := i + 1 ;  
        until (i > n) ;  
        writeln('La somme de la suite = ', somme) ;  
    end  
    else writeln('La valeur du nombre doit être positive.') ;  
end.
```

*Boucle For :*

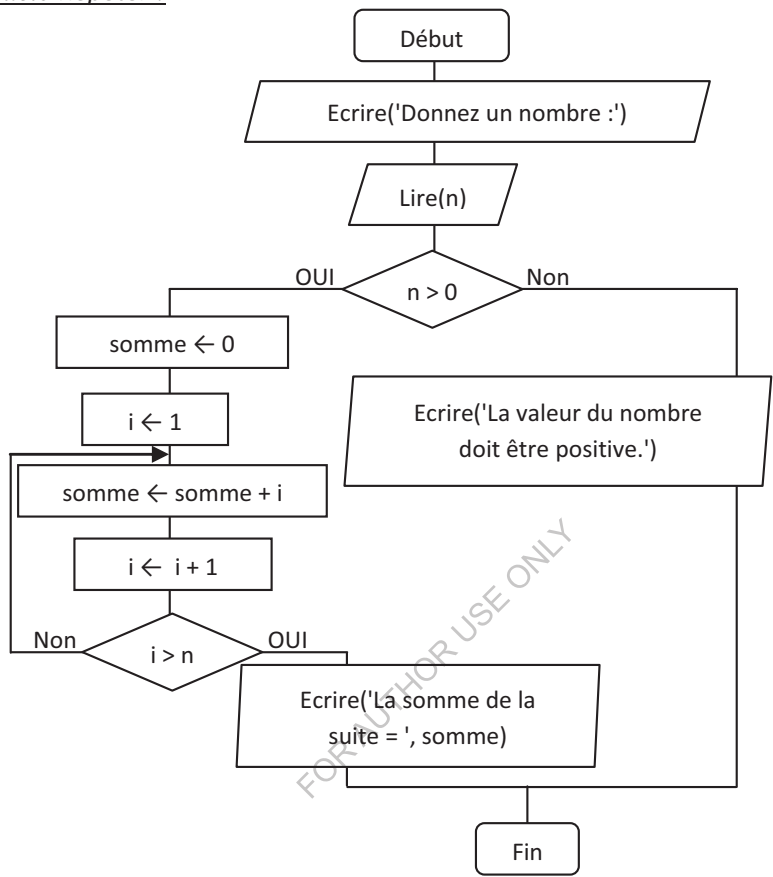
```
program somme_suite3;  
var  
    i, n, somme : integer ;  
begin  
    writeln('Donnez un nombre :') ;  
    readln(n) ;  
    if (n > 0) then begin  
        somme := 0;  
        for i := 1 to n do somme := somme + i ;  
        writeln('La somme de la suite = ', somme) ;  
    end  
    else write('La valeur du nombre doit être positive.') ;  
end.
```

Les organigrammes correspondant aux trois algorithmes sont les suivants :

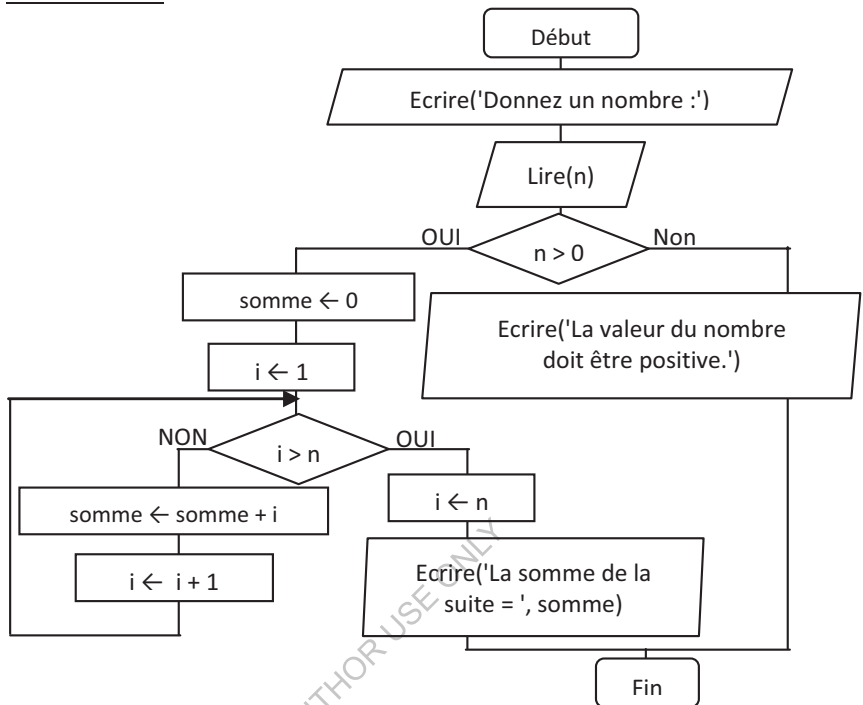
Boucle Tant que :



Boucle Répéter :



Boucle Pour :



**Solution 2 :**

Algorithme factorielle\_n ;

Variables

i, n, factorielle : entier ;

Début

Ecrire('Entez un nombre :') ;

Lire(n) ;

Si (n < 0) Alors Ecrire('La factorielle d'un nombre négatif n'existe pas.')

Sinon début

factorielle ← 1 ;

i ← 1 ;

Tant que (i ≤ n) Faire début

factorielle ← factorielle \* i ;

i ← i + 1

fin ;

Ecrire('La factorielle de ', n, ' = ', factorielle) ;

fin ;

Fin.

Le programme Pascal :

```
program factorielle_n ;
var
  i, n, factorielle : integer ;
begin
  writeln('Enter un nombre :)') ;
  readln(n) ;
  if (n < 0) then writeln('la factorielle d'un nombre négatif n'existe pas.')
  else begin
    factorielle := 1 ;
    i := 1 ;
    while (i <= n) do begin
      factorielle := factorielle * i ;
      i := i + 1 ;
    end ;
    writeln('La factorielle de ', n, ' = ', factorielle) ;
  end ;
end.
```

**Solution 3 :**

Algorithme puissance ;

Variables

n, i : entier ;  
x, puiss : réel ;

Début

Ecrire('Calcul de la puissance n ième du réel x par multiplications successives.') ;

Ecrire('Introduisez le nombre réel x :)') ;

Lire(x) ;

Ecrire('Introduisez l'exposant de x (entier positif :)') ;

Lire(n) ;

puiss  $\leftarrow$  1 ;

Pour i  $\leftarrow$  1 à n Faire puiss  $\leftarrow$  puiss \* x ;

Ecrire('La puissance ', n, ' ième de ', x, ' est ', puiss) ;

Fin.

Le programme Pascal :

```
program puissance;
```

```
var
```

```
  n, i : integer;
  x, puiss: real;
```

```
begin
```

```
writeln('Calcul de la puissance n ième du réel x par multiplications successives.');
```

```
writeln('Introduisez le nombre réel x :');
```

```
readln(x);
```

```
writeln('Introduisez l'exposant de x (entier positif) :');
```

```
readln(n);
```

```
puiss := 1;
```

```
for i := 1 to n do puiss := puiss * x ;
```

```
writeln('La puissance ',n,' ième de ', x,' est ',puiss) ;
```

```
end.
```

#### **Solution 4 :**

Algorithme valeur\_1\_3 ;

Variables

n : entier ;

Début

Répéter

Ecrire('Entrez un nombre entre 1 et 3 :');

Lire(n) ;

Si ((n < 1) OU (n > 3)) Alors Ecrire('Saisie erronée. Recommencez')

Sinon Ecrire('Nombre accepté.');

Jusqu'à (n >= 1) ET (n <= 3);

Fin.

Le programme Pascal :

```
program valeur_1_3 ;
```

```
var
```

```
n : integer ;
```

```
begin
```

```
repeat
```

```
writeln('Entrez un nombre entre 1 et 3 :');
```

```
readln(n) ;
```

```
if ((n < 1) OR (n > 3)) then writeln('Saisie erronée. Recommencez')
```

```
else writeln('Nombre accepté.');
```

```
until (n >= 1) and (n <= 3);
```

```
end.
```

#### **Solution 5 :**

Algorithme valeur\_10\_20 ;

Variables

n : entier ;

Début

Répéter



```
Ecrire('Entrez un nombre entre 10 et 20 :') ;  
Lire(n) ;  
Si (n < 10) Alors Ecrire('Plus grand !')  
    Sinon Si (n > 20) Alors Ecrire('Plus petit !')  
        Sinon Ecrire('Nombre accepté.');
```

Jusqu'à (n >= 10) ET (n <= 20);

Fin.

Le programme Pascal :

```
program valeur_10_20 ;  
var  
    n : integer ;  
begin  
    repeat  
        writeln('Entrez un nombre entre 10 et 20 :') ;  
        readln(n) ;  
        if (n < 10) then writeln('Plus grand !')  
            else if (n > 20) then writeln('Plus petit !')  
                else writeln('Nombre accepté.');        until (n >= 10) AND (n <= 20);  
    end.
```

**Solution 6 :**

Algorithme afficher\_10\_suivants ;

Variables

n, i : entier ;

Début

```
Ecrire('Entrez un nombre :') ;  
Lire(n) ;  
Ecrire('Les 10 nombres suivants sont :') ;  
Pour i ← n + 1 à n + 10 Faire Ecrire(i) ;
```

Fin.

Le programme Pascal :

```
program afficher_10_suivants ;  
var n, i : integer;  
begin  
    writeln('Entrez un nombre :') ;  
    readln(n) ;  
    writeln('Les 10 nombres suivants sont :') ;  
    for i := n + 1 to n + 10 do writeln(i) ;  
end.
```

**Solution 7 :**

Algorithme table\_multiplication ;

Variables

n, i : entier ;

Début

Ecrire('Entrez un nombre :') ;

Lire(n) ;

Ecrire('La table de multiplication de ',n, ' est :') ;

Pour i  $\leftarrow$  1 à 10 Faire Ecrire(n, ' x ',i, ' = ', n\*i) ;

Fin.

Le programme Pascal :

program table\_multiplication ;

var n, i : integer ;

begin

writeln('Entrez un nombre :') ;

readln(n) ;

writeln('La table de multiplication de ',n, ' est :') ;

for i := 1 to 10 do writeln(n, ' x ',i, ' = ', n\*i) ;

end.

**Solution 8 :**

Algorithme Plus\_Grand\_Commune\_Diviseur ;

Variables m, n, a, b, r, PGCD : entier ;

Début

Ecrire('Nous allons calculer le PGCD de deux nombres entiers.') ;

Ecrire('Introduisez le premier nombre :') ;

Lire(m) ;

Ecrire('Introduisez le deuxième nombre :') ;

Lire(n) ;

a  $\leftarrow$  m ;

b  $\leftarrow$  n ;

r  $\leftarrow$  a % b ;

Tant que NON(r = 0) Faire début

a  $\leftarrow$  b ;

b  $\leftarrow$  r ;

r  $\leftarrow$  a % b ;

fin ;

PGCD  $\leftarrow$  b ;

Ecrire('Le PGCD de ',m,' et ',n,' est ',PGCD) ;

Fin.

Le programme Pascal :

```
program Plus_Grand_Commune_Diviseur ;
var
  m, n, a, b, r, PGCD : integer;
begin
  writeln('Nous allons calculer le PGCD de deux nombres entiers. ');
  writeln('Introduisez le premier nombre : ');
  readln(m);
  writeln('Introduisez le deuxième nombre : ');
  readln(n);
  a := m;
  b := n;
  r := a mod b;
  while NOT(r = 0) do begin
    a := b ;
    b := r ;
    r := a mod b ;
  end ;
  PGCD := b ;
  writeln('Le PGCD de ',m,' et ',n,' est ',PGCD) ;
end.
```

Pour le déroulement de ce programme, on va noter les instructions du programme principal comme suit :

Instruction	Notation
writeln('Nous allons calculer le PGCD de deux nombres entiers. ');	P1
writeln('Introduisez le premier nombre : ');	P2
readln(m);	P3
writeln('Introduisez le deuxième nombre : ');	P4
readln(n);	P5
a := m;	P6
b := n;	P7
r := a mod b ;	P8
a := b ;	P9
b := r ;	P10
r := a mod b ;	P11
PGCD := b ;	P12
writeln('Le PGCD de ',m,' et ',n,' est ',PGCD) ;	P13

Le tableau suivant correspond au schéma d'évolution d'état des variables, instruction par instruction :

<b>Variable Instruction</b>	<b>m</b>	<b>n</b>	<b>a</b>	<b>b</b>	<b>r</b>	<b>NOT(r = 0)</b>	<b>PGCD</b>
P1							
P2							
P3	24						
P4							
P5		18					
P6			24				
P7				18			
P8					6	TRUE	
P9			18				
P10				6			
P11					0	FALSE	
P12							6
P13	24	18					6

**Solution 9 :**

Algorithme calculs\_menu ;

Variables

i, choix : entier ;

s, n, moyenne, minimum, maximum : réel ;

Début

Ecrire('Entrez successivement des nombres positifs.') ;

Ecrire('Entrez un nombre négatif pour finir.') ;

Lire(n) ;

Si (n >= 0) Alors début

s ← n ;

minimum ← n ;

maximum ← n ;

i ← 1 ;

Répéter

Lire(n);

Si (n >= 0) Alors début

Si (n < minimum) Alors minimum ← n ;

Si (n > maximum) Alors maximum ← n ;

s ← s + n ;

i ← i + 1 ;

fin ;

Jusqu'à (n < 0) ;

moyenne ← s/i ;

### Répéter

```
Ecrire('Choisissez entre :') ;
Ecrire('1 : minimum') ;
Ecrire('2 : maximum') ;
Ecrire('3 : somme') ;
Ecrire('4 : moyenne') ;
Ecrire('0 : stop') ;
Ecrire('Entrez votre choix :') ;
Lire(choix) ;
```

### Cas choix de

```
1 : Ecrire('Le minimum est : ', minimum) ;
2 : Ecrire('Le maximum est : ', maximum) ;
3 : Ecrire('La somme est : ', s) ;
4 : Ecrire('La moyenne est : ', moyenne) ;
0 : début fin
Sinon Ecrire('Choix non accepté.') ;
```

```
fin ;
```

```
Jusqu'à (choix = 0) ;
```

```
fin ;
```

Fin.

### Le programme Pascal :

```
program calculs_menu ;
```

```
var
```

```
  i, choix : integer ;
```

```
  s, n, moyenne, minimum, maximum : real ;
```

```
begin
```

```
  writeln('Entrez successivement des nombres positifs :') ;
```

```
  writeln('Entrez un nombre négatif pour finir.') ;
```

```
  readln(n);
```

```
  if (n >= 0) then begin
```

```
    s := n;
```

```
    minimum := n;
```

```
    maximum := n;
```

```
    i := 1;
```

```
    repeat
```

```
      readln(n);
```

```
      if (n >= 0) then begin
```

```
        if (n < minimum) then minimum := n ;
```

```
        if (n > maximum) then maximum := n ;
```

```

    s := s + n;
    i := i + 1;
end;
until (n < 0);
moyenne := s/i;
repeat
    writeln('Choisissez entre :') ;
    writeln('1 : minimum') ;
    writeln('2 : maximum') ;
    writeln('3 : somme') ;
    writeln('4 : moyenne') ;
    writeln('0 : stop') ;
    writeln('Entrez votre choix :') ;
    readln(choix) ;
    case (choix) of
        1 : writeln('Le minimum est : ', minimum) ;
        2 : writeln('Le maximum est : ', maximum) ;
        3 : writeln('La somme est : ', s) ;
        4 : writeln('La moyenne est : ', moyenne) ;
        0 : begin end
        else writeln('Choix non accepté.') ;
    end ;
until (choix=0) ;
end;
end.
```

### **Solution 10 :**

Algorithme plus\_grand\_nombre ;

Variables

n, i, PG : entier ;

Début

PG  $\leftarrow$  0 ;

Pour i  $\leftarrow$  1 à 20 Faire début

    Ecrire('Entrez un nombre :') ;

    Lire(n) ;

    Si (i = 1) OU (n > PG) Alors PG  $\leftarrow$  n ;

fin ;

    Ecrire('Le plus grand nombre était : ', PG) ;

Fin.

Une version améliorée :

Algorithme plus\_grand\_nombre2 ;

Variables

n, i, PG, pos : entier ;

Début

PG  $\leftarrow$  0 ;

Pour i  $\leftarrow$  1 à 20 Faire début

Ecrire('Entrez un nombre :') ;

Lire(n) ;

Si (i = 1) OU (n > PG) Alors début

PG  $\leftarrow$  n ;

pos  $\leftarrow$  i ;

fin ;

fin ;

Ecrire('Le plus grand nombre était : ', PG) ;

Ecrire('C'est le nombre numéro : ', pos) ;

Fin.

Le programme Pascal :

program plus\_grand\_nombre ;

var

n, i, PG : integer ;

begin

PG := 0 ;

for i := 1 to 20 do begin

writeln('Entrez un nombre :') ;

readln(n) ;

if ((i = 1) OR (n > PG)) then PG := n ;

end;

writeln('Le plus grand nombre était : ', PG) ;

end.

La version améliorée :

program plus\_grand\_nombre2 ;

var

n, i, PG, pos : integer ;

begin

PG := 0 ;

for i := 1 to 20 do begin

writeln('Entrez un nombre :') ;

readln(n) ;

```

    if ((i = 1) OR (n > PG)) then begin
        PG := n ;
        pos := i ;
    end;
end;
writeln('Le plus grand nombre était : ', PG) ;
writeln('C'est le nombre numéro : ', pos) ;
end.

```

**Solution 11 :**

Algorithme plus\_grand\_nombre2 ;

Variables

n, i, PG, pos : entier ;

Début

PG ← 0 ;

i ← 0 ;

Répéter

i ← i + 1 ;

Ecrire('Entrez un nombre :') ;

Lire(n) ;

Si (i = 1) OU (n > PG) Alors début

PG ← n ;

pos ← i ;

fin ;

Jusqu'à (n = 0)

Ecrire('Le plus grand nombre était : ', PG) ;

Ecrire('C'est le nombre numéro : ', pos) ;

Fin.

Le programme Pascal :

program plus\_grand\_nombre2 ;

var

n, i, PG, pos : integer ;

begin

PG := 0 ;

i := 0 ;

repeat

i := i + 1;

writeln('Entrez un nombre :') ;

readln(n) ;

if ((i = 1) OR (n > PG)) then begin



```

    PG := n ;
    pos := i ;
end;
until (n=0);
writeln('Le plus grand nombre était : ', PG) ;
writeln('C'est le nombre numéro : ', pos) ;
end.

```

## **Solution 12 :**

### **Solution 12.1**

Algorithme Achat ;

Variables

prix, somme, M, Reste, Nb10D, Nb5D : entier ;

Début

prix  $\leftarrow$  1 ;

somme  $\leftarrow$  0 ;

Tant que (prix  $\neq$  0) Faire début

    Ecrire('Entrez le prix :') ;

    Lire(prix) ;

    somme  $\leftarrow$  somme + prix ;

fin ;

Ecrire('Vous devez : ', somme, ' DA') ;

Ecrire('Montant versé :') ;

Lire(M) ;

Reste  $\leftarrow$  M – somme ;

Nb10D  $\leftarrow$  0 ;

Tant que (Reste  $\geq$  10) Faire début

    Nb10D  $\leftarrow$  Nb10D + 1 ;

    Reste  $\leftarrow$  Reste – 10 ;

fin ;

Nb5D  $\leftarrow$  0 ;

Si (Reste  $\geq$  5) Alors début

    Nb5D  $\leftarrow$  1 ;

    Reste  $\leftarrow$  Reste – 5 ;

fin ;

Ecrire('Remise de la monnaie :') ;

Ecrire('Pièces de 10 DA : ', Nb10D) ;

Ecrire('Pièces de 5 DA : ', Nb5D) ;

Ecrire('Pièces de 1 D : ', Reste) ;

Fin.

Le programme Pascal :

```
program Achat ;
var
  prix, somme, M, Reste, Nb10D, Nb5D : integer ;
begin
  prix := 1 ;
  somme := 0 ;
  while (prix <> 0) do begin
    writeln('Entrez le prix :') ;
    readln(prix) ;
    somme := somme + prix ;
  end ;
  writeln('Vous devez :', somme, ' DA') ;
  writeln('Montant versé :') ;
  readln(M) ;
  Reste := M - somme ;
  Nb10D := 0 ;
  while (Reste >= 10) do begin
    Nb10D := Nb10D + 1 ;
    Reste := Reste - 10 ;
  end ;
  Nb5D := 0 ;
  if (Reste >= 5) then begin
    Nb5D := 1 ;
    Reste := Reste - 5 ;
  end ;
  writeln('Remise de la monnaie :') ;
  writeln('Pièces de 10 DA : ', Nb10D) ;
  writeln('Pièces de 5 DA : ', Nb5D) ;
  writeln('Pièces de 1 DA : ', Reste) ;
end.
```

**Solution 12.2**

Algorithme Achat ;

Variables

prix, somme, M, Reste, Nb10D, Nb5D, Nb1D : entier ;

Début

somme  $\leftarrow$  0 ;

Répéter

```

    Ecrire('Entrez le prix : ');
    Lire(prix);
    Si (prix > 0) Alors somme ← somme + prix ;
Jusqu'à (prix = 0) ;
Ecrire('Vous devez : ', somme, ' DA') ;
Ecrire('Montant versé : ');
Lire(M) ;
Reste ← M – somme ;
Nb10D ← Reste ÷ 10 ;
Nb5D ← (Reste % 10) ÷ 5 ;
Nb1D ← Reste % 5;
Ecrire('Remise de la monnaie :');
Ecrire('Pièces de 10 DA : ', Nb10D) ;
Ecrire('Pièces de 5 DA : ', Nb5D) ;
Ecrire('Pièces de 1 D : ',Nb1D) ;
Fin.

```

Le programme Pascal :

```

program Achat ;
var
    prix, somme, M, Reste, Nb10D, Nb5D, Nb1D : integer ;
begin
    somme := 0 ;
    repeat
        write('Entrez le prix : ');
        readln(prix);
        if (prix > 0) then somme := somme + prix ;
    until (prix = 0) ;
    writeln('Vous devez : ', somme, ' DA') ;
    write('Montant versé : ');
    readln(M) ;
    Reste := M - somme ;
    Nb10D := Reste DIV 10 ;
    Nb5D := (Reste MOD 10) DIV 5 ;
    Nb1D := Reste MOD 5;
    writeln('Remise de la monnaie :');
    writeln('Pièces de 10 DA : ', Nb10D) ;
    writeln('Pièces de 5 DA : ', Nb5D) ;
    writeln('Pièces de 1 D : ', Nb1D) ;
end.

```

### Solution 13 :

Algorithme Fibonacci ;

Variables

n, fibo, pred0, pred1, i : entier ;

Début

Ecrire('Donnez un entier :') ;

Lire(n) ;

Si (n = 0) Alors fibo  $\leftarrow$  0

Sinon Si (n = 1) Alors fibo  $\leftarrow$  1

Sinon Si (n > 1) Alors début

pred0  $\leftarrow$  0 ;

pred1  $\leftarrow$  1 ;

Pour i  $\leftarrow$  2 à n Faire début

fibo  $\leftarrow$  pred0 + pred1 ;

pred0  $\leftarrow$  pred1 ;

pred1  $\leftarrow$  fibo ;

fin ;

fin ;

Ecrire('Fibonacci(', n, ') = ', fibo) ;

Fin.

### Le programme Pascal :

program Fibonacci ;

var n, fibo, pred0, pred1, i : integer;

begin

writeln('Donnez un entier :') ;

readln(n) ;

if (n = 0) then fibo := 0

else if (n = 1) then fibo := 1

else if (n > 1) then begin

pred0 := 0;

pred1 := 1;

for i:= 2 to n do begin

fibo := pred0 + pred1;

pred0 := pred1;

pred1 := fibo ;

end;

end;

writeln('Fibonacci(', n, ') = ', fibo) ;

end.

**Solution 14 :**

Algorithme somme\_chiffres ;

Variables   chiffre, somme, a : entier ;

Début

  Ecrire('Donnez un nombre entier positif :');

  Lire(chiffre);

  somme  $\leftarrow$  0 ;

  a  $\leftarrow$  chiffre;

  Tant que (a > 0) Faire début

    somme  $\leftarrow$  somme + (a MOD 10);

    a  $\leftarrow$  a  $\div$  10 ;

  fin ;

  Ecrire('La somme des chiffres du nombre ', chiffre, ' est égale à ', somme);

Fin.

Le programme Pascal :

program somme\_chiffres ;

var   chiffre, somme, a : integer;

begin

  writeln('Donnez un nombre entier positif ');

  readln(chiffre);

  somme := 0 ;

  a := chiffre;

  while (a > 0) do begin

    somme := somme + (a MOD 10);

    a := a DIV 10 ;

  end ;

  writeln('La somme des chiffres du nombre ', chiffre, ' est égale à ', somme);

end.

**Solution 15 :**

Ce programme affiche :

1

2

3

\*\*\*

bon courage

merci

\*\*\*

## Chapitre 5 : Les tableaux et les chaînes de caractères

### 1. Introduction

Les types présentés jusqu'à maintenant permettent de définir des variables simples qui ne peuvent pas être décomposées en valeurs plus simples, mais de nombreux problèmes nécessitent de manipuler des variables qui peuvent avoir des valeurs plus complexes (structurées), et qui ne peuvent pas être présentées par une valeur simple.

On appelle type structuré tout type dont la définition fait référence à d'autre type. Les types structurés permettent d'effectuer des traitements plus complexes avec un maximum d'efficacité. Commençant par le type qu'on juge le plus important des types structurés, à savoir le type tableau.

### 2. Le type tableau

Un tableau est une structure homogène composée d'un ensemble d'éléments de même type de données. Chaque élément est accessible via un indice.

Format général : `Nom_tab : Tableau [indice_min..indice_max] de type_données ;`

Un tableau possède un nom (`Nom_tab`). Il est aussi caractérisé par une valeur d'indice minimal et une valeur d'indice maximal (`indice_min`, `indice_max`). Ces deux valeurs doivent être de type ordinal, généralement sont de type entier, et elles permettent de déterminer le nombre de cases de ce tableau (`nb_case = indice_max - indice_min + 1`). Le type des éléments du tableau est indiqué par `type_données`.

L'exemple suivant permet de déclarer un tableau nommé `Note` de dix éléments réels :

`Note : Tableau [1..10] de réel ;`

Cet exemple nous a permis de substituer la déclaration de 10 variables `Note1`, `Note2`, ..., `Note10` de type réel par une seule structure, à savoir le tableau `Note`.

En Pascal, un tableau se déclare comme suit :

`Note : Array [1..10] of real ;`

La capacité d'un tableau (le nombre d'éléments que peut contenir un tableau) ne peut pas être changée au cours d'exécution d'un programme. Il est donc nécessaire de prendre une valeur suffisamment grande pour le traitement. Il faut alors trouver un majorant du nombre d'éléments. Pour une souplesse de programmation, on peut utiliser une constante de la manière suivante :

```
const
  Max = 10 ;
var
  Note : Array[1..Max] of real ;
```

## 2.1. Manipulation d'un tableau

Un tableau peut être représenté par un ensemble de cases, l'une à côté de l'autre, schématisées comme suit :

1	2	3	4	5	6	7	8	9	10

Représentation du tableau Note

Un élément du tableau est accessible par sa position (indice) dans le tableau. Le tableau Note est un tableau à une seule dimension là où chaque élément est accessible par un seul indice. L'accès peut être pour une lecture, écriture, modification, etc.

Par exemple, pour affecter une valeur au premier élément du tableau, on met :  $\text{Note}[1] \leftarrow 0.25$  ; et le tableau Note devient :

1	2	3	4	5	6	7	8	9	10
0.25									

L'indice peut être exprimé directement comme un nombre en clair, comme il peut être une valeur d'une variable, ou une expression calculée. Pour la manipulation des tableaux, on utilise fréquemment les boucles.

Voyons l'exemple suivant :

```
var
  A : Array [1..10] of real ;
  B : Array [0..9] of boolean ;
  C : Array[boolean] of boolean ;
  H : Array ['a'..'z'] of Array [1..10] of real;
  E : Array[1..7] of string ;
begin
  A[1] :=1 ; B[0] :=TRUE ; C[FALSE]:=TRUE; H['b'][1]:=3; E[1]:='Samedi';
  ...
```

## 2.2. Tri d'un tableau

En informatique ou en mathématiques, un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon un ordre déterminé. Les objets à trier font donc partie d'un ensemble muni d'une relation d'ordre (de manière générale, un ordre total). Les ordres les plus utilisés sont l'ordre numérique et l'ordre lexicographique (dictionnaire). Il est évident que suivant la relation d'ordre considérée, une même collection d'objets peut donner lieu à divers arrangements. Pourtant, il est possible de définir un algorithme de tri indépendamment de la

fonction d'ordre utilisée. Celui-ci ne fera qu'utiliser une certaine fonction d'ordre correspondant à une relation d'ordre qui doit permettre de comparer tout couple d'éléments de la collection.

**Problème :** soit un tableau de N entiers rangés n'importe comment. On cherche à modifier le tableau de telle manière que les entiers y soient rangés par ordre croissant.

**Remarque :** On travaille avec des entiers et l'ordre < (la relation d'ordre "<"), mais on peut très facilement adapter les algorithmes que l'on va étudier à d'autres types d'éléments à trier suivant un certain ordre, par exemple des chaînes à trier par ordre alphabétique, des dates par ordre chronologique, des personnes (nom + prénom + numéro SS + adresse + . . .) par âge décroissant, etc.

**Exemples :** Les algorithmes suivants sont utilisés pour résoudre le problème de tri. Ils seront détaillés dans la partie exercices.

- Tri à bulles.
- Tri par sélection.
- Tri par insertion.
- Tri rapide (*quick sort*).
- Tri fusion (*merge sort*).
- Tri par tas (*heap sort*).

### 2.3. Tableau à deux dimensions

Un élément du tableau peut être lui-même de type tableau, comme c'est le cas pour Array ['a'..'z'] of Array [1..10] of real ;

Il existe une forme plus simple (Array ['a'..'z', 1..10] of real ;), et on parle dans ce cas de tableau à deux dimensions.

Pour une variable H de type tableau défini ci-dessus, un élément est accessible de deux manières : H['a'][3] := 0.25 ; ou bien H['a',3] := 0.25 ;

Autrement dit, lorsqu'on veut traiter plusieurs tableaux ayant la même dimension et le même type de données, on peut utiliser un seul tableau à deux dimensions. On obtient un seul tableau avec un nom unique. Ses éléments sont accessibles via deux indices : l'un indique la ligne et l'autre indique la colonne.

Format général : Nom\_tab : Tableau [ind\_lig\_min..ind\_lig\_max, ind\_col\_min..ind\_col\_max] de type\_données ;

L'exemple suivant permet de déclarer un tableau nommé Note à deux dimensions 3\*10 : Note : Tableau [1..3, 1..10] de réel ;

En Pascal, le tableau Note à deux dimensions se déclare comme suit : Note : Array [1..3, 1..10] of real ;

Un tableau à deux dimensions peut être représenté par un ensemble de cases organisées en lignes et colonnes, schématisées comme suit :



	1	2	3	4	5	6	7	8	9	10
1										
2										
3										

Un élément du tableau est accessible par deux indices (ligne et colonne).  
Par exemple : `Note[2,3] := 5.44` ; et le tableau `Note` devient :

	1	2	3	4	5	6	7	8	9	10
1										
2			5.44							
3										

Les tableaux les plus utilisés sont à une ou deux dimensions, mais il est possible de définir des tableaux à trois ou à quatre dimensions, voire plus, par exemple, `Note : Tableau [1..3, 1..10, 'a'..'z', boolean]` de réel ; Il faut savoir que plus le nombre de dimensions du tableau augmente, plus la conception des algorithmes devient difficile.

**Remarques :**

- Il n’y a aucune différence qualitative entre un tableau à deux dimensions `[i, j]` et un tableau à une dimension `[i*j]`. Tout problème qui peut être modélisé d’une manière peut aussi être modélisé de l’autre. Simplement, l’une ou l’autre de ces techniques correspond plus spontanément à tel ou tel problème, et facilite donc (ou complique, si on a choisi la mauvaise option) l’écriture et la lisibilité de l’algorithme.
- Une question classique à propos des tableaux à deux dimensions est de savoir si le premier indice représente les lignes ou le deuxième les colonnes, ou l’inverse. Alors, on doit savoir que *Lignes* et *Colonnes* sont des concepts graphiques, visuels, qui s’appliquent à des objets du monde réel, alors que les indices des tableaux ne sont que des coordonnées logiques, pointant vers des adresses de la mémoire vive.
- En mathématiques, on parle de vecteur quand il s’agit d’un tableau à une dimension, et d’une matrice quand il s’agit d’un tableau à deux dimensions.

**3. Les chaînes de caractères**

Une chaîne de caractères est une suite de caractères. Il s’agit donc d’une donnée alphanumérique représentant un autre type structuré. Elle peut être considérée tout simplement comme un tableau de caractères sur lequel on peut effectuer des opérations supplémentaires. Ces opérations dépendent du langage de programmation considéré.

**3.1. Opérations sur les chaînes de caractères**

En général, il existe :

- L'opération de concaténation (juxtaposition de 2 chaînes pour en former une nouvelle) est symbolisée par // séparant les 2 chaînes originelles.
- La fonction qui permet d'extraire une sous-chaîne est représentée par le nom de la variable avec en indice les positions des lettres à extraire. Ainsi, la sous-chaîne formée des caractères occupant les positions 2, 3, 4 dans la variable *CH* sera symbolisée par :  $CH_{2 \leftarrow 4}$ .
- La fonction qui fournit la longueur (le nombre de caractères) de la chaîne contenue dans la variable *CH* est symbolisée par  $|CH|$ .
- On peut aussi effectuer des comparaisons sur les chaînes de caractères par <, >, =... Le résultat de la comparaison dépend de l'ordre lexicographique.

Par définition, les indices valides pour une chaîne de caractères sont des entiers compris entre 1 et la longueur de la chaîne. Chaque caractère est accessible (en lecture, en écriture, etc.) par son indice.

### 3.2. Déclaration d'une chaîne de caractères

En Pascal, la déclaration d'une chaîne de caractères se fait comme suit : `var Chaine : packed array [1..10] of char` ; (packed pour dire compacté ; en turbo Pascal, ce mot n'a aucun effet car le compactage est effectué automatiquement quand c'est possible). Cette déclaration permet d'accéder à un élément de la chaîne par son indice, par exemple `Chaine[7]` correspond au septième caractère de la chaîne, mais elle ne permet les opérations supplémentaires décrites précédemment. Pour cela, Pascal offre une autre possibilité pour déclarer une chaîne de caractères, et cela en utilisant le mot clé `STRING`. Ce langage permet aussi de préciser la taille maximale que pourra avoir la chaîne qui sera affectée à une variable de ce type. Pour une chaîne de 25 caractères maximum, on met `STRING[25]` ; En l'absence de la précision de la longueur, Pascal réserve automatiquement la taille maximale, à savoir 255 caractères.

### 3.3. Manipulation des chaînes de caractères

On peut affecter à toute variable chaîne, une expression chaîne de caractères, en utilisant le symbole d'affectation traditionnel `:=`.

Voyons l'exemple suivant qui montre différentes possibilités d'affectation sur les chaînes de caractères :

```
var
  CH1, CH2, CH3, CH4 : string[10] ;
  C : char ;
begin
  CH1 := 'bonjour' ; C := 'a' ; CH2 := C ; CH3 := '' ; CH4 := 'a' ;
  ...
```

Comme vous remarquez, une chaîne de caractères doit être mise entre deux guillemets simples pour la distinguer d'un identificateur de variable. Alors dans l'exemple ci-dessus, on a affecté à la variable CH1 la chaîne de caractères mise entre deux guillemets 'bonjour'. Si cette chaîne contenait elle-même un guillemet, alors il faut le doubler, par exemple CH1 := 'Aujourd'hui' ;. La variable CH2 reçoit la valeur de la variable de type caractère C (l'inverse n'est pas accepté, i.e. on ne peut pas affecter une chaîne de caractères à une variable de type caractère). La variable CH3 reçoit la chaîne vide, et la variable CH4 reçoit un seul caractère ('a').

En mémoire, la chaîne CH1 peut être représentée comme suit :

b	o	n	j	o	u	r			
1	2	3	4	5	6	7	8	9	10

Le premier élément correspond à l'indice 1. Le dernier élément correspond à l'indice 7 (=Long(CH1)). Le reste de l'espace réservé à la chaîne CH1 sera rempli par des espaces possédant le code ASCII 0.

On peut aussi comparer deux chaînes de caractères, par exemple l'expression (CH1 < CH2) retourne une valeur booléenne qui dépend de l'ordre alphabétique des deux chaînes. Selon l'exemple précédent, l'expression (CH1 < CH2) retourne la valeur FALSE.

Les fonctions avec lesquelles on peut manipuler des chaînes de caractères sont détaillées dans le tableau suivant :

Notation en LDA	Fonction Pascal	Type du résultat	Signification
Ch  Long(Ch)	LENGTH(Ch)	Entier	Nombre de caractères dans Ch
Ch1 // Ch2	CONCAT(Ch1,Ch2) Ch1 + Ch2	Chaîne	Concaténation (juxtaposition) de Ch1 et Ch2
Ch <sub>i</sub> ← <sub>j</sub>	COPY(Ch, i, j-i+1)	Chaîne	Extraction, à partir de Ch, des caractères de la position i à la position j
Ch[i] ou Ch <sub>i</sub>	COPY(Ch, i, 1)	Chaîne	Extraction, à partir de Ch, du caractère de la position i
	Ch[i]	Caractère	

Il existe aussi en Pascal la fonction upcase(car) qui permet de convertir le caractère simple car en en majuscule.

### 3.4. Edition d'une chaîne de caractères

L'exemple suivant illustre quelques formats possibles pour afficher une chaîne de caractères.

```
program afficher ;
const msg = 'Pascal 7' ;
begin
  writeln(msg);
  writeln('Pascal 7');
```

```
writeln(msg:10);
end.
```

La première instruction affiche la chaîne de caractères Pascal 7. Même chose pour la deuxième instruction. La dernière instruction permet d'afficher le contenu de la variable msg sur dix positions, en remplissant la partie gauche de la chaîne affichée par des espaces.

Ceci donne l'affichage : Pascal 7.

### 3.5. Tableau de chaînes de caractères

On peut aussi déclarer des tableaux de chaînes de caractères qui sont des tableaux de tableaux. Pour déclarer, par exemple, un tableau de 5 chaînes de 20 caractères, on écrit : `var tab : array [1..5] of string[20] ;`. On accède à chacune des chaînes par `tab[i]`, et on accède à n'importe quel caractère j de la chaîne i par `tab[i,j]` ou `tab[i][j]`.

## 4. Exercices corrigés

### 4.1. Exercices

**Exercice 1 :** (Création, initialisation et édition d'un tableau)

*Problème :* Ecrire un algorithme permettant de créer un tableau de dix entiers, d'initialiser ses éléments à 0, ensuite de les afficher. Traduire l'algorithme en Pascal,

*Solution :* La création d'un tableau consiste en sa déclaration. L'initialisation, dans ce cas là, consiste à mettre un zéro partout dans le tableau. L'édition ou l'affichage d'un tableau consiste à parcourir les différentes cases en faisant varier l'indice, et on affiche leur contenu au fur et à mesure.

**Exercice 2 :** (Création, remplissage et édition d'un tableau)

*Problème :* Ecrire un algorithme permettant de créer un tableau de dix caractères, de lire ses éléments à partir du clavier, ensuite de les afficher. Ecrire le programme Pascal correspondant.

*Solution :* Il s'agit de la même solution que pour l'exercice précédent, sauf que pour le premier, le tableau prend ses valeurs (des zéros) par le programmeur (au niveau du code source du programme). Par contre, dans ce deuxième cas, le tableau est rempli par l'utilisateur, à partir du clavier, au cours d'exécution du programme.

**Exercice 3 :** (La recherche dans un tableau)

*Problème :* Ecrire un algorithme permettant la recherche d'un nombre lu à partir du clavier dans un tableau de dix réels. Traduire l'algorithme en Pascal.

*Solution :* Il existe plusieurs stratégies possibles pour la recherche dans un tableau ; nous en verrons deux : la recherche en utilisant la technique de Flag et la recherche dichotomique.

***Commençons par la technique de Flag :***

Le flag, en anglais, est un petit drapeau qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas. Et, aussitôt que cet événement a lieu, le petit drapeau se lève. Cela se traduit par une variable booléenne initialisée à FAUX (drapeau baissé), et qui aura la valeur VRAI dès que l'événement attendu se produit (drapeau levé). Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Dans notre cas, on va :

- Utiliser une variable booléenne dite Flag, et un tableau de dix réels, appelé NOMBRES, ainsi que le nombre recherché, appelé NBR.
- La variable Flag s'initialise à FAUX.
- Lire les éléments du tableau NOMBRES.
- Lire le nombre recherché NBR.
- Parcourir le tableau NOMBRES. Si un élément du tableau est égal à NBR, la variable Flag devient VRAI.
- Enfin, on teste Flag pour afficher le message d'existence ou d'absence du nombre recherché.

La technique de Flag (que nous pourrions élégamment surnommer *gestion asymétrique de variable booléenne*), peut être utilisée chaque fois que l'on se trouve devant une situation pareille.

***La recherche dichotomique :***

La dichotomie, c'est "couper en deux" en Grec. La recherche dichotomique exige que les éléments du tableau soient ordonnés préalablement. La recherche dichotomique est mieux adaptée dans le cas où le tableau contient un nombre d'éléments très élevé, car la recherche en utilisant la technique classique de Flag consomme un temps considérable.

La recherche dichotomique implique la technique de Flag, car on a besoin d'un test d'arrêt en cas où l'élément recherché est trouvé.

La recherche dichotomique consiste à comparer le nombre à rechercher avec le nombre qui se trouve au milieu du tableau. Si le nombre recherché est inférieur, on devra le rechercher dorénavant dans la première moitié du dico. Sinon, on devra continuer la recherche dans la deuxième moitié.

A partir de là, on prend la moitié du tableau qui nous reste, et on recommence : on compare le nombre à rechercher avec celui qui se trouve au milieu du morceau du tableau restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

Suite à couper notre tableau en deux, puis encore en deux, etc., on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul nombre. Et si on n'est pas tombé sur le bon nombre à un moment ou à un autre, c'est que le nombre recherché n'existe pas dans le tableau. Si notre tableau contient par exemple les valeurs : 19, 31, 31, 40.5, 59, 60, 61, 75, 80, 99, et soit 61 le nombre à rechercher, alors on commence par rechercher le milieu du tableau qui doit être une case dont l'ordre est une valeur entière.

- Le milieu à ce niveau correspond à la valeur 59 qui occupe la cinquième position dans le tableau initial. La valeur recherchée 61 est différente à la valeur du milieu 59, elle est plutôt supérieure à cette valeur. Par conséquent, la première moitié sera suspendue et on contenu la recherche dans la deuxième moitié qui correspond aux valeurs : 60, 61, 75, 80, 99.
- Le milieu est maintenant 75 qui occupe la huitième position dans le tableau initial. La valeur recherchée 61 est différente à la valeur du milieu 75, elle est cette fois-ci inférieure à cette valeur. Par conséquent, la deuxième moitié sera suspendue et on contenu la recherche dans la première moitié qui correspond aux valeurs : 60, 61.
- Le milieu à ce niveau correspond à la valeur 60 qui occupe la sixième position dans le tableau initial. La valeur recherchée 61 est différente à la valeur du milieu 60, elle est supérieure à cette valeur. Par conséquent, la première moitié sera suspendue et on contenu la recherche dans la deuxième moitié qui correspond cette fois-ci à une seule valeur : 61.
- Le milieu à ce niveau correspond à la valeur 61 qui occupe la septième position dans le tableau initial. La valeur recherchée 61 est égale à la valeur du milieu 61. Alors on s'arrête.

Dans ce cas-là, on va :

- Utiliser une variable booléenne dite TROUVE qui indique si l'élément recherché a été trouvé ou non, et un tableau de dix nombres ordonnés appelé NOMBRES. Aussi, une variable est utilisée pour le nombre à rechercher, appelée NBR. En plus, on déclare les variables Inf et Sup qui gardent les bornes de la moitié là où on va faire notre recherche. On utilise aussi la variable Milieu pour déterminer à chaque fois le milieu d'une partie.
- Lire les éléments du tableau NOMBRES.
- Lire le nombre à rechercher NBR.
- On initialise les variables : TROUVE à FAUX, Sup à 10, et Inf à 1.

- Tant que l'élément recherché n'a pas été trouvé, et il y a toujours une partie là où on peut continuer la recherche, tester l'élément recherché par rapport au milieu pour arrêter la recherche dans le cas d'égalité, ou continuer la recherche dans la première ou la seconde moitié selon le résultat de la comparaison.
- Enfin, on teste la variable TROUVE pour afficher le message d'existence ou d'absence du nombre recherché.

Cette technique n'est pas favorisée dans cet exercice, car on n'a pas un nombre important d'élément, mais si nous avions un tableau de 40000 nombres, alors il sera préférable d'utiliser la recherche dichotomique. Regardons ce que cela donne en termes de nombre d'opérations à effectuer, en choisissant le pire des cas : celui où le nombre est absent du tableau.

- Au départ, on cherche le nombre parmi 40 000.
- Après le test n°1, on ne le cherche plus que parmi 20 000.
- Après le test n°2, on ne le cherche plus que parmi 10 000.
- Après le test n°3, on ne le cherche plus que parmi 5 000.
- etc.
- Après le test n°15, on ne le cherche plus que parmi 2.
- Après le test n°16, on ne le cherche plus que parmi 1.

Maintenant, on sait que le nombre n'existe pas. Donc on a obtenu notre réponse après 16 opérations contre 40000 en utilisant la technique de Flag.

**Exercice 4 :** (Calcul de la somme, le produit et la moyenne des éléments numériques d'un tableau)

*Problème :* Ecrire un algorithme permettant de calculer la somme, le produit et la moyenne d'un tableau de dix réels. Traduire l'algorithme en Pascal.

*Solution :* Pour calculer la somme des nombres contenus dans le tableau, il faut ajouter un à un le contenu des cases depuis la première jusqu'à la dernière, en utilisant une variable initialisée à zéro.

Pour calculer le produit des nombres contenus dans le tableau, il faut multiplier un par un le contenu des cases depuis la première jusqu'à la dernière, en utilisant une variable initialisée à un.

Pour calculer la moyenne, il suffit de diviser la somme par le nombre de cases du tableau.

**Exercice 5 :** (La recherche du plus petit et plus grand élément dans un tableau)

*Problème :* Ecrire un algorithme permettant de déterminer la valeur maximale et minimale dans un tableau de dix entiers, avec leurs positions dans le tableau. Traduire l'algorithme en Pascal.

**Solution :** Le principe de la recherche du plus petit élément d'un tableau consiste à :

- On suppose que la première case contient le minimum relatif.
- On compare le contenu de la deuxième case avec le minimum relatif. Si celui-ci est inférieur, il devient le minimum relatif.
- On reprend la même opération pour les cases suivantes.

Pour le plus grand élément du tableau, il suffit de substituer le test Si ( $\min > T[i]$ ) Alors ... par Si ( $\max < T[i]$ ) Alors...

**Exercice 6 :**

Que fait l'algorithme suivant ?

Algorithme X ;

Variables

NB : Tableau [1..5] d'entier ;

i : entier ;

Début

Pour i  $\leftarrow$  1 à 5 Faire Nb[i]  $\leftarrow$  i \* i ;

Pour i  $\leftarrow$  1 à 5 Faire Ecrire(Nb[i]) ;

Fin.

Peut-on simplifier cet algorithme pour avoir le même résultat ?

**Exercice 7 :**

Que fait l'algorithme suivant ?

Algorithme X ;

Variables

N : Tableau[1..6] d'entier ;

i, k : entier ;

Début

N[1]  $\leftarrow$  1 ;

Pour k  $\leftarrow$  2 à 6 Faire N[k]  $\leftarrow$  N[k-1] + 2 ;

Pour i  $\leftarrow$  1 à 6 Faire Ecrire N[i] ;

Fin.

Peut-on simplifier cet algorithme pour avoir le même résultat ?

**Exercice 8 :**

Que fait l'algorithme suivant ?

Algorithme X ;

Variables

Suite : Tableau [1..7] d'entier ;

i : entier ;

Début

Suite[1]  $\leftarrow$  1 ; Suite[2]  $\leftarrow$  1 ;



Pour  $i \leftarrow 3$  à 7 Faire Suite[i]  $\leftarrow$  Suite[i-1] + Suite[i-2] ;

Pour  $i \leftarrow 1$  à 7 Faire Ecrire(Suite[i]) ;

Fin.

Représentez le tableau Suite après l'exécution des opérations de l'algorithme X.

Peut-on simplifier cet algorithme pour avoir le même résultat ?

**Exercice 9 :**

Ecrire un algorithme permettant l'affichage des deux plus petits éléments d'un tableau de dix entiers. Traduire l'algorithme en Pascal.

**Exercice 10 :**

Ecrire un algorithme permettant la recherche du minimum et le nombre d'occurrence de ce minimum dans un tableau de dix entiers. Reprendre l'algorithme, mais cette fois-ci en utilisant une seule boucle. Traduire les deux algorithmes en Pascal.

**Exercice 11 :**

Ecrire un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives. Traduire l'algorithme en Pascal.

**Exercice 12 :**

Ecrire un algorithme permettant de remplir deux tableaux de dix éléments, de calculer leur somme, et de stocker le résultat dans un troisième tableau.

Exemple :

Tableau 1 : 4, 8, 7, 9, 1, 5, 4, 6.

Tableau 2 : 7, 6, 5, 2, 1, 3, 7, 4.

Tableau à constituer : 11, 14, 12, 11, 2, 8, 11, 10.

Traduire l'algorithme en Pascal.

**Exercice 13 :**

Toujours à partir des deux tableaux précédemment saisis, écrivez un algorithme qui calcule le schtroumpf des deux tableaux. Ensuite, représentez l'algorithme par l'organigramme correspondant.

Pour calculer le schtroumpf, il faut multiplier chaque élément du tableau 1 par chaque élément du tableau 2, et additionner le tout.

Par exemple :

Tableau 1 : 4, 8, 7, 12.

Tableau 2 : 3, 6.

Le Schtroumpf :  $3*4 + 3*8 + 3*7 + 3*12 + 6*4 + 6*8 + 6*7 + 6*12 = 279$ .

Traduire l'algorithme en Pascal.

**Exercice 14 :** (Le tri d'un tableau)

**Problème :** Ecrire un algorithme permettant de trier un tableau de dix entiers en ordre croissant. Traduire l'algorithme en Pascal.

**Solution :** Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons deux : le tri par sélection et le tri à bulles.

**Commençons par le tri par sélection :**

Dans ce cas, le tri d'un tableau consiste à mettre en bonne position l'élément numéro 1, c'est-à-dire le plus petit, puis, on met en bonne position l'élément suivant, et ainsi de suite, jusqu'au dernier.

Par exemple, si l'on part de : 10, 5, 77, -4, 8, 11, 0, 1, 15, 13.

On prend l'élément occupant la première case, et on le compare avec le reste du tableau (à partir de la deuxième case, jusqu'à la dixième). Chaque fois où cet élément est supérieur, on le permute avec l'élément comparé. Le tableau devient ainsi : -4, 10, 77, 5, 8, 11, 0, 1, 15, 13.

On prend maintenant l'élément occupant la deuxième case, et on le compare avec le reste du tableau (à partir de la troisième case, jusqu'à la dixième). Chaque fois où cet élément est supérieur, on le permute avec l'élément comparé. Le tableau devient ainsi : -4, 0, 77, 10, 8, 11, 5, 1, 15, 13.

Remarquons qu'à ce niveau les deux plus petits éléments (-4 et 0) ont pris leurs positions dans le tableau.

On prend maintenant l'élément occupant la troisième case, et on le compare avec le reste du tableau (à partir de la quatrième case, jusqu'à la dixième). Chaque fois où cet élément est supérieur, on le permute avec l'élément comparé. Le tableau devient ainsi : -4, 0, 1, 77, 10, 11, 8, 5, 15, 13.

Cette opération se répète jusqu'à arriver au neuvième élément, le comparer avec le dixième, s'il est supérieur, alors permuter ces deux derniers éléments.

Le processus de tri d'un tableau par sélection consiste en deux boucles imbriquées :

- Une boucle principale : allant du premier élément du tableau, puis le second, etc. jusqu'à l'avant dernier.
- Une boucle secondaire : allant du compteur de la boucle principale plus un, jusqu'au dernier élément du tableau. On compare l'élément du tableau (ayant comme indice la valeur courante du compteur de la boucle principale) avec l'élément du tableau (ayant comme indice la valeur courante du compteur de la boucle secondaire). Si le premier est supérieur au deuxième, on permute les deux éléments.

**Le tri à bulles :**

Dans ce cas, on fait le tri d'un tableau en utilisant la technique de Flag vue précédemment, ce qui donne la formule : tri de tableau + flag = tri à bulles.

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel tout élément est plus petit que celui qui le suit.

En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands *remontent* ainsi peu à peu vers les dernières places, ce qui explique la charmante dénomination de *tri à bulles*.

On ne sait jamais d'avance combien de remontées de bulles on doit effectuer. En fait, tout ce qu'on peut dire, c'est qu'on devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés. Ceci est typiquement un cas de question *asymétrique* : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié. Cela implique l'utilisation d'un Flag qui va nous indiquer quand va-on s'arrêter.

Le processus de tri à bulles d'un tableau consiste en deux boucles imbriquées :

- Une boucle principale : Il s'agit d'une boucle Tant que qui dépend d'une variable booléenne Flag. Cette variable booléenne va nous indiquer si nous venons ou non de procéder à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est un signe que le tableau est trié, et donc, on peut arrêter la machine à bulles).
- Une boucle secondaire qui permet de prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, de comparer chaque élément avec son suivant, et procéder à une permutation si nécessaire.

La variable booléenne Flag va nous signaler le fait qu'il y a eu au moins une permutation effectuée. Sa gestion sera comme suit :

- Attribuer la valeur VRAI au Flag dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
- Le remettre à FAUX à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés).
- Il ne faut pas oublier de lancer la boucle principale, et pour cela de donner la valeur VRAI au Flag au départ de l'algorithme.

*Remarque* : Pour trier un tableau en ordre décroissant, il suffit de substituer le test Si ( $T[i] > T[j]$ ) Alors... par Si ( $T[i] < T[j]$ ) Alors...

**Exercice 15 :**

Ecrire un algorithme permettant de lire un tableau de 7 entiers, et de dire si les éléments du tableau sont tous consécutifs ou non.

Par exemple, si le tableau contient les valeurs : 12, 13, 14, 15, 16, 17, 18, alors ses éléments sont tous consécutifs. Si le tableau contient les valeurs : 9, 10, 11, 15, 16, 17, 18, ses éléments ne sont pas tous consécutifs.

Traduire l'algorithme en Pascal.

**Exercice 16 :**

Ecrire un algorithme qui inverse l'ordre des éléments d'un tableau de dix entiers lus à partir du clavier (les premiers seront les derniers...).

Traduire l'algorithme en Pascal.

**Exercice 17 :**

Ecrire un algorithme qui calcule itérativement le  $n^{\text{ième}}$  terme de la suite de Fibonacci définie comme suit : Si  $n = 0$  alors  $F_n = 0$ . Si  $n = 1$  alors  $F_n = 1$ . Si  $n > 1$  alors  $F_n = F_{n-1} + F_{n-2}$ . Utilisez cette fois-ci un tableau pour résoudre l'exercice. Traduire l'algorithme en Pascal.

**Exercice 18 :**

Ecrire un algorithme permettant de supprimer une case dont la position est lue à partir du clavier dans un tableau de dix entiers. La case supprimée sera substituée par un zéro ajouté à la fin du tableau.

Par exemple, après la suppression de la troisième case du tableau : 3, 15, 8, 18, 34, 1, 0, 4, 5, 21, le tableau devient : 3, 15, 18, 34, 1, 0, 4, 5, 21, 0.

Traduire l'algorithme en Pascal.

**Exercice 19 :**

Ecrire un algorithme permettant d'insérer une valeur dans un tableau de dix entiers trié par ordre croissant. On suppose que les valeurs du tableau sont triées par l'utilisateur lors de leurs saisies. Après l'insertion, la valeur de la dernière case sera perdue.

Par exemple, après l'insertion de la valeur 19 dans le tableau : 3, 15, 18, 28, 34, 61, 65, 74, 105, 121, le tableau devient : 3, 15, 18, 19, 28, 34, 61, 65, 74, 105.

Traduire l'algorithme en Pascal.

**Exercice 20 :**

Ecrire un algorithme qui calcule la  $n^{\text{ième}}$  ligne du triangle de Pascal.

Par exemple, pour  $n = 5$  :

Ligne 1 : 1

Ligne 2 : 1 1

Ligne 3 : 1 2 1

Ligne 4 : 1 3 3 1

Ligne 5 : 1 4 6 4 1

L'algorithme reçoit en entrée un entier  $n$  et génère comme résultat un tableau d'entiers Ligne, qui représente la  $n^{\text{ième}}$  ligne du triangle de Pascal.

Les cas triviaux sont :

- Si  $n = 1$  alors Ligne[1] = 1.
- Si  $(n = 2)$  alors Ligne[1] = 1, Ligne[2] = 1.

Le cas général :

- Si  $(n > 2)$  alors Ligne[1] = 1, Ligne[n] = 1.
- Ligne[j] = Ligne\_preced[j-1] + Ligne\_preced[j] pour  $2 \leq j < n$ , avec Ligne\_preced est la  $(n-1)^{\text{ième}}$  ligne du triangle de Pascal.

Traduire l'algorithme en langage Pascal.

Notons qu'en algorithmique, un cas trivial est un cas particulier, simple très basique, qui doit être traité à part, avant de procéder au cas général. Parfois, c'est le calcul du cas trivial qui va nous permettre de calculer le cas général, comme c'est le cas pour le triangle de Pascal.

#### Exercice 21 :

Ecrire un algorithme permettant de créer un tableau d'entiers à deux dimensions ( $3 \times 10$ ), d'initialiser ses valeurs à zéro, ensuite de les afficher. Traduire l'algorithme en Pascal.

#### Exercice 22 :

Que fait cet algorithme ?

Algorithme XX ;

Variables

X : tableau[1..2, 1..3] d'entier ;

i, j, val : entier ;

Début

Val  $\leftarrow$  1 ;

Pour i  $\leftarrow$  1 à 2 Faire Pour j  $\leftarrow$  1 à 3 Faire début

X[i, j]  $\leftarrow$  Val ;

Val  $\leftarrow$  Val + 1 ;

fin ;

Pour i  $\leftarrow$  1 à 2 Faire Pour j  $\leftarrow$  1 à 3 Faire Ecrire(X[i, j]) ;

Fin.

#### Exercice 23 :

Quel résultat produira cet algorithme ?

Algorithme XX ;

Variables

X : tableau[1..2, 1..3] d'entier ;

i, j, val : entier ;

```

Début
  Val  $\leftarrow$  1 ;
  Pour i  $\leftarrow$  1 à 2 Faire Pour j  $\leftarrow$  1 à 3 Faire début
    X[i, j]  $\leftarrow$  Val ;
    Val  $\leftarrow$  Val + 1 ;
  fin ;
  Pour j  $\leftarrow$  1 à 3 Faire Pour i  $\leftarrow$  1 à 2 Faire Ecrire(X[i, j]) ;
Fin.

```

#### Exercice 24 :

Que fait cet algorithme ?

Algorithme XX ;

Variables

T : tableau[1..4, 1..2] d'entier ;

k, m : entier ;

Début

Pour k  $\leftarrow$  1 à 4 Faire Pour m  $\leftarrow$  1 à 2 Faire T[k, m]  $\leftarrow$  k+m ;

Pour k  $\leftarrow$  1 à 4 Faire Pour m  $\leftarrow$  1 à 2 Faire Ecrire(T[k, m]) ;

Fin.

Représentez le tableau T après l'exécution des opérations de l'algorithme XX.

#### Exercice 25 :

Mêmes questions, en remplaçant la ligne : T[k, m]  $\leftarrow$  k+m par : a)

T[k, m]  $\leftarrow$  2 \* k + (m + 1), ensuite par b) T[k, m]  $\leftarrow$  (k + 1) + 4 \* m.

#### Exercice 26 :

Ecrire un algorithme permettant de remplir un tableau à deux dimensions (3\*10) à partir du clavier, et de déterminer le nombre d'apparition d'une certaine valeur saisie par l'utilisateur. Traduire l'algorithme en Pascal.

#### Exercice 27 :

Ecrire un algorithme permettant de remplir un tableau à deux dimensions (10\*20) à partir du clavier, et de calculer les totaux par lignes et colonnes dans les tableaux TotLig et TotCol. Traduire l'algorithme en Pascal.

#### Exercice 28 :

Ecrire un algorithme permettant le calcul de la transposée d'une matrice d'entier (3\*3). Traduire l'algorithme en Pascal.

#### Exercice 29 :

Ecrire un algorithme permettant le calcul du produit matriciel de deux matrices (3\*3). Le résultat sera rangé dans une troisième matrice. Traduire l'algorithme en Pascal.

**Exercice 30 :**

Que fait le programme suivant :

```
program keyboard;
  azerty : string ;
begin
  azerty := 'qwerty';
  writeln('azerty',azerty) ;
end.
```

**Exercice 31 :**

Que fait le programme suivant :

```
program XX ;
var
  ch: string ;
  long: integer ;
begin
  readln(ch) ;
  ch[1] := upcase(ch[1]) ;
  long := length(CH) ;
  if (long > 3) then ch := copy(ch,1,3) ;
  writeln(ch) ;
end.
```

**Exercice 32 :**

Ecrire un algorithme permettant de lire un prénom et un nom, ensuite de les concaténer en une seule chaîne de caractères. Le nom et le prénom doivent être séparés par un point. Affichez enfin la longueur de la chaîne résultante. Traduire l'algorithme en Pascal.

**Exercice 33 :**

Ecrire un algorithme permettant de lire deux chaînes de caractères, et de les afficher par ordre alphabétique. Traduire l'algorithme en Pascal.

**Exercice 34 :**

Ecrire un algorithme permettant de lire une chaîne de 20 caractères maximum, et de calculer le nombre d'apparition d'un caractère lu à partir du clavier. Traduire l'algorithme en Pascal.

**Exercice 35 :**

Ecrire un algorithme permettant de calculer le nombre de mots dans un texte lu à partir du clavier.

On suppose que : le texte est une suite de caractères simples ('a'..'z', 'A'..'Z'). Les mots sont séparés par des espaces (' '). Un espace n'existe que pour séparer deux mots successifs. Un texte contient au moins un mot. Traduire l'algorithme en Pascal.

**Exercice 36 :**

Ecrire un algorithme permettant de calculer le nombre d'apparition d'un mot dans un texte. Le mot et le texte étant tous les deux lus à partir du clavier.

On suppose que le texte est une suite de caractères simples ('a'..'z', 'A'..'Z'). Les mots sont séparés par des espaces (' '). Un espace n'existe que pour séparer deux mots successifs. Un texte contient au moins un mot.

Traduire l'algorithme en Pascal.

**Exercice 37 :**

Ecrire un algorithme qui affiche la conversion d'un entier positif en binaire. Le nombre binaire doit être stocké sous forme de chaîne de caractères. Traduire l'algorithme en Pascal.

**Exercice 38 :**

Ecrire un programme Pascal permettant de lire un tableau de 10 chaînes de 5 caractères maximum, puis affiche le dernier caractère de chaque chaîne.

**Exercice 39 :**

Ecrire un programme Pascal permettant de saisir un tableau de 10 mots de 20 caractères maximum, et d'afficher à nouveau le tableau avec son contenu en majuscule, et trié par ordre alphabétique.

Il est à noter que le principe de tri d'un tableau de mots par ordre alphabétique est identique au tri d'un tableau de nombres par ordre croissant.

**Exercice 40 :**

Ecrire un algorithme permettant de saisir dix noms d'étudiants avec leurs moyennes, puis d'afficher le nom de l'étudiant qui a la plus grande moyenne ainsi que celle-ci. La liste des noms et celle des notes étant stockées dans des tableaux. Traduire l'algorithme en Pascal.

**Exercice 41 :**

On considère des candidats inscrits à une formation diplômante. Si la note obtenue à cette formation est supérieure ou égale à 10, alors le candidat est admis. Ecrire l'algorithme permettant d'afficher la liste des candidats admis avec leurs notes (les noms des candidats et les notes étant lus à partir du clavier, et stockés dans des tableaux). Traduire l'algorithme en Pascal.

**Exercice 42 :**

On considère des candidats inscrits à une formation diplômante. Chaque candidat va obtenir une note pour cette formation. Ecrire l'algorithme permettant d'afficher la liste des candidats dont la note est supérieure ou égale à la moyenne des notes de tous les candidats (les noms des



candidats et les notes étant lus à partir du clavier, et stockés dans des tableaux). Traduire l'algorithme en Pascal.

#### 4.2. Corrigés

##### **Solution 1 :**

Algorithme init\_tab ;

Variables

T : tableau [1..10] d'entier ;

i : entier ;

Début

Pour i ← 1 à 10 Faire T[i] ← 0 ;

Ecrire('Voici les éléments du tableau :') ;

Pour i ← 1 à 10 Faire Ecrire(T[i]) ;

Fin.

Le programme Pascal :

program init\_tab ;

var

T : array [1..10] of integer ;

i : integer;

begin

for i := 1 to 10 do T[i] := 0 ;

writeln('Voici les éléments du tableau :') ;

for i := 1 to 10 do writeln(T[i]);

end.

##### **Solution 2 :**

Algorithme lire\_tab ;

Variables

T : tableau [1..10] de caractère ;

i : entier ;

Début

Ecrire('Donnez les éléments du tableau :') ;

Pour i ← 1 à 10 Faire Lire(T[i]) ;

Ecrire('Voici les éléments du tableau :') ;

Pour i ← 1 à 10 Faire Ecrire(T[i]) ;

Fin.

Le programme Pascal :

program lire\_tab ;

var

T : array [1..10] of char ;

i : integer;

```
begin
  writeln('Donnez les éléments du tableau :');
  for i := 1 to 10 do readln(T[i]);
  writeln('Voici les éléments du tableau :');
  for i := 1 to 10 do writeln(T[i]);
end.
```

### **Solution 3 :**

#### ***La recherche en utilisant la technique de Flag :***

Algorithme Recherche\_Flag ;

Variables

NOMBRES : tableau [1..10] de réel ;

NBR : réel ;

Flag : booléen ;

i : entier ;

Début

Flag ← FAUX ;

Ecrire('Entrez les éléments du tableau :');

Pour i ← 1 à 10 Faire Lire(NOMBRES[i]) ;

Ecrire('Entrez le nombre à rechercher :');

Lire(NBR) ;

Pour i ← 1 à 10 Faire Si (NOMBRES[i]=NBR) Alors Flag ← VRAI ;

Si (Flag) Alors Ecrire(NBR, ' fait partie du tableau.')

    Sinon Ecrire(NBR, ' ne fait pas partie du tableau.');

Fin.

#### **Le programme Pascal :**

```
program Recherche_Flag ;
```

```
var
```

```
  NOMBRES : array [1..10] of real ;
```

```
  NBR : real ;
```

```
  Flag : boolean ;
```

```
  i : integer ;
```

```
begin
```

```
  Flag := FALSE ;
```

```
  writeln('Entrez les éléments du tableau :');
```

```
  for i := 1 to 10 do readln(NOMBRES[i]) ;
```

```
  writeln('Entrez le nombre à rechercher :');
```

```
  readln(NBR) ;
```

```
  for i := 1 to 10 do if (NOMBRES[i]=NBR) then Flag := TRUE ;
```

```
  if (Flag) then writeln(NBR, ' fait partie du tableau.')
```

```
else writeln(NBR, ' ne fait pas partie du tableau.');
```

end.

La solution précédente impose un parcours obligatoire de tout le tableau, même si l'élément recherché est trouvé avant la fin du tableau. Pour que la recherche se termine immédiatement quand l'élément recherché est trouvé, il suffit de substituer la boucle Pour par une boucle Tant que, et cela comme suit :

Algorithme Recherche\_Flag2 ;

Variables

NOMBRES : tableau [1..10] de réel ;

NBR : réel ;

Flag : booléen ;

i : entier ;

Début

Flag  $\leftarrow$  FAUX ;

Ecrire('Entrez les éléments du tableau :');

Pour i  $\leftarrow$  1 à 10 Faire Lire(NOMBRES[i]) ;

Ecrire('Entrez le nombre à rechercher :');

Lire(NBR) ;

i  $\leftarrow$  1 ;

Tant que (i  $\leq$  10) ET (NON Flag) Faire

Si (NOMBRES[i]=NBR) Alors Flag  $\leftarrow$  VRAI

Sinon i  $\leftarrow$  i + 1 ;

Si (Flag) Alors Ecrire(NBR, ' fait partie du tableau.')

Sinon Ecrire(NBR, ' ne fait pas partie du tableau.');

Fin.

### ***La recherche dichotomique :***

Algorithme Recherche\_dichotomique ;

Variables

NOMBRES : tableau [1..10] de réel ;

NBR : réel ;

TROUVE : booléen ;

i, Sup, Inf, Milieu : entier ;

Début

Ecrire('Entrez les nombres triés du tableau :');

Pour i  $\leftarrow$  1 à 10 Faire Lire(NOMBRES[i]) ;

Ecrire('Entrez le nombre à rechercher :');

Lire(NBR) ;

```

Sup ← 10 ;
Inf ← 1 ;
TROUVE ← FAUX ;
Tant que (NON TROUVE) ET (Sup >= Inf) Faire début
    Milieu ← (Sup + Inf) ÷ 2 ;
    Si (NBR = NOMBRES[Milieu]) Alors TROUVE ← VRAI
    Sinon Si (NBR < NOMBRES[Milieu]) Alors Sup ← Milieu - 1
        Sinon Inf ← Milieu + 1 ;
fin ;
Si TROUVE Alors Ecrire('Le nombre ', NBR, ' existe dans le tableau.')
    Sinon Ecrire('Le nombre ', NBR, ' n'existe pas dans le tableau.') ;
Fin.

```

Le programme Pascal :

```

program Recherche_dichotomique ;
var
    NOMBRES : array [1..10] of real ;
    NBR : real ;
    TROUVE : boolean;
    i, Sup, Inf, Milieu : integer ;
begin
    writeln('Entrez les nombres triés du tableau :');
    for i := 1 to 10 do readln(NOMBRES[i]) ;
    writeln('Entrez le nombre à rechercher :') ;
    readln(NBR) ;
    Sup := 10 ;
    Inf := 1 ;
    TROUVE := FALSE ;
    while (NOT TROUVE) AND (Sup >= Inf) do begin
        Milieu := (Sup + Inf) DIV 2 ;
        if (NBR = NOMBRES[Milieu]) then TROUVE := TRUE
        else if (NBR < NOMBRES[Milieu]) then Sup := Milieu - 1
            else Inf := Milieu + 1 ;
    end ;
    if TROUVE then writeln('Le nombre ', NBR, ' existe dans le tableau.')
    else writeln('Le nombre ', NBR, ' n'existe dans le tableau.') ;
end.

```

**Solution 4 :**

Algorithme Som\_Moy\_Produit\_tab ;  
 Variables

```

T : tableau [1..10] de réel ;
i : entier ;
somme, produit, moyenne : réel ;
Début
  Ecrire('Donnez les éléments du tableau :') ;
  Pour ← 1 à 10 Faire Lire(T[i]) ;
  somme ← 0;
  produit ← 1 ;
  Pour i ← 1 à 10 Faire début
    somme ← somme + T[i] ;
    produit ← produit * T[i] ;
  fin ;
  moyenne ← somme / 10 ;
  Ecrire('La somme = ', somme) ;
  Ecrire('Le produit = ', produit) ;
  Ecrire('La moyenne = ', moyenne) ;
Fin.

```

Le programme Pascal :

```

program Som_Moy_Produit_tab ;
var
  T : array [1..10] of real ;
  i : integer;
  somme, produit, moyenne : real;
begin
  writeln('Donnez les éléments du tableau :') ;
  for i := 1 to 10 do readln(T[i]);
  somme := 0;
  produit := 1 ;
  for i := 1 to 10 do begin
    somme := somme + T[i];
    produit := produit * T[i];
  end;
  moyenne := somme / 10;
  writeln('La somme = ', somme);
  writeln('Le produit = ', produit);
  writeln('La moyenne = ', moyenne);
end.

```

**Solution 5 :**

```

Algorithme MaxMin_tab ;

```

### Variables

T : tableau [1..10] d'entier ;

max, min, pos\_min, pos\_max, i : entier ;

### Début

Ecrire('Donnez les éléments du tableau :') ;

Pour i ← 1 à 10 Faire Lire(T[i]) ;

max ← T[1] ;

pos\_max ← 1 ;

Pour i ← 2 à 10 Faire Si (max < T[i]) Alors début

max ← T[i] ;

pos\_max ← i ;

fin ;

min ← T[1] ;

pos\_min ← 1 ;

Pour i ← 2 à 10 Faire Si (min > T[i]) Alors début

min ← T[i] ;

pos\_min ← i ;

fin ;

Ecrire('La valeur maximale = ', max, ' occupant la position ', pos\_max) ;

Ecrire('La valeur minimale = ', min, ' occupant la position ', pos\_min) ;

Fin.

### Le programme Pascal :

```
program MaxMin_tab;
```

```
var
```

```
T : array [1..10] of integer ;
```

```
max, min, pos_max, pos_min, i : integer;
```

```
begin
```

```
writeln('Donnez les éléments du tableau :') ;
```

```
for i := 1 to 10 do readln(T[i]);
```

```
max := T[1];
```

```
pos_max := 1 ;
```

```
for i := 2 to 10 do if (max < T[i]) then begin
```

```
max := T[i];
```

```
pos_max := i ;
```

```
end ;
```

```
min := T[1];
```

```
pos_min := 1;
```

```
for i := 2 to 10 do if (min > T[i]) then begin
```

```
min := T[i];
```

```

        pos_min := i ;
    end ;
    writeln('La valeur maximale = ', max, ' occupant la position ', pos_max);
    writeln('La valeur minimale = ', min, ' occupant la position ', pos_min);
end.

```

### **Solution 6 :**

Cet algorithme remplit un tableau avec cinq valeurs : 1, 4, 9, 16, 25. Il les affiche ensuite à l'écran.

Voici une simplification :

Algorithme X ;

Variables

NB : Tableau [1..5] d'entier ;

i : entier ;

Début

Pour i  $\leftarrow$  1 à 5 Faire début

    Nb[i]  $\leftarrow$  i \* i ;

    Ecrire(Nb[i]) ;

fin ;

Fin.

### **Solution 7 :**

Cet algorithme remplit un tableau avec les six valeurs : 1, 3, 5, 7, 9, 11. Il les affiche ensuite à l'écran.

Voici une simplification :

Algorithme X ;

Variables

N : Tableau[1..6] d'entier ;

i, k : entier ;

Début

N[1]  $\leftarrow$  1 ;

Ecrire N[1] ;

Pour k  $\leftarrow$  2 à 6 Faire début

    N[k]  $\leftarrow$  N[k-1] + 2 ;

    Ecrire N[i] ;

fin ;

Fin.

### **Solution 8 :**

Cet algorithme crée un tableau de 7 éléments de type entier. Ensuite, il remplit le tableau par 7 valeurs : 1, 1, 2, 3, 5, 8, 13. Enfin, il affiche le contenu de ce tableau.

Le tableau Suite peut être représenté donc comme suit :

1	1
2	1
3	2
4	3
5	5
6	8
7	13

Simplification :

Algorithme X ;

Variables

Suite : Tableau [1..7] d'entier ;

i : entier ;

Début

Suite[1]  $\leftarrow$  1 ;

Suite[2]  $\leftarrow$  1 ;

Ecrire(Suite[1]) ;

Ecrire(Suite[2]) ;

Pour i  $\leftarrow$  3 à 7 Faire début

Suite[i]  $\leftarrow$  Suite[i-1] + Suite[i-2] ;

Ecrire(Suite[i]) ;

fin ;

Fin.

**Solution 9 :**

Algorithme Min2\_tab ;

Variables

T : tableau [1..10] d'entier ;

min1, min2, i : entier ;

Début

Ecrire('Donnez les éléments du tableau :') ;

Pour i  $\leftarrow$  1 à 10 Faire Lire(T[i]) ;

min1  $\leftarrow$  T[1] ;

min2  $\leftarrow$  T[2] ;

Pour i  $\leftarrow$  3 à 10 Faire

Si (min1 > min2) Alors début Si (min1 > T[i]) Alors min1  $\leftarrow$  T[i]; fin

Sinon Si (min2 > T[i]) Alors min2  $\leftarrow$  T[i] ;

Ecrire('Les deux valeurs minimales sont : ') ;



Si (min1 < min2) Alors Ecrire(min1, ' ', min2)

Sinon Ecrire(min2, ' ', min1);

Fin.

Le programme Pascal :

program Min2\_tab ;

var

T : array [1..10] of integer ;

min1, min2, i : integer;

begin

writeln('Donnez les éléments du tableau :');

for i := 1 to 10 do readln(T[i]);

min1 := T[1];

min2 := T[2];

for i := 3 to 10 do

if (min1 > min2) then begin if (min1 > T[i]) then min1 := T[i]; end

else if (min2 > T[i]) then min2 := T[i];

write('Les deux valeurs minimales sont : ');

if (min1 < min2) then writeln(min1, ' ', min2)

else writeln(min2, ' ', min1);

end.

**Solution 10 :**

**En utilisant deux boucles :**

Algorithme Min\_tab ;

Variables

T : tableau [1..10] d'entier ;

min, nb, i : entier ;

Début

Ecrire('Donnez les éléments du tableau :');

Pour i ← 1 à 10 Faire Lire(T[i]);

min ← T[1];

nb ← 1 ;

Pour i ← 2 à 10 Faire Si (min > T[i]) Alors min ← T[i] ;

Pour i ← 2 à 10 Faire Si (min = T[i]) Alors nb ← nb + 1 ;

Ecrire('Le minimum = ', min, '. Il apparait ', nb, ' fois.');

Fin.

Le programme Pascal :

program Min\_tab ;

var

T : array [1..10] of integer ;

```

    min, nb, i: integer ;
begin
    writeln('Donnez les éléments du tableau :') ;
    for i := 1 to 10 do readln(T[i]);
    min := T[1];
    nb := 1 ;
    for i := 2 to 10 do if (min > T[i]) then min := T[i];
    for i := 2 to 10 do if (min = T[i]) then nb := nb + 1;
    writeln('Le minimum = ', min, '. Il apparait ', nb, ' fois. ');
end.

```

***En utilisant une seule boucle :***

Algorithme Min\_tab ;

Variables

T : tableau [1..10] d'entier ;

min, nb, i : entier ;

Début

Ecrire('Donnez les éléments du tableau :') ;

Pour i ← 1 à 10 Faire Lire(T[i]) ;

min ← T[1] ;

nb ← 1 ;

Pour i ← 2 à 10 Faire

Si (min = T[i]) Alors nb ← nb+1

    Sinon Si (min > T[i]) Alors début

        min ← T[i] ;

        nb ← 1 ;

    fin ;

Ecrire('Le minimum = ', min, '. Il apparait ', nb, ' fois. ');

Fin.

**Le programme Pascal :**

program Min\_tab ;

var

T : array [1..10] of integer ;

min, nb, i: integer;

begin

writeln('Donnez les éléments du tableau :') ;

for i := 1 to 10 do readln(T[i]);

min := T[1];

nb := 1;

for i := 2 to 10 do

```

if (min = T[i]) then nb:=nb+1
else if (min > T[i]) then begin
    min := T[i];
    nb := 1 ;
end;
writeln('Le minimum = ', min, '. Il apparait ', nb, ' fois.');
```

end.

### **Solution 11 :**

Algorithme valeurs\_nég\_pos ;

Constantes

Max = 10 ;

Variables

Nb, Nbpos, Nbneg, i : entier ;

T : Tableau[1..Max] d'entier ;

Début

Pour i ← 1 à Max Faire T[i] ← 0 ;

Ecrire('Entrez le nombre de valeurs :') ;

Lire(Nb) ;

Si (Nb <= Max) Alors début

Nbpos ← 0 ;

Nbneg ← 0 ;

Pour i ← 1 à Nb Faire début

Ecrire('Entrez la valeur n° ', i) ;

Lire(T[i]) ;

Si (T[i] >= 0) Alors Nbpos ← Nbpos + 1

Sinon Nbneg ← Nbneg + 1 ;

fin ;

Ecrire('Nombre de valeurs positives : ', Nbpos) ;

Ecrire('Nombre de valeurs négatives : ', Nbneg) ;

fin

Sinon Ecrire('Le nombre doit être inférieur ou égal à 10.')

Fin.

Le programme Pascal :

program valeurs\_neg\_pos ;

const

Max = 10 ;

var

Nb, Nbpos, Nbneg, i : integer ;

T : array[1..Max] of integer ;

```

begin
  for i := 1 to Max do T[i] := 0;
  writeln('Entrez le nombre de valeurs :');
  readln(Nb);
  if(Nb <= Max) then begin
    Nbpos := 0;
    Nbneg := 0;
    for i := 1 to Nb do begin
      writeln('Entrez la valeur n° ', i);
      readln(T[i]);
      if (T[i] >= 0) then Nbpos := Nbpos + 1
      else Nbneg := Nbneg + 1;
    end;
    writeln('Nombre de valeurs positives : ', Nbpos);
    writeln('Nombre de valeurs négatives : ', Nbneg);
  end
  else writeln('Le nombre doit être inférieur ou égal à 10. ');
end.

```

**Solution 12 :**

Algorithme Somme\_tab ;

Variables

T1, T2, T3 : tableau [1..10] d'entier ;

i : entier ;

Début

Ecrire('Donnez les éléments du premier tableau :') ;

Pour i ← 1 à 10 Faire Lire(T1[i]) ;

Ecrire('Donnez les éléments du deuxième tableau :') ;

Pour i ← 1 à 10 Faire Lire(T2[i]) ;

Pour i ← 1 à 10 Faire T3[i] ← T1[i] + T2[i] ;

Ecrire('Voici la somme des deux tableaux :') ;

Pour i ← 1 à 10 Faire Ecrire(T3[i]) ;

Fin.

Le programme Pascal :

```

program Somme_tab ;

```

```

var

```

```

  T1, T2, T3 : array [1..10] of integer ;

```

```

  i : integer;

```

```
begin
  writeln('Donnez les éléments du premier tableau :');
  for i := 1 to 10 do readln(T1[i]);
  writeln('Donnez les éléments du deuxième tableau :');
  for i := 1 to 10 do readln(T2[i]);

  for i := 1 to 10 do T3[i] := T1[i] + T2[i];
  writeln('Voici la somme des deux tableaux :');
  for i := 1 to 10 do writeln(T3[i]);
end.
```

### **Solution 13 :**

Algorithme Schtroumpf ;

Variables

i, j, S : entier ;

T1, T2 : tableau[1..10] d'entier ;

Début

Ecrire('Donnez les éléments du premier tableau :');

Pour i ← 1 à 10 Faire Lire(T1[i]);

Ecrire('Donnez les éléments du deuxième tableau :');

Pour i ← 1 à 10 Faire Lire(T2[i]);

S ← 0 ;

Pour i ← 1 à 10 Faire

    Pour j ← 1 à 10 Faire S ← S + T1[i] \* T2[j] ;

Ecrire('Le schtroumpf est : ', S) ;

Fin.

### Le programme Pascal :

```
program Schtroumpf ;
```

```
var
```

```
  i, j, S : integer ;
```

```
  T1, T2 : array[1..10] of integer ;
```

```
begin
```

```
  writeln('Donnez les éléments du premier tableau :');
```

```
  for i := 1 to 10 do readln(T1[i]);
```

```
  writeln('Donnez les éléments du deuxième tableau :');
```

```
  for i := 1 to 10 do readln(T2[i]);
```

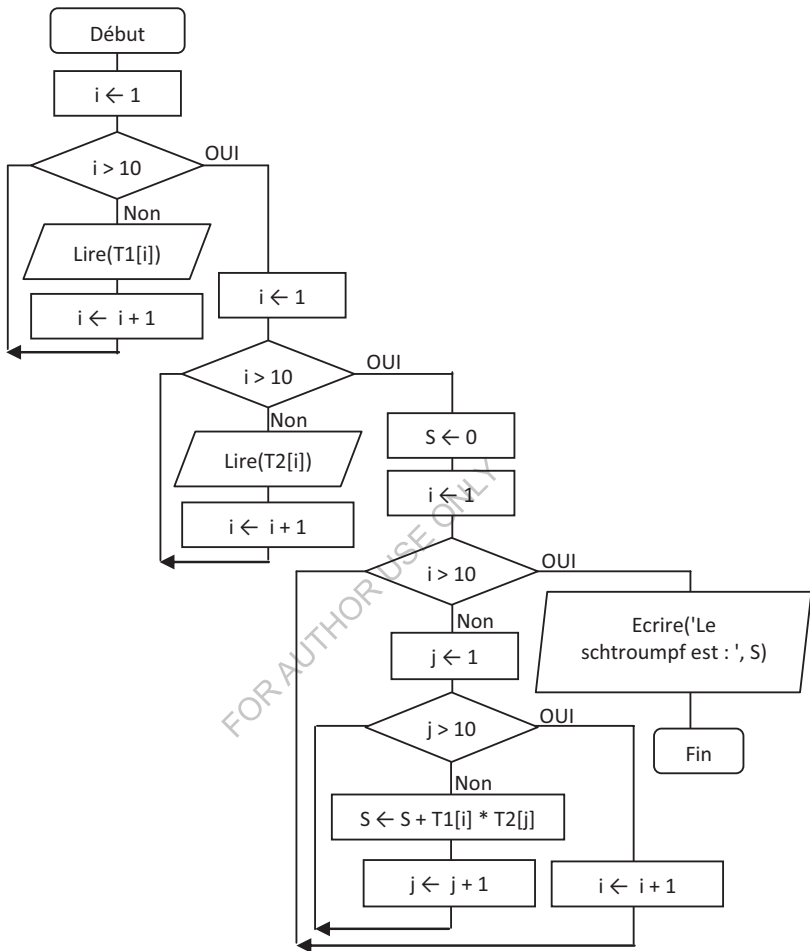
```
  S := 0 ;
```

```
  for i := 1 to 10 do for j := 1 to 10 do S := S + T1[i] * T2[j];
```

```
  writeln('Le schtroumpf est : ', S);
```

```
end.
```

L'organigramme correspondant à l'algorithme :



**Solution 14 :**

**Le tri par sélection :**

Algorithme Tri\_tab\_par\_sélection ;

Variables

T : tableau [1..10] d'entier ;

x, i, j : entier ;

Début

```

Ecrire('Donnez les éléments du tableau :') ;
Pour i ← 1 à 10 Faire Lire(T[i]) ;
Pour i ← 1 à 9 Faire
    Pour j ← i+1 à 10 Faire Si (T[i] > T[j]) Alors début
        x ← T[i] ;
        T[i] ← T[j] ;
        T[j] ← x ;
    fin ;
Ecrire('Voici les éléments du tableau après le tri :') ;
Pour i ← 1 à 10 Faire Ecrire(T[i]) ;
Fin.

```

*Le programme Pascal :*

```

program Tri_tab_par_selection ;
var
    T : array [1..10] of integer ;
    x, i, j : integer;
begin
    writeln('Donnez les éléments du tableau :') ;
    for i := 1 to 10 do readln(T[i]);
    for i := 1 to 9 do
        for j := i+1 to 10 do if (T[i] > T[j]) then begin
            x := T[i] ;
            T[i] := T[j] ;
            T[j] := x ;
        end;
    writeln('Voici les éléments du tableau après le tri :') ;
    for i := 1 to 10 do writeln(T[i]) ;
end.

```

***Le tri à bulles :***

Algorithme Tri\_tab\_bulles ;

Variables

```

T : tableau [1..10] d'entier ;
i, x : entier ;
Yapermut : booléen ;

```

Début

```

Ecrire('Donnez les éléments du tableau :') ;
Pour i ← 1 à 10 Faire Lire(T[i]) ;
Yapermut ← VRAI ;
Tant que (Yapermut) Faire début

```

```

Yapermut ← FAUX ;
Pour i ← 1 à 9 Faire Si (t[i] > t[i+1]) Alors début
    x ← t[i] ;
    t[i] ← t[i+1] ;
    t[i+1] ← x ;
    Yapermut ← VRAI ;
fin ;

fin ;
Ecrire('Voici les éléments du tableau après le tri :') ;
Pour i ← 1 à 10 Faire Ecrire(T[i]);
Fin.

```

*Le programme Pascal :*

```

program Tri_tab_bulles ;
var
    T : array[1..10] of integer ;
    i, x : integer;
    Yapermut : boolean ;
begin
    writeln('Donnez les éléments du tableau :') ;
    for i := 1 to 10 do readln(T[i]);
    Yapermut := TRUE ;
    while (Yapermut) do begin
        Yapermut := FALSE ;
        for i := 1 to 9 do if (t[i] > t[i+1]) then begin
            x := t[i];
            t[i] := t[i+1];
            t[i+1] := x;
            Yapermut := TRUE;
        end;
    end ;
    writeln('Voici les éléments du tableau après le tri :') ;
    for i := 1 to 10 do writeln(T[i]);
end.

```

***Le tri à bulles (en utilisant une boucle Répéter) :***

```

Algorithme Tri_tab_bulles ;
Variables
    T : tableau [1..10] d'entier ;
    i, x : entier ;
    Permut : booléen ;

```



Début

Ecrire('Donnez les éléments du tableau :') ;

Pour i ← 1 à 10 Faire Lire(T[i]) ;

Répéter

Permut ← FAUX ;

Pour i ← 1 à 9 Faire Si (t[i] > t[i+1]) Alors début

    x ← t[i] ;

    t[i] ← t[i+1] ;

    t[i+1] ← x ;

    Permut ← VRAI ;

fin ;

Jusqu'à Non Permut ;

Ecrire('Voici les éléments du tableau après le tri :') ;

Pour i ← 1 à 10 Faire Ecrire(T[i]);

Fin.

En Pascal :

program Tri\_tab\_bulles ;

var

T : array [1..10] of integer ;

i, x : integer ;

Permut : boolean ;

Begin

writeln('Donnez les éléments du tableau :') ;

for i := 1 to 10 do readln(T[i]) ;

repeat

    Permut := false ;

    for i := 1 to 9 do if (t[i] > t[i+1]) then begin

        x := t[i] ;

        t[i] := t[i+1] ;

        t[i+1] := x ;

        Permut := true ;

    end ;

until not Permut ;

writeln('Voici les éléments du tableau après le tri :') ;

for i := 1 to 10 do writeln(T[i]);

end.

**Solution 15 :**

Algorithme Eléments\_consécutifs1 ;

Variables

```

T : tableau [1..7] d'entier ;
i : entier ;
Cons : booléen ;
Début
  Ecrire('Entrez les valeurs du tableau :') ;
  Pour i ← 1 à 7 Faire début
    Ecrire('Entrez la valeur n° ', i, ' : ' ) ;
    Lire(T[i]) ;
  fin ;
  Cons ← VRAI ;
  Pour i ← 1 à 6 Faire
    Si (T[i] <> T[i + 1] - 1) Alors Cons ← FAUX ;
  Si Cons Alors Ecrire('Les éléments sont consécutifs.')
  Sinon Ecrire('Les éléments ne sont pas consécutifs.') ;
Fin.

```

Le programme Pascal :

```

program Elements_consecutifs1 ;
var
  T : array[1..7] of integer ;
  i : integer ;
  Cons : boolean ;
begin
  writeln('Entrez les valeurs du tableau :') ;
  for i := 1 to 7 do begin
    writeln('Entrez la valeur n° ', i, ' : ' ) ;
    readln(T[i]) ;
  end ;
  Cons := TRUE ;
  for i := 1 to 6 do if (T[i] <> T[i + 1] - 1) then Cons := FALSE ;
  if Cons then writeln('Les éléments sont consécutifs.')
  else writeln('Les éléments ne sont pas consécutifs.') ;
end.

```

Cette solution est sans doute la plus spontanée, mais elle présente le défaut d'examiner la totalité du tableau, même lorsqu'on découvre dès le départ deux éléments non consécutifs. Aussi, dans le cas d'un grand tableau, cette solution est coûteuse en temps de traitement. Une autre manière de procéder serait de sortir de la boucle dès que deux éléments non consécutifs sont détectés. Une deuxième solution est donc possible. Algorithme Eléments\_consecutifs2 ;

### Variables

T : tableau [1..7] d'entier ;

i : entier ;

Cons : booléen ;

### Début

Ecrire('Entrez les valeurs du tableau :') ;

Pour i ← 1 à 7 Faire début

    Ecrire('Entrez la valeur n° ', i, ' : ' ) ;

    Lire(T[i]) ;

fin ;

Cons ← VRAI ;

i ← 1 ;

Tant que ((Cons) ET (i < 7)) Faire

    Si (T[i] <> T[i + 1] - 1) Alors Cons ← FAUX

    Sinon i ← i + 1 ;

Si Cons Alors Ecrire('Les éléments sont consécutifs.')

    Sinon Ecrire('Les éléments ne sont pas consécutifs.') ;

Fin.

### **Solution 16 :**

Algorithme Inverse\_tab ;

### Variables

T : tableau [1..10] d'entier ;

i, x : entier ;

### Début

Ecrire('Entrez les valeurs du tableau :') ;

Pour i ← 1 à 10 Faire Lire(T[i]) ;

Pour i ← 1 à 5 Faire début

    x ← T[i] ;

    T[i] ← T[10 - i + 1] ;

    T[10 - i + 1] ← x ;

fin ;

Ecrire('Voici les valeurs du tableau après l'inversement :') ;

Pour i ← 1 à 10 Faire Ecrire(T[i]) ;

Fin.

### Le programme Pascal :

program Inverse\_tab ;

var

    T : array [1..10] of integer ;

    i, x : integer ;

```
begin
  writeln('Entrez les valeurs du tableau :') ;
  for i := 1 to 10 do readln(T[i]) ;
  for i := 1 to 5 do begin
    x := T[i] ;
    T[i] := T[10 - i + 1] ;
    T[10 - i + 1] := x ;
  end ;
  writeln('Voici les valeurs du tableau après l'inversement :') ;
  for i := 1 to 10 do writeln(T[i]) ;
end.
```

**Solution 17 :**

Algorithme fibonacci ;

Constantes

max = 10 ;

Variables

fibonacci : tableau[0..max] d'entier ;

n, i : entier ;

Début

Ecrire('Donnez un entier :') ;

Lire(n) ;

Si (n <= max) alors début

fibonacci[0] ← 0;

fibonacci[1] ← 1;

Pour i ← 2 à n Faire fibonacci[i] ← fibonacci[i-1] + fibonacci[i-2];

Ecrire('Fibonacci(', n, ') = ', fibonacci[n]) ;

fin;

Fin.

Le programme Pascal :

program fibonacci ;

const

max = 10 ;

var

fibonacci : array[0..max] of integer ;

n, i : integer;

begin

writeln('Donnez un entier :') ;

readln(n) ;

if (n <= max) then begin

```

    fibo[0] := 0;
    fibo[1] := 1;
    for i := 2 to n do fibo[i] := fibo[i-1] + fibo[i-2];
    writeln('Fibonacci(', n, ') = ', fibo[n]);
end;
end.

```

### **Solution 18 :**

Algorithme Supprimer\_case\_tab ;

Variables

T : tableau [1..10] d'entier ;

i, j : entier ;

Début

Ecrire('Saisir les dix éléments entiers du tableau :');

Pour i ← 1 à 10 Faire Lire(T[i]);

Ecrire('Donnez la position de la case à supprimer (prise entre 1 et 10 :)'); Lire(j);

Pour i ← j à 9 Faire T[i] ← T[i+1];

T[10] ← 0;

Ecrire('Voici les éléments du tableau après la suppression de la case num ', j);

Pour i ← 1 à 10 Faire Ecrire(T[i]);

Le programme Pascal :

program Supprimer\_case\_tab ;

var

T : array [1..10] of integer ;

i, j : integer;

begin

writeln('Saisir les dix éléments entiers du tableau :');

for i := 1 to 10 do readln(T[i]);

writeln('Donnez la position de la case à supprimer (prise entre 1 et 10 :)'); readln(j);

for i := j to 9 do T[i] := T[i+1];

T[10] := 0;

writeln('Voici les éléments du tableau après le suppression de la case num ', j); for i := 1 to 10 do writeln(T[i]);

end.

### **Solution 19 :**

Algorithme Insérer\_case\_tab ;

Variables

T : tableau [1..10] d'entier ;

i, val, pos : entier ;

Flag : booléen ;

Début

Ecrire('Saisir les éléments triés par ordre croissant d'un tableau de dix entiers :');

Pour i ← 1 à 10 Faire Lire(T[i]);

Ecrire('Donnez la valeur à insérer :');

Lire(val) ;

i ← 1 ;

pos ← 11 ;

Flag ← FAUX ;

Tant que (i <= 10) ET (NON Flag) Faire début

Si val <= T[i] Alors début

pos ← i ;

Flag ← VRAI ;

fin ;

i ← i + 1 ;

fin ;

if (pos <> 11) Alors début

Pour i ← 10 à pos+1 Faire T[i] ← T[i-1] ;

T[pos] ← val ;

fin ;

Ecrire('Voici les éléments du tableau après l'insertion de la valeur ', val);

pour i ← 1 à 10 Faire Ecrire(T[i]);

Fin.

Le programme Pascal :

program Insérer\_case\_tab ;

var

T : array [1..10] of integer ;

i, val, pos : integer;

Flag : boolean ;

begin

writeln('Saisir les éléments triés par ordre croissant d'un tableau de dix entiers :');

for i := 1 to 10 do readln(T[i]);

writeln('Donnez la valeur à insérer :');

readln(val) ;

i := 1 ;

pos := 11;

Flag := FALSE;

```
while (i <= 10) AND (NOT Flag) do begin
  if val <= T[i] then begin
    pos := i ;
    Flag := TRUE;
  end;
  i := i + 1 ;
end ;
if (pos <> 11) then begin
  for i := 10 downto pos+1 do T[i] := T[i-1] ;
  T[pos] := val ;
end;
writeln('Voici les éléments du tableau après l\'insertion de la valeur ', val) ;
for i := 1 to 10 do writeln(T[i]) ;
end.
```

**Solution 20 :**

Algorithme triangle\_Pascal ;

Constantes

max = 10 ;

Variables

Ligne : tableau [1..max] d'entier ;

Ligne\_preced : tableau [1..max] d'entier ;

n, i, j : entier ;

Début

Pour i ← 1 à max Faire Ligne[i] ← 0 ;

Pour i ← 1 à max Faire Ligne\_preced[i] ← 0 ;

Ecrire('Entrez le numéro de la ligne du triangle de Pascal :') ;

Lire(n) ;

Si (n = 1) Alors Ligne[1]:=1

  Sinon Si (n = 2) Alors début

    Ligne[1] ← 1 ;

    Ligne[2] ← 1;

  fin

  Sinon Si (n > 2) Alors début

    Ligne\_preced[1] ← 1 ;

    Ligne\_preced[2] ← 1 ;

    Ligne[1] ← 1 ;

    Pour i ← 3 à n Faire début

      Pour j ← 2 à i-1 Faire

        Ligne[j] ← Ligne\_preced[j-1] + Ligne\_preced[j] ;

```

        Ligne[i] ← 1 ;
        Pour j ← 1 à i Faire Ligne_preced[j] ← Ligne[j] ;
    fin ;
fin ;
Ecrire('Voici les éléments de cette ligne : ' ) ;
Pour i ← 1 à n Faire Ecrire(Ligne[i], ' ');
Fin.

```

Le programme Pascal :

```

program triangle_Pascal ;
const
    max = 10;
var
    Ligne: array[1..max] of integer;
    Ligne_preced : array[1..max] of integer;
    n, i, j : integer;
begin
    for i := 1 to max do Ligne[i] := 0;
    for i := 1 to max do Ligne_preced[i] := 0;
    writeln('Entrez le numéro de la ligne du triangle de Pascal :');
    readln(n) ;
    if (n = 1) then Ligne[1]:=1
    else if (n = 2) then begin
        Ligne[1]:=1;
        Ligne[2] := 1;
    end
    else if (n > 2) then begin
        Ligne_preced[1] := 1;
        Ligne_preced[2] := 1;
        Ligne[1] := 1;
        for i := 3 to n do begin
            for j := 2 to i-1 do
                Ligne[j] := Ligne_preced[j-1] + Ligne_preced[j];
            Ligne[i] := 1;
            for j := 1 to i do Ligne_preced[j] := Ligne[j] ;
        end;
    end;
    writeln('Voici les éléments de cette ligne : ' ) ;
    for i := 1 to n do write(Ligne[i], ' ');
    writeln;
end.

```



### Solution 21 :

Algorithme Init\_matrice ;

Variables

T: tableau [1..3, 1..10] d'entier ;

i, j : entier ;

Début

Pour i ← 1 à 3 Faire

    Pour j ← 1 à 10 Faire T[i,j] ← 0 ;

    Ecrire('Voici la matrice initialisée :') ;

    Pour i ← 1 à 3 Faire

        Pour j ← 1 à 10 Faire Ecrire(T[i,j], ' ');

Fin.

### Le programme Pascal :

program Init\_matrice ;

var

    T : array [1..3, 1..10] of integer ;

    i, j : integer;

begin

    for i := 1 to 3 do for j := 1 to 10 do T[i,j]:=0;

    writeln('Voici la matrice initialisée :') ;

    for i := 1 to 3 do begin

        for j := 1 to 10 do write(T[i,j], ' ');

        writeln;

    end;

end.

### Solution 22 :

Cet algorithme remplit le tableau X de la manière suivante :

X[1,1] = 1, X[1,2] = 2, X[1,3] = 3, X[2,1] = 4, X[2,2] = 5, X[2,3] = 6

Il écrit ensuite ses valeurs à l'écran, dans cet ordre : 1, 2, 3, 4, 5, 6.

### Solution 23 :

Cet algorithme remplit le tableau X de la manière suivante :

X[1,1] = 1, X[1,2] = 2, X[1,3] = 3, X[2,1] = 4, X[2,2] = 5, X[2,3] = 6.

Il écrit ensuite ses valeurs à l'écran, dans cet ordre : 1, 4, 2, 5, 3, 6.

### Solution 24 :

Cet algorithme remplit un tableau d'entiers T à deux dimensions 4\*2, de la manière suivante :

T[1,1] = 2, T[1,2] = 3, T[2,1] = 3, T[2,2] = 4, T[3,1] = 4, T[3,2] = 5, T[4,1] = 5, T[4,2] = 6.

Il écrit ensuite ses valeurs à l'écran, dans cet ordre : 2, 3, 3, 4, 4, 5, 5,

6. Le tableau T peut être représenté donc comme suit :

	1	2
1	2	3
2	3	4
3	4	5
4	5	6

**Solution 25 :**

**Version a :**  $T[k, m] \leftarrow 2 * k + (m + 1)$ .

Cet algorithme remplit le tableau T de la manière suivante :

$T[1,1] = 4$ ,  $T[1,2] = 5$ ,  $T[2,1] = 6$ ,  $T[2,2] = 7$ ,  $T[3,1] = 8$ ,  $T[3,2] = 9$ ,  
 $T[4,1] = 10$ ,  $T[4,2] = 11$ .

Il écrit ensuite ses valeurs à l'écran, dans cet ordre : 4, 5, 6, 7, 8, 9, 10, 11.

**Version b :**  $T[k, m] \leftarrow (k + 1) + 4 * m$ .

Cet algorithme remplit le tableau T de la manière suivante :

$T[1,1] = 6$ ,  $T[1,2] = 10$ ,  $T[2,1] = 7$ ,  $T[2,2] = 11$ ,  $T[3,1] = 8$ ,  $T[3,2] = 12$ ,  
 $T[4,1] = 9$ ,  $T[4,2] = 13$ .

Il écrit ensuite ses valeurs à l'écran, dans cet ordre : 6, 10, 7, 11, 8, 12, 9, 13.

**Solution 26 :**

Algorithme Nbr\_app ;

Variables

T: tableau [1..3, 1..10] d'entier ;

i, j, x, nb : entier ;

Début

Ecrire('Donnez les éléments de la matrice :') ;

Pour i  $\leftarrow$  1 à 3 Faire Pour j  $\leftarrow$  1 à 10 Faire Lire(T[i,j]) ;

Ecrire('Donnez la valeur dont vous désirez calculer le nombre d'apparition :') ;

Lire(x) ;

nb  $\leftarrow$  0 ;

Pour i  $\leftarrow$  1 à 3 Faire

    Pour j  $\leftarrow$  1 à 10 Faire

        Si (x = T[i,j]) Alors nb  $\leftarrow$  nb + 1 ;

Ecrire('Le nombre d'apparition de la valeur ', x, ' dans la matrice est = ', nb) ;

Fin.

Le programme Pascal :

program Nbr\_app ;

var

T : array [1..3, 1..10] of integer ;

i, j, x, nb : integer;

begin

    writeln('Donnez les éléments de la matrice :') ;

    for i := 1 to 3 do for j := 1 to 10 do readln(T[i,j]);

```
writeln('Donnez la valeur dont vous désirez calculer le nombre d'apparition :');  
readln(x);  
nb := 0;  
for i := 1 to 3 do  
    for j := 1 to 10 do  
        if (x = T[i,j]) then nb := nb + 1;  
    writeln('Le nombre d'apparition de la valeur ', x, ' dans la matrice est = ', nb);  
end.
```

**Solution 27 :**

Algorithme Somme\_lig\_col ;

Variables

T: tableau [1..10, 1..20] de réel ;

TotLig : tableau [1..10] de réel ;

TotCol : tableau [1..20] de réel ;

i, j : entier ;

Début

Ecrire('Donnez les valeurs de la matrice :');

Pour i ← 1 à 10 Faire Pour j ← 1 à 20 Faire Lire(T[i,j]) ;

Pour i ← 1 à 10 Faire TotLig[i] ← 0 ;

Pour i ← 1 à 20 Faire TotCol[i] ← 0 ;

Pour i ← 1 à 10 Faire Pour j ← 1 à 20 Faire début

TotLig[i] ← TotLig[i] + T[i,j] ;

TotCol[j] ← TotCol[j] + T[i,j] ;

fin ;

Ecrire('Total lignes :');

Pour i ← 1 à 10 Faire Ecrire(TotLig[i]) ;

Ecrire('Total colonnes :');

Pour i ← 1 à 20 Faire Ecrire(TotCol[i]) ;

Fin.

Le programme Pascal :

program Somme\_lig\_col ;

var

T: array[1..10, 1..20] of real ;

TotLig : array [1..10] of real ;

TotCol : array[1..20] of real ;

i, j : integer ;

begin

writeln('Donnez les valeurs de la matrice :');

for i := 1 to 10 do for j := 1 to 20 do readln(T[i,j]) ;

```

for i := 1 to 10 do TotLig[i] := 0;
for i := 1 to 20 do TotCol[i] := 0 ;
for i := 1 to 10 do for j := 1 to 20 do begin
    TotLig[i] := TotLig[i] + T[i,j];
    TotCol[j] := TotCol[j] + T[i,j] ;
end ;
writeln('Total lignes :');
for i := 1 to 10 do writeln(TotLig[i]);
writeln('Total colonnes :');
for i := 1 to 20 do writeln(TotCol[i]);
end.

```

**Solution 28 :**

Algorithme Transposee\_matrice ;

Variables

T : tableau [1..3, 1..3] d'entier ;

i, j, x : entier ;

Début

Ecrire('Donnez la matrice initiale :');

Pour i ← 1 à 3 Faire Pour j ← 1 à 3 Faire Lire(T[i,j]) ;

Ecrire('Matrice initiale :');

Pour i ← 1 à 3 Faire Pour j ← 1 à 3 Faire Ecrire(T[i,j]);

Pour i ← 1 à 2 Faire

    Pour j ← i+1 à 3 Faire début

        x ← T[i,j] ;

        T[i,j] ← T[j,i] ;

        T[j,i] ← x ;

    fin ;

Ecrire('Matrice transposée :');

Pour i ← 1 à 3 Faire Pour j ← 1 à 3 Faire Ecrire(T[i,j]) ;

Fin .

Le programme Pascal :

program Transposee\_matrice ;

var

T : array [1..3, 1..3] of integer ;

i, j, x: integer;

begin

writeln('Donnez la matrice initiale :');

for i := 1 to 3 do for j := 1 to 3 do readln(T[i,j]);

writeln('Matrice initiale :');

```

for i := 1 to 3 do begin
  for j := 1 to 3 do write(T[i,j]:3);
  writeln;
end;
for i := 1 to 2 do
  for j := i+1 to 3 do begin
    x := T[i,j];
    T[i,j] := T[j,i];
    T[j,i] := x;
  end;
writeln('Matrice transposée : ');
for i := 1 to 3 do begin
  for j := 1 to 3 do write(T[i,j]:3);
  writeln;
end;
end.

```

### **Solution 29 :**

Algorithme produit\_matriciel ;

Variables

mat1, mat2, produit : tableau [1..3, 1..3] d'entier ;

i, j, k, x, s: entier ;

Début

Ecrire('Donnez la première matrice :');

Pour i ← 1 à 3 Faire Pour j ← 1 à 3 Faire Lire(mat1[i,j]);

Ecrire('Donnez la deuxième matrice :');

Pour i ← 1 à 3 Faire Pour j ← 1 à 3 Faire Lire(mat2[i,j]);

Pour i ← 1 à 3 Faire

  Pour j ← 1 à 3 Faire début

    s := 0 ;

    Pour k ← 1 à 3 Faire s ← s + mat1[i,k] \* mat2[k,j] ;

    produit[i,j] ← s ;

  fin ;

Ecrire('Voici le résultat du produit matriciel :');

Pour i ← 1 à 3 Faire Pour j ← 1 à 3 Faire Ecrire(produit[i,j]) ;

Fin.

### Le programme Pascal :

program produit\_matriciel ;

var

  mat1, mat2, produit : array [1..3, 1..3] of integer ;

```

i, j, k, x, s: integer;
begin
  writeln('Donnez la première matrice :');
  for i := 1 to 3 do for j := 1 to 3 do readln(mat1[i,j]);
  writeln('Donnez la deuxième matrice :');
  for i := 1 to 3 do for j := 1 to 3 do readln(mat2[i,j]);
  for i := 1 to 3 do
    for j := 1 to 3 do begin
      s := 0;
      for k := 1 to 3 do s := s + mat1[i,k] * mat2[k,j];
      produit[i,j] := s ;
    end;
  writeln('Voici le résultat du produit matriciel :');
  for i := 1 to 3 do begin
    for j := 1 to 3 do write(produit[i,j]:3);
    writeln;
  end;
end.

```

**Solution 30 :**

Le programme affiche : azertyqwerty

**Solution 31 :**

Le programme permet de lire une chaîne de caractères à partir du clavier, de convertir son premier caractère en majuscule, de calculer sa longueur, si celle-ci est supérieure à trois, alors réduire la chaîne en ses trois premiers caractères, et enfin afficher cette chaîne de caractères. Par exemple, si le chaîne saisie est 'bonjour', elle devient 'Bonjour', sa longueur est 7, alors elle sera réduite en 'Bon', et on affiche enfin Bon.

**Solution 32 :**

Algorithme identification ;

Variables

lpr : entier ;

prénom, nom, ident : chaîne de caractères;

Début

Ecrire('Quel est votre prénom?') ;

Lire(prénom) ;

Ecrire('et votre nom?') ;

Lire(nom) ;

ident ← nom // "." // prénom ;

lpr ← |ident| ;

```
Ecrire('Votre identification est : ', ident) ;  
Ecrire('La longueur de la chaîne résultante est : ', lpr) ;  
Fin.
```

Le programme Pascal :

```
program identification ;  
var  
  lpr : integer ;  
  prenom, nom, ident : string ;  
begin  
  writeln('Quel est votre prénom ?') ;  
  readln(prenom) ;  
  writeln('et votre nom ?') ;  
  readln(nom) ;  
  ident := CONCAT(nom, '.') ;  
  ident := CONCAT(ident, prenom) ;  
  lpr := length(ident) ;  
  writeln('Votre identification est : ', ident) ;  
  writeln('La longueur de la chaîne résultante est : ', lpr) ;  
end.
```

**Solution 33 :**

Algorithme Ordre\_châînes ;

Variables

CH1, CH2 : chaîne de caractères;

Début

```
Ecrire('Donnez la première chaîne :') ;  
Lire(CH1) ;  
Ecrire('Donnez la deuxième chaîne :') ;  
Lire(CH2) ;  
Si (CH1 < CH2) Alors Ecrire(CH1, ' ', CH2)  
  Sinon Ecrire(CH2, ' ', CH1) ;
```

Fin.

Le programme Pascal :

```
program Ordre_châînes ;  
var  
  CH1, CH2 : string ;  
begin  
  writeln('Donnez la première chaîne :') ;  
  readln(CH1) ;  
  writeln('Donnez la deuxième chaîne :') ;
```

```

readln(CH2) ;
if (CH1 < CH2) then writeln(CH1, ' ', CH2)
    else writeln(CH2, ' ', CH1) ;
end.

```

**Solution 34 :**

Algorithme nbr\_caractères ;

Variables

CH : chaîne de caractères[20] ;

C : caractère ;

nb, i : entier ;

Début

Ecrire('Donnez une chaîne de 20 caractères maximum :') ;

Lire(CH) ;

Ecrire('Entrez un caractère :') ;

Lire(C) ;

nb ← 0 ;

Pour i ← 1 à long(CH) Faire Si (CH[i] = C) Alors nb ← nb + 1 ;

Ecrire('Le caractère ', C, ' est apparu ', nb, ' fois dans la chaîne ', CH) ;

Fin.

Le programme Pascal :

program nbr\_caracteres ;

var

CH : string[20];

C : char ;

nb, i : integer ;

begin

writeln('Donnez une chaîne de 20 caractères maximum :') ;

readln(CH) ;

writeln('Entrez un caractère :') ;

readln(C) ;

nb := 0 ;

for i := 1 to length(CH) do if (CH[i] = C) then nb := nb + 1 ;

writeln('Le caractère ', C, ' est apparu ', nb, ' fois dans la chaîne ', CH) ;

end.

**Solution 35 :**

Algorithme nbr\_mots ;

Variables

CH : chaîne de caractères;

nb, i : entier ;



Début

```
Ecrire('Donnez une chaîne de caractères :') ; Lire(CH) ;
nb ← 1 ;
Pour i ← 1 à long(CH) Faire Si (CH[i]=' ') Alors nb ← nb + 1 ;
Ecrire('Le nombre de mots dans ce texte est : ', nb) ;
```

Fin.

Le programme Pascal :

```
program nbr_mots ;
```

```
var
```

```
CH : string ;
nb, i : integer ;
```

```
begin
```

```
writeln('Donnez une chaîne de caractères :') ; readln(CH) ;
nb := 1 ;
for i := 1 to length(CH) do if (CH[i]=' ') then nb := nb + 1 ;
writeln('Le nombre de mots dans ce texte est : ', nb) ;
```

```
end.
```

**Solution 36 :**

Algorithme nbr\_occurrence ;

Variables

```
texte, CH, str : chaîne de caractères ;
nb, i : entier ;
```

Début

```
Ecrire('Donnez un texte :') ;
Lire(texte) ;
Ecrire('Donnez un mot :') ;
Lire(CH) ;
nb := 0 ; i := 1 ;
Tant que i <= length(texte) Faire début
  str := '' ;
  Tant que (texte[i] <> ' ') ET (i <= length(texte)) Faire début
    str := str + texte[i] ;
    i := i + 1 ;
  fin ;
  i := i + 1 ;
  Si (str = CH) Alors nb := nb + 1 ;
fin ;
```

```
Ecrire('Le nombre d'apparition du mot : ', CH, ' dans le texte est égal à ', nb) ;
```

Fin.

Le programme Pascal :

```

program nbr_occurrence ;
var
    texte, CH, str : string ;
    nb, i : integer ;
begin
    writeln('Donnez un texte :) ;
    readln(texte) ;
    writeln('Donnez un mot :) ;
    readln(CH) ;
    nb := 0 ;
    i := 1 ;
    while i <= length(texte) do begin
        str := '' ;
        while (texte[i] <> ' ') AND (i <= length(texte)) do begin
            str := str + texte[i] ;
            i := i + 1 ;
        end ;
        i := i + 1 ;
        if (str = CH) then nb := nb + 1 ;
    end ;
    writeln('Le nombre d'apparition du mot : ', CH, ' dans le texte est égal à ', nb) ;
end.

```

**Solution 37 :**

Algorithme conv\_en\_binaire ;

Variable

n, Q, R : entier ;  
 binaire : chaîne de caractères;

Début

Ecrire('Donnez un entier :) ;  
 Lire(n) ;  
 $Q \leftarrow n \div 2$ ;  
 $R \leftarrow n \bmod 2$ ;  
 Si  $(R = 0)$  Alors binaire  $\leftarrow$  '0'  
 Sinon binaire  $\leftarrow$  '1' ;  
 Tant que  $(Q > 0)$  Faire début  
 $n \leftarrow Q$  ;  
 $Q \leftarrow n \div 2$ ;  
 $R \leftarrow n \bmod 2$  ;

```

Si (R = 0) Alors binaire ← '0' + binaire
Sinon binaire ← '1' + binaire ;
fin ;
Ecrire('Le nombre ', n, ' en binaire est : ', binaire) ;
Fin.

```

Le programme Pascal :

```

program conv_en_binaire ;
var
  n, Q, R : integer;
  binaire : string ;
begin
  writeln('Donnez un entier :) ;
  readln(n) ;
  Q := n DIV 2;
  R := n MOD 2;
  if (R = 0) then binaire := '0'
  else binaire := '1' ;
  while (Q > 0) do begin
    n:= Q;
    Q := n DIV 2;
    R := n MOD 2;
    if (R = 0) then binaire := '0' + binaire
    else binaire := '1' + binaire ;
  end;
  writeln('Le nombre ', n, ' en binaire est : ', binaire) ;
end.

```

**Solution 38 :**

```

program Saisie ;
var
  T : array [1..10] of string[5];
  i : integer;
begin
  writeln('Saisir les chaînes de caractères :');
  for i:= 1 to 10 do readln(T[i]);
  writeln('Les derniers caractères des chaînes :');
  for i:= 1 to 10 do
    writeln('Le dernier caractère de la chaîne ', T[i], ' est ', T[i][length(T[i])]);
  end.

```

**Solution 39 :**

```

program tri_chaines ;
var
  mots : array [1..10] of string[20];
  i,j : integer;
  CH : string[20];
begin
  { Lire les chaînes de caractères }
  writeln('Donnez une liste de chaînes de caractères :');
  for i:= 1 to 10 do begin
    writeln('Donnez le mot n° ', i, ' :');
    readln(mots[i]);
  end;
  { Convertir les mots en majuscule }
  for i:= 1 to 10 do
    for j:=1 to length(mots[i])do mots[i,j]:=upcase(mots[i,j]);
  { Trier les chaînes par ordre alphabétique }
  for i:= 1 to 9 do
    for j:= i+1 to 10 do
      if (mots[i]>mots[j])then begin
        CH:= mots[i];
        mots[i]:= mots[j];
        mots[j]:= CH;
      end;
    { Afficher la liste des mots triés}
    writeln('Voici la liste des mots après le tri :');
    for i := 1 to 10 do writeln(mots[i]);
  end.

```

**Solution 40 :**

```

Algorithme candidat_sup ;
Variables
  noms : tableau [1..10] de chaîne de caractères ;
  notes : tableau [1..10] de réel ;
  pos, i : entier ;
Début
  Ecrire('Donnez les noms et les notes des candidats :') ;
  Pour i←1 à 10 Faire début
    Ecrire('Saisir le nom du candidat n° : ', i, ' :');
    Lire(noms[i]) ;

```

```
Ecrire('Saisir la note de ', noms[i], ' :') ;  
Lire(notes[i]) ;  
fin ;  
pos ← 1 ;  
Pour i←2 à 10 Faire Si (notes[pos] <= notes[i]) Alors pos ← i ;  
Ecrire(noms[pos], ' a eu une meilleure moyenne de ', notes[pos]) ;  
Fin.
```

Le programme Pascal :

```
program candidat_sup ;  
var  
  noms : array [1..10] of string ;  
  notes : array [1..10] of real ;  
  pos, i : integer ;  
begin  
  writeln('Donnez les noms et les notes des candidats :)') ;  
  for i :=1 to 10 do begin  
    writeln('Saisir le nom du candidat n° : ', i, ' :') ;  
    readln(noms[i]) ;  
    writeln('Saisir la note de ', noms[i], ' :') ;  
    readln(notes[i]) ;  
  end ;  
  pos := 1;  
  for i :=2 to 10 do if (notes[pos] <= notes[i]) then pos := i ;  
    writeln(noms[pos], ' a eu une meilleure moyenne de ', notes[pos]) ;  
  end.
```

**Solution 41 :**

Algorithme Liste\_candidats\_admis ;

Constantes MAX = 10 ;

Variables

noms : tableau [1..MAX] de chaîne de caractères;

notes : tableau [1..MAX] de réel ;

i : entier ;

Début

(\*Saisir les noms des candidats\*)

Ecrire('Donnez les noms des candidats :');

Pour i←1 à MAX Faire début

Ecrire('Saisir le nom du candidat n° : ', i, ' :') ;

Lire(noms[i]) ;

fin ;

```
(*Saisir les notes des candidats*)
Ecrire('Donnez les notes des candidats :') ;
Pour i ← 1 à MAX Faire début
    Ecrire('Saisir la note de ', noms[i], ' : ' ) ;
    Lire(notes[i]) ;
fin ;
(*Afficher la liste des candidats admis*)
Ecrire('Voici la liste des candidats admis :') ;
Pour i ← 1 à MAX Faire
    Si (notes[i] >= 10) Alors
        Ecrire(noms[i], ' admis avec une note de ', notes[i]) ;
Fin.
```

Le programme Pascal :

```
program Liste_candidats_admis;
const MAX = 10 ;
var
    noms : array[1..MAX] of string ;
    notes : array[1..MAX] of real ;
    i : integer;
begin
    (*Saisir les noms des candidats*)
    writeln('Donnez les noms des candidats :')
    for i := 1 to MAX do begin
        writeln('Saisir le nom du candidat n° : ', i, ' : ' ) ;
        readln(noms[i]) ;
    end ;
    (*Saisir les notes des candidats*)
    writeln('Donnez les notes des candidats :') ;
    for i := 1 to MAX do begin
        writeln('Saisir la note de ', noms[i], ' : ' ) ;
        readln(notes[i]) ;
    end ;
    (*Afficher la liste des candidats admis*)
    writeln('Voici la liste des candidats admis :') ;
    for i := 1 to MAX do
        if (notes[i] >= 10) then
            writeln(noms[i], ' admis avec une note de ', notes[i]) ;
    end.
end.
```

**Solution 42 :**

Algorithme Liste\_candidats\_sup ;

Constantes

MAX = 10 ;

Variables

noms : tableau [1..MAX] de chaîne de caractères;

notes : tableau [1..MAX] de réel ;

somme, moy : réel ;

i : entier ;

Début

(\*Saisir les noms des candidats\*)

Ecrire('Donnez les noms des candidats :');

Pour i ← 1 à MAX Faire début

Ecrire('Saisir le nom du candidat n° : ', i, ' : ');

Lire(noms[i]);

fin ;

(\*Saisir les notes des candidats et calculer la somme des notes au même temps\*)

Ecrire('Donnez les notes des candidats :');

somme ← 0 ;

Pour i ← 1 à MAX Faire début

Ecrire('Saisir la note de ', noms[i], ' :');

Lire(notes[i]);

somme ← somme + notes[i];

fin ;

(\*Afficher la moyenne des notes des candidats\*)

moy ← somme / Max ;

Ecrire('La moyenne des notes des candidats est égale à ', moy);

(\*Afficher les candidats dont la note est >= à la moyenne des notes de tous les candidats\*)

Ecrire('Voici la liste des candidats dont la note >= moy :');

Pour i ← 1 à MAX Faire

Si (notes[i] >= moy) Alors

Ecrire(noms[i], ' avec une note de ', notes[i]);

Fin.

Le programme Pascal :

program Liste\_candidat\_sup ;

const

MAX = 10 ;

var

noms : array[1..MAX] of string ;

```

notes : array[1..MAX] of real ;
somme, moy : real;
i : integer ;
begin
  (*Saisir les noms des candidats*)
  writeln('Donnez les noms des candidats :');
  for i :=1 to MAX do begin
    writeln('Saisir le nom du candidat n° : ', i, ' : ');
    readln(noms[i]) ;
  end ;
  (*Saisir les notes des candidats et calculer la somme des notes au même temps*)
  writeln('Donnez les notes des candidats :') ;
  somme := 0;
  for i :=1 to MAX do begin
    writeln('Saisir la note de ', noms[i], ' :') ;
    readln(notes[i]) ;
    somme := somme + notes[i] ;
  end ;
  (*Afficher la moyenne des notes des candidats*)
  moy := somme / Max ;
  writeln('La moyenne des notes des candidats est égale à ', moy) ;
  (*Afficher les candidats dont la note est >= à la moyenne des notes de tous les candidats*)
  writeln('Voici la liste des candidats dont la note >= moy :') ;
  for i := 1 to MAX do
    if (notes[i] >= moy) then
      writeln(noms[i], ' avec une note de ', notes[i]) ;
  end.

```

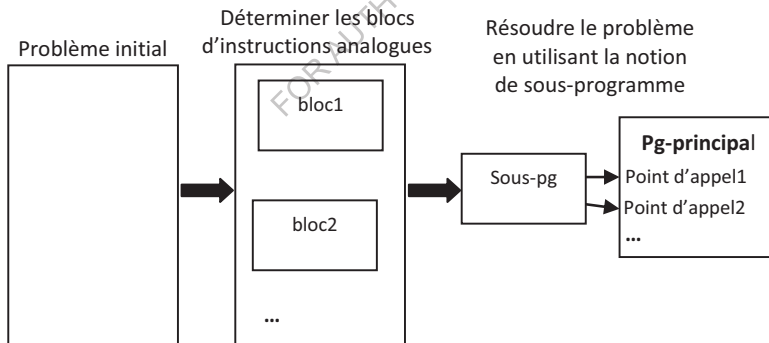


## Chapitre 6 : Les sous-programmes : procédures et fonctions

### 1. Introduction

Un des problèmes majeurs dans le développement des programmes réside dans la quantité de détail à considérer à tout moment. Plus cette quantité est grande, plus le risque d'erreur et de confusion est important. Pour améliorer et faciliter la conception de programmes fiables, lisibles et faciles à maintenir, il faut réduire et/ou organiser cette quantité de détail. Nous allons voir dans ce qui suit un moyen pour réduire et contrôler la complexité des grands programmes afin de les rendre simples et faciles à comprendre et à maintenir. Il s'agit des sous-programmes qui sont utilisés pour deux objectifs :

1. *Réduction de la taille des programmes* : il est possible de déterminer les blocs analogues, les substituer par un sous-programme, ensuite appeler le sous-programme par son nom, au lieu de reprendre l'écriture de toutes les instructions. Le terme "bloc" est souvent utilisé dans ce document pour désigner un ensemble d'instructions. Deux blocs analogues sont deux blocs contenant les mêmes instructions.



Soit le programme Pascal suivant :

```

program Sommes ;
var
  x, y, s : integer ;
begin
  writeln('*** bloc 1 ***') ;
  writeln('Donnez la première valeur :') ;
  readln(x) ;

```

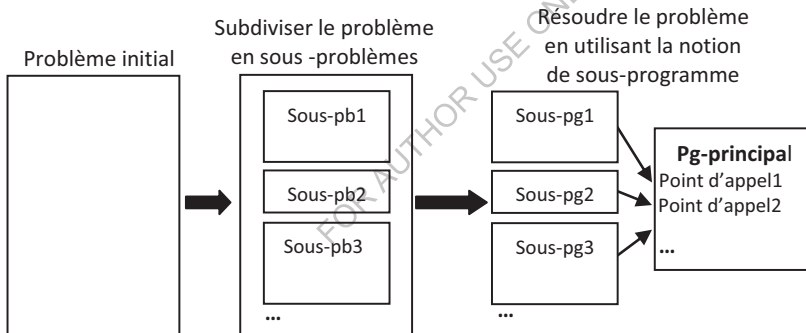
```
writeln('Donnez la deuxième valeur :') ;  
readln(y) ;  
s := x + y ;  
writeln('La somme = ', s) ;  
writeln('*** bloc 2 ***') ;  
writeln('Donnez la première valeur :') ;  
readln(x) ;  
writeln('Donnez la deuxième valeur :') ;  
readln(y) ;  
s := x + y ;  
writeln('La somme = ', s) ;  
end.
```

Le programme permet d'afficher la chaîne de caractères **\*\*\* bloc 1 \*\*\***, de calculer la somme de deux entiers lus à partir du clavier, ensuite d'afficher la chaîne de caractères **\*\*\* bloc 2 \*\*\***, et enfin de calculer la somme de deux entiers lus à partir du clavier une deuxième fois. Pour réduire le code de ce programme, il est possible d'utiliser un sous-programme permettant de calculer la somme de deux entiers lus à partir du clavier, et d'invoquer ce sous-programme deux fois dans le programme principal. On obtient alors le programme suivant :

```
program Sommes ;  
var  
  x, y, s : integer ;  
procedure somme;  
begin  
  writeln('Donnez la première valeur :') ;  
  readln(x) ;  
  writeln('Donnez la deuxième valeur :') ;  
  readln(y) ;  
  s := x + y ;  
  writeln('La somme = ', s) ;  
end;  
begin  
  writeln('*** bloc 1 ***') ;  
  somme;  
  writeln('*** bloc 2 ***') ;  
  somme;  
end.
```

Dans cette deuxième solution, on a regroupé les blocs analogues dans un seul sous-programme (procédure dans cet exemple). La procédure est invoquée au niveau du programme principal par son nom uniquement. Ceci a permis de réduire la taille du programme. Nous avons utilisé dans cet exemple la forme de sous-programme dite procédure, mais il existe une autre forme qui sera vue ultérieurement.

2. *Organisation du code* : pour cet objectif on applique une méthode descendante (top-down). On part du problème à résoudre, et on essaye de le découper en sous-problèmes plus simples à résoudre que le problème initial. Puis, on découpe les sous-problèmes jusqu'à ce qu'on obtient des problèmes très simples à résoudre par des sous-programmes de quelques instructions. Le programme final sera constitué d'un certain nombre de modules. Chaque module correspond à un sous-problème. C'est le principe de la programmation modulaire qui repose sur l'écriture des sous-programmes.



Soit le programme Pascal suivant :

```

program addition_multiplication_soustraction ;
var
  x, y, s, p, d : integer ;
begin
  writeln('Donnez la première valeur :') ;
  readln(x) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(y) ;
  s := x + y ;
  writeln('La somme = ', s) ;
  p := x * y ;
  writeln('Le produit = ', p) ;
end
  
```

```
d := x - y ;  
writeln('La différence = ', d) ;  
end.
```

Le programme permet de calculer la somme, le produit et la différence de deux nombres entiers lus à partir du clavier. Pour plus d'organisation, il est possible de subdiviser le problème en trois sous-problèmes (addition, multiplication, soustraction), et on obtient le programme suivant :

```
program addition_multiplication_soustraction ;  
var  
  x, y, s, p, d : integer ;  
procedure somme;  
begin  
  s := x + y ;  
  writeln('La somme = ', s) ;  
end;  
procedure produit;  
begin  
  p := x * y ;  
  writeln('Le produit = ', p) ;  
end;  
procedure difference;  
begin  
  d := x - y ;  
  writeln('La différence = ', d) ;  
end;  
begin  
  writeln('Donnez la première valeur :) ;  
  readln(x) ;  
  writeln('Donnez la deuxième valeur :) ;  
  readln(y) ;  
  somme;  
  produit;  
  difference;  
end.
```

Dans cette deuxième solution, le problème était subdivisé en sous-problèmes, chacun est résolu au niveau d'un sous-programme. Les sous-programmes (procédures dans ce cas-là) sont appelés dans le programme principal par leurs noms dans des points d'appel.

## 2. Les sous-programmes

Un programme peut contenir dans sa partie déclaration des étiquettes, des constantes, des types, des variables, et enfin des sous-programmes. Un sous-programme est invoqué (appelé) dans la partie instructions d'un programme principal au niveau des points d'appel.

Un sous-programme n'est en fin de compte qu'un petit programme qui peut à son tour contenir une partie déclaration et une partie instructions. Le programme Sommes peut être écrit comme suit :

```
program Sommes ;
var
  s : integer ;
procedure somme;
var
  x, y : integer ;
begin
  writeln('Donnez la première valeur :') ;
  readln(x) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(y) ;
  s := x + y ;
  writeln('La somme = ', s) ;
end;
begin
  writeln('*** bloc 1 ***') ;
  somme;
  writeln('*** bloc 2 ***') ;
  somme;
end.
```

Un sous-programme peut faire appel à d'autres sous-programmes et dans ce cas, on appelle *sous-programme appelant* le sous-programme qui contient dans sa partie déclaration un *sous-programme appelé*, et dans sa partie instructions les points d'appel. Le programme addition\_multiplication\_soustraction peut être écrit comme suit :

```
program addition_multiplication_soustraction ;
var
  x, y : integer ;
procedure addition_multiplication;
var
  s : integer;
```

```
procedure produit;
var
  p : integer;
begin
  p := x * y ;
  writeln('Le produit = ', p) ;
end;
begin
  s := x + y ;
  writeln('La somme = ', s) ;
  produit;
end;
procedure difference;
var
  d : integer;
begin
  d := x - y ;
  writeln('La différence = ', d) ;
end;
begin
  writeln('Donnez la première valeur :') ;
  readln(x) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(y) ;
  addition_multiplication;
  difference;
end.
```

Lorsque le processeur rencontre l'appel d'une procédure, il arrête momentanément l'exécution du programme appelant pour aller exécuter les instructions de la procédure. Quand il termine l'exécution de la procédure, le processeur reprend l'exécution du programme appelant là où il s'est arrêté.

Dans le cas où on a des blocs d'instructions qui se ressemblent en termes d'opérations, mais ils utilisent des variables différentes, on peut utiliser un sous-programme avec des paramètres pour réduire le code du programme.

Soit le programme Pascal suivant :

```
program Sommes ;
var
```

```
x, y, z, h, s : integer ;
begin
  writeln('*** bloc 1 ***') ;
  writeln('Donnez la première valeur :') ;
  readln(x) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(y) ;
  s := x + y ;
  writeln('La somme = ', s) ;
  writeln('*** bloc 2 ***') ;
  writeln('Donnez la première valeur :') ;
  readln(z) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(h) ;
  s := z + h ;
  writeln('La somme = ', s) ;
end.
```

Le programme permet d'afficher la chaîne de caractères `*** bloc 1 ***`, de calculer la somme de deux entiers lus à partir du clavier, ensuite d'afficher la chaîne de caractères `*** bloc 2 ***`, et enfin de calculer la somme de deux entiers lus à partir du clavier une deuxième fois, mais cette fois-ci avec des variables différentes. Pour réduire le code de ce programme, il est possible d'utiliser un sous-programme avec deux paramètres, permettant de calculer la somme de deux entiers lus à partir du clavier, et d'invoquer ce sous-programme deux fois dans le programme principal. On obtient alors le programme suivant :

```
program Sommes ;
var
  x, y, z, h, s : integer ;
procedure somme (a, b : integer);
begin
  writeln('Donnez la première valeur :') ;
  readln(a) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(b) ;
  s := a + b ;
  writeln('La somme = ', s) ;
end;
begin
  writeln('*** bloc 1 ***') ;
  somme(x, y);
```

```
writeln('*** bloc 2 ***') ;  
somme(z, h);  
end.
```

Dans ce programme, nous avons regroupé les blocs analogues dans un seul sous-programme (procédure) avec deux paramètres. La procédure est invoquée au niveau du programme principal, la première fois avec les paramètres x, y, et la deuxième fois avec les paramètres z, h. Les paramètres x, y et z, h sont dits effectifs (ou réels), utilisés lors de l'invocation du sous-programme dans le programme principal au niveau des points d'appel. Par contre, les paramètres a, b sont dits formels, utilisés lors de la déclaration du sous-programme.

### Remarques :

- En Pascal, les paramètres formels sont séparés par des points-virgules. Toutefois, si plusieurs paramètres sont du même type, on peut les placer ensemble, séparés par des virgules et suivis de leurs types. Par contre, les paramètres effectifs sont tous séparés par des virgules.
- Lorsqu'il y a plusieurs paramètres dans la définition d'une procédure, il faut absolument qu'il y en ait le même nombre à l'appel, et que l'ordre soit respecté.
- Le type d'une valeur ou variable utilisée comme paramètre effectif doit être le même que le type du paramètre formel correspondant.

Il existe deux types de sous-programme : les procédures et les fonctions.

### 2.1. Les procédures

Une procédure est un ensemble d'instructions regroupées sous un nom, et qui réalise un traitement particulier dans un programme lorsqu'on l'appelle. Pour déclencher l'exécution d'une procédure dans un programme, il suffit de l'appeler par son nom au moyen d'une instruction dite instruction procédure (point d'appel). Les exemples qu'on a vus jusqu'à maintenant utilisent des procédures.

En Pascal, une procédure est déclarée comme suit:

1. Entête : contient le mot clé `procedure`, suivi du nom de la procédure et éventuellement des paramètres mises entre parenthèses.
2. La partie déclaration : peut contenir des étiquettes, des constantes, des types, des variables et des sous-programmes.
3. La partie instructions : contient les instructions de la procédure prises entre `begin` et `end`.

### Exercice :

Qu'affiche le programme Pascal suivant ?

```
program affichage(output) ;  
var
```



```
I : integer ;
procedure afficher;
var
  J : integer;
begin
  J:=100;
  writeln('ici la procédure : afficher J = ',j) ;
end ;
begin
  I := 10 ;
  writeln('ici le programme : affichage I = ', I);
  afficher ;
  I := 20 ;
  writeln('ici le programme : affichage I = ', I) ;
end.
```

**Solution :**

Le résultat d'exécution de ce programme est le suivant :

ici le programme : affichage I = 10

ici la procédure : afficher J = 100

ici le programme : affichage I = 20

**2.2. Les fonctions**

Les fonctions sont des sous-programmes qui retournent un et un seul résultat au programme appelant. De ce fait, les fonctions sont appelées pour récupérer une valeur, alors que les procédures ne renvoient aucune valeur au programme appelant. Le point d'appel d'une fonction apparaît toujours dans une expression.

La valeur retournée par la fonction porte le nom de la fonction elle-même, donc la fonction doit contenir dans sa partie instructions, au moins une fois, son nom dans la partie gauche d'une instruction d'affectation pour retourner un résultat au programme ou au sous-programme appelant.

En Pascal, une fonction est déclarée comme suit :

1. Entête : contient le mot clé `function`, suivi du nom de la fonction et éventuellement des paramètres mises entre parenthèses, et enfin le type de la fonction.
2. La partie déclaration : peut contenir des étiquettes, des constantes, des types, des variables et des sous-programmes.
3. La partie instructions : contient les instructions de la fonction prises entre `begin` et `end`.

Si on reprend l'exemple Sommes, mais cette fois-ci en utilisant une fonction, on obtient :

```
program Sommes ;
var
  x, y, s : integer ;
function somme : integer;
begin
  writeln('Donnez la première valeur :') ;
  readln(x) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(y) ;
  somme := x + y ;
end;
begin
  writeln('*** bloc 1 ***') ;
  s := somme ;
  writeln('La somme = ', s) ;
  writeln('*** bloc 2 ***') ;
  s := somme ;
  writeln('La somme = ', s) ;
end.
```

Dans ce programme, nous avons regroupé les blocs analogues dans un seul sous-programme (fonction). La fonction est invoquée deux fois dans le programme principal. A chaque fois, la fonction retourne une valeur affectée à la variable s qui est ensuite affichée à l'écran.

Voyons un deuxième exemple :

```
program Comparaison ;
var
  x, y : integer ;
function Sup (a, b : integer) : boolean;
begin
  if (a > b) then Sup := true
  else Sup := false;
end;
begin
  writeln('Donnez la première valeur :') ;
  readln(x) ;
  writeln('Donnez la deuxième valeur :') ;
  readln(y) ;
```

```
if Sup(x,y) then writeln(x, ' est supérieure à ', y)
else writeln(y, ' est supérieure à ', x);
end.
```

Ce programme permet de comparer deux nombres entiers en indiquant quelle est la valeur supérieure, et cela en utilisant une fonction qui reçoit la valeur booléenne true si son premier paramètre est supérieur au deuxième, et la valeur false si l'inverse.

**Exercice :**

Qu'affiche le programme Pascal suivant ?

```
program affichage(output);
var
  I : integer;
function constante : integer;
begin
  constante :=4;
end;
procedure afficher;
var
  J : integer;
begin
  J:=100;
  writeln('constante * 2 = ',constante * 2);
end;
begin
  I := constante * 2;
  writeln('I = ', I);
  afficher;
  if ((constante * 3) < 15) then writeln('constante * 3 = ', constante * 3);
end.
```

**Solution :**

Le résultat d'exécution de ce programme est le suivant :

```
I = 8
constante * 2 = 8
constante * 3 = 12
```

En Pascal, il existe plusieurs fonctions prédéfinies ou standards permettant de faire des calculs mathématiques. Parmi ces fonctions on cite ABS(), SQRT(), SQR(), EXP(), etc. Elles peuvent être utilisées directement dans n'importe quel programme.

**Exercice :**

Essayez d'implémenter votre propre fonction ABS(). Nommez la Absolue. La fonction Absolue doit retourner la valeur absolue d'un paramètre entier.

**Solution :**

```
program Val_ABS ;
var
  x : integer;
function Absolue(a : integer) : integer;
begin
  if (a >= 0) then Absolue := a
  else Absolue := -a ;
end;
begin
  writeln('Donnez une valeur :');
  readln(x);
  writeln('La valeur absolue de ', x, ' est : ', Absolue(x));
end.
```

**3. Les variables locales et les variables globales**

Un sous-programme peut dans sa partie déclaration définir des objets (étiquettes, constantes, types, variables et sous-programmes) qui sont propres à lui. Ces objets sont dits objets locaux, et souvent on utilise la notion de *variables locales*. Cela permet d'isoler dans un sous-programme, non seulement les instructions relatives à une action déterminée, mais aussi des objets qui sont spécifiques à ce sous-programme et ne peuvent être utilisés qu'au niveau de la partie instructions de ce sous-programme.

Les objets déclarés au niveau du programme appelant sont dits objets globaux, et on utilise la notion de *variables globales*. Ces objets peuvent être utilisés dans n'importe endroit dans ce programme, y compris les parties instructions de ses sous-programmes.

Les variables locales, ainsi que les paramètres formels, n'appartiennent qu'à la procédure. La procédure a son propre espace mémoire pour le travail.

**Exercice :**

Soit le problème suivant : Remplacer le maximum d'un tableau de cinquante réels par la moyenne de ses nombres positifs.

Subdivisez ce problème en sous-problèmes simples. Ensuite, essayez de les résoudre par la notion de sous-programme en langage Pascal.

Ce problème peut être subdivisé en trois sous-problèmes :

1. Déterminer le max d'un tableau, ainsi que sa position.
2. Déterminer la moyenne des nombres positifs d'un tableau.
3. Remplacer le maximum par la moyenne.

**Solution :**

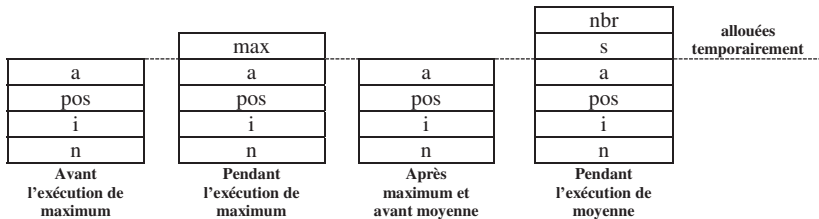
Le problème peut être résolu comme suit :

```
program Rem_Tab (input, output) ;
const  n=50;
var    (*Variables globales*)
  i, pos : integer ;
  A : array [1..n] of real;
  procedure Maximum;
  var    (*Variables locales de la procédure maximum*)
    max : real;
  begin
    max := A[1]; pos := 1;
    for i := 2 to n do if (max < A[i]) then begin
      max := A[i];
      pos := i;
    end;
    writeln('Le maximum du tableau est = ', max, ' sa position est : ', pos) ;
  end;
  function Moyenne : real ;
  var    (*Variables locales de la fonction moyenne*)
    s : real ;
    nbr : integer;
  begin
    nbr:=0;
    s:=0;
    for i := 1 to n do  if (A[i]>0) then begin
      s:=s+A[i];
      nbr := nbr +1;
    end;
    if (nbr<>0) then Moyenne := s/nbr
    else Moyenne := -1;
  end;
begin (*Programme principal*)
  writeln('Entrez les 50 valeurs du tableau :');
  for i := 1 to n do readln(A[i]);
  Maximum;
```

```

A[pos] := Moyenne;
for i := 1 to n do writeln(A[i]);
end.

```



Après l'exécution de la procédure maximum, la variable max déclarée comme variable locale n'existe plus. Si cette procédure est appelée une seconde fois, alors ses variables locales sont à nouveau allouées.

Il est incorrect d'utiliser les variables nbr, max et s dans le programme principal, parce que ces variables ne sont pas définies à ce niveau.

La partie du programme dans laquelle une variable peut être utilisée est appelée le champ de visibilité ou encore la portée de cette variable. Ce champ commence à partir de la déclaration de cette variable jusqu'à la fin du bloc d'instructions concerné par cette déclaration.

### Exemple :

```

program IMBR(input, output);
var
  A, B : real;
procedure EXT;
var
  C, D : real;
begin
  ... (* Les variables accessibles sont A, B, C, D*)
end ;
procedure ITN;
var
  E, F : real;
begin
  ... (* Les variables accessibles sont A, B, E, F*)
end ;
begin
  ... (* Les variables accessibles sont A, B*)
end.

```

```
program IMBR(input, output) ;
var
  A, B : real;
procedure EXT;
var
  C, D : real;
procedure ITN;
var
  E, F : real;
begin
... (* Les variables accessibles sont A, B, C, D, E, F*)
end ;
begin
... (* Les variables accessibles sont A, B, C, D*)
end ;
begin
... (* Les variables accessibles sont A, B*)
end.
```

Les procédures et les fonctions peuvent modifier des variables non locales. Dans ce cas là, on dit qu'elles ont un effet de bord ou un effet secondaire. Souvent, les effets secondaires sont nocifs ou non désirable. En tout cas, les variables non locales obscurcissent le programme et sa structure, et rendent ainsi le programme difficile à comprendre. Pour cela, on ne déclare comme globales que les variables qui doivent absolument l'être. Et chaque fois que possible, lorsqu'on crée un sous-programme, on utilise le passage de paramètres plutôt que des variables globales.

Pour enlever l'ambiguïté, il est préférable d'éviter d'utiliser le même identificateur de variable dans différents niveaux. Evitez aussi d'utiliser des identificateurs peu significatifs, par exemple au lieu de x, y, etc. utilisez puissance, nombre, etc.

### **Exercice :**

Qu'affiche le programme Pascal suivant ?

```
program Saisie ;
var i : integer;
procedure S;
var i : integer;
begin
  i := 5;
```

```
writeln('i = ', i);  
end;  
begin  
  i := 10;  
  S;  
  writeln('i = ', i);  
end.
```

**Solution :**

Le programme affiche :

i = 5

i = 10

Il est possible de substituer une fonction par une procédure en utilisant une variable globale permettant de récupérer la valeur retournée par la fonction. La variable globale doit avoir le même type que la fonction remplacée. Si on remplace la fonction Moyenne dans le programme Rem\_Tab par une procédure, on ajoute une variable globale moy de type réel. On obtient alors :

```
program Rem_Tab(input, output) ;  
const  
  n=50;  
var (*Variables globales*)  
  i, pos : integer ;  
  moy : real ;  
  A : array [1..n] of real;  
  procedure Maximum;  
  var (*Variables locales de la procédure maximum*)  
    max : real;  
  begin  
    max := A[1]; pos := 1;  
    for i := 2 to n do if (max < A[i]) then begin  
      max := A[i];  
      pos := i;  
    end;  
    writeln('Le maximum du tableau est = ', max, ' sa position est : ', pos) ;  
  end;  
  procedure Moyenne ;  
  var (*Variables locales de la fonction moyenne*)  
    s : real ;  
    nbr : integer;
```



```
begin
  nbr:=0;
  s:=0;
  for i := 1 to n do if A[i]>0 then begin
    s:=s+A[i];
    nbr := nbr +1;
  end;
  if (nbr <> 0) then  moy := s/nbr
  else moy := -1;
end;
```

```
begin (*Programme principal*)
  writeln('Entrez les 50 valeurs du tableau :');
  for i := 1 to n do readln(A[i]);
  Maximum;
  Moyenne ;
  A[pos] := moy;
  for i := 1 to n do writeln(A[i]);
end.
```

Tous les changements dans le programme précédent ont été illustrés par une couleur de surbrillance différente.

#### 4. Le passage des paramètres

Pour améliorer le programme précédent, il est possible de déclarer la procédure Maximum comme suit : `procedure Maximum (A : array of real ; var pos : integer);`. A et pos sont les paramètres (ou arguments) formels de la procédure Maximum. Le mot clé `var` est utilisé pour transmettre la valeur de pos vers le programme principal.

Le paramètre avec le mot clé `var` est appelé paramètre à *passage par variable* (*par adresse* ou *par référence*), et celui qui n'a pas de `var` au début est dit paramètre à *passage par valeur*. Si le tableau est de grande taille, il faudra mieux le déclarer en tant que paramètre à passage par variable : `(var A : array of real ; var pos : integer)`. Les paramètres à passage par valeur sont aussi dits *paramètres en entrée*. Les paramètres à passage par variable sont aussi dits *paramètres en sortie*.

Voyons l'exemple suivant :

```
program variable1 (output) ;
var
  H : integer ;
```

```
procedure change1;
begin
  H := 1;
end;
begin
  H := 0;
  change1 ;
  writeln(H) ;
end.
```

Ce programme n'a qu'une seule variable H, et c'est une variable globale. H est initialisée à 0 par le programme principal. Ce programme appelle la procédure change1 qui change effectivement la variable H à 1, ensuite elle l'affiche, c.-à-d. elle affiche 1.

Voyons maintenant un autre exemple :

```
program variable2 (output) ;
var
  H : integer ;
procedure change2;
var
  H : integer;
begin
  H := 1;
end;
begin
  H := 0;
  change2 ;
  writeln(H) ;
end.
```

Dans ce programme, il y a deux variables. Les deux sont appelées H. La première est une variable globale, et la deuxième est une variable locale de la procédure change2. La déclaration d'une variable locale H de même nom que la variable globale H empêche la procédure d'accéder à la variable globale H. L'affectation `H := 1` ; n'a aucun effet sur la variable globale H, et par conséquent, la valeur affichée par le programme est 0.

Dans un troisième exemple, on a :

```
program variable3 (output) ;
var
  H : integer ;
procedure change3(var Y : integer);
```

```
begin
  Y := 1;
end;
begin
  H := 0;
  change3(H) ;
  writeln(H) ;
end.
```

Ce programme a une variable globale H et une procédure change3 déclarée avec un paramètre formel Y à passage par adresse. La valeur de la variable H sera changée de 0 (valeur affectée au niveau du programme principal) à 1 (valeur affectée au niveau de la procédure change3) et la valeur affichée sera 1.

La déclaration de Y dans l'entête de la procédure est précédée par var. Cette déclaration définit Y comme un paramètre formel variable. Au niveau de la mémoire, on ne donne pas un espace mémoire pour Y ; on utilise uniquement l'espace réservé à H.

Voyons maintenant un autre exemple :

```
program variable4 (output) ;
var
  H : integer ;
procedure change4(Y : integer);
begin
  Y := 1;
end;
begin
  H := 0;
  change4(H) ;
  writeln(H) ;
end.
```

Ce programme a une variable globale H et une procédure change4 déclarée avec un paramètre formel Y à passage par valeur. La procédure change4 exécute l'instruction Y := 1, ce qui se traduit par H := 1 lors de l'invocation de la procédure change4 avec le paramètre effectif H. Mais cela n'a aucun effet sur la valeur de la variable H qui a reçu 0 au niveau du programme principal, et c'est cette valeur qui sera affichée en sortie, c.-à-d. 0.

### Exercice :

1. Qu'affiche le programme Pascal suivant ?

```

program Somme ;
var
  x, y, Som : integer;
procedure S(a, b : integer);
begin
  Som := a + b ;
  a := a + b;
  b := 2 ;
end;
begin
  x := 2 ;
  y := 9 ;
  S(x,y);
  writeln(x, ' + ', y, ' = ', Som);
end.

```

2. Qu'affiche programme précédent, mais cette fois-ci, si on remplace  $S(a, b : \text{integer})$  par  $S(a : \text{integer}; \text{var } b : \text{integer})$ , par  $S(\text{var } a : \text{integer}; b : \text{integer})$ , et enfin par  $S(\text{var } a, b : \text{integer})$  ?

**Solution :**

1. Le programme affiche :  $2 + 9 = 11$ .
2. Le programme affiche :
  - Pour  $S(a : \text{integer}; \text{var } b : \text{integer})$  :  $2 + 2 = 11$ .
  - Pour  $S(\text{var } a : \text{integer}; b : \text{integer})$  :  $11 + 9 = 11$ .
  - Pour  $S(\text{var } a, b : \text{integer})$  :  $11 + 2 = 11$ .

**Remarques :**

- Dans la plus part des cas, le paramètre effectif correspondant à un paramètre formel à passage par valeur ne peut être qu'une entité qui peut être placée à droite du signe d'affectation. Par contre, le paramètre effectif correspondant à un paramètre formel à passage par variable ne peut être qu'une entité qui peut être placée à gauche du signe d'affectation.
- Quand on utilise le passage par valeur, il est possible d'invoquer le sous-programme par des paramètres effectifs exprimés sous forme d'expressions arithmétiques, par exemple  $\text{change4}(H*2)$ ,  $\text{change4}(2)$ . Ce n'est pas le cas quand on utilise le passage par adresse, alors  $\text{change3}(H*2)$  et  $\text{change3}(2)$  ne sont pas acceptées.
- Il est également possible d'utiliser le résultat d'une fonction directement comme paramètre effectif d'une autre fonction (si le paramètre formel correspondant de la fonction appelante est à passage par valeur).

### **Règles :**

- Si le but de la procédure est de changer le paramètre, on utilise alors un paramètre à passage par variable.
- S'il est raisonnable de passer une expression à la procédure, alors n'employez pas le passage par variable.
- Dans le doute, il est préférable en général d'utiliser var.

### **5. Construction d'un algorithme complexe**

Pour faciliter l'implémentation d'un algorithme complexe, il est préférable de suivre les étapes suivantes :

1. Construction du dictionnaire de données : le but de cette étape est d'identifier les informations qui seront nécessaires au traitement du problème, et de choisir le type de codage qui sera le plus satisfaisant pour traiter ces informations. Donc, avant même d'écrire quoi que ce soit, les questions qu'il faut se poser sont les suivantes : de quelles informations le programme va-t-il avoir besoin pour venir à bout de sa tâche ? Pour chacune de ces informations, quel est le meilleur codage ? Autrement dit, celui qui sans gaspiller de la place mémoire, permettra d'écrire l'algorithme le plus simple ?
2. Construction de l'algorithme fonctionnel : c'est le découpage en blocs et/ou la représentation graphique de notre problème, ayant comme objectif de faire comprendre quelle procédure fait quoi, et quelle procédure appelle quelle autre. L'algorithme fonctionnel est donc en quelque sorte la construction du squelette de l'application. Il se situe à un niveau plus général, plus abstrait, que l'algorithme normal, qui lui, détaille pas à pas les traitements effectués au sein de chaque procédure.
3. Construction de l'algorithme détaillé : normalement, il ne nous reste plus qu'à traiter chaque procédure isolément. On commencera par les procédures et fonctions, pour terminer par la rédaction du programme principal.

### **6. La récursivité (récursion)**

#### **6.1. Définition**

On a dit précédemment qu'un sous-programme peut être appelé un autre sous-programme. Une procédure ou fonction est dite récursive si elle fait appel à elle-même.

#### **6.2. Exemples**

##### **6.2.1. La factorielle**

L'exemple suivant est utilisé pour calculer la factorielle d'un nombre. La factorielle d'un nombre  $n$  se note  $n!$ . On rappelle que la factorielle

d'un nombre  $n$  est égale au produit de tous les nombres de 1 jusqu'à  $n$ , c.-à-d.  $n! = 1*2*\dots*(n-1)*n$ , ainsi,  $5! = 1*2*3*4*5$ . On a aussi  $0! = 1$ .

La fonction itérative (en utilisant une boucle) pour calculer la factorielle est donnée ci-dessous :

```
function fact (n : integer) : integer ;
```

```
var
```

```
  i, j : integer;
```

```
begin
```

```
  j := 1;
```

```
  for i :=1 to n do j := j*i ;
```

```
  fact := j;
```

```
end ;
```

La définition mathématique de la factorielle en tant que formule récurrente est la suivante : pour tout  $n$  entier, si  $n > 0$   $fact(n) = n*fact(n-1)$ , en plus  $fact(0) = 1$ . L'implémentation directe de cette fonction telle qu'elle est dans sa nouvelle définition nous donne :

```
function fact (n : integer) : integer ;
```

```
begin
```

```
  if (n=0) then fact := 1
```

```
    else fact := n*fact(n-1) ;
```

```
end ;
```

La fonction fact est une fonction récursive pour le calcul de la factorielle. Apparemment, elle est simple car elle n'utilise ni variable local ni itération. En fait, l'appel récursif cache des itérations, et il existe simultanément  $n+1$  appel (activation) de la fonction lors du calcul de la factorielle. L'espace occupé en mémoire est trop élevé, puisque la pile d'activation doit contenir  $n+1$  fois le contexte de la fonction.

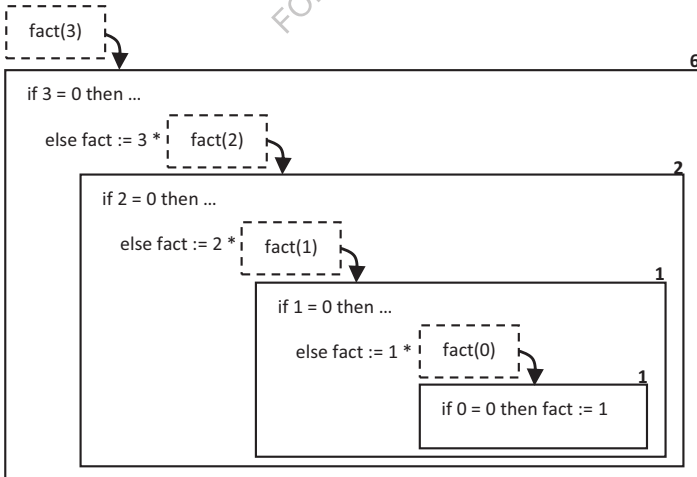
La pile est une zone mémoire réservée à chaque programme; sa taille peut être fixée manuellement par l'utilisateur. Son rôle est de stocker les variables locales et les paramètres d'une procédure. Supposons que nous sommes dans une procédure proc1 dans laquelle nous avons des variables locales, ensuite, nous faisons appel à une procédure proc2; comme le microprocesseur va commencer à exécuter proc2, mais qu'ensuite il reviendra continuer l'exécution de proc1, il faut bien stocker quelque part les variables de la procédure proc1; c'est le rôle de la pile. Tout ceci est géré de façon transparente pour l'utilisateur. Dans une procédure récursive, toutes les variables locales sont stockées dans la pile, et empilées autant de fois qu'il y a d'appels récursifs. Donc la pile se remplit progressivement, et si on ne fait pas attention, on arrive à un *débordement de pile*. Ensuite, les variables sont dépilées.

Une fonction récursive doit vérifier les deux conditions suivantes :

1. Il doit exister des critères pour lesquels les appels cessent. Le critère d'arrêt dans la fonction récursive `fact` est ( $n=0$ ). Dans ce cas, on exécute un bloc dit *point terminal* ou *point d'appui* ou encore *point d'arrêt*, qui indique que le reste des instructions ne doit plus être exécuté. Dans la fonction `fact`, il s'agit du bloc contenant l'instruction `fact := 1` ;
2. Chaque fois que la procédure ou la fonction fait appel à elle-même (directement ou indirectement), elle doit être proche de ces critères d'arrêt, c.-à-d. que les paramètres de l'appel récursif doivent changer, en devenant de plus en plus simples, et en convergeant vers le critère d'arrêt. En effet, à chaque appel, l'ordinateur stocke dans la pile les variables locales; le fait de ne rien changer dans les paramètres ferait que l'ordinateur effectuerait un appel infini à cette procédure, ce qui se traduirait en réalité par un débordement de pile, et d'arrêt d'exécution de la procédure en cours. Grâce à ces changements, tôt ou tard l'ordinateur rencontrera un ensemble de paramètres vérifiant le test d'arrêt, et donc à ce moment, la procédure récursive aura atteint le *fond* (point terminal). Ensuite, les paramètres ainsi que les variables locales sont dépilés au fur et à mesure qu'on remonte les niveaux.

Le mécanisme interne et le stockage des paramètres et des variables locales pour la fonction récursive `fact` peut être représenté comme suit :

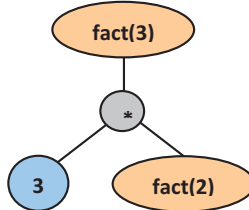
Pour  $n = 3$  :



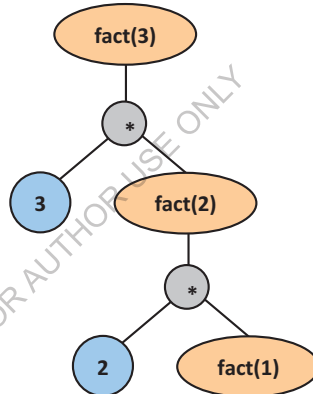
Les boîtes illustre les quatre exécutions. Les nombres à droite représentent les résultats commutatifs où chaque résultat doit être passé à un niveau plus haut.

Pour mieux comprendre, utilisant une deuxième représentation pour exécuter la fonction récursive pour le calcul de la factorielle. Alors, lors de l'appel de  $\text{fact}(3)$ , on aura la suite des appels récursifs suivants :

- $\text{fact}(3) = 3 * \text{fact}(2)$ . Le calcul de  $\text{fact}(3)$  fait appel à  $\text{fact}(2)$ .

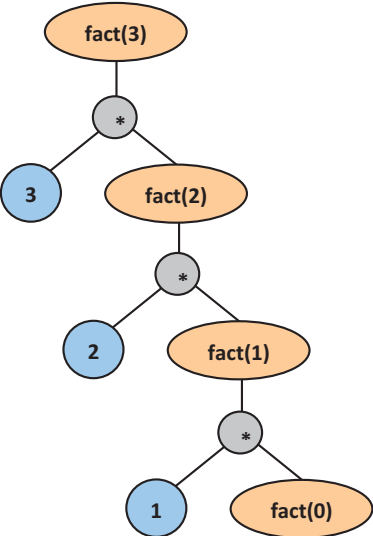


- $\text{fact}(2) = 2 * \text{fact}(1)$ . Le calcul de  $\text{fact}(2)$  fait appel à  $\text{fact}(1)$ .

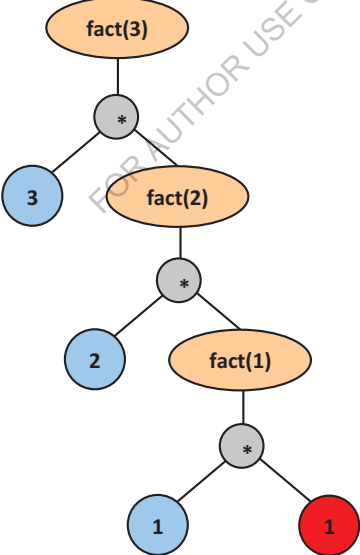


- $\text{fact}(1) = 1 * \text{fact}(0)$ . Le calcul de  $\text{fact}(1)$  fait appel à  $\text{fact}(0)$ .

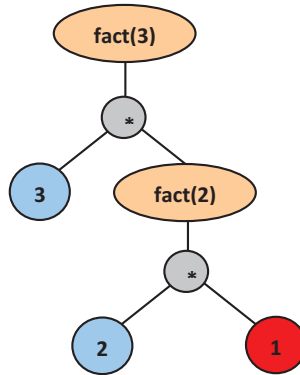




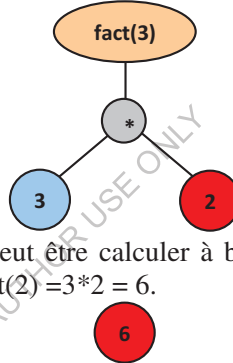
- A ce niveau, on a atteint le point terminal, et le calcul peut commencer par `fact(0) = 1`.



- `fact(1)` peut être maintenant calculer à base du résultat de `fact(0)`.  
Donc `fact(1) = 1*fact(0) = 1*1 = 1`.



- fact(2) peut être maintenant calculer à base du résultat de fact(1).  
Donc  $\text{fact}(2) = 2 * \text{fact}(1) = 2 * 1 = 2$ .



- Finalement fact(3) peut être calculer à base du résultat de fact(2).  
Donc  $\text{fact}(3) = 3 * \text{fact}(2) = 3 * 2 = 6$ .

### 6.2.2. Le PGCD

Comme vu précédemment, le PGCD (le plus grand diviseur commun) de deux nombres entiers  $m$  et  $n$  positifs ou nuls, peut être déterminé en utilisant l'algorithme itératif d'Euclide qui prend d'abord le reste de la division de  $m$  par  $n$ , puis le reste de la division de  $n$  par ce premier reste, etc., jusqu'à ce qu'on trouve un reste nul. Le dernier diviseur utilisé est le PGCD de  $m$  et  $n$ .

```

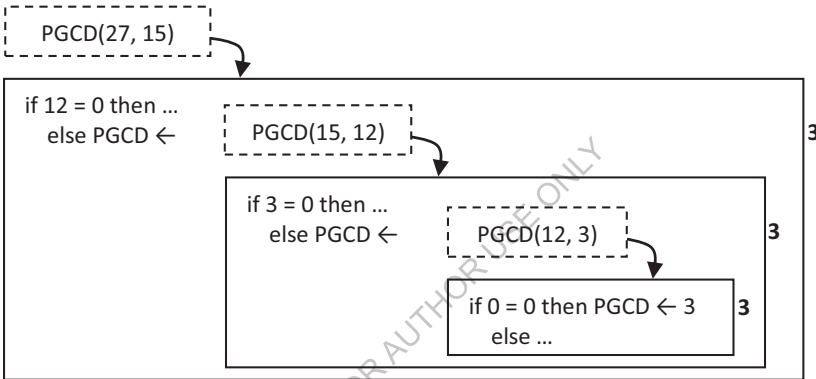
function PGCD(m, n : integer) : integer;
var
  r : integer ;
begin
  while NOT (m mod n = 0) do begin
    r := m mod n ;
    m := n ;
    n := r ;
  end;
end;
    
```

```
PGCD := n ;  
end ;
```

La version récursive de cette fonction peut être représentée de la manière suivante :

```
function PGCD(m, n : integer) : integer;  
begin  
  if m mod n = 0 then PGCD := n  
  else PGCD := PGCD(n, m mod n);  
end ;
```

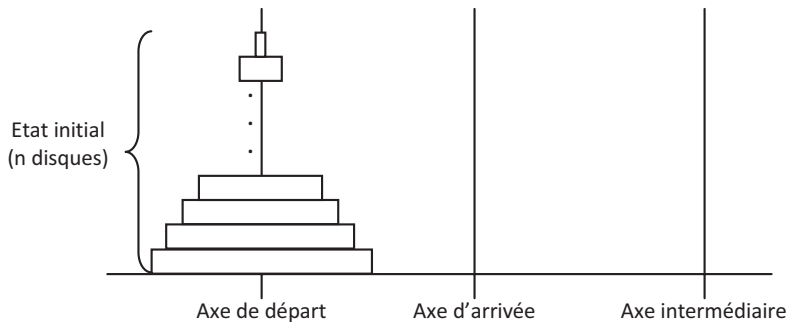
Pour  $m = 27$ ,  $n = 15$ , la représentation du mécanisme interne et du stockage des paramètres et des variables locales pour la fonction récursive PGCD peut être représenté comme suit.



### 6.2.3. Tour de Hanoï

Hanoï est une ville vietnamienne. Ses citoyens jouaient un jeu qui consiste à déplacer un ensemble de disques superposés d'un axe vers un autre. Les règles du jeu sont les suivantes :

- Les disques sont initialement superposés sur l'axe de départ, et ordonnés du plus grand en bas vers le plus petit en haut.
- Ces disques doivent être en fin du jeu déplacer vers un axe d'arrivée dans le même ordre que celui de l'état initial.
- On a le droit de déplacer un seul disque à la fois.
- On peut utiliser un axe intermédiaire.
- A un moment donné, on ne peut trouver un grand disque sur un autre plus petit.



La solution de ce problème n'est pas évidente. Il s'agit d'une œuvre des années de travail. Le principe de la solution de ce problème est de :

- Déplacer  $n-1$  disques de l'axe de départ vers l'axe intermédiaire.
- Bouger un disque de l'axe de départ vers l'axe d'arrivée.
- Déplacer les  $n-1$  disques de l'axe intermédiaire vers l'axe d'arrivée.

Ce qui peut être traduit par la fonction récursive suivante:

Déplacer (n, dep, arr, inter) ;

Début

    Déplacer (n-1, dep, inter, arr) ;

    Bouger (dep, arr) ;

    Déplacer(n-1, inter, arr, dep) ;

Fin ;

Le programme entier pour résoudre ce problème est le suivant :

```
program tourhanoi(input, output) ;
```

```
var
```

```
  nombre : integer ;
```

```
procedure bougerdisque(dep, arr : integer) ;
```

```
begin
```

```
  writeln(dep, ' -> ', arr); (*ce bloc peut être remplacé par un autre
  permettant de visualiser le déplacement des disques*)
```

```
end;
```

```
procedure deplacer(hauteur, dep, arr, inter : integer) ;
```

```
begin
```

```
  if (hauteur > 0) then begin
```

```
    deplacer(hauteur-1, dep, inter, arr) ;
```

```
    bougerdisque(dep, arr) ;
```

```
    deplacer(hauteur-1, inter, arr, dep) ;
```

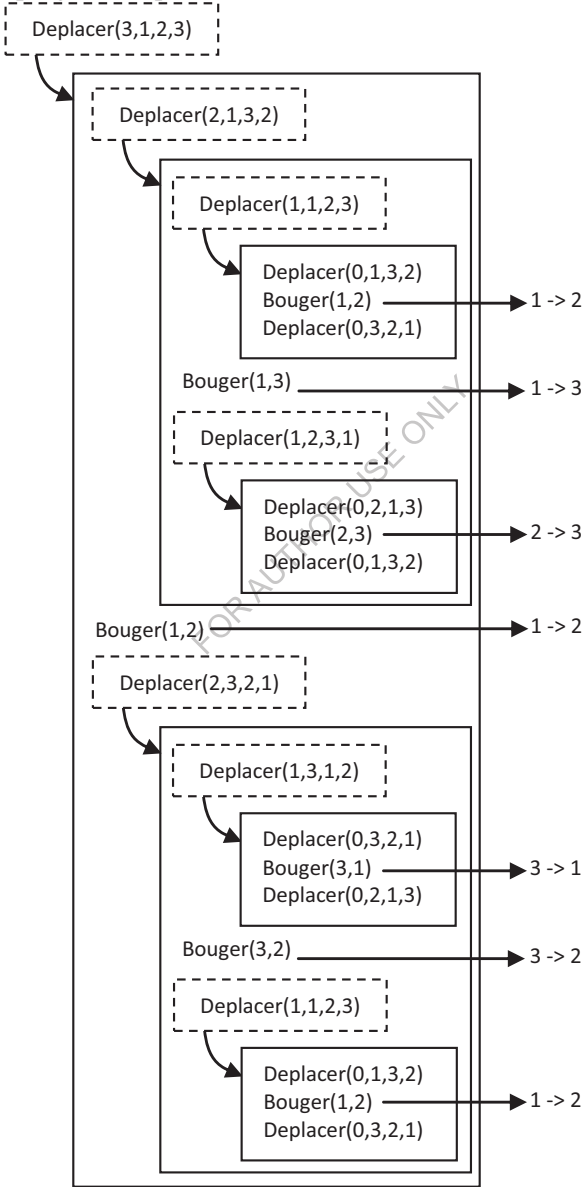
```
  end ;
```

```
end ;
```

```
begin
```

```
writeln('donnez le nombre des disques :)';  
readln(nombre);  
deplacer(nombre, 1, 2, 3);  
end.
```

Déroulant le programme pour nombre = 3.



**Remarques :**

1. L'exécution du programme pour  $n = 64$  mène à deux exécutions  $n = 63$ . Le nombre d'exécutions de déplacement =  $2^{64} - 1$ . Si chaque déplacement d'un disque est exécuté pendant une seconde, le temps total pour exécuter cette tâche est de  $6 * 10^{11}$  années.
2. La définition récursive permet de définir un nombre infini d'objets avec un nombre fini de règles. Par exemple : 1 est un nombre naturel, et le successeur d'un nombre naturel est un nombre naturel. De même, un programme récursif permet de définir un nombre infini de processus avec un nombre fini d'instructions, sans même utiliser les instructions de répétition.

**6.3. Transformation des boucles en procédures récursives**

Soit la procédure suivante contenant une boucle simple :

```
procedure compter ;  
var  
  i : integer ;  
begin  
  for i := 1 to 10 do writeln(i) ;  
end ;
```

Cette procédure peut être transformée en une procédure récursive avec un paramètre; l'instruction qui l'appellera sera compter(1) :

```
procedure compter(i : integer) ;  
begin  
  if (i < 11) then begin  
    writeln(i) ;  
    compter(i + 1) ;  
  end ;  
end ;
```

Supposons qu'on a maintenant deux boucles imbriquées. Nous allons traduire progressivement cette procédure itérative en une procédure récursive avec deux paramètres :

```
procedure affiche ;  
var  
  a, b : integer ;  
begin  
  for a := 0 to 3 do  
    for b := 0 to 9 do writeln(a * 10 + b) ;  
  end ;
```

Pour supprimer les deux boucles, on commence par supprimer la première en suivant l'exemple ci-dessus; on obtient la procédure suivante que l'on appelle avec `affiche(0)` :

```
procedure affiche(a : integer) ;  
var  
  b : integer ;  
begin  
  if (a < 4) then begin  
    for b := 0 to 9 do writeln(a * 10 + b) ;  
    affiche(a+1) ;  
  end ;  
end ;
```

Il ne nous reste plus qu'à supprimer la deuxième boucle. Sachant que lorsque  $b=10$  dans la procédure initiale, le programme revient à la boucle sur  $a$ , et remet  $b$  à zéro, alors on a deux appels récursifs :

- Le premier est dans le cas où  $b$  est inférieur à 10. Alors on appelle la procédure avec  $a$  inchangé et  $b$  incrémenté de 1.
- Le deuxième cas est là où  $b=10$ . Alors on appelle la procédure avec  $a$  incrémenté de 1 et  $b$  initialisé à 0.

L'appel est effectué par `affiche(0,0)`.

```
procedure affiche(a, b : integer);  
begin  
  if (a < 4) then  
    if (b < 10) then begin  
      writeln(a * 10 + b) ;  
      affiche(a, b + 1) ;  
    end  
    else affiche(a + 1, 0) ;  
  end ;  
end ;
```

#### 6.4. La récursion croisée (indirecte)

Si un sous-programme  $p$  fait appel à un sous-programme  $q$  qui à son tour appelle le sous-programme  $p$ , on dit qu'il y a une récursivité croisée indirecte entre ces deux sous-programmes.

En Pascal, on utilise le mot clé `forward`, pour indiquer une déclaration plus loin.

##### Exemple :

```
program recursion_croisee ;  
function F(x : integer) : integer ; forward ;  
function P(x : integer) : integer ;
```

```
begin
  if (x = 0) then P := 1
    else P := F(x-1) + P(x-1) ;
end ;
function F (x : integer) : integer ;
begin
  if (x = 0) then F := 1
    else F := F(x-1) + P(x-1);
end ;
begin
  writeln(F(5)) ;
end.
```

Le résultat affiché est 32.

## 7. Exercices corrigés

### 7.1. Exercices

#### Exercice 1 :

Ecrire un programme Pascal permettant de calculer la surface d'un rectangle en utilisant une fonction ayant comme paramètres la longueur et la largeur de ce rectangle.

#### Exercice 2 :

Ecrire un programme Pascal permettant d'afficher la somme de deux entiers et la concaténation de deux chaînes de caractères en utilisant une procédure qui a comme paramètres deux entiers et deux chaînes de caractères.

#### Exercice 3 :

Dans le programme Pascal suivant, remplacez la fonction par une procédure :

```
program somme ;
var x, y : integer ;
function S (a, b : integer) : integer ;
begin
  S := a + b ;
end;
begin
  writeln('Donnez la première valeur :');
  readln(x) ;
  writeln('Donnez la deuxième valeur :');
  readln(y) ;
  writeln('Somme = ', S(x,y) ) ;
end.
```



**Exercice 4 :**

Ecrire un programme Pascal permettant à partir de la saisie de trois entiers  $a$ ,  $b$  et  $c$ , de calculer la factorielle de  $a$ , de  $b$  ou de  $c$ . L'utilisateur doit introduire son choix à partir d'un menu. Le calcul de la factorielle doit être effectué par une fonction. Remplacer ensuite (dans un autre programme) la fonction par une procédure.

**Exercice 5 :**

Ecrire un programme Pascal permettant lire deux entiers, de permuter leurs contenus en utilisant une procédure qui recevra ces deux entiers en paramètres, et de les afficher afin de vérifier la permutation.

**Exercice 6 :**

Quel est le résultat d'exécution des deux programmes suivants ?

program Afficher1 ;

var

  i : integer ;

procedure imp(a : integer);

begin

  writeln(a);

  a := a+2 ;

end ;

begin

  i := 1;

  while (i<10) do begin

    imp(i);

    i := i+3;

  end;

end.

-----  
program Afficher2 ;

var

  i : integer ;

procedure imp(var a : integer);

begin

  writeln(a);

  a := a+2 ;

end ;

begin

  i := 1;

  while (i<10) do begin

```
    imp(i);  
    i := i+3;  
end;  
end.
```

**Exercice 7 :**

1. Qu'affiche le programme Pascal suivant ?

```
program Moyenne ;  
var  
    x, y, Moy : integer;  
Function M(a, b : integer) : integer ;  
begin  
    M := (a + b) div 2 ;  
    a := b;  
    b := 2 ;  
end;  
begin  
    x := 11 ;  
    y := 9 ;  
    Moy := M(x,y);  
    writeln('La moyenne de ', x, ' et ', y, ' est : ', Moy);  
end.
```

2. Qu'affiche programme précédent, mais cette fois-ci, si on remplace  $M(a, b : \text{integer})$  par  $M(a : \text{integer} ; \text{var } b : \text{integer})$ , par  $M(\text{var } a : \text{integer} ; b : \text{integer})$ , et enfin par  $M(\text{var } a, b : \text{integer})$  ?

**Exercice 8 :**

Ecrire en langage Pascal une fonction booléenne qui prend trois paramètres réels : les deux premiers sont les bornes d'un intervalle, et le troisième est (éventuellement) modifié de manière à rester dans l'intervalle spécifié. La fonction renvoie true si et seulement si le troisième paramètre a été effectivement modifié, sinon elle renvoie false.

**Exercice 9 :**

Ecrire un programme Pascal qui calcule le maximum de 4 réels saisis au clavier. Le calcul du maximum de deux valeurs sera effectué par une fonction.

Dérouler le programme (décrire le changement des variables, instruction par instruction) si l'utilisateur introduit les valeurs suivantes : 50, 9, 80, 5.

**Exercice 10 :**

En utilisant la notion de sous-programme, essayez de réduire la taille du programme Pascal suivant :

```

program sm ;
var x, y, z, s : integer ;
begin
  readln(x,y,z);
  s:=x+y;
  writeln(x, ' + ', y, ' = ',s) ;
  s:=y+z;
  writeln(y, ' + ', z, ' = ',s) ;
  s:=z+x;
  writeln(z, ' + ', x, ' = ',s) ;
end.

```

**Exercice 11 :**

Soit le problème : Trouvez le maximum avec sa position et le minimum avec sa position dans un tableau de dix entiers.

Subdivisez ce problème en sous-problèmes simples. Ensuite, essayez de les résoudre par la notion de sous-programmes en langage Pascal.

**Exercice 12 :**

Même question pour le problème suivant : trie le sous-tableau pris entre deux composantes dont les indices correspondent aux valeurs *max* et *min* en ordre croissant dans un tableau de dix entiers.

**Exercice 13 :**

Même question pour le problème suivant : un étudiant doit, pour obtenir son diplôme, passer un écrit et un oral dans deux modules. Le coefficient du premier module est égal à 1. Le coefficient du second module est égal à 2. La moyenne d'un module, afin de ne pas pénaliser trop les éventuels échecs accidentels, accorde le coefficient 2 à la meilleure des deux notes obtenues et le coefficient 1 à l'autre note. Après saisie des quatre notes, la décision finale est affichée (diplôme obtenu si la moyenne est supérieure ou égale à 10 ; aucun module ne devant avoir une moyenne inférieure à 8).

**Exercice 14 :**

1. Qu'affiche le programme Pascal suivant ?

```

program calcul ;
var
  Fact : integer ;
procedure module_recuratif(F, N : integer);
begin
  F := F*N;
  if (N<>1) then module_recuratif(F, N-1);
end ;

```

```

procedure essai(N : integer) ;
begin
  Fact := 1 ;
  module_recuratif(Fact, N);
  writeln(Fact, '..', N);
end ;
begin
  essai(3);
end.

```

2. Qu'affiche programme précédent, mais cette fois-ci, si on remplace module\_recuratif(F, N : integer) par module\_recuratif(var F : integer ; N: integer) ?
3. Que va-t-il se passer pour module\_recuratif(var F : integer ; var N: integer) ?

### Exercice 15 :

Expliquez ce que font les procédures suivantes avec  $n \geq 0$ .

```

procedure P1 ( n : integer) ;
begin
  if (n>0) then begin
    writeln(n) ;
    P1(n-1) ;
    writeln(n) ;
  end ;
end ;

```

```

-----
procedure P2( n : integer) ;
begin
  if (n>0) then P2(n-1)
  else writeln(n) ;
end ;

```

```

-----
procedure P3( n : integer) ;
begin
  if (n>0) then begin
    writeln(n);
    P3(n div 2) ;
  end;
end ;

```

### Exercice 16 :

Ecrire un programme Pascal permettant la conversion d'un nombre décimal en binaire en utilisant une procédure récursive.

**Exercice 17 :**

Ecrire un programme Pascal permettant la conversion d'un nombre décimal en chaîne de caractères en utilisant une procédure récursive.

**Exercice 18 :**

Ecrire un programme Pascal qui utilise une fonction récursive pour déterminer si une chaîne de caractères est un palindrome ou non.

Une chaîne est palindrome si l'ordre de ses lettres reste le même qu'on le lise de gauche à droite ou de droite à gauche. Par exemple : "elle".

**Exercice 19 :**

Ecrire un programme Pascal qui contient une fonction récursive dont la valeur serait le miroir d'une chaîne donnée.

Par exemple, le miroir de la chaîne "ahmed" est "demha".

**Exercice 20 :**

Ecrire un programme Pascal permettant le calcul de la puissance en utilisant une fonction récursive.

**Exercice 21 :**

Ecrire un programme Pascal contenant une fonction récursive permettant de calculer le  $n^{\text{ième}}$  terme de la suite de Fibonacci définie comme suit : Si  $n = 0$  alors  $F_n = 0$ . Si  $n = 1$  alors  $F_n = 1$ . Si  $n > 1$  alors  $F_n = F_{n-1} + F_{n-2}$ .

**Exercice 22 :**

Ecrire un programme Pascal permettant le calcul du maximum dans un tableau de dix entiers en utilisant une procédure récursive.

**Exercice 23 :**

Ecrire un programme Pascal permettant la recherche dichotomique dans un tableau de cinq réels triés, en utilisant une procédure récursive.

**7.2. Corrigés**

**Solution 1 :**

```
program surface_rec ;
var
  long, larg : real ;
function surface ( a, b : real ) : real ;
begin
  surface := a*b;
end;
begin
  writeln('Donnez la longueur et la largeur d'un rectangle :');
  readln(long, larg);
  writeln('La surface = ', surface(long, larg)) ;
end.
```

**Solution 2 :**

```
program somme_concat ;
var
  x, y : integer ;
  ch1, ch2 : string;
procedure som_con(a, b : integer ; c1, c2 : string) ;
begin
  writeln('Le résultat de la somme = ', a+b) ;
  writeln('Le résultat de la concaténation = ', c1+c2) ;
end;
begin
  writeln('Donnez deux nombres :');
  readln(x, y);
  writeln('Donnez deux chaînes de caractères :');
  readln(ch1, ch2);
  som_con(x, y, ch1, ch2) ;
end.
```

**Solution 3 :**

La fonction S peut être substituer par une procédure S en ajoutant une variable globale som. On obtient alors :

```
program somme ;
var x, y, som : integer ;
procedure S (a, b : integer) ;
begin
  som := a + b ;
end;
begin
  writeln('Donnez la première valeur :');
  readln(x) ;
  writeln('Donnez la deuxième valeur :');
  readln(y) ;
  S(x,y) ;
  writeln('Somme = ', som) ;
end.
```

**Solution 4 :**

Le programme en utilisant une fonction :

```
program factorielle ;
var
  a, b, c, choix : integer ;
```

```

function facto (x : integer) : integer ;
var
  i, f : integer;
begin
  f := 1 ;
  for i := 1 to x do f := f * i ;
  facto := f ;
end;
begin
  writeln('Donnez trois entiers :');
  readln(a, b, c);
  writeln('Tapez 1 pour claculer la factorielle de ', a);
  writeln('Tapez 2 pour claculer la factorielle de ', b);
  writeln('Tapez 3 pour claculer la factorielle de ', c);
  readln(choix) ;
  case choix of
    1 : if (a >= 0) then writeln('La factorielle de ', a, ' est : ', facto(a))
        else writeln('Calcul impossible. ');
    2 : if (b >= 0) then writeln('La factorielle de ', b, ' est : ', facto(b))
        else writeln('Calcul impossible. ');
    3 : if (c >= 0) then writeln('La factorielle de ', c, ' est : ', facto(c))
        else writeln('Calcul impossible. ');
    else writeln('Choix invalide. ');
  end;
end.

```

Le programme en utilisant une procedure :

Programme 1 :

```

program factorielle ;
var
  a, b, c, choix, Z : integer ;
procedure facto (x : integer) ;
var
  i, f : integer;
begin
  f := 1 ;
  for i := 1 to x do f := f * i ;
  Z := f ;
end;

```

```

begin
  writeln('Donnez trois entiers :');
  readln(a, b, c);
  writeln('Tapez 1 pour claculer la factorielle de ', a);
  writeln('Tapez 2 pour claculer la factorielle de ', b);
  writeln('Tapez 3 pour claculer la factorielle de ', c);
  readln(choix) ;
  case choix of
    1 : if (a >= 0) then
          begin facto(a) ;writeln('La factorielle de ',a, ' est : ', z) end
        else writeln('Calcul impossible.');
```

```

    2 : if (b >= 0) then
          begin facto(b) ;writeln('La factorielle de ',b, ' est : ', z) end
        else writeln('Calcul impossible.');
```

```

    3 : if (c >= 0) then
          begin facto(c) ;writeln('La factorielle de ',c, ' est : ', z) end
        else writeln('Calcul impossible.')
```

```

  else writeln('Choix invalide');
```

```
end;
```

```
end.
```

### Programme 2 :

```
program factorielle ;
```

```
var
```

```
  a, b, c, f, choix : integer ;
```

```
procedure facto (x : integer) ;
```

```
var
```

```
  i : integer;
```

```
begin
```

```
  f := 1 ;
```

```
  if (x >= 0) then begin for i := 1 to x do f := f * i ;
```

```
    writeln('La factorielle de ',x, ' est : ', f); end
```

```
  else writeln('Calcul impossible.');
```

```
end;
```

```
begin
```

```
  writeln('Donnez trois entiers :');
```

```
  readln(a, b, c);
```

```
  writeln('Tapez 1 pour claculer la factorielle de ', a);
```

```
  writeln('Tapez 2 pour claculer la factorielle de ', b);
```

```
  writeln('Tapez 3 pour claculer la factorielle de ', c);
```



```
readln(choix) ;
case choix of
  1 : facto(a);
  2 : facto(b);
  3 : facto(c)
  else writeln('Choix invalide. ');
end;
end.
```

**Solution 5 :**

```
program permutation ;
var
  val1, val2 : integer ;
procedure permute ( var x, y : integer ) ;
var
  z : integer;
begin
  z := x ;
  x := y ;
  y := z ;
end;
begin
  writeln('Donnez deux entiers :');
  write('val1 = ' ) ; readln(val1);
  write('val2 = ' ) ; readln(val2);
  permute(val1, val2);
  writeln('Après la permutation val1 =', val1, ' et val2 = ', val2) ;
end.
```

**Solution 6 :**

Pour le premier programme, le résultat affiché est :

1  
4  
7

Pour le deuxième programme, le résultat affiché est :

1  
6

**Solution 7 :**

1. Le programme affiche : La moyenne de 11 et 9 est : 10
2. Le programme affiche :
  - Pour M(a : integer ; var b : integer) : La moyenne de 11 et 2 est : 10

- Pour M(var a : integer ; b : integer) : La moyenne de 9 et 9 est : 10
- Pour M(var a, b : integer) : La moyenne de 9 et 2 est : 10

**Solution 8 :**

```
function Intervalle (inf, sup : real; var x : real) : boolean ;
var
  res : boolean ;
begin
  res := false ;
  if x < inf then begin
    x := inf ;
    res := true ;
  end ;
  if x > sup then begin
    x := sup ;
    res := true ;
  end ;
  Intervalle := res ;
end ;
```

**Solution 9 :**

```
program calculerMax4 ;
var
  i : integer ;
  maximum, nombre : real ;
function calculerMax2(x, y : real) : real ;
var res : real ;
begin
  if x > y then res := x
  else res := y ;
  calculerMax2 := res ;
end ;
begin
  readln(maximum) ;
  for i := 1 to 3 do begin
    readln(nombre) ;
    maximum := calculerMax2(nombre, maximum) ;
  end ;
  writeln('Le maximum = ', maximum) ;
end.
```

Pour le déroulement du programme, on va noter les instructions du programme principal comme suit :

Instruction	Notation
readln(maximum)	P1
i := ?	P2
readln(nombre)	P3
maximum := calculerMax2	P4
writeln('Le maximum = ', maximum)	P5

On va noter les instructions de la fonction comme suit :

Instruction	Notation
calculerMax2(nombre, maximum)	F1
if x > y then	F2
res := x	F3
res := y	F4
calculerMax2 := res	F5

Le tableau suivant correspond au schéma d'évolution d'état de la mémoire, instruction par instruction :

Variable Instruction	Programme principal			La fonction				
	i	maximum	nombre	x	y	x>y	res	calculerMax2
P1		50						
P2	1							
P3			9					
F1				50	9			
F2						true		
F3							50	
F5								50
P4		50						
P2	2							
P3			80					
F1				50	80			
F2						false		
F4							80	
F5								80
P4		80						
P2	3							
P3			5					
F1				80	5			
F2						true		
F3							80	
F5								80
P4		80						
P5		80						

**Solution 10 :**

```

program sm ;
var
  x, y, z, s : integer ;
procedure somme ( a, b : integer) ;
begin
  s := a+b ;
  writeln(a, ' + ', b, ' = ', s) ;
end;
begin
  readln(x, y, z);
  somme(x, y);
  somme(y, z);
  somme(z, x);
end.

```

**Solution 11 :**

Ce problème peut être subdivisé en trois sous-problèmes :

1. Déterminer le max d'un tableau.
2. Déterminer le min d'un tableau.
3. Déterminer la position d'une valeur.

Et le programme Pascal sera le suivant :

```

program maxmin ;
var
  tab : array[1..10] of integer ;
  max, min, i : integer ;
procedure maximum ;
begin
  max := tab[1];
  for i := 2 to 10 do if max<tab[i] then max := tab[i] ;
end;
procedure minimum ;
begin
  min := tab[1];
  for i := 2 to 10 do if min>tab[i] then min := tab[i] ;
end;
function position (x:integer): integer;
begin
  for i := 1 to 10 do if tab[i]=x then position := i ;
end;

```

```
begin
  for i:= 1 to 10 do readln(tab[i]);
  maximum ;
  minimum ;
  writeln('Le maximum est ', max, ' sa position est ', position(max)) ;
  writeln('Le minimum est ', min, ' sa position est ', position(min)) ;
end.
```

### **Solution 12 :**

Ce problème peut être subdivisé en quatre sous-problèmes :

1. Déterminer le max d'un tableau.
2. Déterminer le min d'un tableau.
3. Déterminer la position d'une valeur.
4. Trier une partie d'un tableau.

Et le programme Pascal sera le suivant :

```
program trimaxmin ;
var
  tab : array[1..10] of integer ;
  max, min, posmax, posmin, i : integer ;
procedure maximum ;
begin
  max := tab[1];
  for i := 1 to 10 do if max<tab[i] then max := tab[i] ;
end;
procedure minimum ;
begin
  min := tab[1];
  for i := 1 to 10 do if min>tab[i] then min := tab[i] ;
end;
function position (x:integer): integer;
begin
  for i := 1 to 10 do if tab[i]=x then position := i ;
end;
procedure tri(x, y : integer);
var
  z, j : integer;
begin
  for i := x to y-1 do
    for j := i+1 to y do if tab[i]>tab[j] then begin
      z := tab[i];
```

```

        tab[i] := tab[j];
        tab[j] := z;
    end;
end;
begin
    writeln('Donnez les éléments du tableau :');
    for i:= 1 to 10 do readln(tab[i]);
    maximum ;
    minimum ;
    posmax :=0 ; posmin := 0 ;
    posmax := position(max) ;
    posmin := position(min) ;
    writeln('Le maximum est ', max, ' sa position est ', posmax) ;
    writeln('Le minimum est ', min, ' sa position est ', posmin) ;
    if (posmin < posmax) then tri(posmin, posmax)
        else tri(posmax, posmin) ;
    writeln('Voici le tableau après l"opération de tri :');
    for i:= 1 to 10 do writeln(tab[i]);
end.

```

### **Solution 13 :**

Ce problème peut être subdivisé en trois sous-problèmes :

1. Calculer la moyenne de deux notes avec les coefficients 1 et 2 successivement.
2. Calculer la moyenne pour un module. Les coefficients étant déterminés à base de la valeur supérieure.
3. Accorder le diplôme selon la moyenne générale et les moyennes des deux modules.

Et le programme Pascal sera comme suit :

```

program Afficher_resultat ;
var
    ne_m1, no_m1, ne_m2, no_m2 : real ;
    obtenu : boolean ;
function calculerMoyenne(n1, n2 : real) : real ;
var
    moy : real ;
begin
    moy := (n1 + 2*n2)/3 ;
    calculerMoyenne := moy ;
end ;

```

```

function calculerNoteModule (n1, n2 : real) : real ;
var note : real ;
begin
  if n1 > n2 then note := calculerMoyenne(n2, n1)
    else note := calculerMoyenne(n1, n2) ;
  calculerNoteModule := note ;
end ;
function accorderDiplome(m1, m2 : real) : boolean ;
var
  moy : real ;
  recu : boolean ;
begin
  moy := calculerMoyenne(m1, m2) ;
  if moy < 10 then recu := false
    else if (m1 < 8) OR (m2 < 8) then recu := false
      else recu := true ;
  accorderDiplome := recu;
end ;
begin
  write('Donnez la note d"écrit du 1ier module : '); readln(ne_m1) ;
  write('Donnez la note d"oral du 1ier module : '); readln(no_m1) ;
  write('Donnez la note d"écrit du 2ième module : '); readln(ne_m2) ;
  write('Donnez la note d"oral du 2ième module : '); readln(no_m2) ;
  obtenu := accorderDiplome(calculerNoteModule(ne_m1, no_m1),
    calculerNoteModule(ne_m2, no_m2)) ;
  if obtenu then writeln('Diplôme accordé.')
    else writeln('Diplôme non accordé.') ;
end.

```

#### **Solution 14 :**

1. Pour module\_recuratif(F, N : integer), le programme affiche : 1..3.
2. Pour module\_recuratif(var F : integer ; N: integer), le programme affiche : 6..3.
3. Pour module\_recuratif(var F : integer ; N: integer), il va y avoir une erreur lors de l'invocation récursive module\_recuratif(F, N-1) car on a mis une expression composée dans un paramètre effectif correspondant à un paramètre formel à passage par adresse.

#### **Solution 15 :**

Pour un entier positif  $n$ , la procédure P1( $n$ ) permet d'afficher les valeurs de  $n$  jusqu'à 1, ensuite de 1 jusqu'à  $n$ .

Par exemple, pour  $n=3$  la procédure affiche :

3  
2  
1  
1  
2  
3

Pour un entier  $n$  positif, la procédure  $P2(n)$  permet d'afficher 0 quelle que soit la valeur de  $n$ . Par exemple, pour  $n=3$  la procédure affiche :

0

Pour un entier  $n$  positif, la procédure  $P3(n)$  permet d'afficher  $n$  suivi des résultats successifs de la division entière par 2. Par exemple, pour  $n=11$  la procédure affiche :

11  
5  
2  
1

**Solution 16 :**

Solution 1)

```
program conv_en_binaire ;
var
  n : integer;
  binaire : string ;
procedure conv(x : integer);
begin
  if (x div 2 = 0) then
    if (x mod 2 = 0) then binaire := '0' + binaire
    else binaire := '1' + binaire
  else begin
    if (x mod 2 = 0) then binaire := '0' + binaire
    else binaire := '1' + binaire;
    conv(x div 2) ;
  end ;
end ;
begin
  writeln('Donnez un entier :)';
  readln(n) ;
  binaire := "";
  conv(n);
```



```
writeln('Le nombre ', n, ' est egal en binaire à ', binaire) ;
end.
```

Solution 2)

```
program conv_en_binaire ;
var
  n, Q, R : integer;
  binaire : string ;
procedure conv(x,y : integer);
var
  z : integer;
begin
  if (x > 0) then begin
    z := x ;
    x := z DIV 2 ;
    y := z MOD 2 ;
    if (y = 0) then binaire := '0' + binaire
      else binaire := '1' + binaire;
    conv(x, y) ;
  end ;
end ;
begin
  writeln('Donnez un entier :) ;
  readln(n) ;
  binaire := "";
  Q := n DIV 2;
  R := n MOD 2;
  if (R = 0) then binaire := '0'
    else binaire := '1' ;
  conv(Q,R);
  writeln('Le nombre ', n, ' est egal en binaire à ', binaire) ;
end.
```

**Solution 17 :**

```
program conv_en_chaine ;
var
  n, Q, R : integer;
  CH : string ;
function lettre(x : integer): char;
begin
  case x of
```

```
0: lettre := '0';
1: lettre := '1';
2: lettre := '2';
3: lettre := '3';
4: lettre := '4';
5: lettre := '5';
6: lettre := '6';
7: lettre := '7';
8: lettre := '8';
9: lettre := '9';
end;
end;
procedure conv(x,y : integer);
var
  z : integer;
begin
  if (x > 0) then begin
    z := x ;
    x := z DIV 10 ;
    y := z MOD 10 ;
    CH := lettre(y) + CH;
    conv(x, y) ;
  end ;
end ;
begin
  writeln('Donnez un entier :)';
  readln(n) ;
  Q := n DIV 10;
  R := n MOD 10;
  CH := lettre(R);
  conv(Q,R);
  writeln('Le nombre ', n, ' correspond à la chaîne ', CH) ;
end.
```

**Solution 18 :**

```
program palindrome ;
var
  chaine : string ;
function pal( ch : string) : boolean;
begin
```

```
if (length(ch)=0)or(length(ch)=1) then pal := true
  else if (ch[1]=ch[length(ch)]) then pal:= pal(COPY(ch, 2, length(ch)-2))
  else pal := false;
end;
begin
  writeln('Donnez une chaîne de caractères :');
  readln(chaine);
  if pal(chaine) then writeln('La chaîne ', chaine, ' est un palaindrome.')
    else writeln('La chaîne ', chaine, ' n"est pas un palindrome.');
```

end.

**Solution 19 :**

```
program chaine_miroir ;
var chaine : string ;
function miroir( ch : string) : string;
begin
  if (length(ch)=0) OR (length(ch)=1) then miroir := ch
    else miroir := ch[length(ch)] + miroir(COPY(ch, 1, length(ch)-1));
end;
begin
  writeln('Donnez une chaîne de caractères :');
  readln(chaine);
  writeln('La chaîne miroir de ', chaine, ' est ', miroir(chaine));
```

end.

**Solution 20 :**

```
program puissance_entier ;
var
  n,p : integer;
function puissance(x,y : integer): integer ;
begin
  if (y = 0) then puissance := 1
    else puissance := x * puissance(x, y-1);
end;
begin
  writeln('Donnez un nombre :');
  readln(n);
  writeln('Donnez la puissance :');
  readln(p);
  writeln(n,' puissance ', p,' = ', puissance(n,p)) ;
```

end.

**Solution 21 :**

Solution 1)

```
program fibonacci ;
var
  n : integer ;
function fibo (n : integer) : integer ;
begin
  if (n = 0) then fibo := 0
  else if (n = 1) then fibo := 1
  else fibo := fibo(n-1) + fibo(n-2) ;
end;
begin
  write('Entrez une valeur : ');
  readln(n) ;
  writeln('Fibo(',n,') = ', fibo(n));
end.
```

Solution 2)

```
program fibonacci ;
var
  n : integer ;
function fibo (n : integer) : integer ;
begin
  if (n = 0) or (n = 1) then fibo := n
  else fibo := fibo(n-1) + fibo(n-2) ;
end;
begin
  write('Entrez une valeur : ');
  readln(n) ;
  writeln('Fibo(',n,') = ', fibo(n));
end.
```

**Solution 22 :**

```
program max_tab ;
var
  tab : array[1..10] of integer ;
  max, i : integer;
procedure maximum(i : integer) ;
begin
  if (i < 11) then begin
    if (max < tab[i]) then max := tab[i];
```

```
    maximum(i+1);
end;
end;
begin
    writeln('Donnez les dix valeurs du tableau :');
    for i:= 1 to 10 do readln(tab[i]);
    max := tab[1];
    maximum(2);
    writeln('Le maximum est ', max);
end.
```

**Solution 23 :**

```
program Recherche_dichotomique ;
var
    NOMBRES : array [1..5] of real ;
    NBR : real ;
    TROUVE : boolean;
    i : integer ;
procedure dico(f, s : integer) ;
var m : integer;
begin
    m := (f + s) DIV 2 ;
    if (f <= s) then
        if (NBR = NOMBRES[m]) then TROUVE := true
        else if (NBR < NOMBRES[m]) then dico(f, m-1)
        else dico(m+1, s);
    end;
begin
    writeln('Entrez les nombres triés du tableau :');
    for i := 1 to 5 do readln(NOMBRES[i]);
    writeln('Entrez le nombre à rechercher :');
    readln(NBR);
    TROUVE := false ; dico(1, 5);
    if TROUVE then writeln('Le nombre ', NBR, ' existe dans le tableau.')
    else writeln('Le nombre ', NBR, ' n'existe pas dans le tableau.') ;
end.
```

## Chapitre 7 : La complexité des algorithmes

### 1. Introduction

Pour un problème donné, il existe plusieurs algorithmes. Il est donc important de pouvoir comparer ces algorithmes et choisir le meilleur. Il y a deux critères principaux utilisés pour la comparaison :

- L'espace mémoire utilisé par l'algorithme, et on parle de la complexité spatiale.
- Le temps d'exécution de l'algorithme, et on parle de la complexité temporelle.

On est souvent amené à améliorer les algorithmes par une opération d'optimisation. L'optimisation consiste à réduire le coût de l'algorithme. L'optimisation en espace consiste à réduire la complexité spatiale. L'optimisation en temps consiste à réduire la complexité temporelle.

Les situations où on doit réduire la complexité spatiale sont plus rares. De ce fait, on s'intéresse dans ce qui suit à l'optimisation en temps des algorithmes, et on utilise le terme complexité pour dire *complexité temporelle*.

### 2. Calcul de la complexité

Chaque instruction a son propre temps d'exécution. Par exemple, l'affectation est quasi-instantanée, l'addition et la soustraction sont très rapides, la multiplication un peu moins, la division est relativement lente, etc.

Pour calculer la complexité, on ne prend que des opérations élémentaires, considérées comme importantes (par exemple, la comparaison, l'affectation, opération d'E/S). On suppose que chaque opération élémentaire prend un temps d'exécution fixe. Le calcul de la complexité consiste à déterminer le nombre d'opérations importantes exécutées par l'algorithme. On distingue trois sortes de complexités :

#### 2.1. Complexité dans le pire des cas (le cas le plus défavorable)

La complexité dans le pire des cas d'un algorithme est une fonction  $C_{max}$  de  $N$  dans  $N$  telle que  $C_{max}(n)$  est le nombre maximum de fois que l'algorithme effectue une opération importante ; le maximum étant pris parmi tous les temps d'exécution.

On note simplement  $C(n)$  lorsqu'il est clair qu'on s'intéresse à la complexité dans le pire des cas.

#### 2.2. Complexité dans le meilleur des cas (le cas le plus favorable)

La complexité dans le meilleur des cas d'un algorithme est une fonction  $C_{min}$  de  $N$  dans  $N$  telle que  $C_{min}(n)$  est le nombre minimum de fois que

l'algorithme effectue une opération importante ; le minimum étant pris parmi tous les temps d'exécution.

### 2.3. Complexité moyenne

La complexité moyenne d'un algorithme est une fonction  $C_{moy}$  de  $N$  dans  $N$  telle que  $C_{moy}(n)$  est le nombre moyen de fois que l'algorithme effectue une opération importante ; la moyenne étant prise parmi tous les temps d'exécution. Dans ce cas, la complexité ne peut être évaluée qu'en faisant des hypothèses statistiques sur les cas d'exécution.

### 3. Etude de cas

On veut maintenant calculer la complexité d'un extrait d'un algorithme : l'idée est d'évaluer le temps d'exécution ou encore le nombre d'opérations exécutées par notre algorithme.

1. Min $\leftarrow i - 1$	1 affectation
2. Pour j $\leftarrow i$ à n Faire	1 affectation + 1 comparaison. #boucle(# nombre de répétition)
3. Si A[j] < A[Min] Alors	1 comparaison. # boucle
4. Min $\leftarrow j$ ;	(Si test VRAI : 1 affectation). #boucle

- Dans le pire des cas (A[j] < A[Min] est toujours VRAI) :  $C_{max}(n) = 1 + 4 * (n-i+1)$ .
- Supposons que, sur les (n-i+1) tests, la moitié est évaluée à VRAI. En moyenne,  $C_{moy}(n) = 1 + 3 * (n-i+1) + (n-i+1)/2$ .
- Dans le meilleur des cas (A[j] < A[Min] est toujours FAUX), on n'exécute jamais l'instruction Min  $\leftarrow j$ , alors  $C_{min}(n) = 1 + 3 * (n-i+1)$ .

### 4. Ordres de grandeur (Classes de complexité)

On évalue la complexité d'un algorithme pour déterminer l'ordre de grandeur de la complexité. Ceci permet de savoir si l'algorithme peut traiter dans un temps raisonnable des données de grande taille.

#### Définitions :

- Pour deux fonctions réelles  $f(n)$  et  $g(n)$ , on écrira :  $f(n) = O(g(n))$ , Si et seulement s'il existe deux constantes strictement positives  $n_0$  et  $c$  avec :  $0 \leq f(n) \leq c \times g(n)$ , pour tout  $n$  supérieur à  $n_0$ . Dans ce cas, on dit que  $f$  positive est asymptotiquement majorée (ou dominée) par  $g$ .
- On définit symétriquement la minoration de  $f$  par  $g$  en notant :  $f(n) = \Omega(g(n))$ .
- Quand  $f(n) = O(g(n))$  et  $f(n) = \Omega(g(n))$ , on dit que  $f$  est de même ordre de grandeur que  $g$  (par fois on dit que  $g$  est un encadrement asymptotique de  $f$ ), et on écrit :  $f(n) = \Theta(g(n))$ . Dans ce cas, il existe trois constantes strictement positives  $n_0$ ,  $c'$  et  $c$  avec :  $c' \times g(n) \leq f(n) \leq c \times g(n)$ , pour tout  $n$  supérieur à  $n_0$ .

D'après les définitions ci-dessus, on peut ainsi établir un ordre (non exhaustif) entre les fonctions mathématiques (du plus petite au plus grande) :  $O(1)$ ,  $O(\log(n))$ ,  $O(nx)$ ,  $O(n)$ ,  $O(n.\log(n))$ ,  $O(n^c)$ ,  $O(c^n)$ ,  $O(n!)$ , avec  $0 < x < 1 < c$ .

On notera en particulier que le taux de croissance est très rapide dans les fonctions exponentielles ( $n^c$ ,  $c^n$ ,  $n!$ ).

La notation  $f = O(g)$  est scabreuse, car elle ne dénote pas une égalité mais plutôt l'appartenance de  $f$  à la classe des fonctions en  $O(g)$ .

Voici les complexités les plus fréquentes avec leurs noms.

$O(1)$	complexité constante
$O(\log(n))$	complexité logarithmique
$O(n)$	complexité linéaire
$O(n.\log(n))$	complexité quasi-linéaire
$O(n^2)$	complexité quadratique
$O(n^3)$	complexité cubique
$O(n^p)$	complexité polynomiale
$O(2^n)$	complexité exponentielle
$O(n!)$	complexité factorielle

**Notes :**

- L'ordre de complexité permet de déterminer le comportement de l'algorithme quand  $n$  tend vers l'infinie.
- L'ordre de complexité d'un algorithme est souvent déterminé par le calcul de la complexité dans le pire des cas.
- On admet généralement que les algorithmes de complexité  $O(1)$ ,  $O(\log(n))$ ,  $O(n)$  et  $O(n.\log(n))$  sont efficaces, i.e. rapides en pratique, même pour des données initiales de grande taille. Les algorithmes de complexité polynomiale peuvent aussi être considérés efficaces à condition que l'exposant ne soit pas trop grand. Enfin, les algorithmes de complexité exponentielle ou factorielle sont inefficaces.
- Lorsqu'on fait un appel à une fonction ou procédure, il faut compter les opérations incluses dans cette fonction ou procédure pour le calcul de la complexité.

**Règles :**

1. Dans l'estimation de l'ordre de grandeur, on garde seulement le terme le plus significatif de  $f(n)$ . Les termes d'ordre inférieur et les facteurs constants sont négligeables. Alors si  $f(n)$  est un polynôme d'ordre  $k$  en  $n$ , l'ordre de grandeur sera  $O(n^k)$ .
2. Toute fonction constante  $C$  est de complexité constante,  $C = O(1)$ .



### Exemples :

- $4n^2+n = O(n^2)$  et  $n^2-3 = O(n^2)$  sans que l'on ait  $4n^2+n = n^2-3$ , à partir d'un  $n$  assez grand. Dans le cas de la complexité quadratique, si  $n$  double, le temps est multiplié par 4.
- Dans le cas de produit matricielle, la complexité est cubique  $O(n^3)$ . Si  $n$  double, le temps est multiplié par 8.
- La complexité de l'algorithme de recherche d'un élément dans un ensemble ordonné fini de cardinal  $n$  (recherche dichotomique) est logarithmique  $O(\log(n))$ . Dans ce cas, le bloc d'opérations est répété  $n/2^k$ .
- Parmi les problèmes complexes les plus connus, celui du commis voyageur : on demande à trouver le chemin le plus court pour un commis voyageur qui doit visiter  $n$  villes. Si on calcule tous les ordres de parcours possibles pour décider le chemin le plus court, il y a  $n!$  ordres possibles de parcours des  $n$  villes. Ceci donne une complexité de l'ordre de  $O(n!)$ . Dans un tel algorithme, augmenter  $n$  de seulement une unité multiplie le temps d'exécution par  $n$ . Déjà pour 20 villes, à une opération par nanoseconde ( $10^{-9}s$ ), le temps d'exécution dépasse mille années !

### Propriétés :

1. Réflexivité :  $f(n) = O(f(n))$ .
2. Transitivité : si  $f(n) = O(g(n))$  et  $g(n) = O(h(n))$  alors  $f(n) = O(h(n))$ .
3. Somme : si  $f(n) = O(h(n))$  et  $g(n) = O(h(n))$  alors  $f(n) + g(n) = O(h(n))$ .
4. Produit : si  $f(n) = O(F(n))$  et  $g(n) = O(G(n))$  alors  $f(n) \times g(n) = O(F(n) \times G(n))$  ; en particulier  $C \times f(n) = O(F(n))$  pour toute constante  $C$ , car toute fonction constante est  $C = O(1)$ .

On a les mêmes propriétés avec  $\Omega$  et  $\Theta$ .

## 5. Exercices corrigés

### 5.1. Exercices

#### Exercice 1 :

Soit le programme Pascal suivant :

```

program Complexite;
var
  T : array [1..5] of real;
  i : integer;
  s : real;
begin
  for i := 1 to 5 do T[i] := i*i;
  s := T[1] + T[5];

```

```

for i := 1 to 5 do writeln(T[i]);
writeln(s);
end.

```

Sachant qu'un élément de type Integer occupe 2 octets en mémoire et un élément de type Real occupe 6 octets en mémoire :

1. Quel est le coût en espace du programme Pascal précédent ?
2. Proposez une optimisation en espace de ce programme.
3. Quel est le coût en espace du programme Pascal après l'optimisation ?

### Exercice 2 :

Soit l'extrait de l'algorithme de recherche séquentielle (technique de Flag) dans un tableau :

Lire(x) ;

b ← FAUX ;

i ← 1 ;

Tant que (i ≤ n) ET NON b Faire début

Si T[i] = x Alors b ← VRAI ;

i ← i + 1 ;

fin ;

1. Déterminer  $C_{max}(n)$ ,  $C_{min}(n)$  et  $C_{moy}(n)$ , sachant que la valeur recherchée se trouve obligatoirement une seule fois dans le tableau.
2. Quel est l'ordre de grandeur de chacun de ces cas ?

### Exercice 3 :

Même question pour un extrait de l'algorithme de tri par sélection d'un tableau en ordre croissant. L'extrait étant introduit comme suit :

Pour i ← 1 à n Faire Lire(T[i]) ;

Pour i ← 1 à n-1 Faire

Pour j ← i+1 à n Faire Si (T[i] > T[j]) Alors début

x ← T[i] ;

T[i] ← T[j] ;

T[j] ← x ;

fin ;

Pour i ← 1 à n Faire Ecrire(T[i]) ;

### Exercice 4 :

Soit l'algorithme suivant :

Algorithme Affichage;

Variables

n : entier;

Procédure AF1 (x : entier);

Variables i, k : entier;

```

Début
  Lire(k);
  Pour i ← 1 à x Faire
    Si i ≤ k Alors Ecrire(i);
  Fin;
Procédure AF2 (y : entier);
  Variables j, p : entier;
  Début
    Pour j ← 1 à y Faire
      Pour p ← 1 à y Faire AF1(y);
  Fin;
Début
  Lire(n);
  AF2(n);
Fin.
1. Déterminer  $C_{max}(n)$ ,  $C_{min}(n)$  et  $C_{moy}(n)$ .
2. L'algorithme ci-dessus est-t-il efficace ?

```

### Exercice 5 :

Déterminez l'ordre de complexité de la procédure de calcul de la factorielle dans sa forme itérative, ensuite dans sa forme récursive, présentées toutes les deux comme suit :

#### Forme itérative :

```

Fonction facto ( n : entier ) : entier;
Variables i, f : entier;
début
  f ← 1 ;
  Si (n > 0) Alors
    Pour i ← 1 à n Faire f ← f * i ;
  facto ← f ;

```

Fin;

#### Forme récursive :

```

Fonction facto(n:entier) : entier;
Début
  Si n ≤ 1 Alors facto ← 1
  Sinon facto ← n * fact(n-1);

```

Fin;

### Exercice 6 :

Déterminez l'ordre de complexité de la procédure de calcul de la suite de fibonacci dans sa forme récursive présentée comme suit :

```

Fonction fibo (n : entier) : entier ;
début
  Si (n = 1) OU (n = 2) Alors fibo ← 1
    Sinon fibo ← fibo(n-1) + fibo(n-2) ;
Fin;

```

### Exercice 7 :

Déterminez l'ordre de complexité de la procédure suivante qui est une solution du problème de tour de Hanoï :

```

procedure hanoi( n: integer; start, aux, finish : char) ;
begin
  if (n<>0) then
    begin
      hanoi(n-1, start, finish, aux);
      writeln('MOVE ', start, ' ', finish);
      hanoi(n-1, aux, start, finish);
    end;
  end;
end;

```

## 5.2. Corrigées

### Solution 1 :

On note le coût en espace par CE :

$$CE = 5 * 6 + 2 + 6 = 38 \text{ octets.}$$

Le programme optimisé est le suivant :

```

program Complexite;
var
  T : array [1..5] of integer;
  i : integer;
begin
  for i := 1 to 5 do T[i] := i*i;
  for i := 1 to 5 do writeln(T[i]);
  writeln(T[1] + T[5]);
end.

```

Dans ce cas  $CE = 5 * 2 + 2 = 12$  octets.

### Solution 2 :

Pour  $C_{max}(n)$ , quand la valeur recherchée est la dernière dans le tableau,  $n$  comparaisons sont nécessaires.

$$C_{max}(n) = 1 + 1 + 1 + n + n + 1 + n + 1 = 3n + 5 = O(n)$$

Pour  $C_{min}(n)$ , quand la valeur recherchée est la première dans le tableau, une seule comparaison est nécessaire.

$$C_{min}(n) = 8 = O(1)$$

Pour  $C_{moy}(n)$ , prenons l'hypothèse suivante : on trouve la valeur recherchée après avoir parcouru la moitié du tableau.

$$C_{moy}(n) = 3/2 n + 5 = O(n)$$

### Solution 3 :

Pour  $C_{max}(n)$ , quand le tableau est déjà ordonné en ordre décroissant, la condition ( $T[i] > T[j]$ ) est toujours VRAI.

$$\begin{aligned} C_{max}(n) &= n + ((n-1) + (n-2) + \dots + 2 + 1) * 4 + n \\ &= 2n + (n * (n-1)/2) * 4 \\ &= 2n + 2n^2 - 2n \\ &= 2n^2 = O(n^2) \end{aligned}$$

Pour  $C_{min}(n)$ , quand le tableau est déjà ordonné en ordre croissant, la condition ( $T[i] > T[j]$ ) est toujours FAUX.

$$\begin{aligned} C_{min}(n) &= n + ((n-1) + (n-2) + \dots + 2 + 1) * 1 + n \\ &= 2n + (n * (n-1)/2) \\ &= 3/2 n + 1/2 n^2 \\ &= O(n^2) \end{aligned}$$

Pour  $C_{moy}(n)$ , prenons l'hypothèse suivante : la moitié du tableau est déjà ordonnée en ordre croissant.

$$\begin{aligned} C_{moy}(n) &= n + ((n-1) + (n-2) + \dots + 2 + 1) + ((n/2-1) + (n/2-2) + \dots + 2 + 1) * 3 + n \\ &= 2n + (n * (n-1)/2) + (n/2 * (n/2-1)/2) * 3 \\ &= 3/4 n + 7/4 n^2 = O(n^2) \end{aligned}$$

### Solution 4 :

1- Le corps de l'algorithme peut être réécrit comme suit :

Lire(n);

Pour j ← 1 à n Faire

    Pour p ← 1 à n Faire début

        Lire(k);

        Pour i ← 1 à n Faire

            Si i <= k Alors Ecrire(i);

    Fin;

Ainsi, pour  $C_{max}$ , on suppose que le test ( $i \leq k$ ) est toujours VRAI, i.e. k est supérieure ou égale à n.

$$\begin{aligned} C_{max}(n) &= 1 + 2n + 2n^2 + n^2 + 2n^3 + n^3 + n^3. \\ &= 4n^3 + 3n^2 + 2n + 1. \end{aligned}$$

Pour  $C_{min}$ , on suppose que le test ( $i \leq k$ ) est toujours FAUX, i.e. k est négative ou nulle.

$$\begin{aligned} C_{min}(n) &= 1 + 2n + 2n^2 + n^2 + 2n^3 + n^3. \\ &= 3n^3 + 3n^2 + 2n + 1. \end{aligned}$$

Pour  $C_{moy}$ , on suppose que le test ( $i \leq k$ ) est VRAI pour la moitié des cas, i.e.  $k = n/2$ .

$$\begin{aligned} C_{moy}(n) &= 1 + 2n + 2n^2 + n^2 + 2n^3 + n^3 + n^3/2. \\ &= 7/2 n^3 + 3n^2 + 2n + 1. \end{aligned}$$

### 2- Efficacité?

Nous avons  $C(n) = C_{max}(n) = 4n^3 + 3n^2 + 2n + 1 = O(n^3)$ .

=> La complexité de l'algorithme est polynomiale avec  $p=3$ , ou tout simplement, elle est cubique => l'algorithme est efficace, i.e. rapide en pratique, même pour n de grande taille.

**Solution 5 :**

Pour la forme itérative :

$$\begin{aligned} C(n) &= 1 + 1 + 2n + n + 1 \\ &= 3 + 3n = O(n) \end{aligned}$$

La complexité est linéaire :  $O(n)$ .

Pour la forme récursive :

$$\begin{aligned} C(n) &= 1 + C(n-1) \\ &= 1 + (1 + C(n-2)) \\ &= 2 + C(n-2) \\ &= 2 + (1 + C(n-3)) \\ &= 3 + C(n-3) \end{aligned}$$

...

$$\begin{aligned} &= n-1 + C(n-(n-1)) \\ &= n-1 + C(1) \\ &= n-1+2 \\ &= n+1 = O(n) \end{aligned}$$

La complexité est linéaire :  $O(n)$ .

**Solution 6 :**

$$\begin{aligned} C(n) &= 1 + C(n-1) + C(n-2) \\ &= 1 + (1 + C(n-2)) + (1 + C(n-3)) \\ &= 1 + 2(1) + C(n-2) + C(n-3) \\ &= 1 + 2(1) + (1 + C(n-3)) + (1 + C(n-4)) \\ &= 1 + 2(2) + C(n-3) + C(n-4) \\ &\dots \\ &= 1 + 2(n-2) + C(n-(n-1)) + C(n-n) \\ &= 1 + 2(n-2) + C(1) + C(0) \\ &= 3 + 2(n-2) \\ &= 2n - 1 = O(n) \end{aligned}$$

La complexité est linéaire :  $O(n)$ .

**Solution 7 :**

$$\begin{aligned} C(n) &= 2 * C(n-1) + 1 \\ &= 2 * (2 * C(n-2) + 1) + 1 \\ &= 4 * C(n-2) + 3 \\ &= 8 * C(n-3) + 7 \\ &= 2^3 * C(n-3) + (2^3 - 1) \\ &= \dots \\ &= 2^k * C(n-k) + (2^k - 1) \\ &= \dots \\ &= 2^n * C(0) + (2^n - 1) \\ &= 2^n - 1 \end{aligned}$$

$$C(n) = 2^n - 1$$

La complexité est exponentielle, de l'ordre de  $O(2^n)$ .

## Chapitre 8 : Les types définis par l'utilisateur

### 1. Introduction

Dans les chapitres précédents, nous avons vu des types simples (entier, réel, caractère, booléen) et des types structurés (tableaux et chaînes de caractères). Ces types sont prédéfinis, c.-à-d. qu'ils existent dans les langages de programmation.

Tous les langages de programmation offrent à l'utilisateur la possibilité de définir de nouveaux types de données plus sophistiqués, permettant d'imaginer des traitements à la fois plus performants et plus souples. Dans ce qui suit, on va voir des types simples définis par l'utilisateur (énuméré et intervalle) et des types structurés définis par l'utilisateur (ensemble et enregistrement).

En langage Pascal, les types définis sont introduits par le mot clé TYPE dans la partie déclaration d'un programme.

### 2. Types simples définis par l'utilisateur

Pascal permet d'utiliser des types simples prédéfinis (entier, réel, caractère et booléen). Ce langage donne aussi la possibilité au programmeur de définir d'autres types simples.

#### 2.1. Le type énuméré

Le type énuméré permet de citer explicitement les valeurs que peut prendre une variable ; on ne peut affecter à cette variable aucune autre valeur que celles prévues dans l'énumération. Les valeurs du type énuméré sont symbolisées par des identificateurs qui sont en fait des identificateurs de constantes. Bien sûr, il est interdit de déclarer plusieurs fois le même identificateur pour éviter l'ambiguïté.

En règle général, on définit un nouveau type de données correspondant à cette énumération, et on attribue un nom à cette énumération. Ensuite, il devient possible de déclarer des variables de ce nouveau type ainsi créé.

#### Exemple :

Type

```
couleur= (vert, noir, blanc, jaune) ; { On crée le type couleur }
```

var

```
c,d : couleur ; { On déclare des variables de type couleur }
```

Dans cet exemple, on vient de créer le type couleur, puis on a déclaré les variables c et d de ce type. Une variable de type couleur ne peut prendre une valeur autre que celles énumérées (constantes énumérées) : vert, noir, blanc, jaune.

On peut mettre directement : var c,d : (vert, noir, blanc, jaune) ;

Attention ! Il ne s'agit pas de créer une variable de type chaîne de caractères couleur qui peut être égale à 'vert', 'noir', 'blanc' ou 'jaune', mais c'est plutôt un nouveau type de variables dont on définit les éléments possibles.

La déclaration précédente pouvait être substituée par une autre basée sur les constantes :

```
const vert=0 ; noir = 1 ; blanc = 2 ; jaune = 3 ;
```

```
var c,d : integer ; { c et d prennent uniquement les valeurs 0, 1, 2 et 3 }
```

Mais cette solution est lourde à manipuler.

Une variable de type énuméré ne peut être ni lue ni écrite par les instructions read ou write (puisque'il ne s'agit pas d'une chaîne de caractères), mais on peut manipuler cette variable de différentes manières :

- L'affectation qui permet d'affecter à une variable de type énuméré une valeur désignée symboliquement par l'un des identificateur énuméré dans la liste de définition de ce type.

Par exemple : `c := vert ; d := blanc ;`

- Les valeurs d'un type énuméré sont liées par une relation d'ordre dépendant de la place de chaque identificateur dans la liste. Ce qui permet aussi d'utiliser des fonctions prédéfinies, telles que PRED(noir) qui donne vert, SUCC(noir) qui donne blanc et ORD(noir) qui donne 1.
- Cette relation d'ordre permet aussi de faire des comparaisons selon l'ordre de déclaration. Par exemple, l'expression logique (noir < vert) donne FALSE.
- On peut utiliser une variable de type énuméré dans une boucle. Par exemple : `for c := vert to jaune...`  
ou dans une instruction de sélection :

```
case c of  
  vert : ... ;  
  noir : ... ;  
  blanc : ... ;  
  jaune : ... ;  
end ;
```

On note que le type booléen peut être manipuler comme un type énuméré déclaré implicitement sous la forme : `Type boolean = (FALSE, TRUE) ;`

## 2.2. Le type intervalle

Le type intervalle est utilisé quand une variable prend ses valeurs dans un intervalle limité par une borne inférieure et une borne supérieure :



**Exemple :**

Type

niveau = 1..10 ;

couleur = (vert, noir, blanc, jaune) ;

var

x : niveau ; y : 'a'..'f' ; cl : vert..blanc ;

...

Cet exemple présente différentes façons de déclarer des variables de type intervalle. La variable x peut prendre les valeurs de 1 à 10, y peut prendre les valeurs de 'a' à 'f', et cl peut prendre les valeurs vert, noir ou blanc. Les variables x, y et cl ne peuvent en aucun cas contenir une valeur dépassant les bornes définies dans le type.

Les deux bornes doivent être de même type simple non réel avec (la borne inférieure  $\leq$  la borne supérieure) ; ces bornes font partie de l'intervalle. Les bornes peuvent être symboliques, comme c'est le cas pour la variable cl dans l'exemple précédent.

Les opérations applicables au type intervalle sont celles applicables à des variables de même type que les bornes.

Pascal permet de manipuler des types simples définis par le programmeur (énuméré, intervalle). Ce langage permet aussi de définir des types complexes (structurés) à base de ces types simples.

**3. Types structurés définis par l'utilisateur****3.1. Le type ensemble**

Un ensemble est une collection d'objets de même type. C'est un ensemble au sens mathématique sur lequel on peut réaliser les opérations classiques de réunion, d'intersection, de complémentation, d'inclusion, etc. Si on déclare une variable S un ensemble de type T, alors S peut prendre comme valeur un sous ensemble de T.

**Exemple :**

type

couleur = (vert, noir, blanc) ;

cl = set of couleur ;

var

cl1, cl2 : cl ;

begin

cl1 := [vert, blanc] ;

cl2 := [vert, noir] ;

...

Dans cet exemple, nous avons défini un type énuméré couleur et un type ensemble cl associé au type couleur. On dit que couleur est le type de base de cl. Ensuite, nous avons déclaré cl1 et cl2 deux variables de type ensemble cl. cl1 et cl2 peuvent prendre comme valeurs des sous-ensembles de cl.

On a pu mettre directement : `var cl1, cl2 : set of (vert, noir, blanc);`

Un type de base peut être un type énuméré, un type intervalle d'entiers ou de caractères, un type caractère, ou un type booléen. Par exemple : `var chiffre : set of 1..10;`

On peut affecter à une variable de type ensemble un ensemble de valeurs mises entre deux crochets [ et ] en utilisant le symbole d'affectation := comme c'est le cas dans l'exemple précédent pour les variables cl1 et cl2. L'écriture `cl1 := [];` indique que la variable cl1 reçoit l'ensemble vide, et l'écriture `cl1 := [vert..blanc];` indique que la variable cl1 reçoit l'ensemble des valeurs vert, noir et blanc.

On peut effectuer plusieurs opérations sur les ensembles :

- L'union, par exemple : `cl1+cl2` donne [vert, noir, blanc].
- La différence ou le complément, par exemple : `cl1-cl2` donne [blanc].
- L'intersection, par exemple : `cl1*cl2` donne [vert].
- Les tests booléens sont aussi possibles (=, <>, <=, >=), par exemple l'expression logique `(cl1=cl2)` retourne la valeur FALSE, car les deux ensembles ne sont pas égaux.
- On peut également tester l'appartenance d'un élément à un ensemble par l'opérateur IN, par exemple l'expression logique `(vert IN cl1)` retourne la valeur TRUE.

### 3.2. Le type enregistrement

La notion d'enregistrement ou de structure permet de regrouper un ensemble de données de différents types dans un même objet. Un enregistrement (structure) est un nouveau type défini par un identificateur possédant un certain nombre de variables.

Intuitivement, un enregistrement permet donc de regrouper dans une même structure un ensemble d'informations ayant entre elles un lien logique. Formellement, le type enregistrement est le produit cartésien d'un ensemble de types.

En LDA, un type enregistrement est décrit comme suit :

`Nom_enreg = enregistrement`

`Nom_var1 : type1;`

`Nom_var2 : type2;`

`....`

`Fin_enreg;`

Où Nom\_enreg est le nom de l'enregistrement défini par l'utilisateur. Nom\_var1, Nom\_var2... sont des variables membres (champs) de l'enregistrement (structure).

Une fois ce type défini, on peut déclarer des variables de ce type comme suit : Nom\_var : Nom\_enreg ; On accède à une information en précisant le nom de la variable de type enregistrement suivi d'une variable membre, généralement séparés par un point comme suit : Nom\_var.Nom\_var1 et Nom\_var.Nom\_var2.

**Exemple :**

Voyons l'exemple suivant exprimé en Pascal :

```
program personnel;
type
  personne = record
    nom : string[20] ;
    sit_familiale : (marie, celibataire, divorce) ;
    telephone : string[10] ;
  end ;
var
  employe, X : personne ;
begin
  employe.nom := 'Ali' ;
  employe.sit_familiale := celibataire ;
  readln(employe.telephone) ;
  writeln(employe.nom, ' ', employe.telephone) ;
  X := employe ;
end.
```

Dans cet exemple, nous avons défini un type enregistrement nommé personne et possédant les champs nom de type chaîne de caractères, sit\_familiale de type énuméré, et telephone de type chaîne de caractères. Ensuite, nous avons déclaré deux variables X et employe de type enregistrement déjà défini. L'espace mémoire réservé pour chacune des deux variables est égal à la somme des espaces réservés pour ses champs.

Les opérations de lecture, écriture, comparaison, etc. ne sont pas applicables directement sur un type enregistrement. Alors, les instructions read(employe); write(employe); et la comparaison (X<employe) ne sont pas acceptées dans le programme précédent. La seule opération possible pour manipuler directement une variable de type enregistrement sans passer par ses champs est l'affectation (eg. X :=

employe ;). Pour d'autres opérations, la variable employe peut être manipulée champ par champ en regroupant le nom de la variable avec le nom d'un seul champ à la fois, séparés par un point pour obtenir une seule composante (eg. employe.nom). La composante obtenue se comporte alors exactement comme toute autre variable de même type que celui du champ correspondant. Maintenant, on peut effectuer sur cette composante une affectation, une lecture, une écriture, une comparaison, etc.

L'exemple précédent peut être simplifié en utilisant la structure with...do... de la manière suivante :

```
program personnel;
type
  personne = record
    nom : string[20] ;
    sit_familiale : (marie, celibataire, divorce) ;
    telephone : string[10] ;
  end ;
var
  employe, X : personne ;
begin
  with employe do begin
    nom := 'Ali' ;
    sit_familiale := celibataire ;
    readln(telephone);
    writeln(nom, ' ', telephone) ;
  end;
  X := employe;
end.
```

Les variables X et employe peuvent être déclarées directement comme suit :

```
var
  X, employe : record
    nom : string[20] ;
    sit_familiale : (marie, celibataire, divorce) ;
    telephone : string[10] ;
  end ;
```

Dans un enregistrement, chaque champ doit avoir un nom différent. Mais pour des enregistrements différents, on peut réutiliser le même nom de champ avec un type différent ou identique, par exemple :

type

```
personne = record
  nom : string[20] ;
  sit_familiale : (marie, celibataire, divorce) ;
  telephone : string[10] ;
end ;
homme = record
  nom : string[20] ;
  sit_familiale, telephone : string[10] ;
end ;
```

Il est aussi possible de déclarer un tableau dont les éléments sont de type enregistrement, par exemple : `var employes : array [1..10] of personne ;`. L'accès au premier élément du tableau s'effectuera comme suit : `employes[1].nom := 'Ali' ; employes[1].sit_familiale := celibataire ; ...`

Il est conseillé d'écrire des fonctions ou des procédures réalisant les opérations usuelles sur le type enregistrement, car une fonction ou une procédure peut avoir une variable de type enregistrement comme paramètre.

## 4. Exercices corrigés

### 4.1. Exercices

#### Exercice 1 :

Ecrire un programme Pascal permettant de définir un type énuméré ayant comme valeurs les jours de semaine, ensuite de lire une variable entière et de dire si ça correspond à un jour férié dans la liste des valeurs énumérées ou non.

#### Exercice 2 :

Ecrire un programme Pascal permettant de définir un type énuméré ayant comme valeurs les couleurs d'un feu permettant la gestion de la circulation dans un carrefour, ensuite de lire une variable entière et d'afficher à quelle couleur cette valeur correspond.

#### Exercice 3 :

En utilisant le type intervalle, améliorez le programme précédent en se limitant uniquement sur les valeurs que peut prendre une valeur entière lue à partir du clavier.

#### Exercice 4 :

Ecrire un programme Pascal permettant de déclarer deux ensembles de caractères majuscules et minuscules, et de dire si un caractère, lu à partir du clavier, appartient au premier ou bien au deuxième ensemble.

**Exercice 5 :**

Ecrire un programme Pascal permettant de déterminer le nombre de voyelles dans un mot. Utilisez l'ensemble  $VL = \{a, e, u, o, i\}$ .

**Exercice 6 :**

Présentez en Pascal les types enregistrements suivants :

- Un nombre complexe est défini par une partie réelle et une partie imaginaire.
- Une date est composée d'un numéro de jour (1..31), d'un numéro de mois (1..12) et d'une année (strictement positive).
- Un stage est défini par un intitulé (chaîne de caractères), une date de début et une date de fin (deux dates), et un nombre de places (entier).
- Une identité décrivant le nom, le prénom et la date naissance.
- Une fiche bibliographique est définie par le titre du livre (chaîne), les auteurs (chacun est identifié par un nom, un prénom et une date de naissance), la date de parution, l'éditeur (nom, prénom et date naissance) et le numéro ISBN (chaîne).

Comment peut-on affecter une valeur à :

- Une partie réelle d'une variable X de type complexe.
- Jour d'une variable Y de type date.
- Mois de la date de début d'une variable Z de type stage.
- Année de la date de naissance d'une variable G de type identité.
- Jour de la date de naissance du premier auteur d'une variable H de type fiche bibliographique.

**Exercice 7 :**

Ecrire un programme Pascal permettant de définir un type enregistrement dit pere possédant les champs nom de type chaîne de caractères, date\_nais de type chaîne de caractères, nbr\_enf de type entier, et liste\_enf de type chaîne de caractères. Le programme doit permettre de lire et d'afficher les informations d'une variable pere1 de type enregistrement déjà défini.

**Exercice 8 :**

Reprendre l'exercice 7, mais cette fois-ci en modifiant la structure pere comme suit : le champ date\_nais doit être à son tour défini comme un type enregistrement possédant les champs : jour, mois et annee. Le champ liste\_enf doit être déclaré comme un tableau de chaînes de caractères.

**Exercice 9 :**

Reprendre l'exercice 7, mais cette fois-ci en utilisant une procédure qui reçoit en paramètre une variable de type enregistrement (pere), puis elle l'affiche.

**Exercice 10 :**

En utilisant la structure pere définie dans l'exercice 7, essayez cette fois-ci de déclarer dans un programme Pascal deux variables pere1 et pere2 de type pere, de lire uniquement le nombre d'enfants des deux pères, et de calculer et d'afficher le nombre total des enfants.

**Exercice 11 :**

Ecrire un programme Pascal permettant de déclarer un tableau de cinq éléments de type enregistrement pere défini dans l'exercice 8, de lire ses éléments, ensuite de les afficher.

**Exercice 12 :**

On considère dix candidats inscrits à une formation diplômante. Chaque candidat va obtenir une note pour cette formation. Ecrire le programme Pascal permettant d'afficher la liste des candidats dont la note est supérieure ou égale à la moyenne des notes de tous les candidats (les noms des candidats et les notes étant lus à partir du clavier). Utilisez un tableau dont les éléments sont de type enregistrement. Le type enregistrement doit contenir un champ indiquant le nom du candidat et un autre indiquant la note du candidat.

**Exercice 13 :**

Simplifiez le programme de l'exercice précédent en utilisant la structure with...do....

**4.2. Corrigés**

**Solution 1 :**

```
program type_enum ;
type
  jour = (Samedi, Dimanche, Lundi, Mardi, Mercredi, Jeudi, Vendredi) ;
var
  num : integer ;
  j : jour ;
begin
  writeln('Entrez un numéro de jour entre 0 et 6 :') ;
  readln(num) ;
  j := jour(num) ;
  if ((j = jeudi) OR (j = vendredi)) then writeln('C'est un jour férié.')
    else writeln('Ce n'est pas un jour férié.') ;
end.
```

**Solution 2 :**

```
program Circulation ;
type
  feu = (vert, orange, rouge) ;
```

```

var
  num : integer ;
  j : feu;
begin
  writeln('Entrez un numéro de feu entre 0 et 2 :');
  readln(num) ;
  j := feu(num) ;
  case j of
    vert : writeln('feu vert');
    orange : writeln('feu orange');
    rouge : writeln('feu rouge');
  end;
end.

```

*Ou bien :*

```

program Circulation ;
type
  feu = (vert, orange, rouge) ;
var
  num : integer ;
  j : feu;
begin
  writeln('Entrez un numéro de feu entre 0 et 2 :');
  readln(num) ;
  case num of
    ord(vert) : writeln('feu vert');
    ord(orange) : writeln('feu orange');
    ord(rouge) : writeln('feu rouge');
  end;
end.

```

**Solution 3 :**

```

program Circulation ;
type feu = (vert, orange, rouge) ;
var
  num : 0..2 ;
  j : feu;
begin
  writeln('Entrez un numéro de feu entre 0 et 2 :');
  readln(num) ;
  j := feu(num) ;

```



```

case j of
  vert : writeln('feu vert');
  orange : writeln('feu orange');
  rouge : writeln('feu rouge');
end;
end.

```

#### **Solution 4 :**

```

program ensemble ;
type
  ens_car = set of char ;
var
  maj, min : ens_car ;
  c : char;
begin
  maj := ['A'..'Z'] ;
  min := ['a'..'z'] ;
  writeln('Tapez une lettre :)') ;
  readln(c) ;
  if (c IN maj) then writeln('C'est une lettre écrite en majuscule.')
  else if (c IN min) then writeln('C'est une lettre écrite en minuscule.')
  else writeln('Il ne s'agit pas d'une lettre.');
```

end.

#### **Solution 5 :**

```

program voyelles ;
var
  VL : set of char ;
  mot : string[20] ;
  x, t, nb : integer ;
begin
  VL := ['a', 'A', 'e', 'E', 'i', 'I', 'o', 'O', 'u', 'U'] ;

  writeln('Entrez un mot de 20 caractères maximum :)') ;
  readln(mot) ;
  t := length(mot) ;
  nb := 0 ;
  for x := 1 to t do if (mot[x] IN VL) then nb := nb + 1;
  writeln('Nombre de voyelles dans le mot est : ', nb) ;
end.

```

**Solution 6 :**

Définition des types enregistrements :

```
type
  nombre_complexe = record
    partie_reelle : real ;
    partie_imaginaire : real ;
  end;
  date = record
    jour : 1..31;
    mois : 1..12;
    annee : integer;
  end;
  stage = record
    intitule : string[50];
    date_debut, date_fin : date;
    nbr_place : integer;
  end;
  identite = record
    nom : string[15];
    prenom : string[15];
    date_nais : date;
  end;
  fiche_biblio = record
    titre : string[50];
    auteurs : array [1..10] of identite ;
    date_parution : date ;
    editeur : identite;
    num_ISBN : string[10];
  end ;
```

Accès aux champs des variables :

```
var
  X : nombre_complexe ;
  Y : date ;
  Z : stage ;
  G : identite ;
  H : fiche_biblio ;
begin
  X.partie_reelle := ... ;
  Y.jour := ... ;
```

```
Z.date_debut.mois := ...;
G.date_nais.annee := ... ;
H.auteurs[1].date_nais.jour := ... ;
end ;
```

**Solution 7 :**

```
program enregistrement ;
type
  pere = record
    nom : string [20];
    date_nais : string[10];
    nbr_enf : integer;
    liste_enf : string[100];
  end;
var
  pere1 : pere ;
begin
  writeln('Donnez le nom du père :) ' ;
  readln(pere1.nom) ;
  writeln('Donnez la date de naissance du père :) ' ;
  readln(pere1.date_nais) ;
  writeln('Donnez le nombre d"enfants du père :) ' ;
  readln(pere1.nbr_enf) ;
  writeln('Donnez la liste des enfants du père :) ' ;
  readln(pere1.liste_enf) ;
  writeln('Voici les informations concernant ce père :) ' ;
  writeln('Nom : ', pere1.nom, ', Date naissance : ', pere1.date_nais,
    ', Nombre d"enfants : ', pere1.nbr_enf);
  writeln('Liste des enfants : ', pere1.liste_enf) ;
end.
```

**Solution 8 :**

```
program enregistrement ;
type
  date = record
    jour : 1..31;
    mois : 1..12;
    annee : integer;
  end;
  pere = record
    nom : string [20];
```

```
    date_nais : date;
    nbr_enf : integer;
    liste_enf : array[1..20] of string[20];
end;
var
    pere1 : pere ;
    i : integer ;
begin
    writeln('Donnez le nom du père :') ;
    readln(pere1.nom) ;
    writeln('Donnez la date de naissance du père :') ;
    write('Le jour : ') ;
    readln(pere1.date_nais.jour) ;
    write('Le mois : ') ;
    readln(pere1.date_nais.mois) ;
    write('L'année : ') ;
    readln(pere1.date_nais.annee) ;
    writeln('Donnez le nombre d'enfants du père :') ;
    readln(pere1.nbr_enf) ;
    for i := 1 to pere1.nbr_enf do begin
        writeln('Donnez le nom de l'enfant num : ', i) ;
        readln(pere1.liste_enf[i]) ;
    end ;
    writeln('Voici les informations concernant ce père :') ;
    writeln('Nom : ', pere1.nom, ', Date naissance : ', pere1.date_nais.jour,
        ' ', pere1.date_nais.mois, ' ', pere1.date_nais.annee);
    writeln('Nombre d'enfants : ', pere1.nbr_enf, ', Liste des enfants : ');
    for i := 1 to pere1.nbr_enf do writeln(' ', pere1.liste_enf[i]) ;
end.
```

**Solution 9 :**

```
program enregistrement ;
type
    pere = record
        nom : string [20];
        date_nais : string[10];
        nbr_enf : integer;
        liste_enf : string[100];
    end;
var
```

```
pere1 : pere ;
procedure affiche(X : pere) ;
begin
  writeln('Voici les informations concernant ce père :)') ;
  write('Nom : ', X.nom, ', Date naissance : ', X.date_nais,
    ', Nombre d'enfants : ', X.nbr_enf);
  writeln(', Liste des enfants : ', X.liste_enf) ;
end ;
begin
  writeln('Donnez le nom du père :)') ;
  readln(pere1.nom) ;
  writeln('Donnez la date de naissance du père :)') ;
  readln(pere1.date_nais) ;
  writeln('Donnez le nombre d'enfants du père :)') ;
  readln(pere1.nbr_enf) ;
  writeln('Donnez la liste des enfants du père :)') ;
  readln(pere1.liste_enf) ;
  affiche(pere1) ;
end.
```

**Solution 10 :**

```
program enregistrement ;
type
  pere = record
    nom : string [20];
    date_nais : string[10];
    nbr_enf : integer;
    liste_enf : string[100];
  end;
var
  pere1, pere2 : pere ;
  somme : integer ;
begin
  writeln('Donnez le nombre d'enfants du premier père :)') ;
  readln(pere1.nbr_enf) ;
  writeln('Donnez le nombre d'enfants du deuxième père :)') ;
  readln(pere2.nbr_enf) ;
  somme := pere1.nbr_enf + pere2.nbr_enf ;
  writeln('Le nombre total des enfants est égal à ', somme) ;
end.
```

**Solution 11 :**

```
program enregistrement ;
type
  date = record
    jour : 1..31;
    mois : 1..12;
    annee : integer;
  end;
  pere = record
    nom : string [20];
    date_nais : date;
    nbr_enf : integer;
    liste_enf : array[1..20] of string[20];
  end;
var
  peres : array [1..5] of pere ;
  i,j : integer ;
begin
  for j := 1 to 5 do begin
    writeln('Donnez le nom du père num ', j) ;
    readln(peres[j].nom) ;
    writeln('Donnez la date de naissance du père num ', j) ;
    write('Le jour : ' ) ;
    readln(peres[j].date_nais.jour) ;
    write('Le mois : ' ) ;
    readln(peres[j].date_nais.mois) ;
    write('L'année : ' ) ;
    readln(peres[j].date_nais.annee) ;
    writeln('Donnez le nombre d'enfants du père num ', j) ;
    readln(peres[j].nbr_enf) ;
    for i := 1 to peres[j].nbr_enf do begin
      writeln('Donnez le nom de l'enfant num : ', i) ;
      readln(peres[j].liste_enf[i]) ;
    end ;
  end ;
  for j := 1 to 5 do begin
    writeln('Voici les informations concernant le père num ', j) ;
    writeln('Nom : ', peres[j].nom) ;
    writeln('La date de naissance : ' ) ;
```

```
writeln('Le jour : ', peres[j].date_nais.jour) ;
writeln('Le mois : ', peres[j].date_nais.mois) ;
writeln('L'année : ', peres[j].date_nais.annee) ;
writeln('Le nombre d'enfants : ', peres[j].nbr_enf) ;
writeln('La liste des enfants :') ;
for i := 1 to peres[j].nbr_enf do
    writeln('Le nom de l'enfant num : ', i, ' : ', peres[j].liste_enf[i]) ;
end ;
end.
```

### **Solution 12 :**

```
program Liste_candidats_sup ;
type
    candidat = record
        nom : string[20];
        note : real;
    end;
var
    liste : array [1..10] of candidat;
    somme, moy : real;
    i : integer;
begin
    somme := 0;
    (*Saisir les noms et les notes des candidats*)
    for i := 1 to 10 do begin
        writeln('Saisir le nom du candidat n° : ', i, ' : ');
        readln(liste[i].nom) ;
        writeln('Saisir la note du candidat ', liste[i].nom, ' : ');
        readln(liste[i].note) ;
        somme := somme + liste[i].note ;
    end ;
    (*Afficher la moyenne des notes des candidats*)
    moy := somme / 10 ;
    writeln('La moyenne des notes des candidats est égale à ', moy) ;
    (*Afficher les candidats dont la note est >= à la moyenne des notes de tous les candidats *)
    writeln('Les candidats dont la note est >= à la moyenne :') ;
    for i := 1 to 10 do
        if (liste[i].note >= moy) then
            writeln(liste[i].nom, ' avec une note de ', liste[i].note) ;
    end.
```

**Solution 13 :**

```
program Liste_candidats_sup ;
type
  candidat = record
    nom : string[20];
    note : real;
  end;
var
  liste : array [1..10] of candidat;
  somme, moy : real;
  i : integer;
begin
  somme := 0;
  (*Saisir les noms et les notes des candidats*)
  for i := 1 to 10 do with liste[i] do begin
    writeln('Saisir le nom du candidat n° : ', i, ' : ');
    readln(nom);
    writeln('Saisir la note du candidat ', liste[i].nom, ' : ');
    readln(note);
    somme := somme + note;
  end;
  (*Afficher la moyenne des notes des candidats*)
  moy := somme / 10;
  writeln('La moyenne des notes des candidats est égale à ', moy);
  (*Afficher les candidats dont la note est >= à la moyenne des notes de tous les candidats *)
  writeln('Les candidats dont la note est >= à la moyenne :');
  for i := 1 to 10 do with liste[i] do
    if (note >= moy) then writeln(nom, ' avec une note de ', note);
  end.
end.
```



## Chapitre 9 : Les fichiers

### 1. Introduction

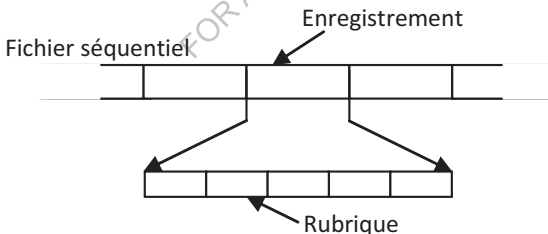
Toutes les structures de données que nous avons utilisées jusqu'à maintenant étaient stockées en mémoire vive (la RAM), c'est-à-dire dans une zone volatile de la machine.

En mémoire vive, la durée de vie d'une variable est égale au temps d'exécution du programme. Or, dans une application informatique, les données doivent être mémorisées plus longtemps que la durée d'un programme. D'où la nécessité d'utiliser les fichiers.

### 2. Les fichiers

Un fichier est une structure de données destinée à contenir des informations homogènes. Il permet de stocker des informations sur un support non volatil (disque dur , clé USB, etc.) pour une durée indéterminée.

Le fichier peut être considéré comme un tableau d'enregistrements stocké sur un disque. Chaque enregistrement contient une collection d'unités logiques d'informations, encore appelées rubriques ou champs. Une rubrique est la plus petite unité logique d'information ayant un sens en tant que tel. Souvent, les enregistrements ont la même structure. La taille d'une rubrique s'exprime en nombre de caractères, et peut être fixe ou variable.



Quand il s'agit d'un ensemble d'enregistrements, on parle de fichier typé, mais il existe d'autres types de fichiers, à savoir les fichiers non typés et les fichiers texte.

### 3. Types de fichiers

Pour utiliser un fichier dans un programme, il faudra l'identifier par une variable dont le type est en fonction de l'utilisation de ce fichier. Il existe trois types de fichiers :

#### 3.1. Les fichiers texte

Un fichier texte est un fichier séquentiel qui contient une suite de caractères regroupés en lignes. Chaque ligne se terminant par un

indicateur de fin de ligne : soit le retour au chariot (Cr) dont le code ASCII est 13, soit le saut de ligne (Lf) dont le code ASCII est 10.

Format général : Variables f : Texte ;

En Pascal : var f : text ;

### 3.2. Les fichiers typés

Un fichier typé est un fichier organisé de telle sorte que l'on puisse exploiter les données d'un enregistrement. Cela augmente la vitesse d'accès aux données du fichier. Mais le plus grand avantage, c'est que l'on obtient ainsi des fichiers parfaitement formatés, c'est-à-dire qu'on peut y lire et écrire directement des variables de type structuré qui contiennent plusieurs champs de données sans avoir à se soucier des divers champs qu'elles contiennent.

Format général : Variables f : Fichier de <Type> ;

En Pascal : var f : file of <Type> ;

#### Exemple :

Type

TPersonne = Enregistrement

Code : entier ;

Nom, Prénom : chaîne de caractères;

fin ;

Variables

personne : Fichier de TPersonne ;

f : Fichier d'entier ;

...

#### En Pascal :

type

TPersonne = record

Code : integer ;

Nom, Prenom : string ;

fin ;

var

personne : file of TPersonne ;

f : file of integer ;

...

Dans l'exemple précédent, nous avons déclaré deux fichiers typés *personne* et *f*. Chaque enregistrement du premier fichier est de type *TPersonne*. Chaque enregistrement du deuxième fichier correspond à un entier. La forme *fichier typé* est la plus favorisée pour la manipulation des fichiers.

### 3.3. Les fichiers non typés

Un fichier non typé est un fichier dont on ne connaît pas le contenu. On n'a aucune information sur la structure et le type des données. La lecture et l'écriture d'un tel fichier sont plus rapides. Il peut être par exemple un fichier son, une image, un programme exécutable, etc.

Format général : Variables f : Fichier ;

En Pascal : var f : file ;

### 4. Structure des enregistrements dans un fichier typé

Il existe plusieurs façons pour structurer les données en enregistrements au sein d'un fichier typé et de l'implémenter :

- Les enregistrements sont séparés par des délimiteurs.
- Les enregistrements sont de taille fixe.
- Chaque enregistrement est précédé par une zone indiquant sa taille.
- Les adresses des différents enregistrements sont contenues dans un deuxième fichier séparé.

#### Exemples :

Prenons le cas du carnet d'adresses, avec dedans le nom, le prénom, le téléphone et l'email. Les données, sur le fichier, peuvent être organisées en structures d'enregistrements comme suit :

##### **Structure n°1 :**

"Mohamed";"Ali";0555421564;med-ali@yahoo.fr  
"Guezal";"Hiba";0765691234;GHiba@gmail.com  
"Alioui";"Ahmed";0662289765;AliouiA@yahoo.fr  
"Berabeh";"Houcine";0551498752;HoucineBerabeh@aol.fr  
ou ainsi :

##### **Structure n°2 :**

Mohamed Ali 0555421564med-ali@yahoo.fr  
Guezal Hiba 0765691234GHiba@gmail.com  
Alioui Ahmed 0662289765AliouiA@yahoo.fr  
Berabeh Houcine 0551498752HoucineBerabeh@aol.fr

La structure n°1 est dite délimitée ; elle utilise un caractère spécial, appelé caractère de délimitation, qui permet de repérer quand finit un champ et quand commence le suivant. Ce caractère de délimitation doit être strictement interdit à l'intérieur de chaque champ, faute de quoi la structure devient proprement illisible.

La structure n°2, elle, est dite à champs de largeur fixe. Il n'y a pas de caractère de délimitation, mais on sait que les X premiers caractères de chaque ligne stockent le nom, les Y suivants le prénom, etc. Cela impose bien entendu de ne pas saisir un renseignement plus long que le champ prévu pour l'accueillir.

L'avantage de la structure n°1 est son faible encombrement en place mémoire ; il n'y a aucun espace perdu, et un fichier codé de cette manière occupe le minimum de place possible. Mais elle possède en revanche un inconvénient majeur, qui est la lenteur de la lecture. En effet, chaque fois que l'on récupère un enregistrement dans le fichier, il faut alors parcourir, un par un, tous les caractères pour repérer chaque occurrence du caractère de séparation avant de pouvoir découper cette ligne en différents champs.

La structure n°2, à l'inverse, gaspille de la place mémoire. Mais d'un autre côté, la récupération des différents champs est très rapide. Lorsqu'on récupère un enregistrement, il suffit de le découper en différentes chaînes de longueur prédéfinie, et le tour est joué.

A l'époque où la place mémoire coûtait cher, la structure délimitée était souvent privilégiée. Mais depuis bien des années, la quasi-totalité des logiciels et des programmeurs optent pour la structure en champs de largeur fixe. Aussi, nous ne travaillerons qu'avec des fichiers bâtis sur cette structure.

De même, il existe plusieurs façons d'implémenter la notion de rubrique dans un enregistrement de fichier :

- Chaque rubrique se termine par un séparateur.
- Chaque rubrique d'un enregistrement a une longueur fixe.
- Chaque rubrique débute par une zone indiquant sa taille.

**Remarque :**

Le problème de structuration des enregistrements se pose quand le programmeur désire gérer personnellement le stockage des enregistrements dans un fichier texte, et non pas un fichier typé. Dans un fichier typé, ce problème est géré systématiquement.

**5. L'organisation des enregistrements dans un fichier typé**

La manière dont on peut accéder à un fichier typé dépend de l'organisation des enregistrements à l'intérieur de ce fichier.

Si comparé à un tableau, il y a des différences importantes au niveau du nombre de composantes et des opérations d'accès aux composantes d'un fichier :

- Dans un tableau, le nombre de composantes est fixé par la définition du tableau, tandis qu'avec un fichier, le nombre est indéterminé a priori et peut varier au cours d'exécution du programme. Donc, il faudra toujours s'assurer que la capacité de stockage du support externe soit suffisante pour l'ensemble des composantes du fichier à enregistrer.

- Si dans un tableau l'accès aux composantes est déterminé par l'utilisation d'indices, avec un fichier, seulement une composante à la fois est directement accessible, mais on peut passer d'une composante à la suivante ou bien se repositionner sur la première.

Classiquement, on distingue trois grandes formes d'organisation des enregistrements dans un fichier :

- Organisation séquentielle.
- Organisation relative.
- Organisation indexée.

Dans la plupart des langages de programmation, l'organisation d'un fichier se décide au moment de sa création.

### **5.1. Organisation séquentielle**

Les enregistrements d'un fichier séquentiel sont stockés sur le support physique dans l'ordre dans lequel ils sont entrés. Il y a une correspondance entre l'ordre physique et l'ordre logique.

Chaque enregistrement (sauf le premier) possède un prédécesseur, et chaque enregistrement (sauf le dernier) possède un successeur.

Les supports qui supportent cette organisation sont par exemple la bande magnétique, les unités de disques, etc.

### **5.2. Organisation relative**

Un fichier en organisation relative se compose d'un certain nombre d'enregistrements de taille fixe. Ces enregistrements portent un numéro relatif au début du fichier, ce numéro allant de 1 (ou 0) à N. Chaque numéro représente l'emplacement de l'enregistrement relativement au début du fichier.

Les supports qui permettent ce genre d'organisation sont tous les supports où le mécanisme de lecture/écriture peut se positionner directement à un endroit précis du fichier, par exemple, unités de disques, etc.

### **5.3. Organisation indexée**

Dans cette organisation, chaque enregistrement est identifié par une clé, et chaque clé est associée à un numéro d'enregistrement qui renvoie à l'enregistrement correspondant.

Une clé est une rubrique du fichier qui identifie de façon unique un enregistrement, et chaque clé du fichier a une valeur unique.

Les supports qui permettent cette organisation sont les mêmes que ceux qui permettent l'organisation relative.

## **6. Les méthodes d'accès aux fichiers**

Les manières avec lesquelles on cherche, on lit, ou on écrit un enregistrement d'un fichier sont appelées les méthodes d'accès.

La méthode d'accès que l'on veut utiliser doit être spécifiée au moment de l'ouverture du fichier. Un même fichier peut être accédé par des méthodes différentes.

### 6.1. Accès séquentiel

L'accès séquentiel à un fichier signifie que les enregistrements du fichier sont traités en séquence. On ne peut donc accéder à un enregistrement qu'en ayant au préalable examiné celui qui le précède :

- Pour un fichier séquentiel, l'ordre physique correspond à la séquence de traitement.
- Pour un fichier en organisation relative, la séquence est celle des numéros d'enregistrements.
- Pour un fichier en organisation indexée, la séquence est l'ordre ascendant des valeurs des clés.

### 6.2. Accès direct

L'accès direct (ou aléatoire) à un fichier signifie qu'on peut accéder directement à l'enregistrement choisi, en précisant le numéro de cet enregistrement. Mais cela veut souvent dire une gestion fastidieuse des déplacements dans le fichier. Un fichier que l'on accède directement peut évidemment aussi être accédé séquentiellement.

### 6.3. Accès indexé

L'accès indexé signifie que les enregistrements du fichier sont accédés dans l'ordre déterminé par la valeur de la clé d'accès. Ce type d'accès combine la rapidité de l'accès direct et la simplicité de l'accès séquentiel. Il est particulièrement adapté au traitement de gros fichiers, comme les bases de données importantes.

#### Remarque :

En fait, tout fichier peut être utilisé avec l'un ou l'autre des trois types d'accès. Le choix du type d'accès n'est pas un choix qui concerne le fichier lui-même, mais uniquement la manière dont il va être traité par la machine. C'est donc dans le programme, et seulement dans le programme, que l'on choisit le type d'accès souhaité.

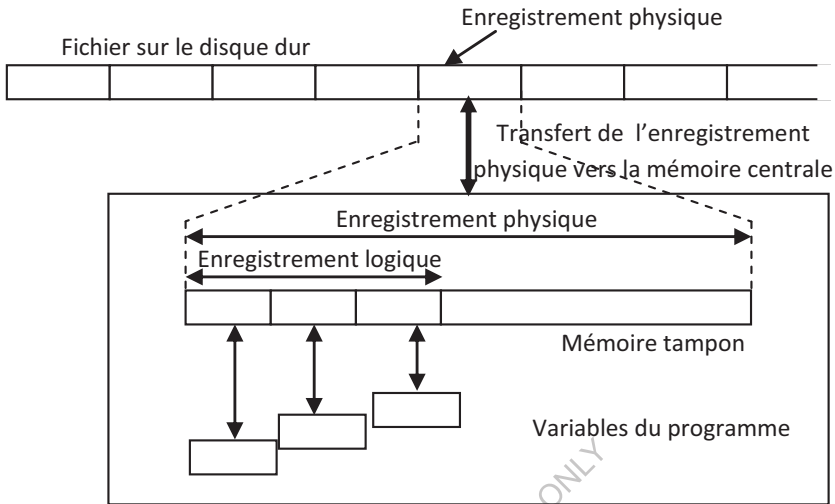
Pour conclure sur tout cela, voici un petit tableau récapitulatif :

Méthode d'accès Organisation	Séquentielle	Directe	Indexée
séquentielle	Oui	Non (*)	Non
relative	Oui	Oui	Non
indexée	Oui	Non (*)	Oui

(\*) Certains constructeurs de compilateurs offrent quand même la facilité de l'accès direct si le support externe le permet.

## 7. Manipulation des fichiers

Le schéma suivant représente les entrées/sorties entre la mémoire et le disque dur pendant la manipulation d'un fichier :



Plusieurs opérations peuvent être alors utilisées pour la manipulation des fichiers, parmi lesquelles on cite :

### 7.1. Assignment

Après avoir déclaré une variable (de type Fichier, Fichier de, ou Texte), il faut l'associer au chemin du fichier à traiter sur le disque dur. Donc, le fichier a deux noms : un nom logique (ou interne) en mémoire centrale, et un nom physique (ou externe) sur le disque. L'opération d'assignation consiste à associer le nom logique à un nom physique.

Format général : Assigner (<nom\_logique>, <nom\_physique>) ;

En Pascal : assign(<nom\_logique>, <nom\_physique>) ;

#### Exemple :

Type

TPersonne = Enregistrement

Code : entier ;

Nom, Prénom : chaîne de caractères ;

fin ;

Variables

personne : Fichier de TPersonne ;

f : Text ;

Début

Assigner(personne, 'c:\application\pr.dat') ;

Assigner(f, 'c:\essai.txt') ;

...

**En Pascal :**

```
type
  TPersonne = record
    Code : integer ;
    Nom, Prenom : string ;
  end ;
var
  personne : file of TPersonne ;
  f : text ;
begin
  assign(personne, 'c:\application\pr.dat') ;
  assign(f, 'c:\essai.txt') ;
  ...
```

**7.2. Ouverture**

Il est possible d'ouvrir un fichier en lecture sans écraser son contenu et positionner le pointeur au début du fichier.

Format général : OuvrirL(<nom\_logique>) ;

En Pascal : reset(<nom\_logique>) ;

Il est aussi possible d'ouvrir un fichier en écriture. Si le fichier existe déjà, alors il sera écrasé, et son contenu sera perdu.

Format général : OuvrirE(<nom\_logique>) ;

En Pascal : rewrite(<nom\_logique>) ;

**7.3. Lecture et écriture**

La lecture consiste à ranger le contenu du composant courant du fichier dans une variable de même type que les composantes du fichier. La variable est une structure de données qui dépend du type de fichier : enregistrement pour un fichier typé, chaîne de caractères pour un fichier texte, ou une variable quelconque pour un fichier non typé.

Format général : Lire(<nom\_logique>, <nom\_variable>) ;

En Pascal : read(<nom\_logique>, <nom\_variable>) ;

L'écriture permet d'écrire dans le fichier à la position courante, le contenu d'une variable de même type que les composantes du fichier.

Format général : Ecrire(<nom\_logique>, <nom\_variable>) ;

En Pascal : write(<nom\_logique>, <nom\_variable>) ;

**7.4. Accès aux enregistrements**

La lecture d'un fichier nécessite un test, car les lectures séquentielles (accès séquentiel) se terminent lorsque tous les enregistrements ont été lus un par un. Un indicateur booléen du système de gestion de fichier nous permet d'identifier le fait qu'il n'y a plus d'enregistrement à lire.



Format général : Fin\_Fichier(<nom\_logique>).

En Pascal : eof(<nom\_logique>).

La fonction Fin\_Fichier détecte la fin du fichier et retourne la valeur VRAI ou FAUX.

Il est aussi possible d'accéder directement à un enregistrement par son numéro d'ordre (accès direct) où le numéro est un nombre entier ou une expression donnant un résultat entier.

Format général : Rechercher(<nom\_logique>, <numéro>);

En Pascal : seek(<nom\_logique>, <numéro>);

### 7.5. Fermeture du fichier

L'ouverture d'un fichier se fait en lui attribuant un canal. On ne peut ouvrir qu'un seul fichier par canal, mais quel que soit le langage, on dispose toujours de plusieurs canaux. Et pour finir, une fois qu'on en a terminé avec un fichier, il ne faut pas oublier de le fermer. On libère ainsi le canal qu'il occupait.

A la fin du traitement du fichier, celui-ci doit être fermé.

Format général : Fermer(<nom\_logique>);

En Pascal : close(<nom\_logique>);

Le tableau suivant récapitule l'ensemble des fonctions et procédures Pascal prédéfinies pour la manipulation des fichiers :

Fonction/Procédure	Description
assign(<nom_log>, <nom_phys>)	Assigne le nom d'un fichier à la variable <nom_log>.
chdir(<nom_phys>)	Change le répertoire courant.
close(<nom_log>)	Ferme le fichier ouvert de la variable <nom_log>.
eof(<nom_log>)	Fonction booléenne indiquant la fin du fichier : elle prend deux valeurs (TRUE ou FALSE), selon que la fin du fichier soit atteinte ou non.
erase(<nom_log>)	Efface le fichier associé à la variable <nom_log>.
getdir(dsk, <nom_phys>)	Revoie le nom du répertoire courant sur l'unité du disque spécifiée.
ioresult	Fonction entière renvoyant l'état de la dernière opération d'E/S. Avec l'option de compilation {\$I-}, précédant une opération, la fonction IOResult renverra zéro si cette opération s'est bien déroulée,

	ou une valeur différente de zéro dans le cas contraire. La commande <code>{!-}</code> interrompt la détection d'erreur d'entrée/sortie par Turbo Pascal, et <code>{!+}</code> la réactive.
<code>mkdir(&lt;nom_phys&gt;)</code>	Crée un sous répertoire.
<code>rename(&lt;nom_log&gt;)</code>	Permet de renommer le fichier associé à <code>&lt;nom_log&gt;</code> .
<code>reset(&lt;nom_log&gt;)</code>	Ouvre un fichier existant.
<code>rewrite(&lt;nom_log&gt;)</code>	Crée puis ouvre un nouveau fichier. Si celui-ci existe, il sera supprimé.
<code>rmdir(&lt;nom_phys&gt;)</code>	Supprime un répertoire.

Le tableau suivant récapitule l'ensemble des fonctions et procédures Pascal prédéfinies pour le traitement des fichiers texte :

Fonction/Procédure	Description
<code>append(&lt;nom_log&gt;)</code>	Ouvre un fichier existant en mode ajout de lignes.
<code>eoln(&lt;nom_log&gt;)</code>	Fonction booléenne testant la fin de ligne d'un fichier.
<code>plush(&lt;nom_log&gt;)</code>	Vide le buffer associé au fichier vers le disque (écriture physique sur le disque).
<code>read(&lt;nom_log&gt;, &lt;nom_var&gt;)</code>	Lit une ou plusieurs valeurs à partir d'un fichier texte et les range dans les variables <code>&lt;nom_var&gt;...</code>
<code>readln(&lt;nom_log&gt;, &lt;nom_var&gt;)</code>	Même effet que <code>read</code> , mais en plus, elle va à la ligne suivante dans le fichier.
<code>seekof(&lt;nom_log&gt;)</code>	Fonction logique renvoyant l'état de la fin du fichier.
<code>seekeol(&lt;nom_log&gt;)</code>	Fonction logique renvoyant l'état de la fin de ligne d'un fichier texte.
<code>settextbuf(&lt;nom_log&gt;, &lt;tamp&gt;, &lt;t&gt;)</code>	Assigne un tampon <code>&lt;tamp&gt;</code> d'une certaine taille <code>&lt;t&gt;</code> à un fichier texte.
<code>write(&lt;nom_log&gt;, &lt;nom_var&gt;)</code>	Ecrit une ou plusieurs variables dans un fichier texte.
<code>writeln(&lt;nom_log&gt;, &lt;nom_var&gt;)</code>	Même effet que <code>write</code> , mais

	ajoute en plus l'indicateur de fin de ligne dans le fichier.
--	--

Le tableau suivant récapitule l'ensemble des fonctions et procédures Pascal prédéfinies pour le traitement des fichiers typés :

Fonction/Procédure	Description
filepos(<nom_log>)	Fonction de type longint renvoyant la position courante dans le fichier.
filesize(<nom_log>)	Fonction longint renvoyant la taille actuelle du fichier.
seek(<nom_log>,<num>)	<num> est une variable de type longint, et correspond au numéro d'enregistrement dans le fichier (le premier enregistrement étant 0). Cette procédure déplace le pointeur vers ce numéro.
read(<nom_log>,<enr>)	Permet de lire à partir du fichier à la position référencée par seek, l'enregistrement <enr>. Le pointeur se déplace d'un enregistrement après l'opération de lecture.
truncate(<nom_log>)	Cette procédure coupe le fichier à la position courante du pointeur. Ce qui était après le pointeur est effacé. eof(<nom_log>) devient TRUE.
write(<nom_log>,<enr>)	Permet d'écrire, à la position courante du pointeur, l'enregistrement <enr> défini. Cette opération déplace le pointeur d'un enregistrement après écriture.

**Exemple 1 :**

Le programme Pascal suivant permet la création d'un fichier typé contenant les enregistrements des employés d'une entreprise. Chaque enregistrement contient : le code, le nom et le prénom de l'employé. L'organisation des enregistrements est séquentielle. Le programme permet aussi l'affichage à l'écran de toutes les informations contenues dans le fichier.

```
program fichier_seq ;
type
  employe = record
    code : integer ;
    nom, prenom : string[20] ;
  end;
```

```
var
  emp : file of employe ;
  e : employe ;
  reponse : char ;
begin
  { Création du fichier emp }
  assign(emp, 'emp_physique');
  rewrite(emp);
  writeln('Entrez la liste des employés :');
  repeat
    write('Donnez le code de l'employé : ');
    readln(e.code);
    write('Donnez le nom de l'employé : ');
    readln(e.nom);
    write('Donnez le prénom de l'employé : ');
    readln(e.prenom);
  { Ecriture sur disque dans le fichier emp }
  write(emp, e);
  writeln('Autre saisie ? o/n');
  readln(reponse);
  until reponse = 'n';
  close(emp);
  { Consultation séquentielle du fichier emp }
  assign(emp, 'emp_physique');
  reset(emp);
  writeln('Voici la liste des employés :');
  while not eof(emp) do begin
    read(emp, e);
    writeln(' Code : ', e.code, ' Nom : ', e.nom, ' Prénom : ', e.prenom);
  end;
  close(emp);
end.
```

**Exemple 2 :**

Le programme Pascal suivant permet de lire le contenu d'un fichier texte nommé texte.txt.

```
program fichier_seq ;
var
  fichier : text ;
  ligne : string[80] ;
```

```
begin
  assign(fichier, 'texte.txt');
  {$I-}
  reset(fichier);
  if (ioresult = 0) then begin
    while not eof(fichier) do begin
      readln(fichier, ligne);
      writeln(ligne);
    end;
    close(fichier);
  end
  else writeln('fichier inexistant.');
```

end.

## 8. Stratégies de traitement des fichiers

En plus des opérations décrites précédemment, on peut aussi effectuer sur un fichier :

- La suppression d'un enregistrement d'un fichier.
- La modification d'un enregistrement (le contenu des rubriques d'un enregistrement du fichier sera changé).
- Tri d'un fichier.
- Fusion de deux ou plusieurs fichiers.

Ces opérations ne sont pas utilisées directement, mais peuvent être implémentées en utilisant plusieurs stratégies. Il existe globalement deux manières de traiter les fichiers :

1. La première consiste à s'en tenir au fichier proprement dit, c'est-à-dire à modifier directement les informations sur le disque dur. C'est parfois un peu acrobatique, lorsqu'on veut supprimer un élément d'un fichier : on programme alors une boucle avec un test, qui recopie dans un deuxième fichier tous les éléments du premier fichier sauf un ; et il faut ensuite recopier intégralement le deuxième fichier à la place du premier fichier.
2. L'autre stratégie consiste à passer par un ou plusieurs tableaux. En fait, le principe fondamental de cette approche est de commencer, avant toute autre chose, par recopier l'intégralité du fichier de départ en mémoire vive. Ensuite, on ne manipule que cette mémoire vive (concrètement, un ou plusieurs tableaux). Et lorsque le traitement est terminé, on recopie à nouveau dans l'autre sens, à partir de la mémoire vive vers le fichier d'origine.

Les avantages de la seconde technique sont nombreux :

- La rapidité : les accès en mémoire vive sont des milliers de fois plus rapides (nanosecondes) que les accès aux mémoires de masse (millisecondes au mieux pour un disque dur). En basculant le fichier du départ dans un tableau, on minimise le nombre ultérieur d'accès au disque, et tous les traitements étant ensuite effectués en mémoire.
- La facilité de programmation : bien qu'il faille écrire les instructions de recopie du fichier dans le tableau, pour peu qu'on doive tripoter les informations dans tous les sens, c'est largement plus facile de faire cela avec un tableau qu'avec des fichiers.

Mais comme même, la seconde technique est déconseillée dans le cas d'immenses fichiers (très rares, cependant) là où la recopie en mémoire peut s'avérer problématique. La recopie d'un très gros fichier en mémoire vive exige des ressources qui peuvent atteindre des dimensions considérables.

## **9. Exercices corrigés**

### **9.1. Exercices**

#### **Exercice 1 :**

Reprenez le programme précédent (fichier\_seq), et ajoutez la partie du programme qui permet d'accéder directement à l'enregistrement dont la position est saisie par l'utilisateur pour l'afficher. Il est à noter que le premier enregistrement se trouve à la position 0, le deuxième à la position 1, et ainsi de suite.

#### **Exercice 2 :**

Ecrire un programme Pascal qui permet de :

- Créer un fichier etudiant contenant les informations suivantes : numéro d'identification (entier), nom (chaîne de 20 caractères) et moyenne (réel).
- Visualiser la liste des étudiants.
- Visualiser un étudiant par numéro d'identification.
- Visualiser la liste des étudiants par ordre décroissant des moyennes obtenues.

#### **Exercice 3 :**

Ecrire un programme Pascal permettant de stocker la liste des étudiants du fichier etudiant dans un deuxième fichier etudiant2, après avoir trié les étudiants par ordre croissant de leurs moyennes.

#### **Exercice 4 :**

Ecrire un programme Pascal permettant d'insérer un étudiant dans le fichier etudiant2, tout en gardant l'ordre croissant des moyennes obtenues.

### Exercice 5 :

Ecrire un programme Pascal permettant de supprimer ou de modifier les informations d'un étudiant dont le numéro est lu à partir du clavier : Il s'agit donc de la modification ou la suppression d'un enregistrement du fichier etudiant créé précédemment. Si le numéro saisi n'existe pas, on doit le signaler.

### Exercice 6 :

Ecrire un programme Pascal permettant de copier un fichier texte texte.txt dans un deuxième fichier essai.txt, ensuite de fusionner ces deux fichiers dans un troisième nommé texteglob.txt.

## 9.2. Corrigés

### Solution 1 :

```
program fichier_seq ;
type
  employe = record
    code : integer ;
    nom, prenom : string[20] ;
  end;
var
  emp : file of employe ;
  e : employe ;
  reponse : char ;
  position : integer ;
begin
  { Consultation directe par position }
  writeln('Donnez la position de l'enregistrement dans le fichier :');
  readln(position);
  assign(emp, 'emp_physique');
  reset(emp);
  seek(emp, position) ;
  read(emp, e);
  writeln(' Code : ', e.code, ' Nom : ', e.nom, ' Prénom : ', e.prenom);
  close(emp);
end.
```

### Solution 2:

```
program scalarite ;
function menu : integer;
var
  choix : integer ;
```

```
begin
  writeln('1. Création du fichier. ');
  writeln('2. Visualiser la liste des étudiants. ');
  writeln('3. Visualiser un étudiant par numéro. ');
  writeln('4. Visualiser la liste par ordre décroissant des moyennes. ');
  writeln('5. Quitter l'application. ');
  writeln('Entrez votre choix : ');
  readln(choix);
  menu := choix;
end;
type
  etudiant = record
    num_id : integer ;
    nom : string[20] ;
    moyenne : real ;
  end;
var
  etud : file of etudiant ;
  e : etudiant ;
  reponse : char ;
  choix, num, nb, i, j : integer ;
  etd : array[1..10] of etudiant ;
begin
  choix := menu ;
  repeat
    case choix of
      1 : begin { Création du fichier }
          assign(etud, 'etudiant');
          rewrite(etud);
          repeat
            writeln('Donnez le numéro de l'étudiant : ');
            readln(e.num_id);
            writeln('Donnez le nom de l'étudiant : ');
            readln(e.nom);
            writeln('Donnez la moyenne de l'étudiant : ');
            readln(e.moyenne);
            write(etud, e);
            writeln('Autre saisie ? o/n');
            readln(reponse);
```



```

    until reponse = 'n';
    close(etud);
end;
2 : begin { Visualiser la liste des étudiants }
    assign(etud, 'etudiant');
    reset(etud);
    while not eof(etud) do begin
        read(etud, e);
        writeln('Numéro = ', e.num_id, ', Nom : ', e.nom, ', Moyenne : ', e.moyenne);
    end;
    close(etud);
end;
3 : begin { Consulter un étudiant par un numéro }
    writeln('Donnez le numéro de l"étudiant :');
    readln(num);
    assign(etud, 'etudiant');
    reset(etud);
    while not eof(etud) do begin
        read(etud, e);
        if (e.num_id = num) then
            writeln('Numéro = ', e.num_id, ', Nom : ', e.nom, ', Moyenne : ', e.moyenne);
        end;
    end;
    close(etud);
end;
4 : begin { Visualiser la liste des étudiants par ordre décroissant des moyennes }
    assign(etud, 'etudiant');
    reset(etud);
    nb := 1;
{ On stocke les enregistrements du fichier dans un tableau
d'enregistrements pour le tri }
    while not eof(etud) do begin
        read(etud, etd[nb]);
        nb := nb + 1 ;
    end;
    close(etud);
{ Tri du tableau par ordre décroissant des moyennes }
    for i := 1 to nb - 2 do
        for j := i+1 to nb - 1 do
            if etd[j].moyenne > etd[i].moyenne then begin

```

```
        e := etd[i];
        etd[i] := etd[j];
        etd[j] := e ;
    end;
{ Affichage du tableau après tri }
for i := 1 to nb - 1 do
    with etd[i] do writeln(num_id, ' ', nom, ' ', moyenne);
end;
5 : begin end
    else writeln('Choix inexistant. Recommencer.');
```

end;

if choix <> 5 then choix := menu;

until choix = 5 ;

end.

**Solution 3 :**

```
program Tri_fichier ;
type
    etudiant = record
        num_id : integer ;
        nom : string[20] ;
        moyenne : real ;
    end;
var
    etud : file of etudiant ;
    e : etudiant ;
    nb, i, j : integer ;
    etd : array[1..10] of etudiant ;
begin
    assign(etud, 'etudiant');
    reset(etud);
    nb := 1;
{ On stocke les enregistrements du fichier etudiant dans un tableau
d'enregistrements pour le tri }
    while not eof(etud) do begin
        read(etud, etd[nb]);
        nb := nb + 1 ;
    end;
    close(etud);
{ Tri du tableau par ordre croissant des moyennes }
```

```
for i := 1 to nb - 2 do
  for j := i+1 to nb - 1 do
    if etd[j].moyenne < etd[i].moyenne then begin
      e := etd[i];
      etd[i] := etd[j];
      etd[j] := e ;
    end;
  { Stockage du tableau dans le fichier etudiant2 }
  assign(etud, 'etudiant2');
  rewrite(etud);
  for i := 1 to nb - 1 do write(etud, etd[i]);
  close(etud);
  { Visualiser la liste des étudiants à partir du fichier etudiant2 }
  writeln('Liste des étudiants dans le fichier etudiant2 :');
  assign(etud, 'etudiant2');
  reset(etud);
  while not eof(etud) do begin
    read(etud, e);
    writeln('Numéro = ', e.num_id, ', Nom : ', e.nom, ', Moyenne : ', e.moyenne);
  end;
  close(etud);
end.
```

**Solution 4 :**

```
program Insertion_fichier ;
type
  etudiant = record
    num_id : integer ;
    nom : string[20] ;
    moyenne : real ;
  end;
var
  etud : file of etudiant ;
  e : etudiant ;
  nb, i, j : integer ;
  etd : array[1..10] of etudiant ;
  b : boolean ;
begin
  { Visualiser la liste des étudiants à partir du fichier etudiant2 avant l'insertion }
  writeln('Liste des étudiants dans le fichier etudiant2 avant insertion :');
```

```

assign(etud, 'etudiant2');
reset(etud);
while not eof(etud) do begin
  read(etud, e);
  writeln('Numéro = ', e.num_id, ', Nom : ', e.nom, ', Moyenne : ', e.moyenne);
end;
close(etud);
{ On stocke les enregistrements du fichier etudiant2 dans un tableau d'enregistrements }
assign(etud, 'etudiant2');
reset(etud);
nb := 1;
while not eof(etud) do begin
  read(etud, etd[nb]);
  nb := nb + 1 ;
end;
close(etud);
{ Lire les informations de l'étudiant à insérer }
writeln('Saisir l''étudiant à insérer :');
writeln('Donnez le numéro de l''étudiant :');
readln(e.num_id);
writeln('Donnez le nom de l''étudiant :');
readln(e.nom);
writeln('Donnez la moyenne de l''étudiant :');
readln(e.moyenne);
{ Insertion de l'étudiant dans le tableau }
i := 1;  b := false;
while (i <= nb - 1) and NOT b do begin
  if (e.moyenne < etd[i].moyenne) then b := true ;
  if NOT b then i := i + 1 ;
end;
for j := nb downto i+1 do etd[j] := etd[j-1];
etd[i] := e;
{ Stockage du tableau dans le fichier etudiant2 }
assign(etud, 'etudiant2');
rewrite(etud);
for i := 1 to nb do write(etud, etd[i]);
close(etud);
{ Visualiser la liste des étudiants à partir du fichier etudiant2 après l'insertion }
writeln('Liste des étudiants dans le fichier etudiant2 après l''insertion :');

```

```
assign(etud, 'etudiant2');
reset(etud);
while not eof(etud) do begin
  read(etud, e);
  writeln('Numéro = ', e.num_id, ', Nom : ', e.nom, ', Moyenne : ', e.moyenne);
end;
close(etud);
end.
```

**Solution 5 :**

```
program scolarite ;
function menu : integer;
var
  choix : integer ;
begin
  writeln('1. Visualiser la liste des étudiants. ');
  writeln('2. Modifier un enregistrement. ');
  writeln('3. Supprimer un enregistrement. ');
  writeln('4. Quitter l''application. ');
  writeln('Entrez votre choix : ');
  readln(choix);
  menu := choix;
end;
type
  etudiant = record
    num_id : integer ;
    nom : string[20] ;
    moyenne : real ;
  end;
var
  etud : file of etudiant ;
  e : etudiant ;
  reponse : char ;
  choix, num, nb, i, j : integer ;
  etd : array[1..20] of etudiant ;
  exist : boolean;
begin
  choix := menu ;
  repeat
    case choix of
```

```
1 : begin { Visualiser la liste des étudiants }
    assign(etud, 'etudiant');
    reset(etud);
    while not eof(etud) do begin
        read(etud, e);
        writeln('Numéro = ', e.num_id, ', Nom : ', e.nom, ', Moyenne : ', e.moyenne);
    end;
    close(etud);
end;

2 : begin { Modification d'un enregistrement }
    assign(etud, 'etudiant');
    reset(etud);
    nb := 1;
{ On stocke les enregistrements du fichier dans un tableau d'enregistrements pour le tri }
    while not eof(etud) do begin
        read(etud, etd[nb]);
        nb := nb + 1 ;
    end;
    close(etud);
    writeln('Donnez le numéro de l"étudiant à modifier :');
    readln(num);
    i := 1;
    exist := false;
    while (i <= nb-1) and not exist do begin
        if etd[i].num_id = num then exist := true ;
        if not exist then i := i + 1 ;
    end;
    if exist then begin
        writeln('Saisir les nouvelles informations :');
        writeln('Donnez le numéro de l"étudiant :');
        readln(e.num_id);
        writeln('Donnez le nom de l"étudiant :');
        readln(e.nom);
        writeln('Donnez la moyenne de l"étudiant :');
        readln(e.moyenne);
        etd[i] := e;
        { Stockage du tableau dans le fichier etudiant }
        assign(etud, 'etudiant');
        rewrite(etud);
```

```

        for i := 1 to nb-1 do write(etud, etd[i]);
        close(etud);
    end
    else writeln('Il n'existe pas un étudiant avec le num : ', num);
end;
3 : begin { Suppression d'un enregistrement }
    assign(etud, 'etudiant');
    reset(etud);
    nb := 1;
{ On stocke les enregistrements du fichier dans un tableau d'enregistrements pour le tri }
    while not eof(etud) do begin
        read(etud, etd[nb]);
        nb := nb + 1 ;
    end;
    close(etud);
    writeln('Donnez le numéro de l'étudiant à supprimer :');
    readln(num);
    i := 1;
    exist := false;
    while (i <= nb-1) and not exist do begin
        if etd[i].num_id = num then exist := true ;
        if not exist then i := i + 1 ;
    end;
    if exist then begin
        for j := i to nb-2 do etd[j] := etd[j+1];
        { Stockage du tableau dans le fichier etudiant }
        assign(etud, 'etudiant');
        rewrite(etud);
        for i := 1 to nb-2 do write(etud, etd[i]);
        close(etud);
    end
    else writeln('Il n'existe pas un étudiant avec le num : ', num);
end;
4 : begin end
else writeln('Choix inexistant. Recommencer.');
```

end;

if choix <> 4 then choix := menu;

until choix = 4 ;

end.

**Solution 6 :**

```
program concat_copier_fichier ;
var
  fichier1, fichier2, fichier3 : text ;
  ligne : string[80] ;
begin
  assign(fichier1, 'texte.txt');
  {$I-}
  reset(fichier1);
  if (ioresult = 0) then begin
    { Copier le fichier texte.txt dans essai.txt }
    assign(fichier2, 'essai.txt');
    rewrite(fichier2);
    while not eof(fichier1) do begin
      readln(fichier1, ligne);
      writeln(fichier2, ligne);
    end;
    close(fichier1);
    close(fichier2);
    { Concaténer texte.txt et essai.txt dans texteglob.txt }
    assign(fichier1, 'texte.txt');
    reset(fichier1);
    assign(fichier2, 'essai.txt');
    reset(fichier2);
    assign(fichier3, 'texteglob.txt');
    rewrite(fichier3);
    while not eof(fichier1) do begin
      readln(fichier1, ligne);
      writeln(fichier3, ligne);
    end;
    close(fichier1);
    while not eof(fichier2) do begin
      readln(fichier2, ligne);
      writeln(fichier3, ligne);
    end;
    close(fichier2);
    close(fichier3);
    writeln('Le contenu du fichier texte.txt :');
    assign(fichier1, 'texte.txt');
```



```
reset(fichier1);
while not eof(fichier1) do begin
  readln(fichier1, ligne);
  writeln(ligne);
end;
close(fichier1);
writeln('Le contenu du fichier essai.txt :');
assign(fichier2, 'essai.txt');
reset(fichier2);
while not eof(fichier2) do begin
  readln(fichier2, ligne);
  writeln(ligne);
end;
close(fichier2);
writeln('Le contenu du fichier texteglob.txt :');
assign(fichier3, 'texteglob.txt');
reset(fichier3);
while not eof(fichier3) do begin
  readln(fichier3, ligne);
  writeln(ligne);
end;
close(fichier2);
end
else writeln('Fichier texte.txt inexistant.');
```

end.

## Chapitre 10 : Les listes chaînées

### 1. Introduction

Une structure de données correspond à un ensemble de deux ou plusieurs données, formant ainsi un groupe d'éléments à traiter. Jusqu'à maintenant, on n'a vu que des structures de données statiques (les tableaux). Ces structures sont définies préalablement dans la partie déclaration, avant l'exécution du programme. Au cours d'exécution, l'espace réservé à ces données ne peut pas être changé. Elles ne peuvent être alors ni étendues en cas de besoin de réservation d'un nouvel espace, ni détruites en cas de non utilisation (pour une optimisation de l'espace mémoire réservé).

Si on utilise par exemple un tableau pour stocker une liste (un ensemble) d'éléments, cela va nous poser plus tard des problèmes, surtout si le nombre d'éléments n'est pas connu d'avance, et qui peut dépasser la taille fixée du tableau. Pour remédier à ce problème, il est possible d'utiliser les listes chaînées qui peuvent être étendues par l'allocation d'un nouvel espace mémoire quand c'est nécessaire. Elles peuvent être aussi réduites par une désallocation de l'espace non utile, libérant ainsi cet espace mémoire pour d'autre utilisation.

La construction d'une liste chaînée consiste à regrouper un ensemble d'objets éparpillés en mémoire, et liés au moyen de variables de type pointeur.

### 2. Les pointeurs

Lorsqu'on déclare une variable, et ce, quel que soit le langage de programmation, le compilateur réserve en mémoire l'espace nécessaire au contenu de cette variable à une adresse donnée. Toute variable possède donc une adresse en mémoire. La plupart du temps, on ne s'intéresse pas à ces adresses. Mais quelquefois, ce type de renseignement peut s'avérer fort utile.

Un pointeur est une variable qui désigne une adresse mémoire. Il est une variable qui au lieu de contenir l'information proprement dite, contient son adresse en mémoire. Il s'agit d'un type de données à l'aide duquel le programmeur localise un objet dans la mémoire.

Format général : `Nom_pointeur : ↑ <type de données>` ;

Les pointeurs sont typés pour indiquer le type de données stockées à l'adresse qu'ils contiennent. Le signe ↑ mis devant le nom du type de données, dans la partie déclaration, permet de définir une variable pointeur pointant vers des variables de ce type.

### Exemple :

Pour déclarer un pointeur Y vers un entier (contient l'adresse d'un entier), on met :

Variables

Y : ↑ entier ;

...

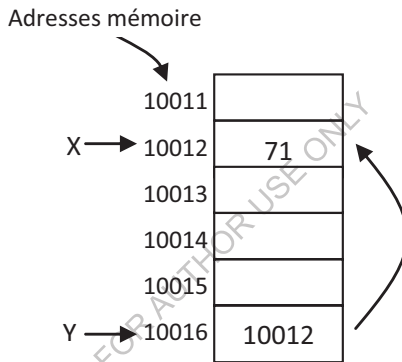
En Pascal, un pointeur est déclaré comme suit :

var

Y : ^ integer;

...

Si nous avons : X une variable entière ayant comme valeur 71, et Y un pointeur vers la variable entière X. Les deux variables X et Y peuvent être représentées en mémoire comme suit :



A la différence de la variable X classique qui contient directement la donnée 71, la variable Y déclarée comme pointeur contient son adresse. On dit que Y pointe vers la donnée 71 grâce à son contenu qui est l'adresse de cette donnée.

Un pointeur peut recevoir la valeur d'un autre pointeur par une opération d'affectation. On peut aussi comparer deux pointeurs par les opérateurs de comparaison (=, >, <, >=, <=, <>).

Les pointeurs mènent à une gestion dynamique de la mémoire.

### 3. Gestion dynamique de la mémoire

A la différence de la gestion statique qui permet la réservation de l'espace mémoire avant l'exécution du programme, la gestion dynamique permet la réservation de cet espace pendant l'exécution du programme.

Quand un pointeur contient l'adresse d'une autre variable, on dit qu'il pointe vers l'emplacement en mémoire de cette variable. Une variable de type pointeur permet donc de référencer (pointer vers) une variable

d'un type déterminé, dite encore variable pointée. Seule une variable de type pointeur permet la réservation (allocation) d'un espace mémoire au cours d'exécution du programme.

La variable pointeur est créée statiquement dans la partie déclaration du programme, alors que la variable pointée est créée dynamiquement dans la partie instructions. Cette gestion dynamique est effectuée en utilisant :

- Allouer(p) qui permet de réserver un espace mémoire pour une variable pointée par le pointeur p. Il s'agit d'une allocation dynamique de la mémoire effectuée pendant l'exécution du programme. En Pascal, c'est new(p) ;
- Désallouer(p) qui permet de libérer l'espace mémoire réservé pour une variable pointée par le pointeur p, quand on en a plus besoin. En Pascal, c'est dispose(p) ;

Ces deux procédures permettent d'obtenir et de rendre un espace mémoire au fur et à mesure des besoins de l'algorithme. On parle de gestion dynamique de la mémoire, contrairement à la gestion statique des tableaux de dimension fixe.

**Exemple :**

Algorithme pointeurs ;

Type

pointeur = ↑integer ;

Variable

y : pointeur ; { y est un pointeur vers un entier }

Début

Allouer(y) ; { On alloue un espace mémoire de la taille d'un entier.

L'adresse de cet espace est assigné au pointeur y }

y↑ ← 7 ; { On place la valeur 7 dans l'espace alloué }

Ecrire('La valeur est ', y↑) ; { puis, on l'affiche }

Désallouer(y) ; { On libère l'espace réservé précédemment }

Fin.

*Le programme Pascal :*

program pointeurs ;

type

pointeur = ^integer;

var

y : pointeur ;

begin

new(y);

y^ := 7 ;

```
writeln('La valeur est ', y^);  
dispose(y);  
end.
```

Le programme Pascal ci-dessus permet de créer une variable *y* de type pointeur, pointant vers un entier. Il alloue un espace pour la variable pointée. Ensuite, il affecte la valeur 7 à la variable pointée, indiquée par *^* qui suit le nom de la variable pointeur. Il affiche cette valeur. Enfin, il libère l'espace réservé à la variable pointée.

Voyons ce que fait l'exemple suivant :

```
program pointeurs ;  
type  
  pointeur = ^integer;  
var  
  y : pointeur ;  
  x : integer;  
begin  
  x := 7;  
  y := @x ;  
  writeln('La valeur est ', y^);  
end.
```

Le programme Pascal précédent permet de créer une variable *y* de type pointeur, pointant vers un entier, et une autre variable entière *x*. Ensuite, il affecte la valeur 7 à la variable *x*. Il affecte l'adresse de la variable *x*, indiquée par *@*, à la variable *y*. Il affiche la valeur de la variable pointée par *y*. On constate qu'il n'y a ni une allocation ni une désallocation de l'espace mémoire réservé, car la variable pointée *x* est créée statiquement dans la partie déclaration.

### Remarques :

- Les pointeurs occupent 4 octets en mémoire.
- Quand un pointeur ne pointe nulle part, sa valeur est égale à nil. Cette valeur représente une adresse inaccessible, et c'est donc une valeur que l'on utilise pour dire que le pointeur n'a pas de valeur déterminée. Attention ! après une désallocation, la valeur du pointeur n'est pas mise automatiquement à nil. Pour cela, il est préconisé d'affecter la valeur nil au pointeur pour dire qu'il ne pointe nulle part après cette désallocation.
- Les deux procédures *new* et *dispose* utilisent une zone mémoire appelée TAS. *new* prend une place du TAS, et *dispose* restitue une place au TAS.

Une structure de donnée dynamique est constituée d'un certain nombre d'éléments reliés par des pointeurs. Principalement, le développeur peut avoir recours aux structures de données dynamiques suivantes : les listes chaînées (simplement chaînées, doublement chaînées, circulaires, les piles et les files), les arbres et les graphes. Les deux dernières structures seront vues dans les prochains chapitres.

#### 4. Les listes chaînées

D'une façon générale, une liste chaînée est une structure de données qui permet de stocker un ensemble d'éléments. Dans une liste chaînée, les éléments sont rangés linéairement. Chaque élément est lié à son successeur, et il n'est donc pas possible d'accéder directement à un élément quelconque de la liste.

Une liste chaînée est donc une structure de données dynamique constituée d'un certain nombre d'éléments qui sont reliés par des pointeurs. On peut ajouter de nouveaux éléments à cette structure, ou les supprimer pendant l'exécution du programme. Les adresses réelles des éléments n'ont pas d'importance parce que les liaisons logiques entre les éléments sont établies par des pointeurs, et non pas par les positions relatives dans la mémoire.

Une liste chaînée est composée de maillons (nœuds, composants ou éléments). Un maillon étant une structure (enregistrement) qui contient des informations à stocker et un pointeur vers le prochain maillon (successeur) de la liste.

Type

Maillon = enregistrement

Inf1 : Type\_element1 ;

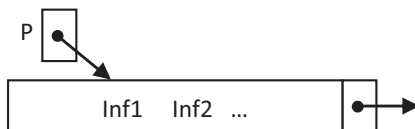
Inf2 : Type\_element2 ;

...

Suivant : ↑ Maillon ;

Fin ;

Une variable P de type pointeur vers Maillon (var P : ↑ Maillon) peut être représentée comme suit :



L'adresse du maillon est rangée dans la variable P. Par abus de langage, on dira : maillon d'adresse P ou maillon pointé par P. Le maillon contient deux parties : la partie information contenant les variables Inf1, Inf2..., et le pointeur Suivant contenant une adresse. On accède aux

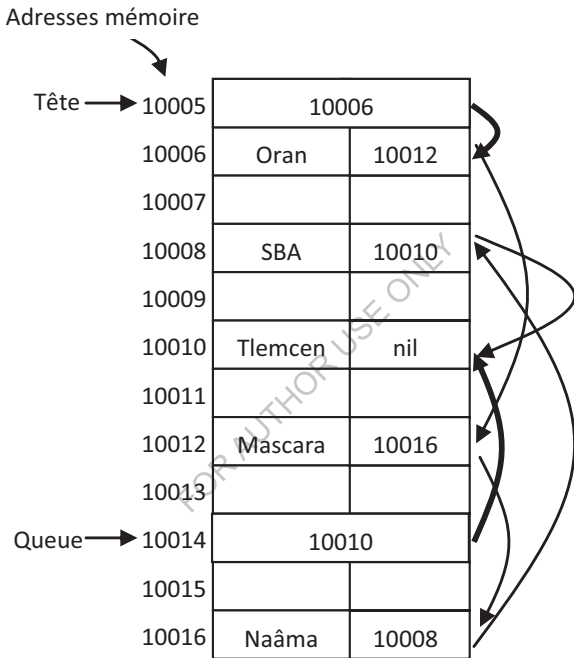
variables de la partie information du maillon de l'adresse P comme suit :  $P↑.Inf1$ ,  $P↑.Inf2...$  On accède de la même façon à l'adresse contenue dans le maillon d'adresse P par  $P↑.Suivant$ .

La création d'une liste chaînée consiste alors à allouer dynamiquement les maillons chaque fois que cela est nécessaire.

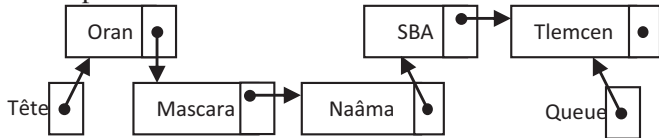
**Exemple :**

Soit une liste contenant les chaînes de caractères suivantes classées dans cet ordre : "Oran", "Mascara", "Saida", "SBA" et "Tlemcen".

Cette liste peut être représentée en mémoire comme suit :



Une deuxième représentation facile à être assimilée est la suivante :



À la différence d'un tableau, les éléments d'une liste chaînée n'ont aucune raison d'être contigus ni ordonnés en mémoire. Il n'est donc pas possible d'accéder directement à un élément quelconque de la liste. C'est l'adresse (pointeur) du premier maillon qui détermine la liste. Cette adresse doit se trouver dans une variable que nous appelons

souvent Tête, Sommet ou Début de la liste. Le pointeur vers le dernier élément de la liste est appelé Queue de la liste.

### Remarque :

Il est possible d'utiliser un tableau pour la modélisation d'une liste chaînée. Ce qui consiste à allouer à l'avance un certain nombre de maillons. Autrement dit, un tableau de maillons, de taille fixe, est alloué pour la création de la liste. Par la suite, la liste est formée en utilisant comme maillons des cases de ce tableau. Avec cette modélisation, un pointeur vers un prochain maillon dans la liste sera tout simplement un indice dans le tableau, ce qui permet un accès direct aux éléments de la liste. Dans ce cas, la liste de l'exemple précédent sera présenter comme suit :

1	2	3	4	5
Oran	Mascara	Saida	SBA	Tlemcen

Les tableaux souffrent du problème de leur taille fixe. Si on utilise alors un tableau pour représenter une liste, on doit savoir combien d'éléments pourra cette liste contenir.

## 5. Opérations sur les listes chaînées

Plusieurs opérations peuvent être effectuées sur une liste. On cite :

- La création d'une liste.
- L'ajout à la fin ou l'insertion d'un élément au milieu d'une liste.
- La suppression d'un élément d'une liste.
- L'affichage des éléments d'une liste.

La représentation par tableau d'une liste chaînée n'a que peu d'intérêt. En effet, l'insertion d'un élément à n'importe quelle position dans une liste chaînée représentée par un tableau implique le décalage vers la droite d'un certain nombre d'éléments. Pour la suppression, la liste chaînée représentée par un tableau a le même défaut qu'un tableau : il faut décaler un certain nombres d'éléments vers la gauche.

La liste chaînée représentée par pointeurs permet une insertion et une suppression rapide des éléments. Cependant, contrairement au tableau, cette liste chaînée interdit l'accès direct aux éléments (mis à part la Tête et la Queue).

Dans ce qui suit, nous allons créer, consulter, insérer et supprimer des éléments d'une liste chaînée représentée par pointeurs des villes présentée ci-dessus. Pour ce faire, considérons que chaque élément est de type Ville contenant une information (la désignation de la ville de type chaîne de caractères) et un pointeur vers l'élément suivant.



5.1. Créer et remplir une liste

Nous allons décrire maintenant étape par étape l’algorithme permettant de créer et remplir la liste des villes :

- 1. Chaque élément de la liste doit contenir la désignation de la ville, plus un pointeur vers l’élément suivant. Ce qui est traduit par la structure Ville décrite comme suit :

Type

Ville = enregistrement  
Des : chaîne de caractères ;  
Suivant : ↑ Ville ;  
Fin ;

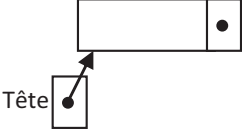
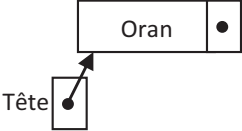
- 2. Les éléments de la liste seront éparpillés dans la mémoire centrale. Le début et la fin de la liste doivent être capturés successivement par deux pointeurs : Tete et Queue. Le contenu de la variable Tete est fixe, car elle contient la même valeur tout au long de l’algorithme (l’adresse du premier élément de la liste). Le contenu de la variable Queue change tout au long de l’algorithme, car elle contient l’adresse du dernier élément ajouté à la liste. Les variables Tete et Queue sont déclarées comme suit :

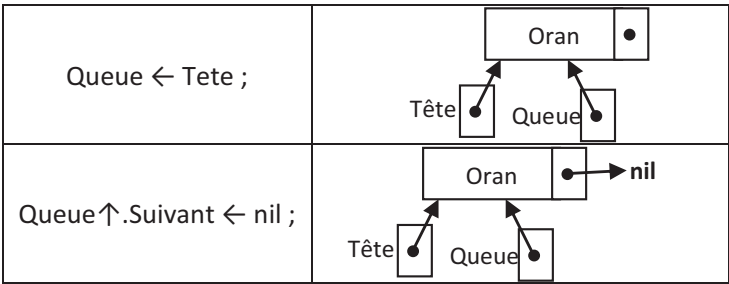
Variables

Tete, Queue : ↑ Ville ;

- 3. Le premier élément de la liste peut être créé comme suit :
  - On réserve un espace mémoire pour une variable pointée par Tete : Allouer(Tete) ;
  - Remplir le premier élément de la liste : Tete↑.Des ← 'Oran' ;
  - A ce niveau, le premier élément est au même temps le dernier. Par conséquent, la queue pointe vers la tête : Queue ← Tete ;
  - Le successeur de la queue est le nil : Queue↑.Suivant ← nil ;

Le tableau suivant décrit le changement d’état de la mémoire selon les opérations de l’algorithme :

Opération	Représentation en mémoire
Allouer(Tete) ;	
Tete↑.Des ← 'Oran' ;	



4. L'ajout d'un deuxième élément à la fin de la liste se fait comme suit :

- On réserve un espace mémoire pour une variable pointée par le successeur de la queue :  $Alouer(Queue \uparrow . Suivant);$
- La queue va pointer maintenant vers son successeur :  $Queue \leftarrow Queue \uparrow . Suivant;$
- Remplir le dernier élément de la liste :  $Queue \uparrow . Des \leftarrow 'Mascara';$
- Le successeur de la queue est le nil :  $Queue \uparrow . Suivant \leftarrow nil ;$

Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de l'algorithme :

Opération	Représentation en mémoire
$Alouer(Queue \uparrow . Suivant);$	
$Queue \leftarrow Queue \uparrow . Suivant;$	
$Queue \uparrow . Des \leftarrow 'Mascara' ;$	
$Queue \uparrow . Suivant \leftarrow nil ;$	

5. Pour ajouter le troisième, le quatrième et le cinquième élément, il suffit de répéter l'étape 4, en substituant le remplissage du dernier élément de la liste successivement par  $Queue \uparrow . Des \leftarrow 'Saida' ;$

Queue↑.Des ← 'SBA' ; et Queue↑.Des ← 'Tlemcen' ;. Il est donc possible de substituer l'étape 4 par une procédure permettant l'ajout d'un élément à la fin de la liste. La procédure possède comme paramètre le nom de la ville à ajouter, et elle est écrite comme suit :

Procédure Ajouter\_Q (V : chaîne de caractères) ;

Début

    Allouer(Queue↑.Suivant);

    Queue ← Queue↑.Suivant;

    Queue↑.Des ← V ;

    Queue↑.Suivant ← nil ;

Fin ;

L'algorithme complet pour créer la liste des villes est le suivant :

Algorithme Villes ;

Type

    Ville = Enregistrement

        Des : chaîne de caractères ;

        Suivant : ↑ Ville;

fin ;

Variables

    Tete, Queue : ↑ Ville;

(\*Procédure d'ajout d'un élément à la liste\*)

Procédure Ajouter\_Q (V : chaîne de caractères) ;

début

    Allouer(Queue↑.Suivant);

    Queue ← Queue↑.Suivant;

    Queue↑.Des ← V ;

    Queue↑.Suivant ← nil ;

fin ;

Début

    (\*Créer et remplir le premier élément de la liste\*)

    Allouer(Tete) ;

    Tete↑.Des ← 'Oran' ;

    Queue ← Tete ;

    Queue↑.Suivant ← nil ;

    (\*Ajouter le reste des éléments à la liste\*)

    Ajouter\_Q('Mascara') ;

    Ajouter\_Q('Saida') ;

    Ajouter\_Q('SBA') ;

    Ajouter\_Q('Tlemcen') ;

Fin.

Le programme Pascal :

```
program Villes ;
type
  Ptr_Ville = ^ Ville ;
  Ville = record
    Des : string ;
    Suivant : Ptr_Ville;
  end ;
var
  Tete, Queue : Ptr_Ville ;
  (*Procédure d'ajout d'un élément à la liste*)
procedure Ajouter_Q (V : String) ;
begin
  new(Queue^.Suivant);
  Queue := Queue^.Suivant;
  Queue^.Des := V ;
  Queue^.Suivant := nil ;
end ;
begin
  (*Créer et remplir le premier élément de la liste*)
  new(Tete);
  Tete^.Des := 'Oran';
  Queue := Tete ;
  Queue^.Suivant := nil ;
  (*Ajouter le reste des éléments à la liste*)
  Ajouter_Q('Mascara') ;
  Ajouter_Q('Saida') ;
  Ajouter_Q('SBA') ;
  Ajouter_Q('Tlemcen') ;
end.
```

**Exercice :**

Redéfinir la procédure Ajouter\_Q, mais cette fois-ci en supposant que la liste est captée par un seul pointeur (celui de la tête). La queue étant déterminer en parcourant la liste de la tête à la fin. Le premier élément est pointé par la variable pointeur Tete. Le dernier élément de la liste est celui qui n'as pas de successeur, c.-à-d. l'élément pour le quel Suivant est égal à nil.

**Solution :**

```

Procédure Ajouter_Q(V : chaîne de caractères) ;
Variables
  P : ↑ Ville;
Début
  P ← Tete;
  Tant que (P↑.Suivant <> nil) Faire P ← P↑.Suivant;
  Allouer(P↑.Suivant);
  P ← P↑.Suivant;
  P↑.Des ← V ;
  P↑.Suivant ← nil ;
Fin ;

```

En Pascal :

```

procedure Ajouter_Q(V : string) ;
var
  P : Ptr_Ville ;
begin
  P := Tete;
  while (P^.Suivant <> nil) do P := P^.Suivant;
  new(P^.Suivant);
  P := P^.Suivant;
  P^.Des := V ;
  P^.Suivant := nil ;
end ;

```

**5.2. Ajouter un élément au début de la liste**

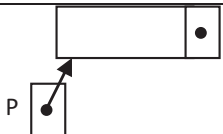
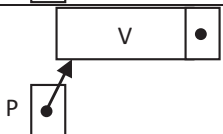
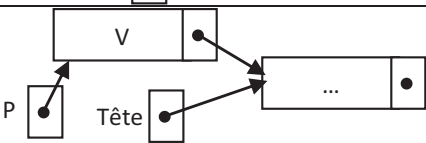
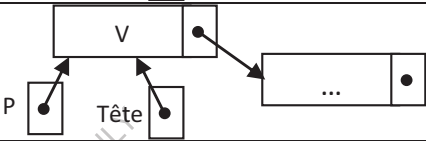
Il est aussi possible de remplir la liste en utilisant une procédure permettant d'ajouter un élément au début de la liste. La procédure possède comme paramètre le nom de la ville à ajouter.

Procédure Ajouter\_T (V : chaîne de caractères) ;

L'ajout d'un élément au début de la liste se fait comme suit :

- On réserve un espace mémoire pour une variable P de type pointeur vers ville : Allouer(P);
- Remplir le nouvel élément pointé par P : P↑.Des ← V ;
- On effectue le chaînage du nouvel élément avec la liste existante. Le successeur de P sera alors la tête : P↑.Suivant ← Tete ;
- La tête va pointer maintenant vers le nouvel élément : Tete ← P;

Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de la procédure :

Opération	Représentation en mémoire
Allouer(P);	
$P \uparrow .Des \leftarrow V$ ;	
$P \uparrow .Suivant \leftarrow Tete$ ;	
$Tete \leftarrow P$ ;	

La procédure permettant l'ajout d'un élément au début de la liste est alors la suivante :

```
Procédure Ajouter_T (V : chaîne de caractères) ;  
Variables P :  $\uparrow$  Ville ;  
Début  
  Allouer(P);  
   $P \uparrow .Des \leftarrow V$  ;  
   $P \uparrow .Suivant \leftarrow Tete$  ;  
   $Tete \leftarrow P$  ;  
  Si (Queue = nil) Alors Queue  $\leftarrow Tete$  ;  
Fin ;
```

En Pascal :

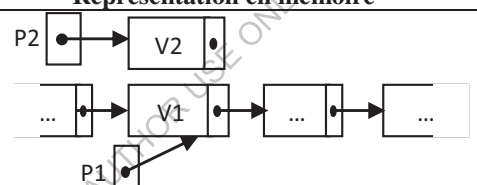
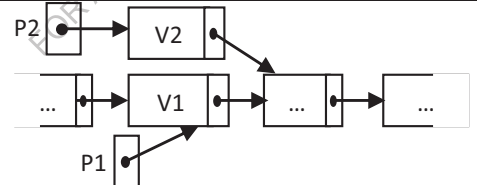
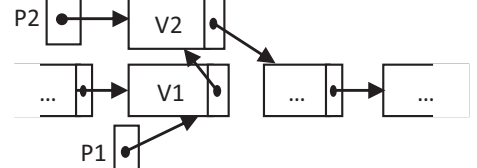
```
procedure Ajouter_T (V : string) ;  
var P : Ptr_Ville ;  
begin  
  new(P);  
  P^.Des := V ;  
  P^.Suivant := Tete ;  
  Tete := P;  
  if (Queue = nil) then Queue := Tete ;  
end ;
```

5.3. Insérer un élément dans la liste

On désire maintenant implémenter une procédure permettant d’insérer un élément après un autre dans une liste. La procédure possède deux paramètres : V1 et V2. V2 étant le nom de la ville à insérer, et V1 le nom de la ville après laquelle on désire insérer V2. Pour ce faire, on aura besoin de déclarer deux variables P1 et P2 pointant vers une ville : P1, P2 : ↑ Ville ;. Le pointeur P2 va pointer vers l’élément à insérer (un nouvel élément contenant la ville V2). Le pointeur P1 va nous permettre de localiser la ville V1 dans la liste. Il va donc pointer vers l’élément dont la désignation est V1. Pour que P1 et P2 prennent leurs positions, on doit parcourir la liste à partir de sa tête. Après ça, on doit suivre les étapes suivantes :

- Le successeur de P2 devient le successeur de P1 :  $P2 \uparrow .\text{Suivant} \leftarrow P1 \uparrow .\text{Suivant}$  ;
- Le successeur de P1 devient P2 lui même :  $P1 \uparrow .\text{Suivant} \leftarrow P2$ ;

Le tableau suivant décrit ces deux dernières étapes :

Opération	Représentation en mémoire
{Avant l'insertion}	
$P2 \uparrow .\text{Suivant} \leftarrow P1 \uparrow .\text{Suivant}$ ;	
$P1 \uparrow .\text{Suivant} \leftarrow P2$ ;	

La procédure permettant d’insérer un élément après un autre est alors la suivante :

Procédure Insérer\_AP\_Ville (V1, V2 : chaîne de caractères) ;

Variables

P1, P2 : ↑ Ville;

```
Début
Si Queue↑.Des = V1 Alors Ajouter_Q(V2)
Sinon début
{ Pointer vers l'élément contenant V1 }
  P1 ← Tete;
  Tant que (P1↑.Des <> V1) Faire P1 ← P1↑.Suivant;
{ Créer un nouvel élément contenant V2 }
  Allouer(P2);
  P2↑.Des ← V2 ;
{ Insérer le nouvel élément dans la liste }
  P2↑.Suivant ← P1↑.Suivant ;
  P1↑.Suivant ← P2;
fin ;
Fin ;
```

En Pascal :

```
procedure Insérer_AP_Ville (V1, V2 : string) ;
var
  P1, P2 : Ptr_Ville;
begin
  if Queue^.Des = V1 then Ajouter_Q(V2)
  else begin
    { Pointer vers l'élément contenant V1 }
    P1 := Tete;
    while (P1^.Des <> V1) do P1 := P1^.Suivant;
    { Créer un nouvel élément contenant V2 }
    new(P2);
    P2^.Des := V2 ;
    { Insérer le nouvel élément dans la liste }
    P2^.Suivant := P1^.Suivant ;
    P1^.Suivant := P2;
  end ;
end ;
```

**Remarque :**

Dans la procédure ci-dessus, on suppose que V1 existe obligatoirement dans la liste. Il est possible de traiter le cas où V1 n'existe pas dans la liste par un message d'erreur.

**Exercice :**

Ecrire une procédure permettant d'insérer un élément avant un autre dans une liste. La procédure possède deux paramètres : V1 et V2. V1



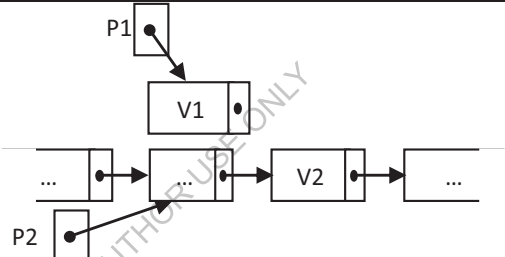
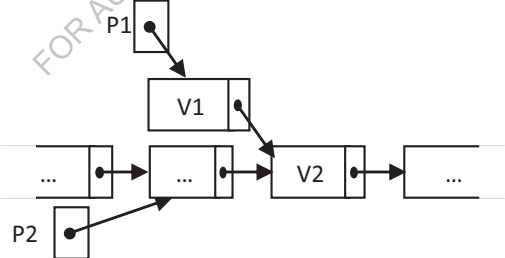
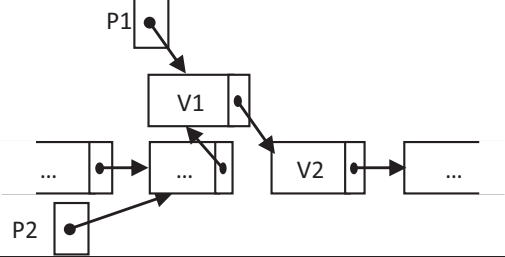
étant le nom de la ville à insérer, et V2 le nom de la ville avant laquelle on désire insérer V1.

**Solution :**

On aura besoin de déclarer deux variables P1 et P2 pointant vers une ville : P1, P2 : ↑ Ville ;. Le pointeur P1 va pointer vers l'élément à insérer (un nouvel élément contenant la ville V1). Le pointeur P2 va nous permettre de localiser la ville V2 dans la liste. Plus précisément, il va pointer juste avant l'élément dont la désignation est V2. Après ça, on doit suivre les étapes suivantes :

- Le successeur de P1 devient le successeur de P2 :  $P1 \uparrow .\text{Suivant} \leftarrow P2 \uparrow .\text{Suivant}$  ;
- Le successeur de P2 devient P1 lui même :  $P2 \uparrow .\text{Suivant} \leftarrow P1$ ;

Le tableau suivant décrit ces deux dernières étapes :

Opération	Représentation en mémoire
{Avant l'insertion}	
$P1 \uparrow .\text{Suivant} \leftarrow P2 \uparrow .\text{Suivant}$ ; $P2 \uparrow .\text{Suivant} \leftarrow P1$ ;	
$P2 \uparrow .\text{Suivant} \leftarrow P1$ ;	

La procédure permettant d'insérer un élément avant un autre est alors la suivante :

```

Procédure Insérer_AV_Ville (V1, V2 : chaîne de caractères) ;
Variables
  P1, P2 : ↑ Ville;
Début
Si Tete↑.Des = V2 Alors Ajouter_T(V1)
Sinon début
  { Pointer avant l'élément contenant V2 }
  P2 ← Tete;
  Tant que (P2↑.Suivant↑.Des <> V2) Faire P2 ← P2↑.Suivant;
  { Créer un nouvel élément contenant V1 }
  Allouer(P1);
  P1↑.Des ← V1 ;
  { Insérer le nouvel élément dans la liste }
  P1↑.Suivant ← P2↑.Suivant ;
  P2↑.Suivant ← P1;
fin ;
Fin ;

```

En Pascal :

```

procédure Insérer_AV_Ville (V1, V2 : string) ;
var
  P1, P2 : Ptr_Ville;
begin
if Tete^.Des = V2 then Ajouter_T(V1)
else begin
  { Pointer avant l'élément contenant V2 }
  P2 := Tete;
  while (P2^.Suivant^.Des <> V2) do P2 := P2^.Suivant;
  { Créer un nouvel élément contenant V1 }
  new(P1);
  P1^.Des := V1 ;
  { Insérer le nouvel élément dans la liste }
  P1^.Suivant := P2^.Suivant ;
  P2^.Suivant := P1;
end ;
end ;

```

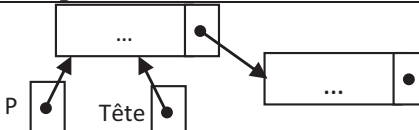
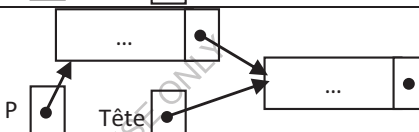
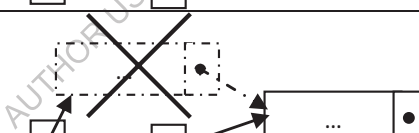
#### 5.4. Supprimer la tête de la liste

Il est aussi possible de supprimer la tête de la liste et récupérer l'espace mémoire occupé par cet élément. Pour ce faire, on utilise une variable P

pointant vers une ville : P :  $\uparrow$  Ville ;, ensuite il faut suivre les étapes suivantes :

- La variable P sera utilisée pour garder l'adresse du premier élément :  $P \leftarrow Tete$  ;
- Maintenant, on met à jour l'adresse de la nouvelle tête de la liste :  $Tete \leftarrow Tete \uparrow .Suivant$  ;
- Enfin, on libère l'espace mémoire réservé par l'ancienne tête :  $Désallouer(p)$  ;

Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de la procédure :

Opération	Représentation en mémoire
$P \leftarrow \text{Tete} ;$	 <p>The diagram shows a linked list with three nodes. The first node is labeled 'Tete' and contains '...'. A pointer 'P' points to this node. The 'Tete' node's 'Suivant' pointer points to the second node, which also contains '...'. The second node's 'Suivant' pointer points to the third node, which contains '...'.</p>
$\text{Tete} \leftarrow \text{Tete} \uparrow .\text{Suivant} ;$	 <p>The diagram shows the same linked list structure. The 'Tete' pointer is now updated to point to the second node (the one previously pointed to by the 'Tete' node's 'Suivant' pointer). The original 'Tete' node is still present but its 'Suivant' pointer is no longer followed by the 'Tete' pointer.</p>
Désallouer(p) ;	 <p>The diagram shows the linked list with the 'Tete' pointer pointing to the second node. A large 'X' is drawn over the first node (the one previously pointed to by 'Tete') and its 'Suivant' pointer, indicating that this node and its link are being deallocated. The 'Tete' pointer remains pointing to the second node, which now becomes the first node in the list.</p>

La procédure permettant la suppression de la tête de la liste est alors la suivante :

```
Procedure Supprimer_T ;
```

## Variables

P : ↑ Ville ;

Début

Si Tete <> nil Alors

Si Tete↑.Suivant = nil Alors début

Désallouer(Tete);

```
Tete ← nil;
```

Queue  $\leftarrow$  nil;

fin

Sinon début

P ← Tete ;

```
Tete ← Tete↑.Suivant ;
```

```

        Désallouer(P) ;
    fin ;
Fin ;

```

En Pascal :

```

procédure Supprimer_T ;
var
    P : Ptr_Ville ;
Begin
    if Tete <> nil then
        if Tete = Queue then begin
            dispose(Tete);
            Tete := nil;
            Queue := nil;
        end
        else begin
            P := Tete ;
            Tete := Tete^.Suivant ;
            dispose(P) ;
        end ;
    end ;
end ;

```

**Remarque :** Avant de traiter le cas général, nous avons traité les deux cas triviaux : liste vide et liste contenant un seul élément.

**Exercice :**

Ecrire une procédure permettant de vider une liste de villes, et récupérer l'espace mémoire occupé par les éléments de cette liste en utilisant la procédure de suppression de la tête décrite ci-dessus.

**Solution :**

```

Procédure Vider_Liste ;
Début
    Tant que (Tete <> nil) Faire Supprimer_T ;
Fin ;

```

En Pascal :

```

procédure Vider_Liste ;
begin
    while (Tete <> nil) do Supprimer_T ;
end ;

```

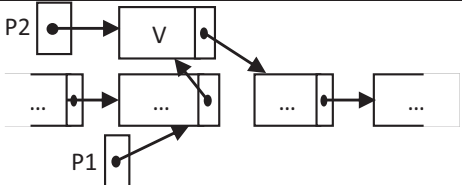
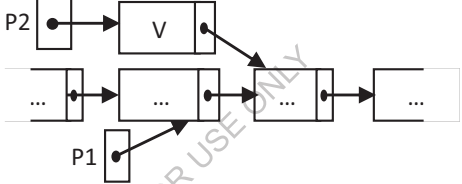
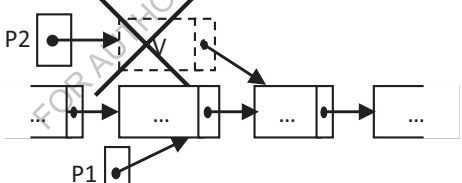
### 5.5. Supprimer un élément de la liste

La procédure permettant la suppression d'un élément de la liste possède un paramètre V indiquant la ville à supprimer. Dans cette procédure, on aura besoin de déclarer deux variables P1 et P2 pointant vers une ville : P1, P2 : ↑ Ville ;. Le pointeur P2 va pointer vers l'élément à supprimer.

Le pointeur P1 pointe juste avant P2. Pour que P1 et P2 prennent leurs positions, on doit parcourir la liste à partir de sa tête. Après ça, on doit suivre les étapes suivantes :

- Le successeur de P1 devient le successeur de P2 :  $P1 \uparrow .\text{Suivant} \leftarrow P2 \uparrow .\text{Suivant}$  ;
- Libérer l'espace mémoire occupé par l'élément supprimé :  $\text{Désallouer}(P2)$  ;

Le tableau suivant décrit ces deux dernières étapes :

Opération	Représentation en mémoire
{Avant la suppression}	
$P1 \uparrow .\text{Suivant} \leftarrow P2 \uparrow .\text{Suivant}$ ;	
Désallouer(P2);	

N'oubliez pas de mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste. La procédure permettant de supprimer un élément de la liste est alors la suivante :

Procédure Supprimer (V : chaîne de caractères) ;

Variables

P1, P2 :  $\uparrow$  Ville;

Début

Si Tete $\uparrow$ .Des = V Alors Supprimer\_T

Sinon début

{ Pointer vers l'élément contenant V }

P1  $\leftarrow$  Tete;

P2  $\leftarrow$  P1 $\uparrow$ .Suivant ;

Tant que (P2 $\uparrow$ .Des  $\neq$  V) Faire début

```

    P1 ← P2;
    P2 ← P2↑.Suivant;
  fin ;
{ Mettre à jour la queue dans le cas où l'élément à supprimer est le dernier
de la liste }
  Si Queue = P2 Alors Queue ← P1 ;
  { Supprimer l'élément contenant V }
  P1↑.Suivant ← P2↑.Suivant ;
  Désallouer(P2) ;
  fin ;
Fin ;

```

En Pascal :

```

procedure Supprimer (V : string) ;
var
  P1, P2 : Ptr_Ville;
begin
  if Tete↑.Des = V then Supprimer_T
  else begin
    { Pointer vers l'élément contenant V }
    P1 := Tete;
    P2 := P1^.Suivant ;
    while (P2^.Des <> V) do begin
      P1 := P2;
      P2 := P2^.Suivant;
    end ;
    { Mettre à jour la queue dans le cas où l'élément à supprimer est le dernier
    de la liste }
    if Queue = P2 then Queue := P1 ;
    { Supprimer l'élément contenant V }
    P1^.Suivant := P2^.Suivant ;
    dispose(P2) ;
  end;
end ;

```

**Remarque :**

Dans la procédure ci-dessus, on suppose que V existe obligatoirement dans la liste. Il est possible de traiter le cas où V n'existe pas dans la liste par un message d'erreur.

### 5.6. Afficher les éléments de la liste

Pour consulter la liste chaînée que nous avons créée précédemment, il faut se positionner au début de la liste, puis la parcourir élément par élément jusqu'à la fin.

Procédure Afficher\_Liste ;

Variables

P : ↑ Ville;

j : entier ;

début

P ← Tete;

j ← 1;

Tant que (P <> nil) Faire début

Ecrire('L'élément num° ', j, ' de la liste est : ', P↑.Des) ;

j ← j + 1 ;

P ← P↑.Suivant;

fin;

fin ;

On utilise alors un pointeur P vers une ville qui se positionne initialement au début de la liste : P ← Tete;. On affiche le contenu de l'élément courant (P↑.Des) et on passe à l'élément suivant : P ← P↑.Suivant;. Ces deux dernières opérations se répètent jusqu'à atteindre la fin de la liste.

En Pascal :

procedure Afficher\_Liste;

var

P : Ptr\_Ville;

j : integer ;

begin

P := Tete;

j := 1;

while (P <> nil) do begin

writeln('L'élément num° ', j, ' de la liste est : ', P<sup>^</sup>.Des) ;

j := j + 1 ;

P := P<sup>^</sup>.Suivant;

end;

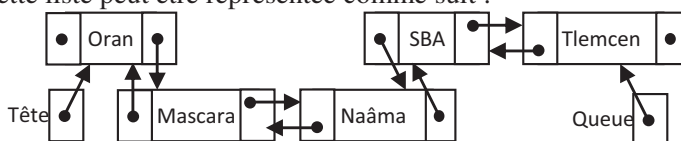
end ;

### 6. Les listes doublement chaînées

Les listes chaînées traitées précédemment sont dites simples parce qu'elles sont parcourues dans un seul sens. Il est possible de parcourir

une liste dans les deux sens en ajoutant un pointeur permettant l'accès à l'élément précédent. On obtient alors une liste doublement chaînée appelée aussi liste bidirectionnelle, contrairement à une liste simplement chaînée ou unidirectionnelle vue précédemment. Cette méthode est coûteuse en espace mémoire, pour cela elle n'est utilisée que lorsqu'on a besoin d'effectuer un retour en arrière vers la tête de la liste.

Si on stocke l'ensemble des villes dans une liste doublement chaînée, alors cette liste peut être représentée comme suit :



### 6.1. Créer et remplir une liste doublement chaînée

Pour créer la liste doublement chaînée représentée ci-dessus, il faut suivre les étapes suivantes :

1. Chaque élément de la liste doit contenir la désignation de la ville, plus un pointeur vers l'élément suivant, et un autre pointant vers l'élément précédent. Ce qui est traduit par la structure Ville suivante :

Type

Ville = enregistrement

Des : chaîne de caractères ;

Precedent, Suivant : ↑ Ville ;

Fin ;

2. Le début et la fin de la liste doivent être capturés successivement par deux pointeurs : Tete et Queue. Cela facilite le parcours de la liste de gauche à droite (de la tête vers la queue) et de la droite vers la gauche (de la queue vers la tête). Ces deux variables sont déclarées comme suit :

Variables

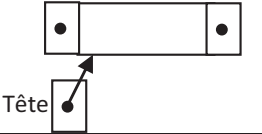
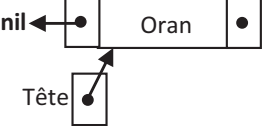
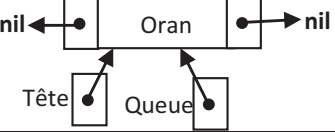
Tete, Queue : ↑ Ville ;

3. Le premier élément de la liste peut être créé comme suit :

- On réserve un espace mémoire pour une variable pointée par Tete : Allouer(Tete) ;
- Remplir le premier élément de la liste :  $Tete \uparrow .Des \leftarrow 'Oran'$  ; Le premier élément n'a pas de précédent, ce qui se traduit par :  $Tete \uparrow .Precedent \leftarrow nil$  ;
- A ce niveau, le premier élément et au même temps le dernier. Par conséquent, la queue pointe vers la tête :  $Queue \leftarrow Tete$  ; Le successeur de la queue est le nil :  $Queue \uparrow .Suivant \leftarrow nil$  ;



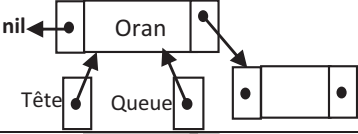
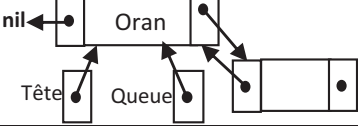
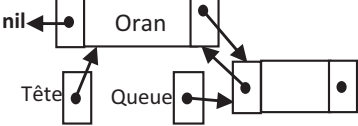
Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de l'algorithme :

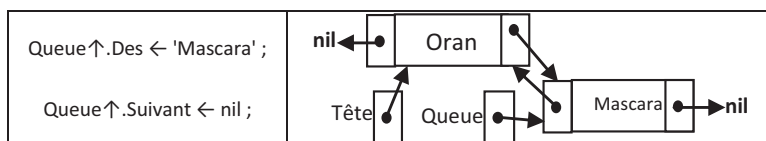
Opération	Représentation en mémoire
Alouer(Tete) ;	
Tete↑.Des ← 'Oran' ; Tete↑.Precedent ← nil ;	
Queue ← Tete ; Queue↑.Suivant ← nil ;	

4. L'ajout d'un deuxième élément à la fin liste doublement chaînée se fait comme suit :

- On réserve un espace mémoire pour une variable pointée par le successeur de la queue : Allouer(Queue↑.Suivant);
- Le nouvel élément créé sera précédé par la queue : Queue↑.Suivant↑.Precedent ← Queue ;
- La queue va pointer maintenant vers son successeur : Queue ← Queue↑.Suivant;
- Remplir le dernier élément de la liste : Queue↑.Des ← 'Mascara' ;  
Le successeur de la queue est le nil : Queue↑.Suivant ← nil ;

Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de l'algorithme :

Opération	Représentation en mémoire
Allouer(Queue↑.Suivant);	
Queue↑.Suivant↑.Precedent ← Queue ;	
Queue ← Queue↑.Suivant;	



5. Pour ajouter le troisième, le quatrième et le cinquième élément, il suffit de répéter l'étape 4, en substituant le remplissage du dernier élément de la liste successivement par  $\text{Queue} \uparrow .\text{Des} \leftarrow \text{'Saida'}$  ;  $\text{Queue} \uparrow .\text{Des} \leftarrow \text{'SBA'}$  ; et  $\text{Queue} \uparrow .\text{Des} \leftarrow \text{'Tlemcen'}$  ;. Il est donc possible de substituer l'étape 4 par une procédure permettant l'ajout d'un élément à la fin de la liste. La procédure possède comme paramètre le nom de la ville à ajouter, et elle est écrite comme suit :

Procédure Ajouter\_Q (V : chaîne de caractères) ;

Début

```

Allouer(Queue↑.Suivant);
Queue↑.Suivant↑.Précédent ← Queue ;
Queue ← Queue↑.Suivant;
Queue↑.Des ← V ;
Queue↑.Suivant ← nil ;

```

Fin ;

L'algorithme complet pour la création de la liste doublement chaînée est alors le suivant :

Algorithme Villes ;

Type

```

Ville = Enregistrement
  Des : chaîne de caractères ;
  Précédent, Suivant : ↑ Ville;
fin ;

```

Variables

```

Tete, Queue : ↑ Ville;
(*Procédure d'ajout d'un élément à la fin de la liste*)

```

Procédure Ajouter\_Q (V : chaîne de caractères) ;

début

```

Allouer(Queue↑.Suivant);
Queue↑.Suivant↑.Précédent ← Queue ;
Queue ← Queue↑.Suivant;
Queue↑.Des ← V ;
Queue↑.Suivant ← nil ;

```

fin ;

Début

```
(*Créer et remplir le premier élément de la liste*)
Allouer(Tete) ;
Tete↑.Des ← 'Oran' ;
Tete↑.Precedent ← nil ;
Queue ← Tete ;
Queue↑.Suivant ← nil ;
(* Ajouter les autres éléments de la liste *)
Ajouter_Q('Mascara') ;
Ajouter_Q('Saida') ;
Ajouter_Q('SBA') ;
Ajouter_Q('Tlemcen') ;
Fin.
```

Le programme Pascal :

```
program Villes ;
type
  Ptr_Ville = ^ Ville ;
  Ville = record
    Des : string ;
    Precedent, Suivant : Ptr_Ville;
  end ;
var
  Tete, Queue : Ptr_Ville ;
(*Procédure d'ajout d'un élément à la fin de la liste*)
procedure Ajouter_Q (V : String) ;
begin
  new(Queue↑.Suivant);
  Queue↑.Suivant↑.Precedent := Queue ;
  Queue := Queue↑.Suivant;
  Queue↑.Des := V ;
  Queue↑.Suivant := nil ;
end ;
begin
  (*Créer et remplir le premier élément de la liste*)
  new(Tete);
  Tete↑.Des := 'Oran' ;
  Tete↑.Precedent := nil ;
  Queue := Tete ;
  Queue↑.Suivant := nil ;
  (*Ajouter les autres éléments de la liste*)
```

```
Ajouter_Q('Mascara') ;  
Ajouter_Q('Saida') ;  
Ajouter_Q('SBA') ;  
Ajouter_Q('Tlemcen') ;  
end.
```

**6.2. Ajouter un élément au début de la liste doublement chaînée**

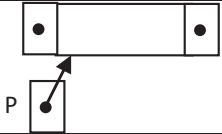
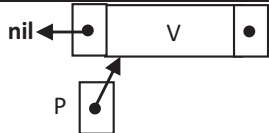
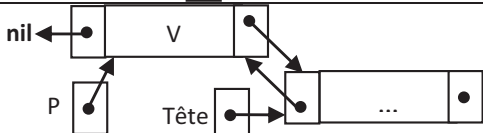
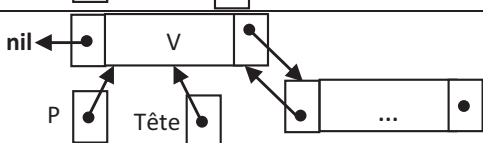
La procédure permettant l'ajout d'un élément en tête de la liste possède comme paramètre le nom de la ville à ajouter :

Procédure Ajouter\_T (V : chaîne de caractères) ;

Les étapes à suivre pour ajouter un élément au début de la liste doublement chaînée sont les suivantes :

- On utilise une variable P pointant vers une ville pour laquelle on réserve un espace mémoire par : Allouer(P);
- Remplir le nouvel élément pointé par P :  $P \uparrow .Des \leftarrow V$  ;, sans oublier que le précédent du nouvel élément est le nil :  $P \uparrow .Precedent \leftarrow nil$  ;
- On effectue le chaînage du nouvel élément avec la liste existante. Le successeur de P sera alors la tête :  $P \uparrow .Suivant \leftarrow Tete$  ;. Le précédent de la tête sera le nouvel élément :  $Tete \uparrow .Precedent \leftarrow P$  ;
- On met à jour l'adresse de la tête de la nouvelle liste :  $Tete \leftarrow P$  ;

Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de la procédure :

Opération	Représentation en mémoire
Allouer(P);	
$P \uparrow .Des \leftarrow V$ ; $P \uparrow .Precedent \leftarrow nil$ ;	
$P \uparrow .Suivant \leftarrow Tete$ ; $Tete \uparrow .Precedent \leftarrow P$ ;	
$Tete \leftarrow P$ ;	

La procédure permettant l'ajout d'un élément au début d'une liste doublement chaînée est alors la suivante :

Procédure Ajouter\_T (V : chaîne de caractères) ;

Variables

P : ↑ Ville ;

Début

Allouer(P);

P↑.Des ← V ;

P↑.Precedent ← nil ;

P↑.Suivant ← Tete ;

Si Tete <> nil Alors Tete↑.Precedent ← P;

Tete ← P;

Si Queue = nil Alors Queue ← Tete;

Fin ;

En Pascal :

procedure Ajouter\_T (V : string) ;

var

P : Ptr\_Ville ;

begin

new(P);

P^.Des := V ;

P^.Precedent := nil ;

P^.Suivant := Tete ;

if Tete <> nil then Tete^.Precedent := P;

Tete := P;

if Queue = nil then Queue := Tete;

end ;

### Exercice :

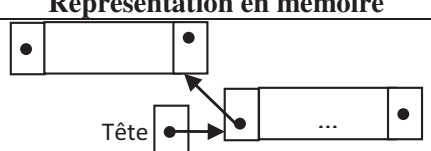
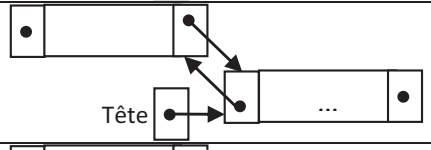
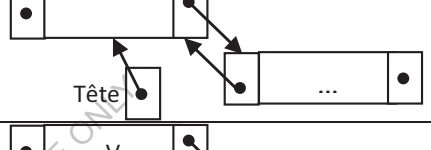
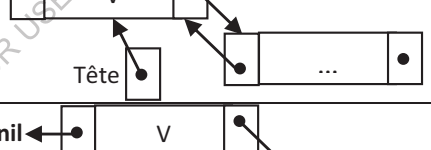
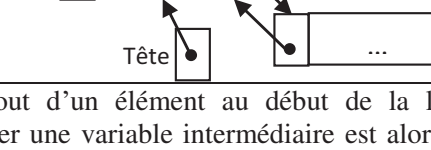
Reprendre la procédure Ajouter\_T, mais cette fois-ci sans utiliser la variable P.

### Solution :

Il est possible d'ajouter un élément en tête de la liste doublement chaînée sans utiliser une variable intermédiaire, et cela comme suit :

- On alloue un espace mémoire pour le précédent de la tête :  
Allouer(Tete↑.Precedent);
- Le successeur du nouvel élément devient Tete :  
Tete↑.Precedent↑.Suivant ← Tete ;
- On met à jour l'adresse de la tête de la nouvelle liste : Tete ← Tete↑.Precedent ;

- Remplir le nouvel élément (la nouvelle tête) par :  $Tete \uparrow .Des \leftarrow V$  ;
  - Le précédent du nouvel élément est le nil :  $Tete \uparrow .Precedent \leftarrow nil$  ;
- Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de la procédure :

Opération	Représentation en mémoire
$Allouer(Tete \uparrow .Precedent);$	
$Tete \uparrow .Precedent \uparrow .Suivant \leftarrow Tete ;$	
$Tete \leftarrow Tete \uparrow .Precedent ;$	
$Tete \uparrow .Des \leftarrow V ;$	
$Tete \uparrow .Precedent \leftarrow nil ;$	

La procédure permettant l'ajout d'un élément au début de la liste doublement chaînée sans utiliser une variable intermédiaire est alors la suivante :

Procédure Ajouter\_T (V : chaîne de caractères) ;

Début

Si (Tete <> nil) Alors début

$Allouer(Tete \uparrow .Precedent);$

$Tete \uparrow .Precedent \uparrow .Suivant \leftarrow Tete ;$

$Tete \leftarrow Tete \uparrow .Precedent ;$

$Tete \uparrow .Des \leftarrow V ;$

$Tete \uparrow .Precedent \leftarrow nil ;$

fin

else début

$Allouer(Tete);$

```
Tete↑.Des ← 'Oran' ;  
Tete↑.Precedent ← nil ;  
Queue ← Tete ;  
Queue↑.Suivant ← nil ;  
fin;  
Fin ;
```

En Pascal :

```
procedure Ajouter_T (V : string) ;  
begin  
  if (Tete <> nil) then begin  
    new(Tete^.Precedent);  
    Tete^.Precedent^.Suivant := Tete ;  
    Tete := Tete^.Precedent ;  
    Tete^.Des := V ;  
    Tete^.Precedent := nil ;  
  end  
  else begin  
    new(Tete);  
    Tete^.Des := 'Oran' ;  
    Tete^.Precedent := nil ;  
    Queue := Tete ;  
    Queue^.Suivant := nil ;  
  end;  
end ;
```

### 6.3. Insérer un élément dans la liste doublement chaînée

La procédure permettant d'insérer un élément après un autre dans une liste doublement chaînée possède deux paramètres : V1 et V2. V2 étant le nom de la ville à insérer, et V1 le nom de la ville après laquelle on désire insérer V2. Pour ce faire, on aura besoin de déclarer deux variables P1 et P2 pointant vers une ville : P1, P2 : ↑ Ville ;. Le pointeur P2 va pointer vers l'élément à insérer (un nouvel élément contenant la ville V2). Le pointeur P1 va nous permettre de localiser la ville V1 dans la liste. Il va donc pointer vers l'élément dont la désignation est V1. Pour que P1 et P2 prennent leurs positions, on doit parcourir la liste à partir de sa tête. Après ça, on doit suivre les étapes suivantes :

- Le successeur de P2 devient le successeur de P1 :  $P2↑.Suivant \leftarrow P1↑.Suivant$  ;
- Le précédent de P2 devient P1 :  $P2↑.Precedent \leftarrow P1$  ;

- Le successeur de P1 devient P2 :  $P1 \uparrow .\text{Suivant} \leftarrow P2$ , et pour terminer l’insertion, on met :  $P2 \uparrow .\text{Suivant} \uparrow .\text{Precedent} \leftarrow P2$  ;
- Le tableau suivant décrit ces dernières étapes :

Opération	Représentation en mémoire
{Avant l’insertion}	
$P2 \uparrow .\text{Suivant} \leftarrow P1 \uparrow .\text{Suivant}$ ;	
$P2 \uparrow .\text{Precedent} \leftarrow P1$ ;	
$P1 \uparrow .\text{Suivant} \leftarrow P2$ ; $P2 \uparrow .\text{Suivant} \uparrow .\text{Precedent} \leftarrow P2$ ;	

La procédure permettant d’insérer un élément après un autre dans une liste doublement chaînée est la suivante :

```
Procédure Insérer_AP_Ville (V1, V2 : chaîne de caractères) ;
Variables P1, P2 : ↑ Ville;
Début
Si Queue↑.Des = V1 Alors Ajouter_Q(V2)
Sinon début
{ Pointer vers l’élément contenant V1 }
P1 ← Tete;
Tant que (P1↑.Des <> V1) Faire P1 ← P1↑.Suivant;
```



```

{ Créer un nouvel élément contenant V2 }
Allouer(P2);
P2↑.Des ← V2 ;
{ Insérer le nouvel élément dans la liste }
P2↑.Suivant ← P1↑.Suivant ;
P2↑.Precedent ← P1 ;
P1↑.Suivant ← P2;
P2↑.Suivant↑.Precedent ← P2 ;
fin ;
Fin ;

```

En Pascal :

```

procedure Insérer_AP_Ville (V1, V2 : string) ;
var P1, P2 : Ptr_Ville;
begin
if Queue^.Des = V1 then Ajouter_Q(V2)
else begin
{ Pointer vers l'élément contenant V1 }
P1 := Tete;
while (P1^.Des <> V1) do P1 := P1^.Suivant;
{ Créer un nouvel élément contenant V2 }
new(P2);
P2^.Des := V2 ;
{ Insérer le nouvel élément dans la liste }
P2^.Suivant := P1^.Suivant ;
P2^.Precedent := P1 ;
P1^.Suivant := P2;
P2^.Suivant^.Precedent := P2 ;
end ;
end ;

```

**Remarque :**

Dans la procédure ci-dessus, on suppose que V1 existe obligatoirement dans la liste. Il est possible de traiter le cas où V1 n'existe pas dans la liste par un message d'erreur.

**Exercice :**

Reprendre la procédure Insérer\_AP\_Ville, mais cette fois-ci en utilisant un seul pointeur intermédiaire.

**Solution :**

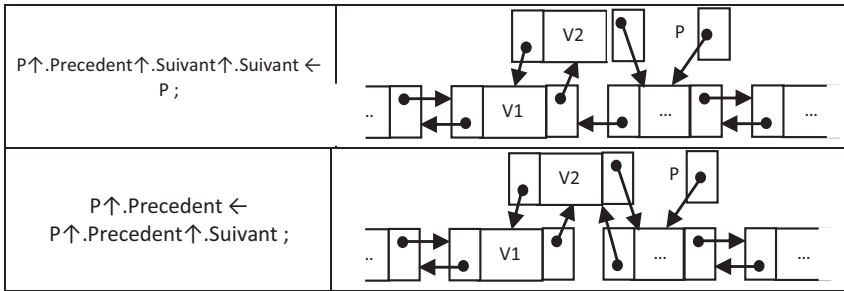
Au lieu d'utiliser deux pointeurs pour insérer un élément après un autre, comme c'est le cas pour les listes simplement chaînées, dans une liste doublement chaînée, il est possible d'utiliser tout simplement un seul

pointeur P pointant vers une ville :  $P : \uparrow \text{ Ville } ;$ . Le pointeur P va pointer vers le successeur de l'élément après lequel on désire faire l'insertion. Le pointeur P va donc nous permettre de localiser le successeur de l'élément contenant la ville V1 dans la liste. Il va donc pointer après l'élément dont la désignation est V1. Pour que P prenne sa position, on doit parcourir la liste à partir de sa tête. Après ça, on doit suivre les étapes suivantes :

- Allouer un espace mémoire pour le précédent de P :  $\text{Allouer}(P \uparrow .\text{Precedent} \uparrow .\text{Suivant}) ;$
- Remplir le nouvel élément créé :  $P \uparrow .\text{Precedent} \uparrow .\text{Suivant} \uparrow .\text{Des} \leftarrow V2 ;$
- Le précédent du nouvel élément créé est le précédent du P :  $P \uparrow .\text{Precedent} \uparrow .\text{Suivant} \uparrow .\text{Precedent} \leftarrow P \uparrow .\text{Precedent} ;$
- Le suivant du nouvel élément créé est le pointeur P :  $P \uparrow .\text{Precedent} \uparrow .\text{Suivant} \uparrow .\text{Suivant} \leftarrow P ;$
- Le précédent de P devient le nouvel élément créé :  $P \uparrow .\text{Precedent} \leftarrow P \uparrow .\text{Precedent} \uparrow .\text{Suivant} ;$

Le tableau suivant décrit ces dernières étapes :

Opération	Représentation en mémoire
{Avant l'insertion}	
Allouer( $P \uparrow .\text{Precedent} \uparrow .\text{Suivant}$ );	
$P \uparrow .\text{Precedent} \uparrow .\text{Suivant} \uparrow .\text{Des} \leftarrow V2 ;$	
$P \uparrow .\text{Precedent} \uparrow .\text{Suivant} \uparrow .\text{Precedent} \leftarrow P \uparrow .\text{Precedent} ;$	



La procédure permettant l'insérer un élément après un autre dans une liste doublement chaînée en utilisant un seul pointeur est la suivante :

Procédure Insérer\_AP\_Ville (V1, V2 : chaîne de caractères) ;

Variables

P :  $\uparrow$  Ville;

Début

Si Queue $\uparrow$ .Des = V1 Alors Ajouter\_Q(V2)

Sinon début

{ Pointer après l'élément contenant V1 }

P  $\leftarrow$  Tete;

Tant que (P $\uparrow$ .Des  $\neq$  V1) Faire P  $\leftarrow$  P $\uparrow$ .Suivant;

P  $\leftarrow$  P $\uparrow$ .Suivant;

{ Créer un nouvel élément contenant V2 }

Allouer(P $\uparrow$ .Precedent $\uparrow$ .Suivant) ;

P $\uparrow$ .Precedent $\uparrow$ .Suivant $\uparrow$ .Des  $\leftarrow$  V2 ;

{ Insérer le nouvel élément dans la liste }

P $\uparrow$ .Precedent $\uparrow$ .Suivant $\uparrow$ .Precedent  $\leftarrow$  P $\uparrow$ .Precedent ;

P $\uparrow$ .Precedent $\uparrow$ .Suivant $\uparrow$ .Suivant  $\leftarrow$  P ;

P $\uparrow$ .Precedent  $\leftarrow$  P $\uparrow$ .Precedent $\uparrow$ .Suivant ;

fin ;

Fin ;

En Pascal :

procedure Insérer\_AP\_Ville (V1, V2 : string) ;

var

P : Ptr\_Ville;

begin

if Queue $\wedge$ .Des = V1 then Ajouter\_Q(V2)

else begin

{ Pointer après l'élément contenant V1 }

P := Tete;

while (P $\wedge$ .Des  $\neq$  V1) do P := P $\wedge$ .Suivant;

```
P := P^.Suivant;  
{ Créer un nouvel élément contenant V2 }  
new(P^.Precedent^.Suivant);  
P^.Precedent^.Suivant^.Des := V2 ;  
{ Insérer le nouvel élément dans la liste }  
P^.Precedent^.Suivant^.Precedent := P^.Precedent ;  
P^.Precedent^.Suivant^.Suivant := P ;  
P^.Precedent := P^.Precedent^.Suivant ;  
end ;  
end ;
```

**6.4. Supprimer la tête de la liste doublement chaînée**

Pour supprimer la tête de la liste doublement chaînée, et récupérer l'espace mémoire occupé par cet élément, il faut suivre les étapes suivantes :

- On change la tête de la liste par :  $Tete \leftarrow Tete \uparrow .Suivant$  ;
- On libère l'espace mémoire occupé par le précédent de tête :  $Désallouer(Tete \uparrow .Precedent)$  ;
- Le précédent de la tête est le nil :  $Tete \uparrow .Precedent \leftarrow nil$  ;

Le tableau suivant décrit le changement d'état de la mémoire selon les opérations de la procédure :

Opération	Représentation en mémoire
$Tete \leftarrow Tete \uparrow .Suivant$ ;	
$Désallouer(Tete \uparrow .Precedent)$ ;	
$Tete \uparrow .Precedent \leftarrow nil$ ;	

La procédure permettant la suppression de la tête de la liste est alors la suivante :

```
Procédure Supprimer_T ;  
Début  
  Si (Tete <> nil) Alors
```

```

Si (Tete↑.Suivant = nil) Alors début
    Désallouer(Tete) ;
    Tete ← nil ;
    Queue ← nil ;
fin
Sinon début
    Tete ← Tete↑.Suivant ;
    Désallouer(Tete↑.Precedent) ;
    Tete↑.Precedent ← nil ;
fin ;
Fin ;

```

En Pascal :

```

procedure Supprimer_T ;
begin
    if (Tete <> nil) then
        if (Tete^.Suivant = nil) then begin
            dispose(Tete);
            Tete := nil;
            Queue := nil;
        end
        else begin
            Tete := Tete^.Suivant ;
            dispose(Tete^.Precedent) ;
            Tete^.Precedent := nil ;
        end ;
    end ;
end ;

```

### 6.5. Supprimer un élément de la liste doublement chaînée

La procédure permet la suppression d'un élément de la liste possède un paramètre V indiquant la ville à supprimer. Dans cette procédure, on utilise un pointeur P pointant vers une ville :  $P : \uparrow \text{ Ville}$  ;. Le pointeur P va pointer vers l'élément à supprimer. Pour que P prend sa position, on doit parcourir la liste à partir de sa tête. Après ça, on doit suivre les étapes suivantes :

- Le successeur du précédent de P devient le successeur de P :  $P \uparrow . \text{Precedent} \uparrow . \text{Suivant} \leftarrow P \uparrow . \text{Suivant}$  ;
- Le précédent du successeur de P (s'il existe) devient le précédent de P :  $P \uparrow . \text{Suivant} \uparrow . \text{Precedent} \leftarrow P \uparrow . \text{Precedent}$  ;
- Libérer l'espace mémoire occupé par l'élément supprimé :  $\text{Désallouer}(P)$  ;

Le tableau suivant décrit ces étapes :

Opération	Représentation en mémoire
{Avant la suppression}	
$P \uparrow .\text{Precedent} \leftarrow P \uparrow .\text{Suivant} ;$	
$P \uparrow .\text{Suivant} \leftarrow P \uparrow .\text{Precedent} ;$	
Désallouer(P) ;	

N'oubliez pas de mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste doublement chaînée. La procédure permettant de supprimer un élément de la liste est alors la suivante :

Procédure Supprimer (V : chaîne de caractères) ;

Variables P :  $\uparrow$  Ville;

Début

Si Tete $\uparrow$ .Des = V Alors Supprimer\_T

Sinon début

{ Pointer vers l'élément contenant V }

P  $\leftarrow$  Tete;

Tant que (P $\uparrow$ .Des  $\neq$  V) Faire P  $\leftarrow$  P $\uparrow$ .Suivant ;

{ Mettre à jour la queue dans le cas où l'élément à supprimer est le dernier de la liste }

Si Queue = P Alors Queue  $\leftarrow$  P $\uparrow$ .Precedent ;

{ Supprimer l'élément contenant V }

P $\uparrow$ .Precedent $\uparrow$ .Suivant  $\leftarrow$  P $\uparrow$ .Suivant ;

Si (P $\uparrow$ .Suivant  $\neq$  nil) Alors P $\uparrow$ .Suivant $\uparrow$ .Precedent  $\leftarrow$  P $\uparrow$ .Precedent ;

Désallouer(P) ;

fin ;

Fin ;

En Pascal :

```
procedure Supprimer (V : string) ;
var P : Ptr_Ville;
begin
  if Tete^.Des = V then Supprimer_T
  else begin
    { Pointer vers l'élément contenant V }
    P := Tete;
    while (P^.Des <> V) do P := P^.Suivant;
  { Mettre à jour la queue dans le cas où l'élément à supprimer est le
  dernier de la liste }
    if Queue = P then Queue := P^.Precedent ;
    { Supprimer l'élément contenant V }
    P^.Precedent^.Suivant := P^.Suivant ;
    if (P^.Suivant <> nil) then P^.Suivant^.Precedent := P^.Precedent ;
    dispose(P) ;
  end;
end ;
```

**Remarque :**

Dans la procédure ci-dessus, on suppose que V existe obligatoirement dans la liste. Il est possible de traiter le cas où V n'existe pas dans la liste par un message d'erreur.

**6.6. Afficher les éléments de la liste doublement chaînée**

Une fois créée, la liste doublement chaînée peut être parcourue de gauche à droite (de la tête vers la queue) ou de la droite vers la gauche (de la queue vers la tête). Le principe est le même que pour les listes simplement chaînées.

La procédure permettant d'afficher la liste doublement chaînée créée ci-dessus de gauche à droite est la suivante :

```
Procédure Afficher_Liste_T ;
Variables
  P : ↑ Ville;
  j : entier ;
début
  P ← Tete;
  j ← 1;
  Tant que (P <> nil) Faire début
    Ecrire('L'élément num° ', j, ' de la liste est : ', P↑.Des) ;
    j ← j + 1 ;
```

```
P ← P↑.Suivant;  
fin;  
fin ;
```

On utilise alors un pointeur P vers une ville qui se positionne initialement au début de la liste :  $P \leftarrow Tete$ ;. On affiche le contenu de l'élément courant ( $P\uparrow.Des$ ), et on passe à l'élément suivant :  $P \leftarrow P\uparrow.Suivant$ ;. Ces deux dernières opérations se répètent jusqu'à atteindre la fin de la liste.

En Pascal :

```
procedure Afficher_Liste_T ;  
var  
  P : Ptr_Ville;  
  j : integer ;  
begin  
  P := Tete;  
  j := 1;  
  while (P <> nil) do begin  
    writeln('L'élément num° ', j, ' de la liste est : ', P^.Des) ;  
    j := j + 1 ;  
    P := P^.Suivant;  
  end;  
end ;
```

La procédure permettant d'afficher la liste doublement chaînée créée ci-dessus de droite à gauche est la suivante :

```
Procedure Afficher_Liste_Q ;  
Variables  
  P : ↑ Ville;  
  j : entier ;  
début  
  P ← Queue;  
  j ← 1;  
  Tant que (P <> nil) Faire début  
    Ecrire('L'élément num° ', j, ' de la liste est : ', P↑.Des) ;  
    j ← j + 1 ;  
    P ← P↑.Precedent;  
  fin;  
fin ;
```

On utilise alors un pointeur P vers une ville qui se positionne initialement à la fin de la liste :  $P \leftarrow Queue$ ;. On affiche le contenu de



l'élément courant ( $P \uparrow .Des$ ), et on passe à l'élément précédent :  $P \leftarrow P \uparrow .Precedent$ ; Ces deux dernières opérations se répètent jusqu'à atteindre le début de la liste.

Voici la procédure Afficher\_Liste\_Q en Pascal :

```

procedure Afficher_Liste_Q ;
var
  P : Ptr_Ville;
  j : integer ;
begin
  P := Queue;
  j := 1;
  while (P <> nil) do begin
    writeln('L'élément num° ', j, ' de la liste est : ', P^.Des) ;
    j := j + 1 ;
    P := P^.Precedent;
  end;
end ;

```

## 7. Les listes chaînées particulières

Il est possible de conditionner les méthodes d'accès aux éléments d'une liste simplement ou doublement chaînée pour l'adapter à des applications spécifiques, ce qui révèle un ensemble de listes chaînées particulières, à savoir les piles et les files.

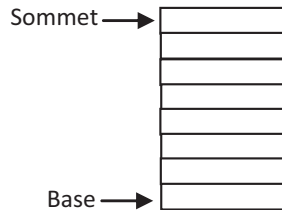
Il est aussi possible de simuler les piles et les files par des tableaux, sauf que l'inconvénient de ces derniers est qu'ils sont limités par leur taille et ne peuvent être étendus en cas de besoin.

### 7.1. Les piles

Une pile est une liste dont les éléments ne peuvent être ajoutés ou supprimés que par une extrémité appelée Sommet de la pile. Quand on ajoute un élément, celui-ci devient le sommet de la pile, c'est-à-dire le seul élément accessible. Quand on retire un élément de la pile, on retire toujours le sommet, et le dernier élément ajouté avant lui devient alors le sommet de la pile. Ceci signifie que les éléments sont retirés de la pile dans l'ordre inverse dans lequel ils ont été ajoutés : dernier entré, premier sorti. En anglais on dira, Last In First Out, plus connu sous le nom LIFO.

La structure de pile est utilisée pour sauvegarder temporairement des informations en respectant leur ordre d'entrée, et les réutiliser en ordre inverse. Une bonne image de cette structure de données est la pile d'assiettes : on peut rajouter une assiette dans une pile d'assiettes, mais on ne peut retirer que celle du dessus sans risquer de tout casser.

La pile peut être représentée tout simplement comme suit :



Le premier élément de la pile est appelé Sommet de la pile, le dernier est appelé Base de la pile.

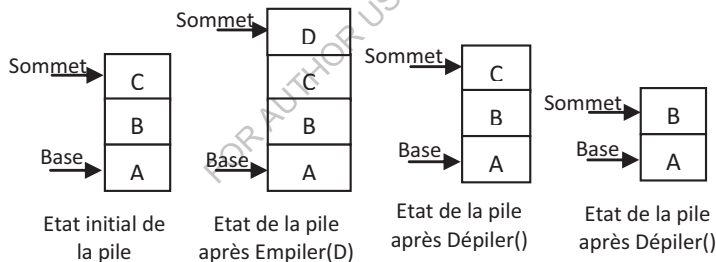
### 7.1.1. Primitives d'accès

On définit habituellement les deux primitives suivantes pour accéder à une pile :

- Empiler(E) qui ajoute un nouvel élément E au sommet de la pile.
- Dépiler() qui supprime l'élément Sommet de la pile

#### Exemple :

Dans une pile de caractères contenant initialement les éléments : A, B et C, on désire empiler encore le caractère D, ensuite dépiler deux caractères. Ces opérations peuvent être représentées comme suit :



### 7.1.2. Représentation d'une pile par une liste doublement chaînée

Une pile peut être représentée par un tableau, une liste simplement ou doublement chaînée. Dans ce qui suit, nous allons choisir une liste doublement chaînée pour représenter une pile :

- L'opération d'empilement se traduit par un ajout en tête de la liste doublement chaînée.
- L'opération de dépilement se traduit par une suppression de la tête de la liste doublement chaînée.
- Le sommet et la base de la pile étant la tête et la queue de la liste doublement chaînée.

Pour la manipulation d'une pile de caractères, on doit suivre les étapes suivantes :

1. On définit la structure Pile comme suit :

Type

Pile = Enregistrement

Car : caractère ;

Precedent, Suivant :  $\uparrow$  Pile ;

fin ;

2. Ensuite, on déclare le sommet et la base de la pile comme suit :

Variables

Base, Sommet :  $\uparrow$  Pile ;

3. La procédure permettant d'empiler un élément dans la pile est la suivante :

Procédure Empiler(C : caractère) ;

Variables

P :  $\uparrow$  Pile ;

Début

Allouer(P);

$P \uparrow .Car \leftarrow C$  ;

$P \uparrow .Precedent \leftarrow nil$  ;

$P \uparrow .Suivant \leftarrow Sommet$  ;

Si (Sommet  $\neq nil$ ) Alors Sommet  $\uparrow .Precedent \leftarrow P$ ;

Sommet  $\leftarrow P$ ;

Si (Base = nil) Alors Base  $\leftarrow Sommet$  ;

Fin ;

4. La procédure permettant de dépiler un élément de la pile est la suivante :

Procédure Depiler ;

Début

Si (Sommet = nil) Alors Ecrire('Dépilement impossible : Pile vide.')

Sinon Si (Sommet  $\uparrow .Suivant = nil$ ) Alors début

Ecrire('Elément dépilé ', Sommet  $\uparrow .Car$ ) ;

Désallouer(Sommet) ;

Sommet  $\leftarrow nil$  ;

Base  $\leftarrow nil$  ;

end

Sinon début

Ecrire('Elément dépilé ', Sommet  $\uparrow .Car$ ) ;

Sommet  $\leftarrow Sommet \uparrow .Suivant$  ;

Désallouer(Sommet  $\uparrow .Precedent$ ) ;

Sommet  $\uparrow .Precedent \leftarrow nil$  ;

fin ;

Fin ;

L'algorithme de gestion de la pile est le suivant :

Algorithme Gestion\_Pile ;

Type

Pile = Enregistrement

Car : caractère ;

Precedent, Suivant :  $\uparrow$  Pile ;

fin ;

Variables

Base, Sommet :  $\uparrow$  Pile ;

Choix : entier ;

Element, reponse : caractère ;

{ Procédure permettant d'empiler un élément dans la pile }

Procédure Empiler(C : caractère) ;

Variables

P :  $\uparrow$  Pile ;

début

Allouer(P);

P $\uparrow$ .Car  $\leftarrow$  C ;

P $\uparrow$ .Precedent  $\leftarrow$  nil ;

P $\uparrow$ .Suivant  $\leftarrow$  Sommet ;

Si (Sommet  $\neq$  nil) Alors Sommet $\uparrow$ .Precedent  $\leftarrow$  P;

Sommet  $\leftarrow$  P;

Si (Base = nil) Alors Base  $\leftarrow$  Sommet ;

fin ;

{ Procédure permettant de dépiler un élément de la pile }

Procédure Depiler ;

début

Si (Sommet = nil) Alors Ecrire('Dépilement impossible : Pile vide.')

Sinon Si (Sommet $\uparrow$ .Suivant = nil) Alors début

Ecrire('Elément dépilé : ', Sommet $\uparrow$ .Car) ;

Désallouer(Sommet) ;

Sommet  $\leftarrow$  nil ;

Base  $\leftarrow$  nil ;

fin

Sinon début

Ecrire('Elément dépilé : ', Sommet $\uparrow$ .Car) ;

Sommet  $\leftarrow$  Sommet $\uparrow$ .Suivant ;

Désallouer(Sommet $\uparrow$ .Precedent) ;

Sommet $\uparrow$ .Precedent  $\leftarrow$  nil ;

```

        fin ;
fin ;
{ Procédure permettant l'affichage des éléments de la pile }
Procédure Afficher_Pile ;
Variables
    P : ↑ Pile;
Début
    Si (Sommet = nil) Alors Ecrire('Aucun élément : Pile vide.')
    Sinon début
        P ← Sommet;
        Ecrire('Les éléments de la pile cités du sommet vers la base :') ;
        Tant que (P <> nil) Faire début
            Ecrire(P↑.Car) ;
            P ← P↑.Suivant;
        fin;
    fin ;
fin ;
Début
{ Initialement la pile est vide }
Sommet ← nil ;
Base ← nil ;
Répéter
    Ecrire('Quelle opération désirez-vous ?') ;
    Ecrire('1- Empiler, 2- Depiler, 3- Afficher :') ;
    Lire(Choix) ;
    Cas Choix de
        1 : début
            Ecrire('Donnez le caractère à empiler :)') ;
            Lire(Element) ;
            Empiler(Element) ;
            fin ;
        2 : Depiler ;
        3 : Afficher_Pile
    Sinon Ecrire('Choix non accepté.') ;
    fin ;
    Ecrire('Voulez-vous un autre test ? o/n') ;
    Lire(reponse) ;
    Jusqu'à (reponse = 'n') ;
Fin.

```

Le programme Pascal est le suivant :

```

program Gestion_Pile ;
Type
  Ptr_Pile = ^ Pile ;
  Pile = record
    Car : char ;
    Precedent, Suivant : Ptr_Pile ;
  end ;
var
  Base, Sommet : Ptr_Pile ;
  Choix : integer ;
  Element, reponse : char ;
{ Procédure permettant d'empiler un élément dans la pile }
procedure Empiler (C : char) ;
var
  P : Ptr_Pile ;
begin
  new(P);
  P^.Car := C ;
  P^.Precedent := nil ;
  P^.Suivant := Sommet ;
  if (Sommet <> nil) then Sommet^.Precedent := P;
  Sommet := P;
  if (Sommet^.Suivant = nil) then Base := Sommet ;
end ;
{ Procédure permettant de dépiler un élément de la pile }
procedure Depiler ;
begin
  if (Sommet = nil) then writeln('Dépilement impossible : Pile vide.')
  else if (Sommet^.Suivant = nil) then begin
    writeln('Élément dépilé : ', Sommet^.Car) ;
    dispose(Sommet) ;
    Sommet := nil ;
    Base := nil ;
  end
  else begin
    writeln('Élément dépilé : ', Sommet^.Car) ;
    Sommet := Sommet^.Suivant ;
    dispose(Sommet^.Precedent) ;
  end
end ;

```

```
        Sommet^.Precedent := nil ;
    end ;
end ;
{ Procédure permettant l'affichage des éléments de la pile }
procedure Afficher_Pile ;
var
    P : Ptr_Pile;
begin
    if (Sommet = nil) then writeln('Aucun élément : Pile vide.')
    else begin
        P := Sommet;
        writeln('Les éléments de la pile cités du sommet vers la base :');
        while (P <> nil) do begin
            writeln(P^.Car) ;
            P := P^.Suivant;
        end ;
    end;
end ;
begin
{ Initialement la pile est vide }
Sommet := nil ;
Base := nil ;
repeat
    writeln('Quelle opération désirez-vous ?') ;
    writeln('1- Empiler, 2- Depiler, 3- Afficher :') ;
    readln(Choix) ;
    case Choix of
        1 : begin
            writeln('Donnez le caractère à empiler :') ;
            readln(Element) ;
            Empiler(Element) ;
        end ;
        2 : Depiler ;
        3 : Afficher_Pile
    else writeln('Choix non accepté.') ;
    end ;
    writeln('Voulez-vous un autre test ? o/n') ;
    readln(reponse) ;
until (reponse = 'n') ;
end.
```

**Exercice :**

Reprendre l'algorithme et le programme Pascal permettant la gestion d'une pile, mais cette fois-ci, on désire utiliser un tableau pour la mise en œuvre de cette pile.

**Solution :**

La deuxième manière pour modéliser une pile consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la pile se fera en enlevant le dernier élément du tableau.

Algorithme Gestion\_Pile ;

Constantes

max = 100 ;

Variables

Pile : Tableau[1..max] de caractère ;

Base, Sommet, Choix : entier ;

Element, reponse : caractère ;

{ Procédure permettant d'empiler un élément dans la pile }

Procédure Empiler(C : caractère) ;

début

Si (Sommet < max) Alors début

Si (Sommet <> 0) Alors début

Sommet ← Sommet + 1 ;

Pile[Sommet] ← C ;

fin

Sinon début

Sommet ← 1 ;

Pile[Sommet] ← C ;

Base ← Sommet ;

fin ;

fin

Sinon Ecrire('Empilement impossible : Pile pleine.');

fin ;

{ Procédure permettant de dépiler un élément de la pile }

Procédure Depiler ;

début

Si (Sommet = 0) Alors Ecrire('Dépilement impossible : Pile vide.')

Sinon Si (Sommet = 1) Alors début

Sommet ← 0 ;

Base ← 0 ;



```

    fin
    Sinon sommet ← sommet - 1 ;
fin ;
{ Procédure permettant l'affichage des éléments de la pile }
Procédure Afficher_Pile ;
Variables
    i : entier ;
début
    Si (Sommet = 0) ET (Base = 0) Alors Ecrire('Aucun élément : Pile vide.')
    Sinon début
        Ecrire('Les éléments de la pile cités du sommet vers la base :) ;
        Pour i ← Sommet à 1 Faire Ecrire(Pile[i]) ;
    fin ;
fin ;
Début
{ Initialement la pile est vide }
Sommet ← 0 ;
Base ← 0 ;
Répéter
    Ecrire('Quelle opération désirez-vous ?') ;
    Ecrire('1- Empiler, 2- Depiler, 3- Afficher :) ;
    Lire(Choix) ;
    Cas Choix de
        1 : début
            Ecrire('Donnez le caractère à empiler :) ;
            Lire(Element) ;
            Empiler(Element) ;
        fin ;
        2 : Depiler ;
        3 : Afficher_Pile
    Sinon Ecrire('Choix non accepté.') ;
    fin ;
    Ecrire('Voulez-vous un autre test ? o/n') ;
    Lire(reponse) ;
    Jusqu'à (reponse = 'n') ;
Fin.
```

En Pascal :

```

program Gestion_Pile ;
const
```

```
    max = 100 ;
var
  Pile : array[1..max] of char ;
  Base, Sommet, Choix : integer ;
  Element, reponse : char ;
{ Procédure permettant d'empiler un élément dans la pile }
procedure Empiler(C : char) ;
begin
  if (Sommet < max) then begin
    if (Sommet <> 0) then begin
      Sommet := Sommet + 1 ;
      Pile[Sommet] := C ;
    end
  else begin
    Sommet := 1 ;
    Pile[Sommet] := C ;
    Base := Sommet ;
  end ;
end
  else writeln('Empilement impossible : Pile pleine.');
```

FOR AUTHOR USE ONLY

```
end ;
{ Procédure permettant de dépiler un élément de la pile }
procedure Depiler ;
begin
  if (Sommet = 0) then writeln('Dépilement impossible : Pile vide.')
```

FOR AUTHOR USE ONLY

```
  else if (Sommet = 1) then begin
    Sommet := 0 ;
    Base := 0 ;
  end
  else sommet := sommet - 1 ;
end ;
{ Procédure permettant l'affichage des éléments de la pile }
procedure Afficher_Pile ;
var
  i : integer ;
begin
  if (Sommet = 0) and (Base = 0) then writeln('Aucun élément : Pile vide.')
```

FOR AUTHOR USE ONLY

```
  else begin
    writeln('Les éléments de la pile cités du sommet vers la base :');
```

```
    for i := Sommet downto 1 do writeln(Pile[i]) ;
  end;
end ;
begin
{ Initialement la pile est vide }
  Sommet := 0 ;
  Base := 0 ;
  repeat
    writeln('Quelle opération désirez-vous ?') ;
    writeln('1- Empiler, 2- Depiler, 3- Afficher :)') ;
    readln(Choix) ;
    case Choix of
      1 : begin
        writeln('Donnez le caractère à empiler :)') ;
        readln(Element) ;
        Empiler(Element) ;
        end ;
      2 : Depiler ;
      3 : Afficher_Pile
    else writeln('Choix non accepté. ');
    end ;
    writeln('Voulez vous un autre test ? o/n') ;
    readln(reponse) ;
  until (reponse = 'n') ;
end.
```

**Remarque :**

Remarquez bien que pour l'opération d'empilement, on doit veiller à ne pas dépasser la taille maximale du tableau. Ce qui n'est pas le cas pour la modélisation d'une pile en utilisant une liste chaînée.

**7.2. Les files**

Une file est une liste dont les éléments sont ajoutés à une extrémité appelée Queue, et retiré de l'autre appelée Tête. Quand on ajoute un élément, celui-ci devient le dernier élément qui sera accessible. Quand on retire un élément de la file, on retire toujours la tête, celle-ci étant le premier élément qui a été placé dans la file. Ceci signifie que les éléments sont retirés de la file dans le même ordre dans lequel ils ont été ajoutés : premier entré, premier sorti. En anglais on dira, First In First Out, plus connu sous le nom FIFO.

Une bonne image de cette structure de données est la file d'attente au cinéma, dans un supermarché ou devant un guichet : on fait la queue et, à moins de tricher, il n'est pas possible de sortir de la queue avant les personnes qui ont commencé à faire la queue avant vous.

La structure de file (souvent appelée file d'attente) est d'un usage très répandu dans la programmation système, où les files servent à gérer, par exemple, l'allocation des ressources. C'est le cas notamment d'une imprimante en réseau, où les tâches d'impression arrivent aléatoirement de n'importe quel ordinateur connecté. Les tâches sont placées dans une file d'attente, ce qui permet de les traiter selon leur ordre d'arrivée.

La file peut être représentée tout simplement comme suit :



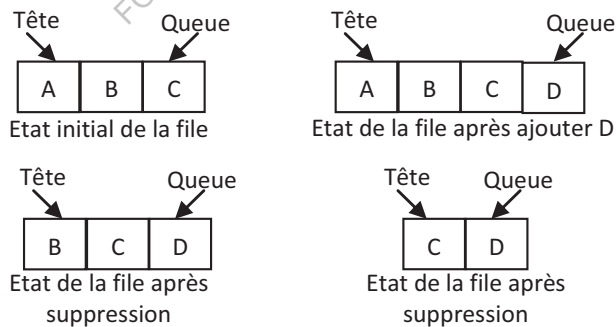
Le premier élément de la file est appelé Tête de la file, le dernier est appelé Queue de la file.

### 7.2.1. Accès à une file

Comme illustré dans l'exemple suivant, on ne peut effectuer un ajout à la file qu'à travers sa queue. Par contre, la suppression est effectuée à partir de la tête de la file.

#### Exemple :

Dans une file de caractères contenant initialement les éléments : A, B et C, on désire ajouter le caractère D, ensuite supprimer deux caractères. Ces opérations peuvent être représentées comme suit :



### 7.2.2. Représentation d'une file par une liste doublement chaînée

Comme une pile, la file peut être représentée par un tableau, une liste simplement ou doublement chaînée. Dans ce qui suit, nous allons choisir une liste doublement chaînée pour représenter une file :

- L'opération d'ajout d'un élément à la file (enfilement) se traduit par un ajout en queue de la liste doublement chaînée.

- L'opération de suppression d'un élément de la file (défilement) se traduit par une suppression de la tête de la liste doublement chaînée.
- La tête et la queue de la file étant la tête et la queue de la liste doublement chaînée.

L'algorithme de manipulation d'une file est alors le suivant :

Algorithme Gestion\_File ;

Type

F = Enregistrement

Car : caractère ;

Precedent, Suivant :  $\uparrow$  F ;

fin ;

Variables

Queue, Tete :  $\uparrow$  F ;

Choix : entier ;

Element, reponse : caractère ;

{ Procédure permettant d'ajouter un élément à la file }

Procédure Ajouter\_File (C : caractère) ;

début

Si (Queue = nil ) Alors début

Allouer(Tete);

Tete $\uparrow$ .Car  $\leftarrow$  C ;

Tete $\uparrow$ .Precedent  $\leftarrow$  nil ;

Queue  $\leftarrow$  Tete ;

Queue $\uparrow$ .Suivant  $\leftarrow$  nil ;

fin

Sinon début

Allouer(Queue $\uparrow$ .Suivant);

Queue $\uparrow$ .Suivant $\uparrow$ .Precedent  $\leftarrow$  Queue ;

Queue  $\leftarrow$  Queue $\uparrow$ .Suivant;

Queue $\uparrow$ .Car  $\leftarrow$  C ;

Queue $\uparrow$ .Suivant  $\leftarrow$  nil ;

fin ;

fin ;

{ Procédure permettant de supprimer un élément de la file }

Procédure Supprimer\_File ;

début

Si (Tete = nil) Alors Ecrire('Suppression impossible : File vide.')

Sinon Si (Tete $\uparrow$ .Suivant = nil) Alors début

Ecrire('Elément supprimé : ', Tete $\uparrow$ .Car) ;

```

        Désallouer(Tete) ;
        Tete ← nil ;
        Queue ← nil ;
    fin
    Sinon début
        Ecrire('Elément supprimé : ', Tete↑.Car) ;
        Tete ← Tete↑.Suivant ;
        Désallouer(Tete↑.Precedent) ;
        Tete↑.Precedent ← nil ;
    fin ;
fin ;
{ Procédure permettant l'affichage des éléments de la file }
Procédure Afficher_File ;
Variables
    P : ↑ F ;
début
    Si (Tete = nil) Alors Ecrire('Aucun élément : File vide.')
    Sinon début
        P ← Tete;
        Ecrire('Les éléments de la file cités de la tête vers la queue :');
        Tant que (P <> nil) Faire début
            Ecrire(P↑.Car) ;
            P ← P↑.Suivant;
        fin ;
    fin ;
fin ;
début
{ Initialement la file est vide }
    Tete ← nil ;
    Queue ← nil ;
    Répéter
        Ecrire('Quelle opération désirez-vous ?') ;
        Ecrire('1- Ajouter, 2- Supprimer, 3- Afficher :') ;
        Lire(Choix) ;
        Cas Choix de
            1 : début
                Ecrire('Donnez le caractère à ajouter à la file :') ;
                Lire(Element) ;
                Ajouter_File(Element) ;

```

```

    fin ;
2 : Supprimer_File ;
3 : Afficher_File
Sinon Ecrire('Choix non accepté.') ;
fin ;
Ecrire('Voulez vous un autre test ? o/n') ;
Lire(reponse) ;
Jusqu'à (reponse = 'n') ;
Fin.

```

Le programme Pascal est le suivant :

```

program Gestion_File ;
Type
Ptr_File = ^ F ;
F = record
    Car : char ;
    Precedent, Suivant : Ptr_File ;
end ;
var
Queue, Tete : Ptr_File ;
Choix : integer ;
Element, reponse : char ;
{ Procédure permettant d'ajouter un élément à la file }
procedure Ajouter_File (C : char) ;
begin
    if (Queue = nil ) then begin
        new(Tete);
        Tete^.Car := C ;
        Tete^.Precedent := nil ;
        Queue := Tete ;
        Queue^.Suivant := nil ;
    end
    else begin
        new(Queue^.Suivant);
        Queue^.Suivant^.Precedent := Queue ;
        Queue := Queue^.Suivant;
        Queue^.Car := C ;
        Queue^.Suivant := nil ;
    end ;
end ;

```

```
{ Procédure permettant de supprimer un élément de la file }
procedure Supprimer_File ;
begin
  if (Tete = nil) then writeln('Suppression impossible : File vide.')
  else if (Tete^.Suivant = nil) then begin
    writeln('Elément supprimé : ', Tete^.Car) ;
    dispose(Tete) ;
    Tete := nil ;
    Queue := nil ;
  end
  else begin
    writeln('Elément supprimé : ', Tete^.Car) ;
    Tete := Tete^.Suivant ;
    dispose(Tete^.Precedent) ;
    Tete^.Precedent := nil ;
  end ;
end ;

{ Procédure permettant l'affichage des éléments de la file }
procedure Afficher_File ;
var
  P : Ptr_File;
begin
  if (Tete = nil) and (Queue = nil) then writeln('Aucun élément : File vide.')
  else begin
    P := Tete;
    writeln('Les éléments de la file cités de la tête vers la queue :') ;
    while (P <> nil) do begin
      writeln(P^.Car) ;
      P := P^.Suivant;
    end ;
  end;
end ;

begin
  { Initialement la file est vide }
  Tete := nil ;
  Queue := nil ;
  repeat
    writeln('Quelle opération désirez-vous ?') ;
    writeln('1- Ajouter, 2- Supprimer, 3- Afficher :)') ;
```



```
readln(Choix) ;
case Choix of
1 : begin
    writeln('Donnez le caractère à ajouter à la file :) ;
    readln(Element) ;
    Ajouter_File(Element) ;
    end ;
2 : Supprimer_File ;
3 : Afficher_File
else writeln('Choix non accepté.') ;
end ;
writeln('Voulez vous un autre test ? o/n') ;
readln(reponse) ;
until (reponse = 'n') ;
end.
```

**Exercice :**

Reprendre l'algorithme et le programme Pascal permettant la gestion d'une file, mais cette fois-ci, on désire utiliser un tableau circulaire pour la mise en œuvre de cette file.

**Solution :**

La deuxième manière de modéliser une file d'attente consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la file se fera en enlevant le premier élément du tableau. Il faudra donc deux indices pour ce tableau : le premier indique le premier élément de la file, et le deuxième indique la fin de la file. On peut noter que progressivement, au cours des opérations d'ajout et de retrait, le tableau se déplace sur la droite dans son espace mémoire. A un moment, il va en atteindre le bout de l'espace. Dans ce cas, le tableau continuera au début de l'espace mémoire comme si la première et la dernière case étaient adjacentes, d'où le terme "tableau circulaire". Ce mécanisme fonctionnera tant que le tableau n'est effectivement pas plein.

Algorithme Gestion\_File ;

Constantes

max = 100 ;

Variables

F : Tableau[1..max] de caractère ;

Queue, Tete, Choix : entier ;

Element, reponse : caractère ;

{ Procédure permettant d'ajouter un élément à la file }

```

Procédure Ajouter_File (C : caractère) ;
début
  Si (Tete = 0) ET (Queue = 0) Alors début
    Tete  $\leftarrow$  1;
    Queue  $\leftarrow$  1;
    F[Queue]  $\leftarrow$  C;
  fin
  Sinon Si (Queue  $\geq$  Tete) Alors
    Si (Queue < max) Alors début
      Queue  $\leftarrow$  Queue + 1 ;
      F[Queue]  $\leftarrow$  C ;
    fin
    Sinon Si (Tete = 1) Alors Ecrire('Ajout impossible : File pleine.')
      Sinon début
        Queue  $\leftarrow$  1 ;
        F[Queue]  $\leftarrow$  C ;
      fin
    Sinon Si (Queue < Tete - 1) Alors début
      Queue  $\leftarrow$  Queue + 1 ;
      F[Queue]  $\leftarrow$  C ;
    fin
    Sinon Ecrire('Ajout impossible : File pleine.');
```

fin ;

{ Procédure permettant de supprimer un élément de la file }

```

Procédure Supprimer_File ;
début
  Si (Tete = 0) ET (Queue = 0) Alors
    Ecrire('Suppression impossible : File vide.')
```

fin ;

```

  Sinon début
    Ecrire('Elément supprimé : ', F[Tete]) ;
    Si (Tete = Queue) Alors début
      Tete  $\leftarrow$  0 ;
      Queue  $\leftarrow$  0 ;
    fin
    Sinon Si (Tete = max) Alors Tete  $\leftarrow$  1
      Sinon Tete  $\leftarrow$  Tete + 1 ;
    fin ;
  fin ;
  { Procédure permettant l'affichage des éléments de la file }
```

```

Procédure Afficher_File ;
Variables
i : entier ;
début
Si (Tete = 0) ET (Queue = 0) Alors Ecrire('Aucun élément : File vide.')
Sinon début
    Ecrire('Les éléments de la file cités de la tête vers la queue :) ;
    Si (Queue >= Tete) Alors
        Pour i ← Tete à Queue Faire Ecrire(F[i])
    Sinon début
        Pour i ← Tete à max Faire Ecrire(F[i]) ;
        Pour i ← 1 à Queue Faire Ecrire(F[i]) ;
    fin ;
fin ;
fin ;
début
{ Initialement la file est vide }
Tete ← 0 ;
Queue ← 0 ;
Répéter
    Ecrire('Quelle opération désirez-vous ?') ;
    Ecrire('1- Ajouter, 2- Supprimer, 3- Afficher :) ;
    Lire(Choix) ;
    Cas Choix de
        1 : début
            Ecrire('Donnez le caractère à ajouter à la file :) ;
            Lire(Element) ;
            Ajouter_File(Element) ;
            fin ;
        2 : Supprimer_File ;
        3 : Afficher_File
    Sinon Ecrire('Choix non accepté.') ;
    fin ;
    Ecrire('Voulez-vous un autre test ? o/n') ;
    Lire(reponse) ;
    Jusqu'à (reponse = 'n') ;
Fin.

```

Le programme Pascal est le suivant :

```

program Gestion_File ;

```

```
const
  max = 100 ;
var
  F : array[1..max] of char ;
  Queue, Tete, Choix : integer ;
  Element, reponse : char ;
{ Procédure permettant d'ajouter un élément à la file }
procedure Ajouter_File (C : char) ;
begin
  if (Tete = 0) and (Queue = 0) then begin
    Tete := 1;
    Queue := 1;
    F[Queue] := C;
  end
  else if (Queue >= Tete) then
    if (Queue < max) then begin
      Queue := Queue + 1 ;
      F[Queue] := C ;
    end
    else if (Tete = 1) then writeln('Ajout impossible : File pleine.')
      else begin
        Queue := 1 ;
        F[Queue] := C ;
      end
    else if (Queue < Tete - 1) then begin
      Queue := Queue + 1 ;
      F[Queue] := C ;
    end
    else writeln('Ajout impossible : File pleine.');
```

end ;

{ Procédure permettant de supprimer un élément de la file }

procedure Supprimer\_File ;

begin

if (Tete = 0) and (Queue = 0) then

writeln('Suppression impossible : File vide.')

else begin

writeln('Elément Supprimé : ', F[Tete]) ;

if (Tete = Queue) then begin

Tete := 0 ;

```

    Queue := 0 ;
end
else if (Tete = max) then Tete := 1
    else Tete := Tete + 1 ;
end ;
end ;
{ Procédure permettant l'affichage des éléments de la file }
procedure Afficher_File ;
var
    i : integer;
begin
    if (Tete = 0) and (Queue = 0) then writeln('Aucun élément : File vide.')
    else begin
        writeln('Les éléments de la file cités de la tête vers la queue :') ;
        if (Queue >= Tete) then
            for i := Tete to Queue do writeln(F[i])
        else begin
            for i := Tete to max do writeln(F[i]) ;
            for i := 1 to Queue do writeln(F[i]) ;
        end;
    end;
end ;
begin
{ Initialement la file est vide }
    Tete := 0 ;
    Queue := 0 ;
    repeat
        writeln('Quelle opération désirez-vous ?') ;
        writeln('1- Ajouter, 2- Supprimer, 3- Afficher :') ;
        readln(Choix) ;
        case Choix of
            1 : begin
                writeln('Donnez le caractère à ajouter à la file :) ;
                readln(Element) ;
                Ajouter_File(Element) ;
            end ;
            2 : Supprimer_File ;
            3 : Afficher_File
        else writeln('Choix non accepté.') ;
    end

```

```

end ;
writeln('Voulez-vous un autre test ? o/n') ;
readln(reponse) ;
until (reponse = 'n') ;
end.

```

## 8. Exercices corrigés

### 8.1. Exercices

#### Exercice 1 :

Soit le programme Pascal suivant :

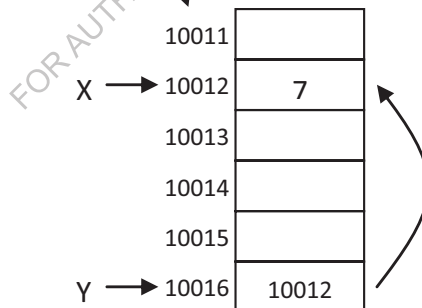
```

program pointeurs ;
var
  y : ^integer ; x : integer;
begin
  x := 7; y := @x ;
  writeln(x, ' ', y^, ' ', y, ' ', @y) ;
end.

```

On suppose qu'après son exécution, le programme ci-dessus a provoqué l'état de la mémoire suivant :

Adresses mémoire



Qu'affiche le programme Pascal précédent (pointeurs) ?

#### Exercice 2 :

Ecrire un algorithme permettant de créer une liste chaînée contenant au moins un élément, ensuite d'afficher les éléments de la liste. Chaque élément est de type enregistrement *Personne* contenant deux informations : *Num* de type entier et *Nom* de type chaîne de caractères. Utilisez une procédure pour l'ajout d'une personne, et une autre pour afficher les éléments de la liste. La tête et la queue de la liste étant

captés par deux pointeurs : Tete et Queue pointant vers Personne. Traduire l'algorithme en Pascal.

Quelle instruction doit-on ajouter pour rendre la liste circulaire ? Dans une liste circulaire, il est possible de revenir à un point de départ en parcourant la liste dans un seul sens, sans revenir dans le sens inverse.

**Exercice 3 :**

Ecrire une fonction entière permettant de calculer la longueur d'une liste. Les éléments de la liste étant de type Personne décrit dans l'exercice 2. La fonction doit posséder comme paramètre la tête de la liste. Traduire la fonction en Pascal.

**Exercice 4 :**

Ecrire une fonction booléenne permettant la recherche d'un élément dans une liste. La fonction doit posséder comme paramètres la tête de la liste, le numéro et le nom de l'élément à rechercher. Les éléments de la liste étant de type Personne décrit dans l'exercice 2. Traduire la fonction en Pascal.

**Exercice 5 :**

Modifier la fonction de l'exercice précédent de telle sorte que cette fonction retourne un pointeur vers l'élément trouvé s'il existe. Dans le cas contraire, elle pointe vers le nil.

**Exercice 6 :**

Ecrire une fonction qui retourne un pointeur vers le  $n^{\text{ième}}$  élément d'une liste. Si  $n$  dépasse la taille de la liste, la fonction retourne nil. La fonction doit posséder comme paramètres la tête de la liste et le numéro d'ordre de l'élément désiré. Les éléments de la liste étant de type Personne décrit dans l'exercice 2. Traduire la fonction en Pascal.

**Exercice 7 :**

Ecrire une procédure permettant d'insérer un élément à la  $k^{\text{ième}}$  position dans une liste. Les éléments de la liste étant de type Personne décrit dans l'exercice 2. L'indice  $k$  doit être pris entre 1 et la longueur de la liste. La procédure doit posséder comme paramètres le numéro et le nom de l'élément à insérer, la position là où le nouvel élément doit être inséré et un pointeur vers la tête de la liste. Traduire la procédure en Pascal.

**Exercice 8 :**

Ecrire une procédure permettant de supprimer l'élément de la  $k^{\text{ième}}$  position dans une liste. Les éléments de la liste étant de type Personne décrit dans l'exercice 2. L'indice  $k$  doit être pris entre 1 et la longueur de la liste. La procédure doit posséder comme paramètres la position de l'élément à supprimer et un pointeur vers la tête de la liste. Traduire la procédure en Pascal.

**Exercice 9 :**

Ecrire la procédure permettant de renverser une liste. Les éléments de la liste étant de type `Personne` décrit dans l'exercice 2. La procédure doit posséder comme paramètres un pointeur vers la tête et un autre vers la queue de la liste à renverser. Traduire la procédure en Pascal.

**Exercice 10 :**

Ecrire un algorithme permettant de créer deux listes chaînées ayant un nombre d'éléments indéfini, ensuite d'afficher les éléments des deux listes. Chaque élément d'une liste est de type `enregistrement Nombre` contenant une information : `Val` de type entier. Utilisez une procédure pour la création d'une liste. Cette procédure doit posséder la tête et la queue d'une liste comme paramètres. Pour l'ajout d'un élément à une liste, utilisez une procédure possédant comme paramètre la queue de la liste à laquelle on veut ajouter le nouvel élément. Une autre procédure possédant la tête d'une liste comme paramètre est utilisée pour afficher les éléments de cette liste. La tête et la queue de la première liste étant captés par deux pointeurs : `Tete1` et `Queue1`. La tête et la queue de la deuxième liste étant captés par deux pointeurs : `Tete2` et `Queue2`. Traduire l'algorithme en Pascal.

**Exercice 11 :**

Ecrire une procédure permettant de concaténer deux listes. Les éléments de la liste étant de type `Nombre` décrit dans l'exercice 10. La procédure doit posséder comme paramètres un pointeur vers la tête de la première liste et un autre vers tête de la deuxième. Traduire la procédure en Pascal.

**Exercice 12 :**

Ecrire une procédure permettant de trier une liste en ordre croissant. Les éléments de la liste étant de type `Nombre` décrit dans l'exercice 10. La procédure doit posséder comme paramètre un pointeur vers la tête de la liste à trier. Traduire la procédure en Pascal.

**Exercice 13 :**

Ecrire une procédure permettant d'insérer un élément dans une liste triée par ordre croissant. Les éléments de la liste étant de type `Nombre` décrit dans l'exercice 10. La procédure doit posséder comme paramètres la valeur à insérer et un pointeur vers la tête de la liste. Traduire la procédure en Pascal.

**Exercice 14 :**

Ecrire une procédure permettant de fusionner deux listes triées en ordre croissant. Les éléments de chaque liste étant de type `Nombre` décrit dans l'exercice 10. La procédure doit posséder comme paramètres un



pointeur vers la tête de la première liste et un pointeur vers la tête de la deuxième liste. Traduire la procédure en Pascal.

### Exercice 15 :

Reprendre l'exercice 2, mais cette fois-ci en stockant les informations dans une liste doublement chaînée.

### Exercice 16 :

Ecrire une procédure permettant l'insertion d'un élément avant un autre dans la liste doublement chaînée créée dans l'exercice précédent. La procédure doit posséder comme paramètres le numéro et le nom de l'élément à insérer, ainsi que le numéro et le nom de l'élément avant lequel on désire faire l'insertion. Traduire la procédure en Pascal.

## 8.2. Corrigés

### Solution 1 :

Le programme affiche :

7 7 10012 10016

### Solution 2 :

Algorithme Personnes ;

Type

Personne = Enregistrement

Num : entier ;

Nom : chaîne de caractères;

Suivant : ↑ Personne ;

fin ;

Variables

Tete, Queue : ↑ Personne ;

c : caractère ;

(\*Procédure d'ajout d'un élément à la liste\*)

Procédure Ajouter\_Personne ;

début

Allouer(Queue↑.Suivant);

Queue ← Queue↑.Suivant;

Ecrire('Donnez le num d'une personne :');

Lire(Queue↑.Num);

Ecrire('Donnez le nom d'une personne :');

Lire(Queue↑.Nom);

Queue↑.Suivant ← nil ;

fin ;

(\*Procédure d'affichage des éléments de la liste\*)

Procédure Afficher\_Liste ;

Variables

P : ↑ Personne ;

début

P ← Tete;

Tant que (P <> nil) Faire début

Ecrire('Élément Num° ', P↑.Num, ' Nom : ', P↑.Nom) ;

P ← P↑.Suivant;

fin;

fin ;

Début

(\*Initialement la liste est vide\*)

Tete ← nil ;

Queue ← nil ;

(\*Créer et remplir le premier élément de la liste\*)

Allouer(Tete);

Ecrire('Donnez le num d'"une personne :');

Lire(Tete↑.Num);

Ecrire('Donnez le nom d'"une personne :');

Lire(Tete↑.Nom);

Queue ← Tete ;

Queue↑.Suivant ← nil ;

Répéter

Ecrire('Voulez-vous ajouter un autre élément ? o/n');)

Lire(c);

Si (c = 'o') Alors Ajouter\_personne;

Jusqu'à (c = 'n');

Afficher\_Liste;

Fin.

Le programme Pascal :

program Personnes ;

type

Ptr\_Personne = ^ Personne ;

Personne = record

Num : integer ;

Nom : string;

Suivant : Ptr\_Personne;

end ;

var

Tete, Queue : Ptr\_Personne ;

```

c : char ;
(*Procédure d'ajout d'un élément à la liste*)
procedure Ajouter_Personne ;
begin
    new(Queue^.Suivant);
    Queue := Queue^.Suivant;
    writeln('Donnez le num d'une personne :');
    readln(Queue^.Num);
    writeln('Donnez le nom d'une personne :');
    readln(Queue^.Nom);
    Queue^.Suivant := nil ;
end ;
(*Procédure d'affichage des éléments de la liste*)
procedure Afficher_Liste ;
var
    P : Ptr_Personne;
begin
    P := Tete;
    while (P <> nil) do begin
        writeln('Elément Num° ', P^.Num, ' Nom : ', P^.Nom) ;
        P := P^.Suivant;
    end;
end ;
begin
    (*Initialement la liste est vide*)
    Tete := nil ;
    Queue := nil ;
    (*Créer et remplir le premier élément de la liste*)
    new(Tete);
    writeln('Donnez le num d'une personne :');
    readln(Tete^.Num);
    writeln('Donnez le nom d'une personne :');
    readln(Tete^.Nom);
    Queue := Tete ;
    Queue^.Suivant := nil ;
    repeat
        writeln('Voulez-vous ajouter un autre élément ? o/n');
        readln(c);
        if (c = 'o') then Ajouter_personne;
    
```

```

until (c = 'n');
Afficher_Liste;
end.

```

Pour obtenir une liste circulaire, il faut ajouter, après la création de la liste complète, l'instruction : `Queue^.Suivant := Tete ;`

### **Solution 3 :**

Fonction Long\_Liste (T : ↑ Personne ) : entier ;

Variables

P : ↑ Personne ;  
 nbr : entier ;

Début

P ← T;  
 nbr ← 0;  
 Tant que (P <> nil) Faire début  
     nbr ← nbr + 1 ;  
     P ← P↑.Suivant;

fin ;

Long\_Liste ← nbr ;

fin ;

En Pascal :

function Long\_Liste (T : Ptr\_Personne): integer ;

var

P : Ptr\_Personne;  
 nbr : integer;

begin

P := T;  
 nbr := 0;  
 while (P <> nil) do begin  
     nbr := nbr + 1;  
     P := P^.Suivant;

end;

Long\_Liste := nbr;

end ;

### **Solution 4 :**

fonction Recherche(Nu : entier ; No : chaîne de caractères ; T : ↑ Personne) : booléen;

Variables

P : ↑ Personne ;  
 Trouve : booléen;

Début

$P \leftarrow T;$

Trouve  $\leftarrow$  FAUX ;

Tant que ( $P \neq \text{nil}$ ) ET NON Trouve Faire

Si  $((Nu = P \uparrow . \text{Num}) \text{ ET } (No = P \uparrow . \text{Nom}))$  Alors Trouve  $\leftarrow$  VRAI

Sinon  $P \leftarrow P \uparrow . \text{Suivant};$

Recherche  $\leftarrow$  Trouve ;

fin ;

En Pascal :

function Recherche(Nu : integer ; No : string ; T : Ptr\_Personne) :

boolean;

var

P : Ptr\_Personne;

Trouve : boolean;

begin

P := T;

Trouve := false ;

while (P  $\neq$  nil) and not trouve do

if  $((Nu = P^{\wedge} . \text{Num}) \text{ and } (No = P^{\wedge} . \text{Nom}))$  then Trouve := true

else P := P^{\wedge} . Suivant;

Recherche := Trouve ;

end ;

**Solution 5 :**

fonction Recherche(Nu : entier ; No : chaîne de caractères ; T :  $\uparrow$

Personne) :  $\uparrow$  Personne ;

Variables

P :  $\uparrow$  Personne ;

Trouve : booléen;

Début

$P \leftarrow T;$

Trouve  $\leftarrow$  FAUX ;

Tant que ( $P \neq \text{nil}$ ) ET NON Trouve Faire

Si  $((Nu = P \uparrow . \text{Num}) \text{ ET } (No = P \uparrow . \text{Nom}))$  Alors Trouve  $\leftarrow$  VRAI

Sinon  $P \leftarrow P \uparrow . \text{Suivant};$

Recherche  $\leftarrow$  P ;

fin ;

En Pascal :

function Recherche(Nu : integer ; No : string ; T : Ptr\_Personne) :

Ptr\_Personne ;

```

var
  P : Ptr_Personne;
  Trouve : boolean;
begin
  P := T;
  Trouve := false ;
  while (P <> nil) and not trouve do
    if ((Nu = P^.Num) and (No = P^.Nom)) then Trouve := true
      else P := P^.Suivant;
  Recherche := P ;
end ;

```

**Solution 6 :**

Fonction Element (n : entier ; T : ↑ Personne) : ↑ Personne ;

Variables

```

P : ↑Personne;
nbr : entier ;
Début
  P ← T;
  nbr ← 1 ;
  Tant que (P <> nil) ET (nbr < n) Faire début
    P ← P↑.Suivant;
    nbr ← nbr + 1 ;
  Fin ;
  Element ← P;

```

Fin ;

En Pascal :

function Element (n : integer ; T : Ptr\_Personne) : Ptr\_Personne ;

```

var
  P : Ptr_Personne;
  nbr : integer ;
begin
  P := T;
  nbr := 1 ;
  while (P <> nil) and (nbr < n ) do begin
    P := P^.Suivant;
    nbr := nbr + 1 ;
  end;
  Element := P;
end ;

```

### Solution 7 :

Procédure Insérer\_Liste (Nu : entier ; No : chaîne de caractères ; k : entier ; var T : ↑ Personne) ;

Variables

P1, P2 : ↑ Personne ;

nbr : entier ;

Début

P1 ← T;

Allouer(P2);

P2↑.Num ← Nu ;

P2↑.Nom ← No ;

Si (k = 1) Alors début

P2↑.Suivant ← T ;

T ← P2;

fin

Sinon Si ((k > 1) ET (k ≤ Long\_Liste(T))) Alors début

nbr ← 2 ;

Tant que (nbr < k ) Faire début

nbr ← nbr + 1;

P1 ← P1↑.Suivant;

fin;

P2↑.Suivant ← P1↑.Suivant ;

P1↑.Suivant ← P2;

fin

Sinon Ecrire('Position non acceptée.');

fin ;

En Pascal :

procedure Insérer\_Liste (Nu : integer ; No : string ; k : integer; var T : Ptr\_Personne) ;

var

P1, P2 : Ptr\_Personne;

nbr : integer;

begin

P1 := T;

new(P2);

P2<sup>^</sup>.Num := Nu ;

P2<sup>^</sup>.Nom := No ;

if (k = 1) then begin

P2<sup>^</sup>.Suivant := T ;

```

    T := P2;
end
else if ((k > 1) and (k <= Long_Liste(T))) then begin
    nbr := 2 ;
    while (nbr < k ) do begin
        nbr := nbr + 1;
        P1 := P1^.Suivant;
    end;
    P2^.Suivant := P1^.Suivant ;
    P1^.Suivant := P2;
end
else writeln('Position non acceptée.');
```

**Solution 8 :**

Procédure Supprimer\_Liste (k : entier ; var T : ↑ Personne) ;

Variables

P1, P2 : ↑ Personne;

nbr : entier ;

Début

P1 ← T;

Si (k = 1) Alors début

T ← P1↑.Suivant ;

Désallouer(p1);

fin

Sinon Si ((k > 1) ET (k <= Long\_Liste(T))) Alors début

nbr ← 2 ;

Tant que (nbr < k ) Faire début

nbr ← nbr + 1;

P1 ← P1↑.Suivant;

fin ;

P2 ← P1↑.Suivant ;

P1↑.Suivant ← P2↑.Suivant ;

Désallouer(P2);

fin

Sinon Ecrire('Position inexistante.');

fin ;

En Pascal :

procedure Supprimer\_Liste (k : integer; var T : Ptr\_Personne) ;

var



```

P1, P2 : Ptr_Personne;
nbr : integer;
begin
  P1 := T;
  if (k = 1) then begin
    T := P1^.Suivant ;
    dispose(p1);
  end
  else if ((k > 1) and (k <= Long_Liste(T))) then begin
    nbr := 2 ;
    while (nbr < k ) do begin
      nbr := nbr + 1;
      P1 := P1^.Suivant;
    end;
    P2 := P1^.Suivant ;
    P1^.Suivant := P2^.Suivant ;
    dispose(P2);
  end
  else writeln('Position inexistante.');
```

### **Solution 9 :**

Procédure Renverser\_Liste (var T, Q : ↑ Personne) ;

Variables

P1, P2 : ↑ Personne ;

Début

Si (T <> nil) Alors début

P1 ← T;

Tant que (P1↑.Suivant <> nil) Faire début

P2 ← P1↑.Suivant ;

P1↑.Suivant ← P2↑.Suivant ;

P2↑.Suivant ← T ;

T ← P2 ;

fin ;

Q ← P1 ;

fin ;

fin ;

En Pascal :

procedure Renverser\_Liste (var T, Q : Ptr\_Personne) ;

var

```

P1, P2 : Ptr_Personne;
begin
  if (T <> nil) then begin
    P1 := T;
    while (P1^.Suivant <> nil) do begin
      P2 := P1^.Suivant ;
      P1^.Suivant := P2^.Suivant ;
      P2^.Suivant := T ;
      T := P2 ;
    end;
    Q := P1 ;
  end;
end ;

```

### **Solution 10 :**

Algorithme Nombres ;

Type

Nombre = Enregistrement

Val : entier ;

Suivant : ↑ Nombre ;

fin ;

Variables

Tete1, Tete2, Queue1, Queue2 : ↑ Nombre ;

c : caractère ;

(\*Procédure d'ajout d'un élément à la liste\*)

Procédure Ajouter\_Nombre( var Q : ↑ Nombre) ;

début

Allouer(Q↑.Suivant);

Q ← Q↑.Suivant;

Ecrire('Donnez la valeur de cet élément :');

Lire(Q↑.Val);

Q↑.Suivant ← nil ;

fin ;

(\*Procédure de création d'une liste\*)

Procédure Creer\_Liste(var T, Q : ↑ Nombre) ;

début

(\*Initialement la liste est vide\*)

T ← nil ;

Q ← nil ;

(\*Créer et remplir le premier élément de la liste courante\*)

```

Allouer(T);
Ecrire('Donnez la valeur de cet élément :');
Lire(T↑.Val);
Q ← T ;
Q↑.Suivant ← nil ;
Répéter
  Ecrire('Voulez-vous ajouter un autre élément ? o/n');
  Lire(c);
  Si (c = 'o') Alors Ajouter_Nombre(Q);
  Jusqu'à (c = 'n');
fin ;
(*Procédure d'affichage des éléments de la liste*)
Procédure Afficher_Liste (T : ↑ Nombre) ;
Variables
  P : ↑ Nombre;
début
  P ← T;
  Tant que (P <> nil) Faire début
    Ecrire('Elément avec val = ', P↑.Val) ;
    P ← P↑.Suivant;
  fin ;
fin ;
Début (*Corps de l'algorithme*)
  Ecrire('Création de la première liste. ');
  Creer_Liste(Tete1, Queue1);
  Ecrire('Création de la deuxième liste. ');
  Creer_Liste(Tete2, Queue2);
  Ecrire('Voici les éléments de la première liste :');
  Afficher_Liste(Tete1);
  Ecrire('Voici les éléments de la deuxième liste :');
  Afficher_Liste(Tete2);
Fin.

```

En Pascal :

```

program Nombres ;
type
  Ptr_Nombre = ^ Nombre ;
  Nombre = record
    Val : integer ;
    Suivant : Ptr_Nombre;

```

```
end ;
var
  Tete1, Tete2, Queue1, Queue2 : Ptr_Nombre ;
  c : char ;
(*Procédure d'ajout d'un élément à la liste*)
procedure Ajouter_Nombre( var Q : Ptr_Nombre) ;
begin
  new(Q^.Suivant);
  Q := Q^.Suivant;
  writeln('Donnez la valeur de cet élément :');
  readln(Q^.Val);
  Q^.Suivant := nil ;
end ;
(*Procédure de création d'une liste*)
procedure Creer_Liste(var T, Q : Ptr_Nombre) ;
begin
  (*Initialement la liste est vide*)
  T := nil ;
  Q := nil ;
  (*Créer et remplir le premier élément de la liste courante*)
  new(T);
  writeln('Donnez la valeur de cet élément :');
  readln(T^.Val);
  Q := T ;
  Q^.Suivant := nil ;
  repeat
    writeln('Voulez vous ajouter un autre élément ? o/n');
    readln(c);
    if (c = 'o') then Ajouter_Nombre(Q);
  until (c = 'n');
end;
(*Procédure d'affichage des éléments de la liste*)
procedure Afficher_Liste (T : Ptr_Nombre) ;
var
  P : Ptr_Nombre;
begin
  P := T;
  while (P <> nil) do begin
    writeln('Elément avec val = ', P^.Val) ;
```

```

    P := P^.Suivant;
end;
end ;
begin (*Programme principal*)
    writeln('Création de la première liste. ');
    Creer_Liste(Tete1, Queue1);
    writeln('Création de la deuxième liste. ');
    Creer_Liste(Tete2, Queue2);
    writeln('Voici les éléments de la première liste : ');
    Afficher_Liste(Tete1);
    writeln('Voici les éléments de la deuxième liste : ');
    Afficher_Liste(Tete2);
end.

```

### **Solution 11 :**

Procédure Concat\_Liste ( T1, T2 : ↑ Nombre) ;

Variables

P : ↑ Nombre ;

début

Si (T1 <> nil) ET (T2 <> nil) Alors début

P ← T1 ;

Tant que (P↑.Suivant <> nil) Faire P := P↑.Suivant ;

P↑.Suivant ← T2 ;

fin ;

fin ;

En Pascal :

procedure Concat\_Liste ( T1, T2 : Ptr\_Nombre) ;

var

P : Ptr\_Nombre;

begin

if (T1 <> nil) and (T2 <> nil) then begin

P := T1 ;

while (P↑.Suivant <> nil) do P := P↑.Suivant ;

P↑.Suivant := T2 ;

end;

end ;

### **Solution 12 :**

Procédure Tri\_Liste (var T1 : ↑ Nombre) ;

Variables

P1, P2 : ↑ Nombre;

```

x : entier ;
début
P1 ← T1 ;
Tant que (P1↑.Suivant <> nil) Faire début
  P2 ← P1↑.Suivant ;
  Tant que (P2 <> nil) Faire début
    Si (P1↑.Val > P2↑.Val) Alors début
      x ← P1↑.Val ;
      P1↑.Val ← P2↑.Val ;
      P2↑.Val ← x ;
    fin ;
    P2 ← P2↑.Suivant ;
  fin ;
  P1 ← P1↑.Suivant ;
fin ;
fin ;

```

En Pascal :

```

procedure Tri_Liste ( var T1 : Ptr_Nombre) ;
var
  P1, P2 : Ptr_Nombre;
  x : integer ;
begin
  P1 := T1 ;
  while (P1^.Suivant <> nil) do begin
    P2 := P1^.Suivant ;
    while (P2 <> nil) do begin
      if (P1^.Val > P2^.Val) then begin
        x := P1^.Val ;
        P1^.Val := P2^.Val ;
        P2^.Val := x ;
      end;
      P2 := P2^.Suivant ;
    end ;
    P1 := P1^.Suivant ;
  end ;
end ;

```

### **Solution 13 :**

Procédure Insérer\_Val (V : entier ; var T : ↑ Nombre) ;  
 Variables

```

P1, P2 : ↑ Nombre ;
début
{ Créer le nouvel élément }
Allouer(P2) ;
P2↑.Val ← V ;
P2↑.Suivant ← nil ;
{ Le nouvel élément est inférieur à la tête }
Si (V ≤ T↑.Val) Alors début
    P2↑.Suivant ← T ;
    T ← P2 ;
fin
Sinon début
{ Le nouvel élément est supérieur à la queue }
P1 ← T ;
Tant que (P1↑.Suivant <> nil) Faire P1 ← P1↑.Suivant ;
Si (V ≥ P1↑.Val) Alors P1↑.Suivant ← P2
    Sinon début
        { Insertion dans la liste }
        P1 ← T ;
        Tant que (V ≥ P1↑.Suivant↑.Val) Faire P1 ← P1↑.Suivant ;
        P2↑.Suivant ← P1↑.Suivant ;
        P1↑.Suivant ← P2 ;
    fin ;
fin ;

```

fin ;

En Pascal :

procedure Insérer\_Val (V : integer ; var T : Ptr\_Nombre) ;

var

P1, P2 : Ptr\_Nombre;

begin

{ Créer le nouvel élément }

new(P2) ;

P2<sup>^</sup>.Val := V ;

P2<sup>^</sup>.Suivant := nil ;

{ Le nouvel élément est inférieur à la tête }

if (V ≤ T<sup>^</sup>.Val) then begin

P2<sup>^</sup>.Suivant := T ;

T := P2 ;

end

```
else begin
{ Le nouvel élément est supérieur à la queue }
P1 := T ;
while (P1^.Suivant <> nil) do P1 := P1^.Suivant ;
if (V >= P1^.Val) then P1^.Suivant := P2
else begin
{ Insertion dans la liste }
P1 := T ;
while (V >= P1^.Suivant^.Val) do P1 := P1^.Suivant ;
P2^.Suivant := P1^.Suivant ;
P1^.Suivant := P2 ;
end ;
end ;
end ;
```

**Solution 14 :**

Procédure Fusionner\_Listes (var T1, T2 : ↑ Nombre) ;

Variables

P1, P2 : Ptr\_Nombre;

début

Tant que (T2 <> nil) Faire début

{ Isoler le premier élément de la deuxième liste }

P2 ← T2 ;

T2 ← T2↑.Suivant ;

P2↑.Suivant ← nil ;

{ Élément isolé inférieur à la tête de la première liste }

Si (P2↑.Val <= T1↑.Val) Alors début

P2↑.Suivant ← T1 ;

T1 ← P2 ;

fin

Sinon début

{ Élément isolé supérieur à la queue de la première liste }

P1 ← T1 ;

Tant que (P1↑.Suivant <> nil) Faire P1 ← P1↑.Suivant ;

Si (P2↑.Val >= P1↑.Val) Alors P1↑.Suivant ← P2

Sinon début

{ Insertion de l'élément isolé dans la première liste }

P1 ← T1 ;

Tant que (P2↑.Val >= P1↑.Suivant↑.Val) Faire P1 ← P1↑.Suivant ;

P2↑.Suivant ← P1↑.Suivant ;



```

    P1↑.Suivant ← P2 ;
  fin ;
fin ;
fin ;
fin ;

```

En Pascal :

```

procedure Fusionner_Listes (var T1, T2 : Ptr_Nombre) ;
var
  P1, P2 : Ptr_Nombre;
begin
  while (T2 <> nil) do begin
    { Isoler le premier élément de la deuxième liste }
    P2 := T2 ;
    T2 := T2^.Suivant ;
    P2^.Suivant := nil ;
    { Élément isolé inférieur à la tête de la première liste }
    if (P2^.Val <= T1^.Val) then begin
      P2^.Suivant := T1 ;
      T1 := P2 ;
    end
  else begin
    { Élément isolé supérieur à la queue de la première liste }
    P1 := T1 ;
    while (P1^.Suivant <> nil) do P1 := P1^.Suivant ;
    if (P2^.Val >= P1^.Val) then P1^.Suivant := P2
  else begin
    { Insertion de l'élément isolé dans la première liste }
    P1 := T1 ;
    while (P2^.Val >= P1^.Suivant^.Val) do P1 := P1^.Suivant ;
    P2^.Suivant := P1^.Suivant ;
    P1^.Suivant := P2 ;
  end ;
end ;
end ;
end ;

```

**Solution 15 :**

Algorithme Personnes ;

Type

Personne = Enregistrement

```

    Num : entier ;
    Nom : chaîne de caractères ;
    Precedent, Suivant : ↑ Personne ;
fin ;
Variables
Tete, Queue : ↑ Personne ;
c : caractère ;
(*Procédure d'ajout d'un élément à la liste doublement chaînée*)
Procédure Ajouter_Personne ;
début
    Allouer(Queue↑.Suivant);
    Queue↑.Suivant↑.Precedent ← Queue ;
    Queue ← Queue↑.Suivant ;
    Ecrire('Donnez le num d"une personne :');
    Lire(Queue↑.Num);
    Ecrire('Donnez le nom d"une personne :');
    Lire(Queue↑.Nom);
    Queue↑.Suivant ← nil ;
fin ;
(*Procédure d'affichage des éléments de la liste doublement chaînée
de la droite vers la gauche*)
Procédure Afficher_Liste ;
Variables
    P : ↑ Personne ;
début
    P ← Tete;
    Tant que (P <> nil) Faire début
        Ecrire('Elément Num° ', P↑.Num, ' Nom : ', P↑.Nom) ;
        P ← P↑.Suivant;
    fin ;
fin ;
début
    (*Initialement la liste est vide*)
    Tete ← nil ;
    Queue ← nil ;
    (*Créer et remplir le premier élément de la liste doublement chaînée*)
    Allouer(Tete);
    Ecrire('Donnez le num d"une personne :');
    Lire(Tete↑.Num);

```

```

Ecrire('Donnez le nom d'une personne :');
Lire(Tete↑.Nom);
Tete↑.Precedent ← nil ;
Queue ← Tete ;
Queue↑.Suivant ← nil ;
Répéter
  Ecrire('Voulez-vous ajouter un autre élément ? o/n');
  Lire(c);
  Si (c = 'o') Alors Ajouter_personne;
  Jusqu'à (c = 'n');
Afficher_Liste;
Fin.
En Pascal :
program Personnes ;
type
  Ptr_Personne = ^ Personne ;
  Personne = record
    Num : integer ;
    Nom : string;
    Precedent, Suivant : Ptr_Personne;
  end ;
var
  Tete, Queue : Ptr_Personne ;
  c : char ;
  (*Procédure d'ajout d'un élément à la liste doublement chaînée*)
procedure Ajouter_Personne ;
begin
  new(Queue↑.Suivant);
  Queue↑.Suivant↑.Precedent := Queue ;
  Queue := Queue↑.Suivant;
  writeln('Donnez le num d'une personne :');
  readln(Queue↑.Num);
  writeln('Donnez le nom d'une personne :');
  readln(Queue↑.Nom);
  Queue↑.Suivant := nil ;
end ;
(*Procédure d'affichage des éléments de la liste doublement chaînée
de la droite vers la gauche*)
procedure Afficher_Liste ;

```

```
var
  P : Ptr_Personne;
begin
  P := Tete;
  while (P <> nil) do begin
    writeln('Elément Num° ', P^.Num, ' Nom : ', P^.Nom);
    P := P^.Suivant;
  end;
end ;
begin
  (*Initialement la liste est vide*)
  Tete := nil ;
  Queue := nil ;
  (*Créer et remplir le premier élément de la liste doublement chaînée*)
  new(Tete);
  writeln('Donnez le num d'une personne :');
  readln(Tete^.Num);
  writeln('Donnez le nom d'une personne :');
  readln(Tete^.Nom);
  Tete^.Precedent := nil ;
  Queue := Tete ;
  Queue^.Suivant := nil ;
  repeat
    writeln('Voulez-vous ajouter un autre élément ? o/n');
    readln(c);
    if (c = 'o') then Ajouter_personne;
  until (c = 'n');
  Afficher_Liste;
end.
```

Pour obtenir une liste circulaire, il faut ajouter, après la construction de la liste, les instructions : Queue^.Suivant := Tete ; Tete^.Precedent := Queue ;

**Solution 16 :**

Procédure Insérer\_AV\_Personne (Nu1 : entier ; No1 : chaîne de caractères ; Nu2 : entier ; No2 : chaîne de caractères) ;

Variables

P1, P2 : ↑ Personne ;

Début

Si (Tete↑. Num = Nu2) ET (Tete↑. Nom = No2) Alors début

```

Allouer(P1);
P1↑.Num ← Nu1 ;
P1↑.Nom ← No1 ;
P1↑.Precedent ← nil ;
P1↑.Suivant ← Tete ;
Tete↑.Precedent ← P1;
Tete ← P1;
fin
Sinon début
{ Pointer avant l'élément Pr2 }
P2 ← Tete ;
Tant que (P2↑.Suivant↑.Num <> Nu2) ET (P2↑.Suivant↑.Nom <> No2) Faire
    P2 ← P2↑.Suivant ;
{ Créer un nouvel élément contenant Pr1 }
Allouer(P1);
P1↑.Num ← Nu1 ;
P1↑.Nom ← No1 ;
{ Insérer le nouvel élément dans la liste }
P1↑.Precedent ← P2 ;
P1↑.Suivant ← P2↑.Suivant ;
P2↑.Suivant↑.Precedent ← P1;
P2↑.Suivant ← P1;
fin ;
fin ;

```

En Pascal :

```

procedure Insérer_AV_Personne (Nu1 : integer; No1 : string; Nu2 :
integer; No2 : string) ;

```

```

var

```

```

    P1, P2 : Ptr_Personne;

```

```

begin

```

```

    if (Tete^. Num = Nu2) and (Tete^. Nom = No2) then begin

```

```

        new(P1);

```

```

        P1^.Num := Nu1 ;

```

```

        P1^.Nom := No1 ;

```

```

        P1^.Precedent := nil ;

```

```

        P1^.Suivant := Tete ;

```

```

        Tete^.Precedent := P1;

```

```

        Tete := P1;

```

```

    end
end

```

```
else begin
{ Pointer avant l'élément Pr2 }
  P2 := Tete ;
  while (P2^.Suivant^.Num <> Nu2) and (P2^.Suivant^.Nom <> No2) do
    P2 := P2^.Suivant;
{ Créer un nouvel élément contenant Pr1 }
  new(P1);
  P1^.Num := Nu1 ;
  P1^.Nom := No1 ;
{ Insérer le nouvel élément dans la liste }
  P1^.Precedent := P2 ;
  P1^.Suivant := P2^.Suivant ;
  P2^.Suivant^.Precedent := P1;
  P2^.Suivant := P1;
end ;
end ;
```

Il est à noter que l'élément avant lequel on désire faire l'insertion est supposé existant. Il est possible de traiter le cas où cet élément n'existe pas dans la liste doublement chaînée par un message d'erreur.

# Chapitre 11 : Les arbres

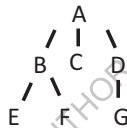
## 1. Introduction

En informatique, on est souvent amené à manipuler des objets que les structures de données étudiées dans les chapitres précédents sont insuffisantes pour les représenter. Il s'agit en particulier des arbres qui sont des structures de données récursives générales.

De nombreuses informations peuvent être organisées sous forme arborescente, par exemple : un arbre généalogique, un dictionnaire, les répertoires sous la plupart des systèmes d'exploitation actuels forment un arbre, etc.

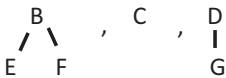
## 2. Définitions

**Un arbre :** C'est un ensemble de nœuds (appelés aussi sommets) reliés par des arêtes (appelées aussi branches ou arcs) formant une hiérarchie. Un arbre est donc soit vide, soit constitué d'un élément auquel sont chaînés un ou plusieurs arbres. Un arbre peut être représenté graphiquement comme suit :



Chaque sommet ou nœud (A, B, C, D, E, F, G) possède un certain nombre de fils (B, C et D pour le nœud A ; E et F pour le nœud B ; G pour le nœud D ; rien pour E, F et G). Lorsqu'un nœud n'a pas de fils (ici E, F, C et G), on dit que c'est une feuille de l'arbre, ou encore un nœud terminal.

Inversement, chaque nœud possède un seul père : A est le père de B, C et D ; B est le père de E et F ; D est le père de G. Le nœud particulier qui n'a pas de père (ici A) est appelé racine de l'arbre.



sont appelés des sous-arbres  
de l'arbre ci-dessus

**Niveau ou profondeur :** On dit que deux nœuds sont à un même niveau dans un arbre, s'ils sont issus d'un même nœud après le même nombre de filiations. Le niveau de la racine d'arbre est égal à 1. Le niveau d'un nœud, autre que la racine, est égal au niveau de son père plus 1. Par exemple, pour l'arbre ci-dessus, les nœuds E, F et G sont de même niveau (3). Ils descendent de A après deux filiations.

**Mot des feuilles d'un arbre :** Le mot des feuilles d'un arbre est la chaîne formée, de gauche à droite, des valeurs des feuilles de l'arbre. Par exemple, pour l'arbre de la figure ci-dessus, le mot des feuilles est égal à : EF CG.

**Taille d'un arbre :** La taille d'un arbre est égale au nombre de nœuds de cet arbre. La taille d'un arbre vide est égale à 0. La taille de l'arbre précédent est égale à 7.

**Hauteur d'un arbre :** La hauteur d'un arbre est égale au maximum des niveaux des feuilles. Ou encore, c'est la distance entre la feuille la plus éloignée et la racine. La hauteur d'un arbre vide est égale à 0. La hauteur de l'arbre précédent est égale à 3.

**Un arbre n-aire :** Un arbre n-aire est un arbre pour lequel chaque nœud possède au plus  $n$  fils, avec  $n > 2$ . L'arbre ci-dessus est donc un arbre 3-aire, ou encore ternaire.

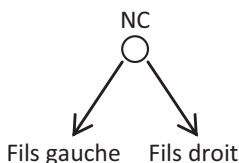
### 3. Arbre binaire

#### 3.1. Définition

Un arbre binaire est un arbre n-aire pour lequel  $n$  est égal à deux. Il peut être représenté sous forme graphique ou sous forme parenthésée.



Pour les arbres binaires, nous parlerons de fils gauche et de fils droit, ainsi de sous-arbre gauche et de sous-arbre droit. A partir de chaque nœud, on peut accéder au fils gauche et au fils droit. Ce qui peut être schématisé comme suit :



#### 3.2. Passage d'un arbre n-aire à un arbre binaire

N'importe quel arbre n-aire peut être représenté sous forme binaire en utilisant trois primitives :

- Nœud courant qu'on notera NC.
- Premier fils gauche qu'on notera PFG.
- Premier frère droit qu'on notera PFD.



On se ramène alors à un arbre binaire où PFG devient fils gauche du NC, et PFD devient le fils droit du NC.

**Exemple :**

Voyons l'exemple suivant illustrant une transformation d'un arbre n-aire à un arbre binaire équivalent :



Puisque chaque arbre n-aire peut être représenté par un arbre binaire équivalent, nous limiterons notre étude dans ce chapitre aux arbres binaires.

**3.3. Représentation chaînée d'un arbre binaire**

Un arbre binaire peut être représenté en mémoire par une liste chaînée. Chaque nœud de la liste est un enregistrement qui contient des informations à stocker et deux pointeurs vers les nœuds fils. Cet enregistrement est alors défini comme suit :

Type

Nœud = enregistrement

Inf1 : Type\_element1 ;

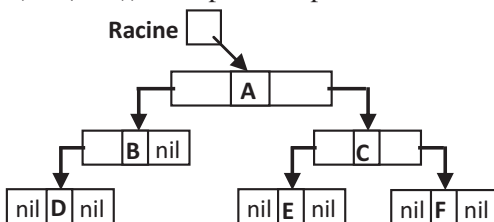
Inf2 : Type\_element2 ;

...

Gauche, Droit : ↑ Nœud ;

Fin ;

L'arbre A(B(D, ), C(E, F)) est représenté par la liste chaînée suivante :

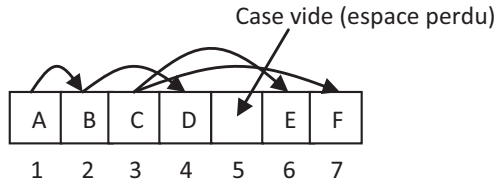


Où racine est une variable de type pointeur contenant l'adresse du nœud racine de l'arbre. C'est le point d'entrée dans l'arbre.

**Remarque :**

Les arbres binaires peuvent aussi être rangés dans des tableaux. Dans cet arrangement compact, un nœud a un indice  $i$ , et ses fils se trouvent aux

indices  $2i$  et  $2i+1$ . Cette méthode permet un accès direct aux données. Toutefois, elle pose un problème dans le cas où le nombre de nœuds dépasse la taille du tableau. Il peut y avoir aussi un espace mémoire perdu.



### Exercice :

Ecrire un algorithme ensuite le programme Pascal permettant de créer l'arbre ci-dessus en utilisant une représentation chaînée.

### Solution :

Algorithme Arbre\_Car ;

Type

Arbre = Enregistrement

Car : caractère ;

F\_gauche, F\_droit : ↑ Arbre;

fin ;

Variables

Racine : ↑ Arbre ;

(\*Procédure d'ajout d'un fils gauche à un noeud\*)

Procédure Ajouter\_FG (R : ↑ Arbre ; C : caractère) ;

début

Allouer(R↑.F\_gauche);

R↑.F\_gauche↑.Car ← C ;

R↑.F\_gauche↑.F\_gauche ← nil ;

R↑.F\_gauche↑.F\_droit ← nil ;

fin ;

(\*Procédure d'ajout d'un fils droit à un noeud\*)

Procédure Ajouter\_FD (R : ↑ Arbre; C : caractère) ;

début

Allouer(R↑.F\_droit);

R↑.F\_droit↑.Car ← C ;

R↑.F\_droit↑.F\_gauche ← nil ;

R↑.F\_droit↑.F\_droit ← nil ;

fin ;

début

```
(*Créer et remplir la racine de l'arbre*)
Allouer(Racine);
Racine↑.Car ← 'A' ;
Racine↑.F_gauche ← nil ;
Racine↑.F_droit ← nil ;
(*Ajouter le reste des éléments à l'arbre*)
Ajouter_FG(Racine, 'B');
Ajouter_FG(Racine↑.F_gauche, 'D');
Ajouter_FD(Racine, 'C');
Ajouter_FG(Racine↑.F_droit, 'E');
Ajouter_FD(Racine↑.F_droit, 'F');
Fin.
```

En Pascal :

```
program Arbre_Car ;
type
  Ptr_Arbre = ^ Arbre ;
  Arbre = record
    Car : char ;
    F_gauche, F_droit : Ptr_Arbre;
  end ;
var
  Racine : Ptr_Arbre ;
(*Procédure d'ajout d'un fils gauche à un noeud*)
procedure Ajouter_FG (R : Ptr_Arbre; C : char) ;
begin
  new(R^.F_gauche);
  R^.F_gauche^.Car := C ;
  R^.F_gauche^.F_gauche := nil ;
  R^.F_gauche^.F_droit := nil ;
end;
(*Procédure d'ajout d'un fils droit à un noeud*)
procedure Ajouter_FD (R : Ptr_Arbre; C : char) ;
begin
  new(R^.F_droit);
  R^.F_droit^.Car := C ;
  R^.F_droit^.F_gauche := nil ;
  R^.F_droit^.F_droit := nil ;
end;
begin
```

```
(*Créer et remplir la racine de l'arbre*)
new(Racine);
Racine^.Car := 'A' ;
Racine^.F_gauche := nil ;
Racine^.F_droit := nil ;
(*Ajouter le reste des éléments à l'arbre*)
Ajouter_FG(Racine, 'B');
Ajouter_FG(Racine^.F_gauche, 'D');
Ajouter_FD(Racine, 'C');
Ajouter_FG(Racine^.F_droit, 'E');
Ajouter_FD(Racine^.F_droit, 'F');
end.
```

### 3.4. Parcours d'un arbre binaire

Il existe deux grands types de parcours d'un arbre binaire :

- *Parcours en profondeur* : Avec ce parcours, nous tentons toujours à visiter le nœud le plus éloigné de la racine que nous pouvons, à la condition qu'il soit le fils d'un nœud que nous avons déjà visité.
- *Parcours en largeur* : Contrairement au précédent, ce parcours essaie toujours de visiter le nœud le plus proche de la racine qui n'a pas été déjà visité. En suivant ce parcours, on va d'abord visiter la racine, puis les nœuds à la profondeur 1, puis 2, etc. D'où le nom parcours en largeur.

Les algorithmes de parcours d'un arbre binaire sont très importants, car ils servent de base à l'écriture de très nombreux algorithmes. On écrira très souvent les algorithmes sur les arbres binaires sous forme récursive qui est beaucoup plus concise et naturelle que la forme itérative. Pour l'écriture récursive, on utilisera la définition suivante d'un arbre : un arbre est soit vide, soit composé d'un élément auquel sont chaînés un sous-arbre gauche et un sous-arbre droit.

Dans ce cours, on s'intéresse à trois manières classiques pour parcourir un arbre binaire qui sont des cas particuliers du parcours en profondeur : parcours préfixé, parcours infixé et parcours postfixé.

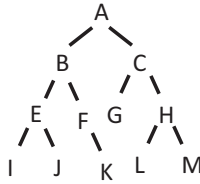
#### 3.4.1. Parcours préfixé (appelé aussi préordre ou RGD)

Ce parcours consiste à effectuer dans l'ordre :

- Traitement de la racine.
- Parcours du sous-arbre gauche.
- Parcours du sous-arbre droit.

#### Exemple :

Soit l'arbre suivant :



Cet arbre peut être traité (en parcours préfixé) dans l'ordre suivant : A B E I J F K C G H L M. La procédure du parcours préfixé d'un arbre binaire s'écrit comme suit :

```

Procédure RGD (R : ↑ Nœud) ;
Début
  Si R <> nil Alors début
    Traiter(R) ;
    RGD(R↑.Gauche) ;
    RGD(R↑.Droit) ;
  fin ;
Fin ;
  
```

Traiter() étant une procédure qui effectue le traitement désiré sur le nœud pris comme paramètre.

### Exercice :

Ecrire la procédure permettant d'afficher en RGD l'arbre de caractères créé ci-dessus.

### Solution :

```

(*Procédure d'affichage des éléments d'un arbre en RGD*)
Procédure Afficher_RGD (R : ↑ Arbre) ;
début
  Si (R <> nil) Alors début
    Ecrire(R↑.Car);
    Afficher_RGD(R↑.F_gauche);
    Afficher_RGD(R↑.F_droit);
  fin;
fin;
  
```

### En Pascal :

```

(*Procédure d'affichage des éléments d'un arbre en RGD*)
procedure Afficher_RGD (R : Ptr_Arbre) ;
begin
  if (R <> nil) then begin
    writeln(R^.Car);
    Afficher_RGD(R^.F_gauche);
    Afficher_RGD(R^.F_droit);
  end;
end;
  
```

### 3.4.2. *Parcours infixe (appelé aussi projectif, symétrique ou encore GRD)*

Ce parcours consiste à effectuer dans l'ordre :

- Parcours du sous-arbre gauche.
- Traitement de la racine.
- Parcours du sous-arbre droit.

Pour l'arbre précédent, on traite les nœuds (en parcours infixe) dans l'ordre suivant : I E J B F K A G C L H M. La procédure du parcours infixe d'un arbre binaire s'écrit comme suit :

Procédure GRD (R : ↑ Nœud) ;

Début

Si R <> nil Alors début

GRD(R↑.Gauche) ;

Traiter(R) ;

GRD(R↑.Droit) ;

fin ;

Fin ;

#### **Exercice :**

Ecrire la procédure permettant d'afficher en GRD l'arbre de caractères créé ci-dessus.

#### **Solution :**

(\*Procédure d'affichage des éléments d'un arbre en GRD\*)

Procédure Afficher\_GRD (R : ↑ Arbre) ;

début

Si (R <> nil) Alors début

Afficher\_GRD(R↑.F\_gauche);

Ecrire(R↑.Car);

Afficher\_GRD(R↑.F\_droit);

fin;

fin;

#### En Pascal :

(\*Procédure d'affichage des éléments d'un arbre en GRD\*)

procedure Afficher\_GRD (R : Ptr\_Arbre) ;

begin

if (R <> nil) then begin

Afficher\_GRD(R^.F\_gauche);

writeln(R^.Car);

Afficher\_GRD(R^.F\_droit);

end;

end;

### 3.4.3. Parcours postfixé (appelé aussi ordre terminal ou GDR)

Ce parcours consiste à effectuer dans l'ordre :

- Parcours du sous-arbre gauche.
- Parcours du sous-arbre droit.
- Traitement de la racine.

Pour l'arbre précédent, on traite les nœuds (en parcours postfixé) dans l'ordre suivant : I J E K F B G L M H C A. La procédure du parcours postfixé d'un arbre binaire s'écrit comme suit :

Procédure GDR (R : ↑ Nœud) ;

Début

Si R <> nil Alors début

GDR(R↑.Gauche) ;

GDR(R↑.Droit) ;

Traiter(R) ;

fin ;

Fin ;

#### Exercice :

Ecrire la procédure permettant d'afficher en GDR l'arbre de caractères créé ci-dessus.

#### Solution :

(\*Procédure d'affichage des éléments d'un arbre en GDR\*)

Procedure Afficher\_GDR (R : ↑ Arbre) ;

début

Si (R <> nil) Alors début

Afficher\_GDR(R↑.F\_gauche);

Afficher\_GDR(R↑.F\_droit);

Ecrire(R↑.Car);

fin;

fin;

#### En Pascal :

(\*Procédure d'affichage des éléments d'un arbre en GDR\*)

procedure Afficher\_GDR (R : Ptr\_Arbre) ;

begin

if (R <> nil) then begin

Afficher\_GDR(R^.F\_gauche);

Afficher\_GDR(R^.F\_droit);

writeln(R^.Car);

end;

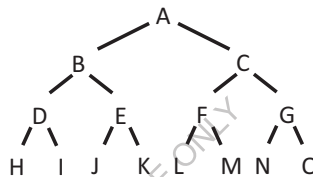
end;

**Remarque :**

Toutes ces procédures récursives utilisent une pile mémoire proportionnelle à la profondeur des arbres. Si nous rajoutons dans chaque nœud une référence à son parent (pointeur vers père), alors nous pouvons implémenter tous ces parcours par des algorithmes itératifs. La référence au parent occupe cependant beaucoup d'espace ; elle n'est réellement utile que si la pile mémoire est particulièrement limitée.

**3.5. Arbres binaires particuliers****3.5.1. Arbre binaire complet**

On dit qu'un arbre binaire est complet si chaque nœud autre qu'une feuille possède deux descendants, et si toutes les feuilles sont au même niveau.

**Exemple :**

La taille d'un arbre binaire complet est égale à  $2^n - 1$ , où  $n$  est le niveau des feuilles.

**3.5.2. Arbre dégénéré**

Un arbre est dit dégénéré si tous les nœuds de cet arbre ont au plus un fils.

**Exemple :**

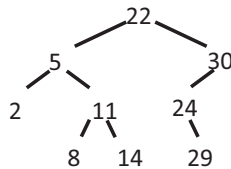
Un arbre dégénéré est équivalent à une liste simplement chaînée.

**3.5.3. Arbre binaire ordonné**

Soit une relation d'ordre (noté  $\leq$ ) sur l'ensemble des valeurs attachées aux nœuds d'un arbre binaire. Un arbre binaire est dit ordonné (appelé aussi arbre binaire de recherche) si la chaîne infixée des valeurs, correspondant au parcours infixé, est ordonnée.



**Exemple :**



Le parcours infixe de l'arbre ci-dessus donne : 2-5-8-11-14-22-24-29-30. Notons que si  $n$  est un nœud dans un arbre binaire de recherche, alors tous les éléments dans le sous-arbre gauche du nœud  $n$  seront inférieurs à  $n$ , et ceux dans le sous-arbre droit seront supérieurs ou égaux à  $n$ .

**4. Exercices corrigés**

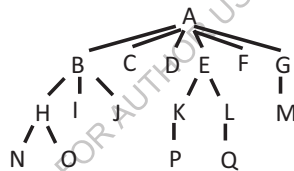
**4.1. Exercices**

**Exercice 1 :**

Proposez une structure pour stocker un arbre  $n$ -aire contenant des valeurs entières.

**Exercice 2 :**

Faites la transformation de l'arbre  $n$ -aire suivant en arbre binaire équivalent.



Arbre 6-aire

**Exercice 3 :**

Ecrire un algorithme permettant de créer et d'afficher les éléments d'un arbre binaire de caractères. Pour la création d'un nœud, utilisez une procédure récursive. Pour l'affichage, faites le parcours RGD de l'arbre binaire. Traduire l'algorithme en Pascal.

**Exercice 4 :**

Ecrire une fonction booléenne permettant la recherche d'un élément dans un arbre binaire. La fonction possède comme paramètres un pointeur vers un nœud de l'arbre binaire et un caractère indiquant l'élément recherché. Traduire la fonction en Pascal.

**Exercice 5 :**

Modifier la fonction de l'exercice précédent de telle sorte qu'elle retourne un pointeur vers l'élément recherché. La fonction possède comme paramètres un pointeur vers un nœud de l'arbre binaire et un caractère indiquant l'élément recherché.

**Exercice 6 :**

Ecrire une fonction qui retourne le niveau d'un nœud dans un arbre binaire. La fonction possède comme paramètres un pointeur vers un nœud de l'arbre binaire et le contenu du nœud pour lequel on désire déterminer le niveau. Traduire la fonction en Pascal.

**Exercice 7 :**

Ecrire une fonction qui retourne le nombre de feuilles d'un arbre binaire. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 8 :**

Ecrire une fonction qui retourne le mot des feuilles d'un arbre binaire. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 9 :**

Ecrire une fonction qui retourne la taille d'un arbre binaire. La taille d'un arbre binaire est égale à 1 (l'élément racine), plus la taille du sous-arbre gauche, plus la taille du sous-arbre droit. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 10 :**

Ecrire une fonction booléenne permettant de déterminer si un arbre binaire est dégénéré ou non. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 11 :**

Ecrire une fonction qui retourne la hauteur d'un arbre binaire. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 12 :**

Ecrire une fonction booléenne permettant de dire si un arbre binaire est équilibré ou non. Un arbre est équilibré si, pour chacun de ses nœuds, la différence entre la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit est d'au plus une unité. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 13 :**

Soit un arbre binaire d'entiers dont la structure de chaque élément est décrite comme suit :

Type

Arbre = Enregistrement

Val : entier ;

F\_gauche, F\_droit : ↑ Arbre;

fin ;

Ecrire une fonction qui calcule la somme des valeurs des nœuds d'un arbre binaire d'entiers. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 14 :**

Ecrire une fonction qui détermine la plus grande des valeurs des nœuds d'un arbre binaire d'entiers. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 15 :**

Ecrire une fonction qui détermine la plus petite des valeurs des nœuds d'un arbre binaire d'entiers. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

**Exercice 16 :**

Ecrire une fonction booléenne permettant de dire si un arbre binaire d'entiers est ordonné ou non. La fonction possède comme paramètre un pointeur vers un nœud de l'arbre binaire. Traduire la fonction en Pascal.

## 4.2. Corrigés

### Solution 1 :

Une première proposition consiste à définir la structure comme suit :

- Une variable entière correspondant au contenu du nœud.
- Un lien vers chacun des nœuds fils.

Type

```
Arbre_naire = Enregistrement
  Val : entier ;
  Fils : Tableau [1..n] de ↑ Arbre_naire;
fin ;
```

On peut également définir la structure comme suit :

- Une variable entière correspondant au contenu du nœud.
- Un lien vers le nœud fils.
- Un autre lien vers le nœud frère.

Type

```
Arbre_naire = Enregistrement
  Val : entier ;
  Fils, Frere : ↑ Arbre_naire;
fin ;
```

Une autre manière pour définir la structure :

- Une variable entière correspondant au contenu du nœud.
- Un lien vers le nœud père.

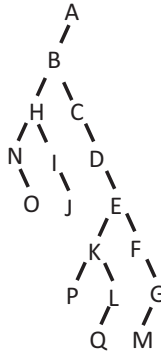
Type

```
Arbre_naire = Enregistrement
  Val : entier ;
```

Pere :  $\uparrow$  Arbre\_naire;  
fin ;

### Solution 2 :

Arbre binaire équivalent :



### Solution 3 :

Algorithme Arbre\_Car ;

Type

Arbre = Enregistrement

Car : caractère ;

F\_gauche, F\_droit :  $\uparrow$  Arbre;

fin ;

Variables

Racine :  $\uparrow$  Arbre ;

(\*Procédure récursive pour la création d'un arbre binaire\*)

Procédure Creer\_Noed (var R :  $\uparrow$  Arbre ) ;

Variables

C, reponse : caractère ;

début

Ecrire('Donnez la valeur du noeud :');

Lire(C);

Allouer(R);

$R \uparrow$ .Car  $\leftarrow$  C ;

$R \uparrow$ .F\_gauche  $\leftarrow$  nil ;

$R \uparrow$ .F\_droit  $\leftarrow$  nil ;

Ecrire('Le noeud ', C, ' possède t-il un fils gauche ? o/n');

Lire(reponse);

Si (reponse = 'o') Alors Creer\_Noed( $R \uparrow$ .F\_gauche);

Ecrire('Le noeud ', C, ' possède t-il un fils droit ? o/n');

```

Lire(reponse);
Si (reponse = 'o') Alors Creer_Noed(R↑.F_droit);
fin ;
(*Procédure d'affichage des éléments d'un arbre binaire en RGD*)
Procédure Afficher_RGD (R : ↑Arbre) ;
début
  Si (R <> nil) Alors début
    Ecrire(R↑.Car);
    Afficher_RGD(R↑.F_gauche);
    Afficher_RGD(R↑.F_droit);
  fin ;
fin ;
Début
  (*Créer et remplir l'arbre*)
  Ecrire('Créer un arbre binaire : le premier élément est la racine de l'arbre. ');
  Creer_Noed(Racine);
  Afficher_RGD(Racine);
Fin.

```

Le programme Pascal :

```

program Arbre_Car ;
type
  Ptr_Arbre = ^ Arbre ;
  Arbre = record
    Car : char ;
    F_gauche, F_droit : Ptr_Arbre;
  end ;
var
  Racine : Ptr_Arbre ;
  (*Procédure récursive pour la création d'un arbre binaire*)
  procédure Creer_Noed (var R : Ptr_Arbre ) ;
var
  C, reponse : char;
begin
  writeln('Donnez la valeur du noeud : ');
  readln(C);
  new(R);
  R^.Car := C ;
  R^.F_gauche := nil ;
  R^.F_droit := nil ;

```

```
writeln('Le noeud ', C, ' possède t-il un fils gauche ? o/n');
readln(reponse);
if (reponse = 'o') then Creer_Noeud(R^.F_gauche);
writeln('Le noeud ', C, ' possède t-il un fils droit ? o/n');
readln(reponse);
if (reponse = 'o') then Creer_Noeud(R^.F_droit);
end;
(*Procédure d'affichage des éléments d'un arbre binaire en RGD*)
procedure Afficher_RGD (R : Ptr_Arbre) ;
begin
  if (R <> nil) then begin
    writeln(R^.Car);
    Afficher_RGD(R^.F_gauche);
    Afficher_RGD(R^.F_droit);
  end;
end;
begin
  (*Créer et remplir l'arbre*)
  writeln('Créer un arbre binaire : le premier élément est la racine de l'arbre.');
```

FOR AUTHOR USE ONLY

**Solution 4 :**

(\*Fonction de recherche d'un élément dans un arbre binaire\*)  
Fonction Recherche\_Arbre (R : ↑Arbre; C : caractère) : boolean ;  
Variables

Trouve : boolean;

Début

Trouve ← FAUX ;

Si (R <> nil) Alors

Si (R↑.Car = C) Alors Trouve ← VRAI

Sinon Trouve ← Recherche\_Arbre(R↑.F\_gauche,C) OU Recherche\_Arbre(R↑.F\_droit,C);

Recherche\_Arbre ← Trouve ;

Fin ;

En Pascal :

(\*Fonction de recherche d'un élément dans un arbre binaire\*)

function Recherche\_Arbre (R : Ptr\_Arbre; C : char) : boolean ;

var

Trouve : boolean;

begin

Trouve := false ;

if (R <> nil) then

if (R^.Car = C) then Trouve := true

else Trouve := Recherche\_Arbre(R^.F\_gauche, C) or Recherche\_Arbre(R^.F\_droit, C);

Recherche\_Arbre := Trouve ;

end;

### **Ou bien :**

(\*Fonction de recherche d'un élément dans un arbre binaire\*)

Fonction Recherche\_Arbre (R : ↑Arbre; C : caractère) : boolean ;

Variables

Trouve : boolean;

Début

Si (R = nil) Alors Trouve ← FAUX

Sinon Si (R↑.Car = C) Alors Trouve ← VRAI

Sinon Trouve ← Recherche\_Arbre(R↑.F\_gauche,C) OU Recherche\_Arbre(R↑.F\_droit,C);

Recherche\_Arbre ← Trouve ;

Fin ;

### **En Pascal :**

(\*Fonction de recherche d'un élément dans un arbre binaire\*)

function Recherche\_Arbre (R : Ptr\_Arbre; C : char) : boolean ;

var

Trouve : boolean;

begin

if (R = nil) then Trouve := false

else if (R^.Car = C) then Trouve := true

else Trouve := Recherche\_Arbre(R^.F\_gauche, C) or Recherche\_Arbre(R^.F\_droit, C);

Recherche\_Arbre := Trouve ;

end;

### **Solution 5 :**

(\*Fonction de recherche d'un élément dans un arbre binaire\*)

Fonction Recherche\_Arbre (R : ↑Arbre; C : caractère) : ↑Arbre ;

Variables

P : ↑Arbre ;

Début

P ← nil ;

Si (R <> nil) Alors

Si (R↑.Car = C) Alors P ← R

Sinon début

```

    P ← Recherche_Arbre(R↑.F_gauche, C) ;
    Si P = nil Alors P ← Recherche_Arbre(R↑.F_droit, C);
  fin ;
  Recherche_Arbre ← P ;
Fin ;

```

En Pascal :

(\*Fonction de recherche d'un élément dans un arbre binaire\*)

```

function Recherche_Arbre (R : Ptr_Arbre; C : char) : Ptr_Arbre ;
var
  P : Ptr_Arbre ;
begin
  P := nil ;
  if (R <> nil) then
    if (R^.Car = C) then P := R
    else begin
      P := Recherche_Arbre(R^.F_gauche, C) ;
      if P = nil then P := Recherche_Arbre(R^.F_droit, C);
    end;
  Recherche_Arbre := P ;
end;

```

**Solution 6 :**

(\*Fonction permettant de déterminer le niveau d'un nœud dans un arbre binaire\*)

Fonction niveau\_noeud(R : ↑Arbre; C : caractère; np : entier) : entier ;

Variables n : entier;

Début

Si (R <> nil) Alors

Si R↑.car = C Alors n ← np + 1

Sinon

Si niveau\_noeud (R↑.F\_droit, C, np+1) < niveau\_noeud (R↑.F\_gauche, C, np+1) Alors

n ← niveau\_noeud (R↑.F\_droit, C, np+1)

Sinon n ← niveau\_noeud (R↑.F\_gauche, C, np+1);

niveau\_noeud ← n ;

fin;

En Pascal :

(\*Fonction permettant de déterminer le niveau d'un nœud dans un arbre binaire\*)

function niveau\_noeud(R : Ptr\_Arbre; C : char; np : integer) : integer ;

var n : integer;

begin

if (R <> nil) then



```

if R^.car = C then n := np + 1
else
  if niveau_noeud (R^.F_droit, C, np+1) < niveau_noeud (R^.F_gauche, C, np+1) then
    n := niveau_noeud (R^.F_droit, C, np+1)
  else n := niveau_noeud (R^.F_gauche, C, np+1);
niveau_noeud := n ;
end;
Avec np est le niveau du père. Lors de l'invocation de la fonction, np = 0.

```

### **Solution 7 :**

(\*Fonction de calcul du nombre de feuilles d'un arbre binaire\*)

fonction Nbr\_Feuilles\_Arbre (R : ↑ Arbre) : entier ;

Variables

Nbr : entier ;

début

Si (R = nil) Alors Nbr ← 0

Sinon Si (R↑.F\_gauche = nil) ET (R↑.F\_droit = nil) Alors Nbr ← 1

Sinon Nbr ← Nbr\_Feuilles\_Arbre(R↑.F\_gauche) + Nbr\_Feuilles\_Arbre(R↑.F\_droit);

Nbr\_Feuilles\_Arbre ← Nbr ;

fin ;

En Pascal :

(\* Fonction de calcul du nombre de feuilles d'un arbre binaire\*)

function Nbr\_Feuilles\_Arbre (R : Ptr\_Arbre) : integer;

var

Nbr : integer;

begin

if (R = nil) then Nbr := 0

else if (R^.F\_gauche = nil) and (R^.F\_droit = nil) then Nbr := 1

else Nbr := Nbr\_Feuilles\_Arbre(R^.F\_gauche) + Nbr\_Feuilles\_Arbre(R^.F\_droit);

Nbr\_Feuilles\_Arbre := Nbr ;

end;

### **Solution 8 :**

(\*Fonction qui retourne le mot des feuilles d'un arbre binaire\*)

Fonction Mot\_Feuilles\_Arbre (R : ↑ Arbre) : chaîne de caractères ;

Variables

Mot : chaîne de caractères ;

début

Si (R = nil) Alors Mot ← ""

Sinon Si (R↑.F\_gauche = nil) ET (R↑.F\_droit = nil) Alors Mot ← R↑.Car

Sinon Mot ← Mot\_Feuilles\_Arbre(R↑.F\_gauche) + Mot\_Feuilles\_Arbre(R↑.F\_droit);

```
Mot_Feuilles_Arbre ← Mot ;  
fin ;  
En Pascal :  
(*Fonction qui retourne le mot des feuilles d'un arbre binaire*)  
function Mot_Feuilles_Arbre (R : Ptr_Arbre) : string ;  
var  
  Mot : string;  
begin  
  if (R = nil) then Mot := "  
    else if (R^.F_gauche = nil)and(R^.F_droit = nil) then Mot := R^.Car  
    else Mot := Mot_Feuilles_Arbre(R^.F_gauche) + Mot_Feuilles_Arbre(R^.F_droit);  
  Mot_Feuilles_Arbre := Mot ;  
end;
```

**Solution 9 :**

```
(*Fonction de calcul de la taille d'un arbre binaire*)  
Fonction Taille_Arbre (R : ↑ Arbre) : entier ;  
Variables  
  T : entier ;  
début  
  Si (R = nil) Alors T := 0  
  Sinon T ← 1 + Taille_Arbre(R↑.F_gauche) + Taille_Arbre(R↑.F_droit);  
  Taille_Arbre ← T ;  
fin;
```

En Pascal :

```
(*Fonction de calcul de la taille d'un arbre binaire*)  
function Taille_Arbre (R : Ptr_Arbre) : integer;  
var  
  T : integer;  
begin  
  if (R = nil) then T := 0  
  else T := 1 + Taille_Arbre(R^.F_gauche) + Taille_Arbre(R^.F_droit);  
  Taille_Arbre := T ;  
end;
```

**Solution 10 :**

```
(*Fonction qui détermine si un arbre est dégénéré ou non*)  
Fonction Degenere_Arbre (R : ↑ Arbre) : boolean ;  
Variables  
  Degenere : boolean;  
début
```

```

Degenere ← VRAI ;
Si (R <> nil) Alors
  Si (R↑.F_gauche <> nil) ET (R↑.F_droit <> nil) Alors Degenere ← FAUX
  Sinon Degenere ← Degenere_Arbre(R↑.F_gauche) ET Degenere_Arbre(R↑.F_droit);
Degenere_Arbre ← Degenere ;
fin ;

```

En Pascal :

```

(*Fonction qui détermine si un arbre est dégénéré ou non*)
function Degenere_Arbre (R : Ptr_Arbre) : boolean ;
var
  Degenere : boolean;
begin
  Degenere := true ;
  if (R <> nil) then
    if (R↑.F_gauche <> nil) and (R↑.F_droit <> nil) then Degenere := false
    else Degenere := Degenere_Arbre(R↑.F_gauche) and Degenere_Arbre(R↑.F_droit);
  Degenere_Arbre := Degenere ;
end;

```

**Solution 11 :**

(\*Fonction permettant de calculer la hauteur d'un arbre binaire\*)

Fonction Hauteur\_Arbre (R : ↑Arbre) : entier ;

Variables

H1, H2 : entier ;

Début

H1 ← 0 ;

H2 ← 0 ;

Si (R <> nil) Alors début

H1 ← 1 + Hauteur\_Arbre(R↑.F\_gauche) ;

H2 ← 1 + Hauteur\_Arbre(R↑.F\_droit);

fin ;

Si H1 > H2 Alors Hauteur\_Arbre ← H1

Sinon Hauteur\_Arbre ← H2 ;

Fin ;

En Pascal :

(\*Fonction permettant de calculer la hauteur d'un arbre binaire\*)

function Hauteur\_Arbre (R : Ptr\_Arbre) : integer ;

var

H1, H2 : integer ;

begin

```
H1 := 0 ;
H2 := 0 ;
if (R <> nil) then begin
    H1 := 1 + Hauteur_Arbre(R^.F_gauche) ;
    H2 := 1 + Hauteur_Arbre(R^.F_droit);
end;
if H1 > H2 then Hauteur_Arbre := H1
else Hauteur_Arbre := H2 ;
```

end;

**Solution 12 :**

(\*Fonction qui détermine si un arbre est équilibré ou non\*)

```
Fonction equilibre_Arbre (R : ↑Arbre) : boolean ;
```

## Variables

```
equilibre : boolean;
```

Début

```
equilibre ← true ;
```

Si (R  $\neq$  nil) Alors

Si  $(|Hauteur\_Arbre(R \uparrow .F\_gauche) - Hauteur\_Arbre(R \uparrow .F\_droit)| > 1)$  Alors  
equilibre  $\leftarrow$  false

Sinon  $\text{equilibre} \leftarrow \text{equilibre\_Arbre}(R \uparrow . F\_gauche)$  ET

```
equilibre_Arbre(R↑.F_droit);
```

```
equilibre_Arbre ← equilibre ;
```

Fin ;

En Pascal :

(\*Fonction qui détermine si un arbre est équilibré ou non\*)

```
function equilibre_Arbre (R : Ptr_Arbre) : boolean ;
```

var

```
equilibre : boolean;
```

begin

```
equilibre := true ;
```

```
if (R <> nil) then
```

```

if (ABS(Hauteur_Arbre(R^.F_gauche) - Hauteur_Arbre(R^.F_droit)) > 1) then
    equilibre := false

```

```
else equilibre := equilibre_Arbre(R^.F_gauche) and equilibre_Arbre(R^.F_droit);
```

```
equilibre_Arbre := equilibre ;
```

end;

**Solution 13 :**

(\*Fonction qui calcule la somme des éléments d'un arbre binaire\*)

Fonction Somme\_Arbre (R :  $\uparrow$ Arbre) : entier ;

## Variables

S : entier ;

Début

Si (R = nil) Alors S  $\leftarrow$  0

Sinon S  $\leftarrow$  R $\uparrow$ .Val + Somme\_Arbre(R $\uparrow$ .F\_gauche) + Somme\_Arbre(R $\uparrow$ .F\_droit);

Somme\_Arbre  $\leftarrow$  S ;

Fin ;

En Pascal :

(\*Fonction qui calcule la somme des éléments d'un arbre binaire\*)

function Somme\_Arbre (R : Ptr\_Arbre) : integer;

var

S : integer;

begin

if (R = nil) then S := 0

else S := R $\wedge$ .Val + Somme\_Arbre(R $\wedge$ .F\_gauche) + Somme\_Arbre(R $\wedge$ .F\_droit);

Somme\_Arbre := S ;

end;

**Solution 14 :**

(\*Fonction qui détermine le maximum dans un arbre binaire\*)

Fonction Max\_Arbre (R :  $\uparrow$ Arbre) : entier ;

Variables

Max : entier ;

Début

Si (R  $\neq$  nil) Alors début

Max  $\leftarrow$  R $\uparrow$ .Val ;

Si (R $\uparrow$ .F\_gauche  $\neq$  nil) Alors

Si (Max < Max\_Arbre(R $\uparrow$ .F\_gauche)) Alors

Max  $\leftarrow$  Max\_Arbre(R $\uparrow$ .F\_gauche) ;

Si (R $\uparrow$ .F\_droit  $\neq$  nil) Alors

Si (Max < Max\_Arbre(R $\uparrow$ .F\_droit)) Alors

Max  $\leftarrow$  Max\_Arbre(R $\uparrow$ .F\_droit) ;

fin ;

Max\_Arbre  $\leftarrow$  Max ;

Fin ;

En Pascal :

(\*Fonction qui détermine le maximum dans un arbre binaire\*)

function Max\_Arbre (R : Ptr\_Arbre) : integer;

var

Max : integer;

begin

```

if (R <> nil) then begin
  Max := R^.Val ;
  if (R^.F_gauche <> nil) then
    if (Max < Max_Arbre(R^.F_gauche)) then
      Max := Max_Arbre(R^.F_gauche) ;
  if (R^.F_droit <> nil) then
    if (Max < Max_Arbre(R^.F_droit)) then
      Max := Max_Arbre(R^.F_droit) ;
end;
Max_Arbre := Max ;
end;

```

**Solution 15 :**

(\*Fonction qui détermine le minimum dans un arbre binaire\*)

Fonction Min\_Arbre (R : ↑Arbre) : entier ;

Variables

Min : entier ;

Début

Si (R <> nil) Alors début

Min ← R↑.Val ;

Si (R↑.F\_gauche <> nil) Alors

Si (Min > Min\_Arbre(R↑.F\_gauche)) Alors

Min ← Min\_Arbre(R↑.F\_gauche) ;

Si (R↑.F\_droit <> nil) Alors

Si (Min > Min\_Arbre(R↑.F\_droit)) Alors

Min ← Min\_Arbre(R↑.F\_droit) ;

fin ;

Min\_Arbre ← Min ;

Fin ;

En Pascal :

(\*Fonction qui détermine le minimum dans un arbre binaire\*)

function Min\_Arbre (R : Ptr\_Arbre) : integer;

var

Min : integer;

begin

if (R <> nil) then begin

Min := R^.Val ;

if (R^.F\_gauche <> nil) then

if (Min > Min\_Arbre(R^.F\_gauche)) then

Min := Min\_Arbre(R^.F\_gauche) ;

```

    if (R^.F_droit <> nil) then
        if (Min > Min_Arbre(R^.F_droit)) then Min := Min_Arbre(R^.F_droit) ;
    end;
    Min_Arbre := Min ;
end;

```

**Solution 16 :**

(\*Fonction qui détermine si un arbre binaire est ordonné ou non\*)

Fonction Ordonne\_Arbre (R : ↑Arbre) : boolean ;

Variables

O : boolean;

Début

O ← VRAI ;

Si (R <> nil) Alors début

Si (R↑.F\_gauche <> nil) Alors

Si (Max\_Arbre(R↑.F\_gauche) > R↑.Val ) Alors O ← FAUX ;

Si (R↑.F\_droit <> nil) Alors

Si (Min\_Arbre(R↑.F\_droit) < R↑.Val ) Alors O ← FAUX ;

O ← Ordonne\_Arbre (R↑.F\_gauche) ET Ordonne\_Arbre(R↑.F\_droit) ET O ;

fin ;

Ordonne\_Arbre ← O ;

Fin ;

En Pascal :

(\*Fonction qui détermine si un arbre binaire est ordonné ou non\*)

function Ordonne\_Arbre (R : Ptr\_Arbre) : boolean;

var

O : boolean;

begin

O := true ;

if (R <> nil) then begin

if (R^.F\_gauche <> nil) then

if (Max\_Arbre(R^.F\_gauche) > R^.Val ) then O := false ;

if (R^.F\_droit <> nil) then

if (Min\_Arbre(R^.F\_droit) < R^.Val ) then O := false ;

O := Ordonne\_Arbre (R^.F\_gauche) and Ordonne\_Arbre(R^.F\_droit) and O;

end;

Ordonne\_Arbre := O ;

end;

## Chapitre 12 : Les graphes

### 1. Introduction

Les graphes sont des structures de données très générales dont les listes et les arbres ne sont que des cas particuliers. Les graphes permettent de modéliser de nombreux problèmes algorithmiques.

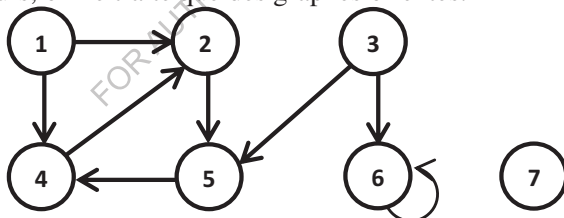
### 2. Définitions

Un *graphe* est un ensemble de nœuds reliés par des liens. Ce n'est plus un arbre dès qu'il existe deux parcours différents pour aller d'au moins un nœud à un autre.

Un graphe est *connexe* lorsqu'il est possible de trouver au moins un parcours permettant de relier les nœuds deux à deux (un arbre est un graphe connexe, deux arbres forment un graphe non connexe).

Un graphe est dit *pondéré* lorsque chaque lien est associé à une valeur (appelée poids). On utilisera un graphe pondéré, par exemple, pour gérer des itinéraires routiers (quelle est la route la plus courte pour aller d'une ville à une autre) ou pour des simulations de trafic routier, pour simuler un circuit électrique, pour prévoir un ordonnancement dans le temps de tâches, etc.

Un graphe est dit *orienté* lorsque les liens sont unidirectionnels. Dans le reste du cours, on ne traite que des graphes orientés.



Exemple d'un graphe orienté.

Un graphe est dit *acyclique* s'il ne contient aucun cycle. Un *cycle* est un chemin permettant de revenir à un sommet de départ en passant par tous les sommets du chemin. Un *chemin* étant une suite d'arcs où l'extrémité d'un arc correspond à l'origine de l'arc suivant, mais pas nécessairement pour le dernier arc du chemin. La longueur d'un chemin est égale au nombre d'arcs qui le composent.

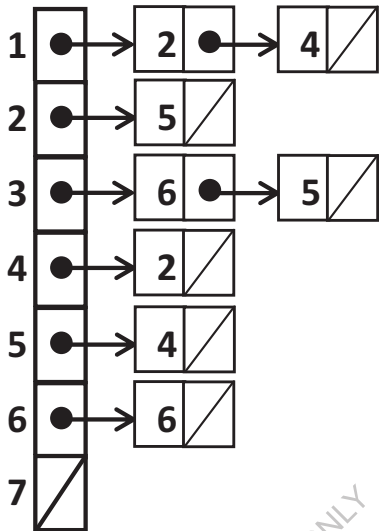
Un graphe est dit *fortement connexe* si pour chaque deux sommets  $i$  et  $j$ , soit  $i=j$ , ou bien il existe un chemin entre  $i$  et  $j$ , et un autre entre  $j$  et  $i$ .

### 3. Représentation d'un graphe

On peut représenter un graphe de manière dynamique par une *liste d'adjacence* qui est un tableau de  $N$  listes, avec  $N$  est le nombre de



sommets. La  $i^{ième}$  liste du tableau correspond aux successeurs du  $i^{ième}$  sommet.



Liste d'adjacence représentant le graphe présenté ci-dessus.

Une autre solution est de numéroté les  $N$  sommets, et d'utiliser une matrice carrée  $N*N$  dite *matrice d'adjacence*, avec la valeur 0 si les nœuds  $i$  et  $j$  ne sont pas reliés, et 1 pour deux nœuds reliés. Pour des graphes non orientés, la matrice est symétrique par rapport à la diagonale.

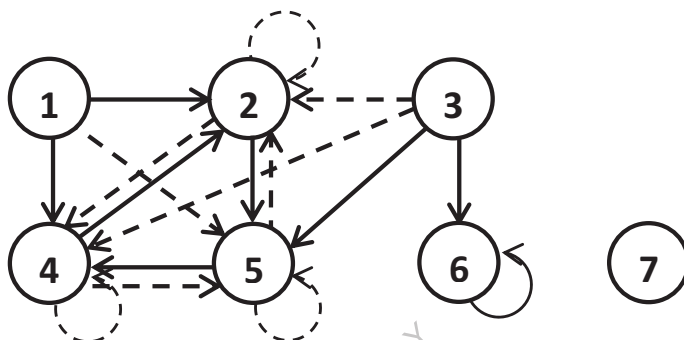
	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	0	0	0	0	1	0	0
3	0	0	0	0	1	1	0
4	0	1	0	0	0	0	0
5	0	0	0	1	0	0	0
6	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0

Matrice d'adjacence représentant le graphe présenté ci-dessus.

Une représentation par matrice est surtout intéressante lorsqu'il y a beaucoup de liens (graphe presque complet). La représentation à l'aide de pointeurs étant moins gourmande en mémoire pour les graphes comportant peu de liens par nœud.

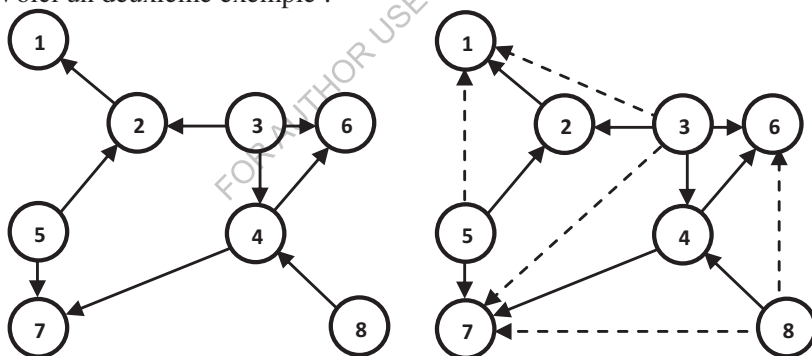
La matrice d'adjacence permet aussi de calculer le nombre de chemins entre deux sommets par  $(M^k)_{i,j}$ , avec  $M$  la matrice d'adjacence, et  $k$  la

longueur du chemin entre les sommets  $i$  et  $j$ . A partir de là, on peut déduire qu'il existe un chemin entre deux sommets  $i$  et  $j$ , si et seulement si  $N_{ij} > 0$ , avec  $N = M + M^2 + \dots + M^n$ . On peut maintenant définir la *fermeture transitive* d'un graphe commettant un nouveau graphe avec les mêmes sommets, où deux sommets sont reliés par un arc, si et seulement s'il existe un chemin entre ces deux sommets dans le graphe original.



Fermeture transitive du graphe présenté ci-dessus.

Voici un deuxième exemple :



Un graphe avec sa fermeture transitive.

#### 4. Parcours d'un graphe

Un problème important est le parcours d'un graphe : il faut éviter les boucles infinies, c'est à dire retourner sur un nœud déjà visité et repartir dans la même direction. Pour éviter ce problème, on peut utiliser par exemple des indicateurs de passage (booléens).

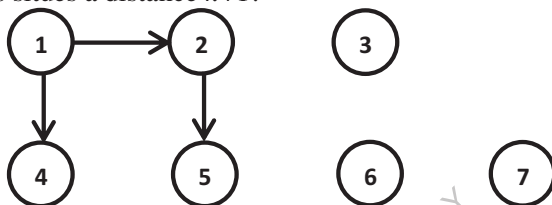
Un graphe peut être parcouru de deux manières selon l'ordre de visite des nœuds : parcours en profondeur d'abord et parcours en largeur d'abord. Grâce à ces stratégies de parcours, il est possible de construire un arbre appelé *l'arbre recouvrant* : c'est un arbre permettant de visiter tous les nœuds, n'utilisant que des liens existants dans le graphe, mais

pas tous. Cet arbre recouvrant n'est évidemment pas unique. En cas de graphe non connexe, il faut rechercher plusieurs arbres recouvrants.

On peut remarquer qu'un arbre recouvrant d'un graphe connexe à  $N$  sommets aura nécessairement  $N-1$  liens. Pour les graphes pondérés, on peut rechercher l'arbre recouvrant de poids minimum (somme minimale des poids des liens). Différents algorithmes existent pour traiter ce problème.

#### 4.1. Parcours en largeur d'abord

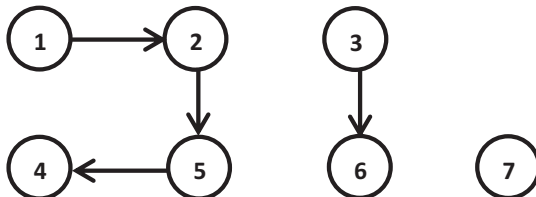
Dans le parcours en largeur d'abord, un sommet  $s$  est fixé comme origine, et l'on visite tous les sommets situés à distance  $k$  de  $s$  avant tous les sommets situés à distance  $k+1$ .



Parcours en largeur d'abord du graphe présenté au début du chapitre. Par exemple, à partir du sommet 1, le parcours du graphe présenté au début de ce chapitre en largeur d'abord permet de visiter les sommets dans l'ordre suivant 1, 2, 4, 5, 3, 6, 7.

#### 4.2. Parcours en profondeur d'abord

Le principe du parcours en profondeur d'abord est de visiter tous les sommets en allant d'abord du plus profondément possible dans le graphe.



Parcours en profondeur d'abord du graphe présenté au début du chapitre. Par exemple, à partir du sommet 1, le parcours du graphe présenté au début de ce chapitre en profondeur d'abord permet de visiter les sommets dans l'ordre suivant : 1, 2, 5, 4, 3, 6, 7.

### 5. Applications des parcours d'un graphe

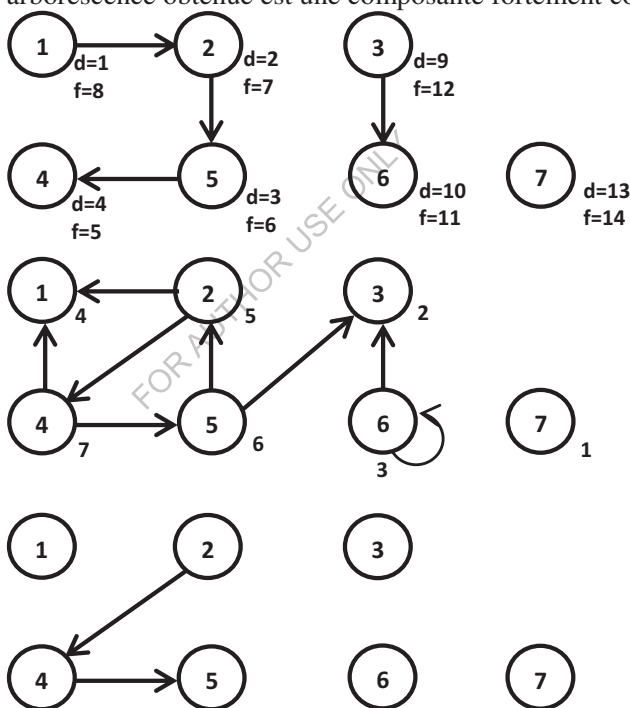
Il est possible d'appliquer les stratégies de parcours d'un graphe pour résoudre plusieurs problèmes, parmi lesquels nous allons voir :

### 5.1. Calcul des composantes fortement connexes

Le but est de subdiviser le graphe en sous graphe où chaque sous graphe est fortement connexe. Pour déterminer les composantes fortement connexes, il est possible d'utiliser le parcours en profondeur comme suit :

1. Effectuer un parcours du graphe en profondeur d'abord. N'oubliez d'utiliser un système de datation permettant de mémoriser la date de début et de fin de visite pour chaque sommet.
2. Calculer le graphe transposé, i.e. le graphe obtenu en inversant le sens de tous les arcs.
3. Effectuer un parcours en profondeur d'abord dans l'ordre décroissant des fins de traitement de l'étape 1.

Chaque arborescence obtenue est une composante fortement connexe :



Recherche des composantes fortement connexes dans le graphe présenté au début du chapitre.

Dans l'exemple précédent, les composantes fortement connexes sont  $\{\{1\}, \{2, 4, 5\}, \{3\}, \{6\}, \{7\}\}$ .

Une autre méthode consiste à utiliser la fermeture transitive pour déterminer les composantes fortement connexes. Pour cela, on doit suivre les étapes suivantes :

1. Déterminer la matrice d'adjacence  $F$  de la fermeture transitive du graphe.
2. Rendre la matrice symétrique par  $F_{ij} = F_{ij} * F_{ji}$ .
3. On ajoute la relation de réflexivité, i.e. pour chaque sommet  $i$ , on met  $F_{ii} = 1$ .

Les composantes fortement connexes sont obtenues ligne par ligne dans la matrice résultante.

## 5.2. Calcul du plus court chemin : algorithme de Dijkstra

L'objectif de cet algorithme est de déterminer le plus court chemin (le chemin le moins coûteux) depuis un sommet appelé sommet initial vers tous les autres sommets dans un graphe pondéré.

Pour cela, nous allons doter chaque sommet d'une étiquette qui indique la valeur du plus court chemin entre le sommet initial et le sommet courant. La valeur de l'étiquette est initialisée à 0 pour le sommet initial et l'infini pour le reste des sommets.

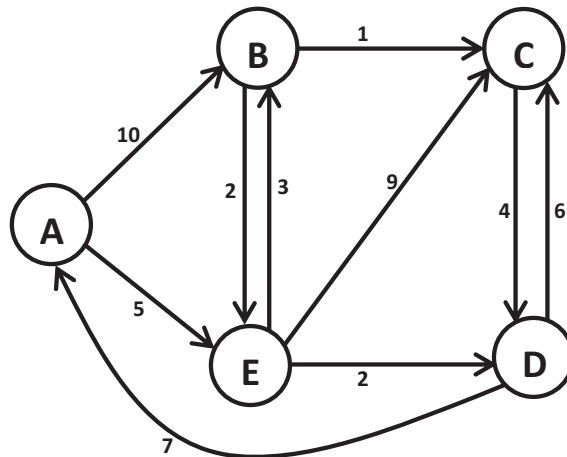
L'algorithme de Dijkstra consiste à :

1. Etablir une matrice où la première ligne contient les sommets du graphe.
2. Dans la deuxième ligne, mettre la valeur 0 dans la case correspondant au sommet initial et l'infini dans les autres cases.
3. Fixer la valeur du sommet initial. Quand une valeur est fixée, elle ne peut pas être changée dans le reste de l'algorithme, i.e. les cases de la colonne correspondant à cette valeur ne vont pas prendre une nouvelle valeur.
4. Le sommet initial va correspondre à un sommet d'origine (sommet d'origine étant une variable intermédiaire qui va être utilisée dans le reste de l'algorithme).
5. Remplir la ligne suivante de la matrice comme suit :
  - a. *Pour chaque* sommet adjacent au sommet d'origine et dont la valeur n'est pas encore fixée *Faire*
    - Calculer la distance entre le sommet initial et le sommet courant (sommet adjacent) en additionnant la valeur fixée dans la ligne précédente avec la distance entre le sommet d'origine et le sommet courant.
    - Si la valeur obtenue est inférieure à la valeur de la même colonne dans la ligne précédente *Alors* la mettre dans la case du sommet courant et marquer cette case par le sommet d'origine, *Sinon* reprendre la valeur de la même colonne dans la ligne précédente.

- b. Pour le reste des cases dont la valeur n'est pas encore fixée, reprendre la valeur de la même colonne dans la ligne précédente.
  - c. Choisir la valeur minimale dans la ligne courante et la fixer. Le sommet correspondant à la valeur fixée devient un sommet d'origine.
6. Si pour tous les sommets on a obtenu une valeur fixée Alors aller à l'étape 7, Sinon revenir à l'étape 5.
7. Les plus courts chemins vont être affichés comme suit : Pour chaque sommet de la première ligne de la matrice, sauf le sommet initial, Faire
  - a. Un chemin est initialement vide.
  - b. Le sommet courant va correspondre à un sommet d'extrémité (sommet d'extrémité étant une variable intermédiaire qui va être utilisée dans le reste de l'algorithme)
  - c. Former l'arc ayant comme extrémité le sommet d'extrémité et comme origine le sommet marquant la valeur fixée correspondant au sommet d'extrémité.
  - d. Ajouter l'arc dans le sens inverse au chemin.
  - e. Si le sommet marquant la valeur fixée est égal au sommet initial Alors afficher le chemin obtenu, Sinon le sommet marquant la valeur fixée devient un sommet d'extrémité et revenir à l'étape c.

**Exemple :**

Soit le graphe pondéré orienté connexe suivant :



En appliquant l’algorithme de Dijkstra, nous allons obtenir la matrice suivante :

A	B	C	D	E
0	$\infty$	$\infty$	$\infty$	$\infty$
•	$10_A$	$\infty$	$\infty$	$5_A$
•	$8_E$	$14_E$	$7_E$	•
•	$8_E$	$13_D$	•	•
•	•	$9_B$	•	•

Les plus courts chemins allant du sommet A vers les autres sommets sont donc :

$A \rightarrow E$	: 5
$A \rightarrow E \rightarrow D$	: 7
$A \rightarrow E \rightarrow B \rightarrow C$	: 9
$A \rightarrow E \rightarrow B$	: 8

6. Exercices corrigés

6.1. Exercices

Exercice 1 :

Ecrire un programme Pascal permettant de créer et d’afficher le premier graphe présenté dans le cours en utilisant une liste d’adjacence.

Exercice 2 :

Reprendre l’exercice précédent, mais cette fois-ci en utilisant une matrice d’adjacence.

Exercice 3 :

Ecrire en Pascal une procédure permettant de transformer la structure du premier exercice en structure du deuxième exercice.

Exercice 4 :

Ecrire en Pascal une procédure permettant d’afficher les nœuds du graphe du deuxième exercice en utilisant la stratégie du parcours en largeur d’abord à partir du sommet numéro 1.

Exercice 5 :

Ecrire en Pascal une procédure permettant d’afficher les nœuds du graphe du deuxième exercice en utilisant la stratégie du parcours en profondeur d’abord à partir du sommet 1.

Exercice 6 :

Ecrire un programme Pascal permettant de créer et d’afficher un graphe représenté par une matrice d’adjacence  $M(5*5)$ , i.e. le graphe possède cinq sommets. Le programme doit utiliser deux procédures, une pour la création et une autre pour l’affichage.

**Exercice 7 :**

Ecrire en Pascal une fonction booléenne permettant de tester l'existence d'un chemin entre deux sommets d'un graphe représenté par une matrice d'adjacence  $M(5*5)$ .

**Exercice 8 :**

Ecrire en Pascal une procédure permettant de déterminer la fermeture transitive d'un graphe représenté par une matrice d'adjacence  $M(5*5)$ .

**Exercice 9 :**

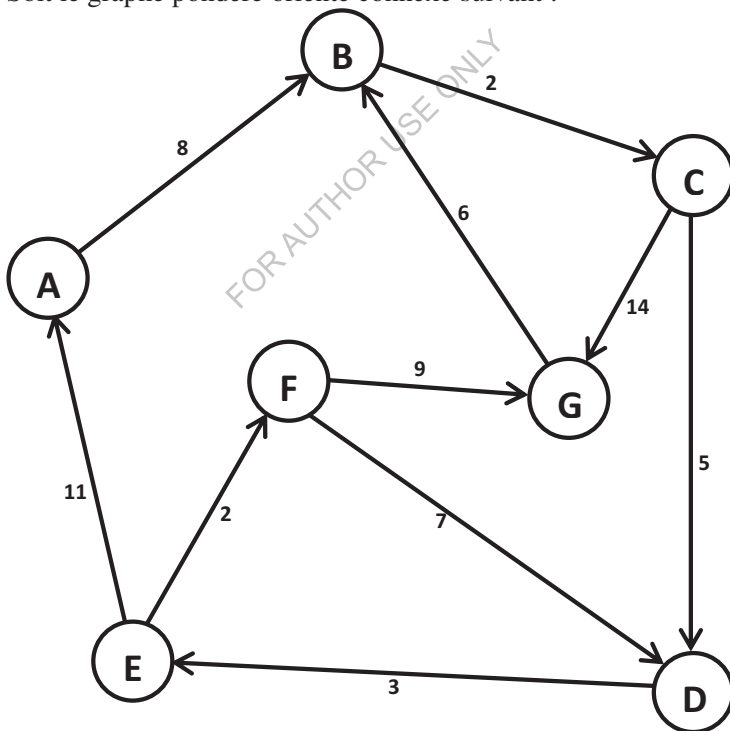
Ecrire en Pascal une fonction permettant de dire si le graphe représenté par une matrice d'adjacence  $M(5*5)$  est fortement connexe ou non.

**Exercice 10 :**

Ecrire en Pascal une procédure permettant de déterminer et d'afficher les composantes fortement connexes d'un graphe représenté par une matrice d'adjacence  $M(5*5)$ .

**Exercice 11 :**

Soit le graphe pondéré orienté connexe suivant :



En appliquant l'algorithme de Dijkstra, cherchez les plus courts chemins allant du sommet C vers les autres sommets.



**6.2. Corrigés****Solution 1 :**

```
program graphe ;
uses crt;
const nbrs = 7 ;
type
  Ptr_Succ = ^ Succ;
  Succ = record
    Sommet : integer ;
    Suivant : Ptr_Succ;
  end ;
var
  G : array [1..nbrs] of Ptr_Succ ;
  i : integer;
{ Procédure d'ajout d'un successeur à un sommet }
procedure Ajouter_Succ (S : integer; Adjacent : integer);
var p : Ptr_Succ ;
begin
  if G[S]<> nil then begin
    p := G[S];
    while p^.Suivant <> nil do p := p^.Suivant;
    new(p^.Suivant);
    p := p^.Suivant;
    p^.Sommet := Adjacent;
    p^.Suivant := nil;
  end
  else begin
    new(G[S]);
    G[S]^Sommet := Adjacent;
    G[S]^Suivant := nil;
  end;
end;
{ Procédure d'affichage des éléments du graphe }
procedure Affichage;
var j : integer;
    p : Ptr_Succ;
begin
  for j := 1 to nbrs do begin
    writeln('Sommet : ', j);
```

```
if G[j] <> nil then begin
  writeln('Ses successeurs sont :');
  p := G[j];
  while p <> nil do begin
    writeln(p^.Sommet);
    p := p^.Suivant;
  end;
end;
end;
end;
begin { Programme Principal }
  clrscr;
  { Initialiser la liste d'adjacence }
  for i := 1 to nbrs do G[i] := nil ;
  { Créer le graphe }
  Ajouter_Succ (1,2);
  Ajouter_Succ (1,4);
  Ajouter_Succ (2,5);
  Ajouter_Succ (3,5);
  Ajouter_Succ (3,6);
  Ajouter_Succ (4,2);
  Ajouter_Succ (5,4);
  Ajouter_Succ (6,6);
  { Afficher le graphe }
  Affichage;
end.
```

**Solution 2 :**

```
program graphe ;
uses crt;
const nbrs = 7;
var M : array[1..nbrs, 1..nbrs] of 0..1;
    k, l : integer;
{ Procédure d'affichage des éléments du graphe}
procedure Affichage;
var i, j : integer;
begin
  for i := 1 to nbrs do begin
    writeln('Sommet : ', i);
    for j := 1 to nbrs do
```

```
    if M[i,j] = 1 then writeln(' Successeur : ', j);
  end;
end;
begin { Programme Principal }
  clrscr;
  { Initialiser la matrice d'adjacence }
  for k := 1 to nbrs do
    for l := 1 to nbrs do M[k,l] := 0;
  { Créer le graphe }
  M[1,2] := 1 ;
  M[1,4] := 1 ;
  M[2,5] := 1 ;
  M[3,5] := 1 ;
  M[3,6] := 1 ;
  M[4,2] := 1 ;
  M[5,4] := 1 ;
  M[6,6] := 1 ;
  { Afficher le graphe }
  Affichage;
end.
```

**Solution 3 :**

```
procedure Transformer;
```

```
var
```

```
  i : integer;
```

```
  p : Ptr_Succ;
```

```
begin
```

```
  for i := 1 to nbrs do begin
```

```
    p := G[i];
```

```
    while p <> nil do begin
```

```
      M[i, p^.Sommet] := 1;
```

```
      p := p^.Suivant;
```

```
    end;
```

```
  end;
```

```
end;
```

**Solution 4 :**

```
procedure Afficher_Larg;
```

```
var i, lmax, lmin : integer;
```

```
  visite : array[1..nbrs] of boolean ;
```

```
  TLarg : array[1..nbrs] of integer ;
```

```
begin
{ Initialisation }
for i := 1 to nbrs do begin
    visite[i] := false;
    TLarg[i] := 0 ;
end;
lmin := 1;
lmax := 1;
TLarg[1] := 1;
visite[1] := true;
{ Remplir le tableau TLarg }
while lmax < nbrs do begin
    for i := 1 to nbrs do
        if (M[TLarg[lmin], i] = 1) and not visite[i] then begin
            lmax := lmax + 1;
            TLarg[lmax] := i;
            visite[i] := true;
        end;
    lmin := lmin + 1;
    if (lmin = lmax+1) then begin
        i := 1;
        while (i <= nbrs) and visite[i] do i := i+1;
        if (i <= nbrs ) then begin
            lmax := lmax + 1 ;
            TLarg[lmax] := i;
            visite[i] := true;
        end;
    end;
end;
end;
for i := 1 to nbrs do writeln(TLarg[i]);
end;
```

**Solution 5 :**

(\*Le tableau Visite, déclaré comme variable globale, indique si un nœud a été déjà visité ou non\*)

Visite: array [1..nbrs] of 0..1;

(\*Procédure récursive d'affichage des éléments d'un nœud en profondeur\*)

Procédure Afficher\_Prof\_R(s: integer);

var i : integer;

begin

```

if (Visite[s]=0) then begin
    Visite[s] := 1;
    writeln(s);
    for i := 1 to nbrs do if (M[s,i]=1) then Afficher_Prof_R(i);
end;
end;
(*Procédure d'affichage des éléments du graphe en profondeur*)
Procédure Afficher_Prof;
var i : integer; b : boolean;
begin
    (*Initialisation du tableau Visite*)
    for i := 1 to nbrs do Visite[i] := 0;
    repeat
        b := false ;
        for i := 1 to nbrs do if (Visite[i]=0) then begin
            Afficher_Prof_R(i);
            b := true;
        end;
    until (not b) ;
end;

```

### **Solution 6 :**

```

program graphe ;
uses crt;
const nbrs = 5;
var M : array[1..nbrs, 1..nbrs] of 0..1;
{ Procédure de création du graphe }
procédure Creation;
var i, j : integer;
    c:char;
begin
    for i := 1 to nbrs do
        for j := 1 to nbrs do begin
            M[i,j] := 0 ;
            writeln('Existe-il un arct entre le sommet ', i, ' et le sommet ', j, ' ? o/n');
            readln(c);
            if c='o' then M[i,j] := 1 ;
        end;
    end;
end;
{ Procédure d'affichage des éléments du graphe }

```

```
procedure Affichage;
var i, j : integer;
begin
  for i := 1 to nbrs do begin
    writeln('Sommet : ', i);
    for j := 1 to nbrs do
      if M[i,j] = 1 then writeln(' Successeur : ', j);
    end;
  end;
end;
begin { Programme Principal }
  clrscr;
  { Créer le graphe }
  Creation;
  { Afficher le graphe }
  Affichage;
end.
```

**Solution 7 :**

Avant d'implémenter la fonction désirée, nous allons déclarer une variable globale H comme suit :

H : array[1..nbrs, 1..nbrs] of integer;

La variable H va être utilisée comme variable intermédiaire dans le programme qui va retourner la puissance de la matrice M par une valeur n.

Pour des raisons de lisibilité du programme, nous allons diviser le problème en une procédure qui détermine la puissance de M par une valeur n, ensuite nous développons la fonction qui détermine l'existence d'un lien entre deux sommets. Et cela comme suit :

{ Procédure qui calcule la puissance de M par n, et range le résultat dans H }

```
procedure M_puissance(n : integer);
var i, j, p, k, s : integer;
    X : array[1..nbrs, 1..nbrs] of integer;
begin
  for i := 1 to nbrs do
    for j := 1 to nbrs do H[i,j] := M[i,j];
  for p := 2 to n do begin
    for i := 1 to nbrs do
      for j := 1 to nbrs do begin
        s := 0;
        for k := 1 to nbrs do s := s + H[i,k] * M[k,j];
        X[i,j] := s ;
      end;
    end;
  end;
end;
```

```
    end;
  for i := 1 to nbrs do
    for j := 1 to nbrs do H[i,j] := X[i,j];
  end;
end;
{ Fonction qui détermine si un chemin existe entre deux sommets ou non }
function Existe_Chemin (s1,s2 : integer) : boolean;
var i,j,k : integer;
    b:boolean;
    X : array[1..nbrs, 1..nbrs] of integer;
begin
  b:= false;
  for i := 1 to nbrs do
    for j := 1 to nbrs do X[i,j] := M[i,j];
  for k := 2 to nbrs do begin
    M_puissance(k);
    for i := 1 to nbrs do
      for j := 1 to nbrs do X[i,j] := X[i,j] + H[i,j];
    end;
  if X[s1,s2] <> 0 then b := true;
  Existe_Chemin := b;
end;
```

**Solution 8 :**

Dans cet exercice, nous allons utiliser une variable globale F qui va retourner le résultat de la fermeture transitive. Cette variable est déclarée comme suit :

F : array[1..nbrs, 1..nbrs] of 0..1;

La procédure de la fermeture transitive est la suivante :

procedure Fermeture ;

var i, j : integer;

begin

for i := 1 to nbrs do

for j := 1 to nbrs do

if Existe\_Chemin(i,j) then F[i,j] := 1

else F[i,j] := 0 ;

end;

**Solution 9 :**

function For\_Con : boolean;

var i, j : integer;

b : boolean;

```

begin
  b:= true;
  for i := 1 to nbrs do
    for j := 1 to nbrs do
      if not (i=j) then
        if not (Existe_Chemin(i,j) and Existe_Chemin(j,i)) then b := false;
      For_Con := b;
    end;
  end;

```

**Solution 10 :**

```

procedure Composantes_For_Con;
var i, j , k: integer;
    cfc : array[1..nbrs] of boolean;
begin
  Fermeture;
  for i:= 1 to nbrs do
    for j := 1 to nbrs do F[i,j]:= F[i,j] * F[j,i];
  for i:= 1 to nbrs do F[i,i] := 1;
  { Affichage des composantes fortement connexes }
  for i:= 1 to nbrs do cfc[i]:= false;
  k := 0;
  for l := 1 to nbrs do
    if not cfc[i] then begin
      k := k+1 ;
      writeln('Composante fortement connexe num ', k, ' contient les sommets :');
      for j := 1 to nbrs do if F[i,j] = 1 then begin
        writeln(j, ' ');
        cfc[j] := true;
      end;
    end;
  end;
end;

```

**Solution 11 :**

La matrice obtenue :

	C	A	B	D	E	F	G
0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
•	$\infty$	$\infty$	$\infty$	$5_C$	$\infty$	$\infty$	$14_C$
•	$\infty$	$\infty$	$\infty$	•	$8_D$	$\infty$	$14_C$
•	$19_E$	$\infty$	$\infty$	•	•	$10_E$	$14_C$
•	$19_E$	$\infty$	$\infty$	•	•	•	$14_C$
•	$19_E$	$20_G$	$\infty$	•	•	•	•
•	•	•	$20_G$	•	•	•	•



Les plus courts chemins sont :

**$C \rightarrow D \rightarrow E \rightarrow A$  : 19**

**$C \rightarrow G \rightarrow B$  : 20**

**$C \rightarrow D$  : 5**

**$C \rightarrow D \rightarrow E$  : 8**

**$C \rightarrow D \rightarrow E \rightarrow F$  : 10**

**$C \rightarrow G$  : 14**

FOR AUTHOR USE ONLY

## Chapitre 13 : Les tables de hachage

### 1. Table de hachage

Une *table de hachage* est une structure de données qui permet une association clé-élément. Elle permet d'optimiser le temps d'accès à un nombre important d'éléments.

Une *clé* permet de désigner le contenu d'un élément de manière non ambiguë. Par exemple, la clé d'un étudiant est son numéro d'inscription, la clé d'un abonné téléphonique est son numéro de téléphone.

On accède à chaque élément de la table via sa *clé*. Il s'agit d'un tableau ne comportant pas d'ordre (un tableau indexé par des entiers). L'accès à un élément se fait en transformant la clé en une valeur de hachage (ou tout simplement hachage) par l'intermédiaire d'une fonction de hachage.

### 2. Exemple d'utilisation

Soit la liste suivante des étudiants : Ahmed, 19, Inf; Ali, 21, Math; Omar, 19, Phys. On désire stocker ces informations dans un tableau d'enregistrements où chaque élément contient les champs : Nom, Age et Filière. Pour déterminer la position de chaque élément dans le tableau, on définit la fonction  $F(ch) = \text{Long}(ch) \bmod 3$ , où  $ch$  est le champ Nom de chaque élément. On obtient donc:

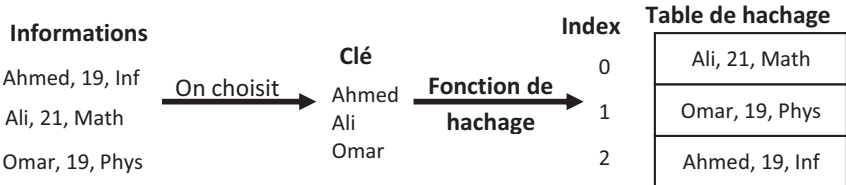
$$F(\text{"Ahmed"}) = 5 \bmod 3 = 2,$$

$$F(\text{"Ali"}) = 3 \bmod 3 = 0,$$

$$F(\text{"Omar"}) = 7 \bmod 3 = 1.$$

- Le tableau d'enregistrement est dit *table de hachage*.
- La fonction  $F$  est dite *fonction de hachage*.
- Le champ Nom choisi comme paramètre de  $F$  est dit *clé*.
- Les positions calculées sont dites *Index* ou *valeurs de hachage*.

Le processus précédant est récapitulé dans le schéma suivant:



### 3. Fonction de hachage

Une fonction de hachage permet de transformer une clé en une valeur de hachage (un index), donnant ainsi la position d'un élément dans le tableau :  $Hach(Clé) = \text{Valeur de hachage, Index, Indice, Adresse ou position}$ .

Si la clé n'est pas un entier naturel, il faut trouver un moyen de la considérer comme un entier naturel. Par exemple, si la clé est de type chaîne de caractères, on peut additionner les positions dans l'alphabet des lettres pour obtenir un entier naturel. Il suffit ensuite de transformer cet entier en index par différentes façons, par exemple, en prenant le reste de la division de l'entier par le nombre d'éléments dans le tableau.

#### 4. Choix de la fonction de hachage

Si nous avons défini la fonction de hachage suivante :

$F(ch) = ASCII(ch[1]) \text{ Mod } 3$ , on obtient:

$F("Ahmed") = 65 \text{ Mod } 3 = 2$ .

$F("Ali") = 65 \text{ Mod } 3 = 2$ .

$F("Omar") = 79 \text{ Mod } 3 = 1$ .

On constate que les deux premiers éléments sont de la même position 2. Ce cas est dit *collision*, c'est-à-dire qu'à partir de deux clés différentes, la fonction de hachage renvoie la même valeur de hachage, et par conséquent donne accès à la même position dans le tableau.

La fonction de hachage doit être choisie de telle sorte que le nombre de collisions soit minimum. Pour minimiser les risques de collision, il faut donc choisir soigneusement sa fonction de hachage.

Le calcul de hachage se fait généralement en deux étapes :

1. Une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la donnée d'origine (clé).
2. Ce nombre entier est converti en une position possible dans la table (Index), en général en calculant le reste de la division (modulo) de ce nombre par la taille de la table.

Pour minimiser le nombre de collisions, la taille de la table de hachage est souvent un nombre premier. C'est le plus petit nombre premier qui peut contenir tous les éléments à stocker. Ceci permet d'éviter les problèmes de diviseurs communs, qui créeraient un nombre important de collisions. Une alternative est d'utiliser une puissance de deux, ce qui permet de réaliser l'opération modulo par de simples décalages, et donc de gagner en rapidité.

En plus des méthodes de hachage dites modulo, il existe d'autres méthodes, telles que :

- Méthode directe : Dans ce cas, la clé et l'adresse sont identiques :  $H(K) = K$ . La structure de données utilisée doit donc prévoir de la place pour chaque clé possible.
- Méthode pseudo-aléatoire : On utilise un générateur de nombre aléatoire de la forme :  $y = ax + c$ , où  $a$  et  $c$  sont des constantes, et  $x$  est la valeur de la clé. Pour un maximum d'efficacité, les constantes  $a$  et  $c$  doivent être des nombres premiers. Le résultat est divisé par

la taille de la table, et le modulo est retourné pour l'adresse de l'entrée.

Un problème fréquent et surprenant est le phénomène de regroupements d'éléments (clustering) qui désigne le fait que des valeurs de hachage se retrouvent côte à côte dans la table, formant des clusters. Plus les éléments sont regroupés, plus le risque de collision est élevé.

Une méthode de hachage doit être judicieusement choisie en fonction des données, afin de limiter au maximum les cas de collision. Les fonctions de hachage réalisant une distribution uniforme des hachages sont donc les meilleures, mais sont en pratique difficile à trouver.

## 5. Résolution des collisions

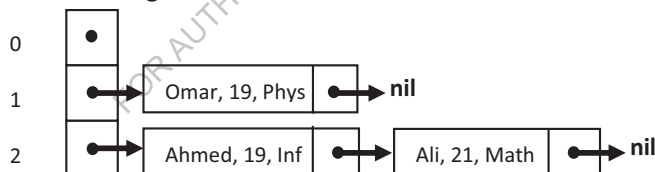
Malgré toutes les précautions, des collisions peuvent surgir. Dans ce cas, plusieurs stratégies de résolution de collision peuvent être utilisées :

### 5.1. Chaînage

Cette méthode est la plus simple. Chaque case de la table est en fait une liste chaînée des clés qui ont le même hachage. Si la fonction de hachage renvoie toujours la même valeur de hachage, quelle que soit la clé, la table de hachage devient alors une liste chaînée.

Pour la fonction  $F(ch) = ASCII(ch[1]) \text{ Mod } 3$ , la table de hachage prendra la forme suivante:

Table de hachage



Des structures de données autres que les listes chaînées peuvent être utilisées, telles que les arbres équilibrés.

### 5.2. Adressage ouvert

L'adressage ouvert consiste dans le cas d'une collision à stocker les valeurs de hachage dans des cases contiguës. La position de ces cases est déterminée par une *méthode de sondage*. Lors d'une recherche, si la case obtenue par hachage direct ne permet pas d'obtenir la bonne position, une recherche sur les cases obtenues par une *méthode de sondage* est effectuée jusqu'à trouver cette position.

Les méthodes de sondage courantes sont :

- **Sondage linéaire :**

Pour une clé d'adresse  $d$ , si la table de hachage a déjà un élément à l'adresse  $d$ , on lui affecte l'adresse  $d0$  de la première case vide suivante

à  $d$ . La table est considérée circulaire, i.e. si  $d$  est l'adresse de la dernière case de la table, alors la prochaine adresse testée sera 0.

La table de hachage précédente prendra la forme suivante :

Table de hachage

0	Ali, 21, Math
1	Omar, 19, Phys
2	Ahmed, 19, Inf

Une alternative est de tester, non pas  $d + 1$ , mais  $d + c$  ou  $d - c$ , où  $c$  est une constante quelconque.

Les avantages de l'adressage linéaire sont sa simplicité (facile à implémenter), et le fait que l'élément est inséré à une localisation proche de son adresse initiale. Les inconvénients sont que cette méthode a tendance à former des regroupements d'éléments (clustering), et donc à favoriser les collisions.

▪ **Sondage quadratique :**

On trouve la prochaine case disponible selon la fonction de hachage :  $H_i(k) = (k + i^2) \text{ MOD } n$ .

Où  $n$  est la taille de la table,  $k$  est l'adresse, et  $i$  est le nombre de collisions rencontrées.

▪ **Double hachage :**

Une deuxième méthode de hachage est utilisée pour *re-hacher* l'adresse qui cause une collision. En particulier, la méthode pseudo-aléatoire de hachage double utilise un générateur aléatoire de la forme de celui déjà vu pour les méthodes de hachage :  $y = ax + c$ . Dans ce cas,  $x$  n'est pas la clé, mais plutôt l'adresse qui crée la collision.

Une indication critique des performances d'une table de hachage est le *facteur de charge* (ou *facteur de remplissage*) qui est la proportion de cases utilisées dans la table. Plus le facteur de charge est proche de 100%, plus le nombre de sondages à effectuer devient important. Lorsque la table est pleine, les algorithmes de sondage peuvent même échouer. Le facteur de charge est en général limité à 80%, même en disposant d'une bonne fonction de hachage. Des facteurs de charge faibles ne sont pas pour autant significatifs de bonnes performances, en particulier, si la fonction de hachage est mauvaise et génère du clustering.

Le facteur de charge se calcule comme suit :  $F = k/n * 100$ , où  $n$  est la taille de la table, et  $k$  le nombre de cases remplies.

**6. Domaine d'utilisation**

Une table de hachage peut être utilisée pour stocker les éléments d'un dictionnaire. Dans ce cas, son but est de savoir si un mot est présent dans un dictionnaire, et de le retrouver rapidement. La clé sera donc le

mot qu'on cherche, et l'élément correspondant dans le tableau sera sa définition.

La table de hachage peut être également utilisée pour les moteurs de recherche, les compilateurs, la cryptographie et la compression des données.

## 7. Exercices corrigés

### 7.1. Exercices

#### Exercice 1 :

On considère la fonction de hachage  $h$  qui associe à toute chaîne de caractères le code ASCII de sa première lettre moins le code ASCII de la lettre  $a$ , le tout modulo  $11$  (par exemple  $h(arbre)=0$ ,  $h(blanc)=1$  ...).

En utilisant la stratégie de chaînage pour résoudre les collisions, écrire le programme Pascal qui contient la fonction de hachage, une fonction de recherche d'un élément, une procédure d'insertion (ajout d'un élément à la table de hachage) et une procédure d'affichage des éléments d'une table de hachage de taille  $11$ . Le corps du programme doit permettre d'effectuer les opérations suivantes : `Inserer('blanc');` ; `Inserer('bleu');` ; `Inserer('noir');` ; `Inserer('vert');` ; `Inserer('rouge');` ; `Inserer('bordeaux');` ; `Inserer('rose');` ; `Inserer('indigo');` ; `Inserer('marron');` ;. Le programme doit permettre aussi l'affichage du contenu de la table de hachage.

#### Exercice 2 :

Représentez graphiquement la table de hachage créée dans l'exercice 1.

#### Exercice 3 :

Ecrire en Pascal la procédure permettant de supprimer un élément de la table de hachage créée dans l'exercice 1.

#### Exercice 4 :

Proposez une fonction de hachage permettant de réduire le taux de collision dans la table de hachage.

#### Exercice 5 :

Reprendre l'exercice 1, mais cette fois-ci en adressage ouvert par un sondage linéaire de la forme  $d+1$ .

#### Exercice 6 :

Représentez la table de hachage de l'exercice 5, ensuite déterminer le facteur de charge de cette table.

#### Exercice 7 :

Ecrire en Pascal la procédure permettant de supprimer un élément de la table de hachage créée dans l'exercice 5.

### 7.2. Corrigés

#### Solution 1 :

program Hachage;

```

uses crt;
const nbre = 11;
type
  Ptr_Element = ^element;
  element = record
    str : string;
    suivant : Ptr_Element;
  end;
var
  tabH : array [0..nbre-1] of Ptr_Element;
  i : integer;
{ Fonction de hachage }
function H (s : string) : integer;
begin
  if length(s) > 0 then H := (ord(s[1]) - ord('a')) MOD nbre
  else H := 0;
end;
{ Fonction de recherche d'un élément dans la table de hachage }
function Recherche(e : string) : boolean ;
var pos : integer;
    p : Ptr_Element;
    trouve : boolean;
begin
  trouve := false;
  pos := H(e);
  p := tabH[pos];
  while (p <> nil) and not trouve do
    if p^.str = e then trouve := true
    else p := p^.Suivant;
  Recherche := trouve;
end;
{ Procédure d'ajout d'un élément à la table de hachage }
procedure Insérer(e : string);
var pos : integer;
    p : Ptr_Element ;
begin
  if not Recherche(e) then begin
    pos := H(e);
    if tabH[pos]<> nil then begin

```

```
p := tabH[pos];
while p^.Suivant <> nil do p := p^.Suivant;
new(p^.Suivant);
p := p^.Suivant;
p^.str := e;
p^.Suivant := nil;
end
else begin
  new(tabH[pos]);
  tabH[pos]^.str := e;
  tabH[pos]^Suivant := nil;
end;
end;
end;
{ Procédure d'affichage des éléments de la table de hachage}
procedure Affichage;
var j : integer;
    p : Ptr_Element;
begin
  for j := 0 to nbre-1 do begin
    if tabH[j] <> nil then begin
      write('Éléments de la position ', j, ' sont : ');
      p := tabH[j];
      while p <> nil do begin
        write(p^.str, ' ');
        p := p^.Suivant;
      end;
      writeln;
    end
    else writeln('Aucun élément dans la position ', j);
  end;
end;
begin (*Programme Principal*)
  clrscr;
  { Initialiser la table de hachage }
  for i := 0 to nbre-1 do tabH[i] := nil ;
  Insérer('blanc');
  Insérer('bleu');
  Insérer('noir');
```

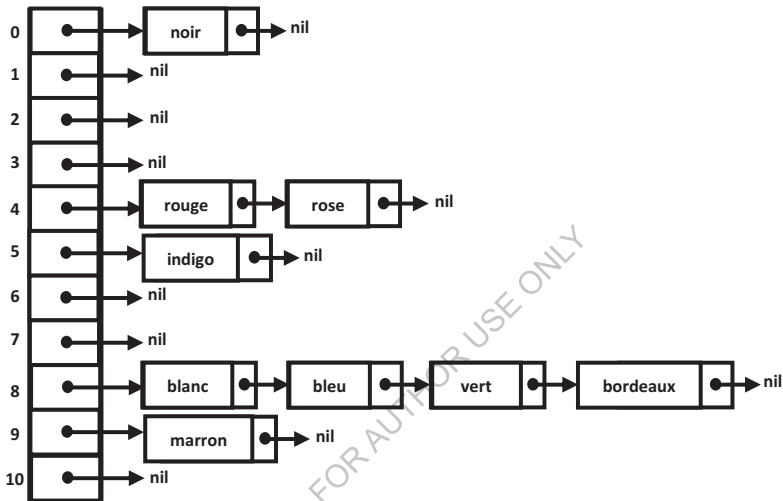


```

Insérer('vert');
Insérer('rouge');
Insérer('bordeaux');
Insérer('rose');
Insérer('indigo');
Insérer('marron');
Affichage;
end.

```

**Solution 2 :**



**Solution 3 :**

{ Procédure de suppression d'un élément de la table de hachage }

```

procédure Supprimer(e : string);
var p1, p2 : Ptr_Element;
    pos : integer;
begin
  if Recherche(e) then begin
    pos := H(e);
    if tabH[pos].str = e then begin
      p2 := tabH[pos];
      tabH[pos] := p2^.Suivant;
      dispose(p2);
    end
  else begin

```

```
p1 := tabH[pos];  
while p1^.Suivant^.str <> e do p1 := p1^.Suivant;  
p2 := p1^.Suivant;  
p1^.Suivant := p2^.Suivant;  
dispose(p2);  
end;  
end;  
end;
```

**Solution 4 :**

```
{ Fonction de hachage }  
function H (s : string) : integer;  
var a,j : integer;  
begin  
  if length(s) > 0 then begin  
    a := 0;  
    for j := 1 to length(s) do a := a + ord(s[j]);  
    H := a MOD nbre  
  end  
  else H := 0;  
end;
```

**Solution 5 :**

```
program Hachage;  
uses crt;  
const nbre = 10;  
      C = 1 ;  
var  
  tabH : array [0..nbre] of string;  
  i : integer;  
{ Fonction de hachage }  
function H (s : string) : integer;  
begin  
  if length(s) > 0 then H := ord(s[1]) MOD nbre  
  else H := 0;  
end;  
{ Fonction de recherche d'un élément dans la table de hachage }  
function Recherche(e : string) : boolean ;  
var pos1, pos2 : integer;  
    trouve : boolean;  
begin
```

```

trouve := false;
pos1 := H(e);
pos2 := pos1;
while not trouve and (pos2 <= nbre) do
    if tabH[pos2] = e then trouve := true
    else pos2 := pos2 + C;
pos2 := 0 ;
while not trouve and (pos2 < pos1) do
    if tabH[pos2] = e then trouve := true
    else pos2 := pos2 + C;
Recherche := trouve;
end;
{ Procédure d'ajout d'un élément à la table de hachage }
procedure Insérer(e : string);
var pos1, pos2 : integer;
    ajout : boolean;
begin
    if not Recherche(e) then begin
        ajout := false;
        pos1 := H(e);
        pos2 := pos1;
        while not ajout and (pos2 <= nbre) do
            if tabH[pos2] = "" then begin
                ajout := true;
                tabH[pos2] := e;
            end
            else pos2 := pos2 + C;
        pos2 := 0 ;
        while not ajout and (pos2 < pos1) do
            if tabH[pos2] = "" then begin
                ajout := true;
                tabH[pos2] := e;
            end
            else pos2 := pos2 + C;
        end;
    end;
end;
{ Procédure d'affichage des éléments de la table de hachage }
procedure Affichage;
var j : integer;

```

```
begin
  for j := 0 to nbre do
    if tabH[j] <> '' then
      writeln('Elément de la position ', j, ' est : ', tabH[j])
    else writeln('La case ', j, ' est vide. ');
  end;
begin (*Programme Principal*)
  clrscr;
  { Initialiser la table de hachage }
  for i := 0 to nbre do tabH[i] := '';
  Insérer('blanc');
  Insérer('bleu');
  Insérer('noir');
  Insérer('vert');
  Insérer('rouge');
  Insérer('bordeaux');
  Insérer('rose');
  Insérer('indigo');
  Insérer('marron');
  Affichage;
end.
```

**Solution 6 :**

FOR AUTHOR USE ONLY

0	noir
1	bordeaux
2	marron
3	
4	rouge
5	rose
6	indigo
7	
8	blanc
9	bleu
10	vert

Le facteur de charge =  $9/11 * 100 = 81,81 \%$

**Solution 7 :**

{ Procédure de suppression d'un élément de la table de hachage }

```
procedure Supprimer(e : string);
var pos1, pos2 : integer;
    supp : boolean;
begin
    if Recherche(e) then begin
        supp := false;
        pos1 := H(e);
        pos2 := pos1;
        while not supp and (pos2 <= nbre) do
            if tabH[pos2] = e then begin
                supp := true;
                tabH[pos2] := "";
            end
            else pos2 := pos2 + C;
        pos2 := 0 ;
        while not supp and (pos2 < pos1) do
            if tabH[pos2] = e then begin
                supp := true;
                tabH[pos2] := "";
            end
            else pos2 := pos2 + C;
        end;
    end;
end;
```

FOR AUTHOR USE ONLY

**INDEX**

---

**A**

ABS · 31, 172  
adressage linéaire · 398  
adressage ouvert · 397, 399  
affectation · 22  
algorithme · 14, 15, 19  
algorithme de Dijkstra · 382, 385  
algorithmique · 15  
Aller à ... · 48  
allocation · 267, 269  
Allouer · 269  
AND · 32  
appel récursif · 183  
APPEND · 251  
arbre · 271, 352, 377  
arbre binaire · 353  
arbre complet · 361  
arbre dégénéré · 361  
arbre équilibré · 363  
arbre n-aire · 353  
arbre ordonné · 361  
arbre recouvrant · 379  
arc · 352, 377  
ARCTAN · 31  
arête · 352  
arguments · 178  
ARRAY · 103  
ASCII · 31  
assembleur · 17  
ASSIGN · 248, 250  
assignation · 22  
automatique · 10

---

**B**

BEGIN · 27  
bloc · 162  
BOOLEAN · 32  
booléen · 20, 21, 32  
boucle infinie · 74, 75  
boucle Pour · 76  
boucle Répéter · 75  
boucle Tant que · 74  
boucles · 74, 104  
boucles imbriquées · 78  
branche · 352  
branchement · 48  
BYTE · 29

---

**C**

caractère · 20, 21, 31  
Cas ... de... · 46  
cas trivial · 118  
CASE ... OF... · 47  
chaînage · 397, 399  
chaîne de caractères · 24, 106  
chaîne palindrome · 198  
champ · 228, 242  
champ de visibilité · 175  
CHAR · 31  
CHDIR · 250  
chemin · 377, 385  
CHR · 32  
clé · 395  
CLOSE · 250  
code · 19

collision · 396, 397  
commentaires · 24, 28  
commis voyageur · 218  
compilateur · 18  
compilation · 17  
complexité spatiale · 215  
complexité temporelle · 215  
compteur · 77, 115  
computer · 12  
CONCAT · 108  
concaténation · 107, 108  
condition · 44, 45, 74  
CONST · 27  
constantes · 19, 30, 166  
COPY · 108  
COS · 31  
critère d'arrêt · 184  
cycle · 377

---

## D

débordement de pile · 183, 184  
déclaration · 19, 21, 166  
défilement · 318  
dépilement · 307  
désallocation · 267  
Désallouer · 269  
dichotomie · 110  
DISPOSE · 269, 270  
DIV · 28  
données · 19  
DOUBLE · 29  
double hachage · 398  
DOWNT0 · 78

---

## E

Ecrire · 24  
écriture · 28  
effet de bord · 176  
effet secondaire · 176  
empilement · 307  
END · 27  
enfilement · 317  
ENIAC · 11  
enregistrement · 227, 271  
ensemble · 226  
entier · 20, 21, 29  
entrées/sorties · 23, 248  
énumération des étapes · 15  
énuméré · 224  
EOF · 250  
EOLN · 251  
équation 1<sup>er</sup> degré · 50  
équation 2<sup>ème</sup> degré · 50  
équations logiques · 21  
ERASE · 250  
ET · 21, 44  
étiquettes · 48, 166  
EXP · 31, 172  
expressions arithmétiques · 22  
expressions logiques · 23  
EXTENDED · 29

---

## F

facteur de charge · 398, 399  
factorielle · 182, 183, 220  
FALSE · 32  
FAUX · 20, 32, 44  
fermeture transitive · 379, 385  
feuille de l'arbre · 352

Fibonacci · 81, 117, 198, 220  
 fichier · 242  
 fichier non typé · 244  
 fichier texte · 242  
 fichier typé · 243  
 FIFO · 316  
 file · 271, 316  
 FILE · 244  
 file d'attente · 317, 322  
 FILE OF · 243  
 FILEPOS · 252  
 FILESIZE · 252  
 fils droit · 353  
 fils gauche · 353  
 Flag · 110, 115  
 fonction · 170  
 fonction de hachage · 395, 396  
 fonctions mathématiques · 31  
 FOR ... DO ... · 78  
 formats d'édition · 28  
 forme graphique · 353  
 forme parenthésée · 353  
 FORWARD · 192  
 FRAC · 31  
 FUNCTION · 170

---

## G

GDR · 360  
 gestion dynamique · 268  
 gestion statique · 268  
 GETDIR · 250  
 GOTO ... · 48  
 graphe · 271, 377  
 graphe acyclique · 377  
 graphe connexe · 377  
 graphe fortement connexe · 377, 385

graphe orienté · 377  
 graphe pondéré · 377  
 GRD · 359

---

## H

HARDWARE · 12  
 hauteur d'un arbre · 353, 363

---

## I

identificateur · 19, 27  
 IF ... THEN ... · 45  
 IF ... THEN ... ELSE... · 46  
 IN · 227  
 incrément · 77  
 index · 395  
 indice · 103, 104, 107  
 information · 10  
 informatique · 10  
 INT · 31  
 INTEGER · 29  
 interprétation · 18  
 interpréteur · 18  
 intervalle · 225  
 IORESULT · 250  
 itérative · 183

---

## J

juxtaposition · 107, 108

---

## L

langage algorithmique · 15



langage de description d'algorithme · 15  
 langage de programmation · 13, 17  
 langage Pascal · 18  
 LDA · 15, 46  
 lecture · 28  
 LENGTH · 108  
 lien · 377  
 LIFO · 306  
 Lire · 24  
 liste chaînée · 267, 271  
 liste circulaire · 328, 333, 349  
 liste d'adjacence · 377, 384  
 liste doublement chaînée · 289  
 LN · 31  
 LONGINT · 29  
 longueur d'un chemin · 377

---

## M

maillon · 271  
 matrice · 106  
 matrice d'adjacence · 378, 384  
 méthode de sondage · 397  
 miroir d'une chaîne · 198  
 MKDIR · 251  
 MOD · 28  
 mot des feuilles · 353, 363  
 mots clés · 28  
 mots réservés · 28

---

## N

NEW · 269, 270  
 NIL · 270  
 niveau d'un nœud · 352, 363  
 nœud · 271, 352, 377

nœud terminal · 352  
 NON · 21, 44, 74  
 NOT · 32

---

## O

opérateurs de comparaison · 21, 32, 44  
 opérations de base · 22  
 optimisation · 215, 267  
 OR · 32  
 ORD · 32, 225  
 ordinateur · 12, 14  
 ordre de grandeur · 216  
 organigramme · 25, 44, 45, 47, 74, 75, 77  
 organisation indexée · 246  
 organisation relative · 246  
 organisation séquentielle · 246  
 OU · 21, 44

---

## P

PACKED · 107  
 paramètres · 167, 168, 169  
 paramètres effectifs · 169  
 paramètres en entrée · 178  
 paramètres en sortie · 178  
 paramètres formels · 169, 173, 178  
 parcours en largeur · 357, 380, 384  
 parcours en profondeur · 357, 380, 384  
 parcours infixe · 359  
 parcours postfixé · 360  
 parcours préfixé · 357  
 pas de progression · 77, 78  
 passage par adresse · 178  
 passage par référence · 178  
 passage par valeur · 178

passage par variable · 178  
 permutation · 33, 116  
 PGCD · 80, 187  
 pile · 271, 306  
 pile d'activation · 183  
 plus court chemin · 382  
 PLUSH · 251  
 poids · 377  
 point d'appel · 165, 169, 170  
 point d'appui · 184  
 point d'arrêt · 184  
 point terminal · 184  
 pointeur · 267  
 portée de la variable · 175  
 Pour ... Faire ... · 76  
 PRED · 32, 225  
 priorité entre opérations · 23, 30  
 PROCEDURE · 169  
 procédure · 169  
 produit matriciel · 119  
 PROGRAM · 27  
 programmation modulaire · 164  
 programme · 13, 17  
 programme appelant · 167, 170  
 programme Pascal · 27

---

## R

racine de l'arbre · 352  
 RAM · 242  
 READ · 27, 28, 249, 251, 252  
 READLN · 27, 28, 251  
 REAL · 29  
 recherche dichotomique · 110, 112,  
     198, 218  
 récursion · 182  
 récursion croisée · 192

récursion indirecte · 192  
 récursivité · 182  
 réel · 20, 21, 29  
 RENAME · 251  
 REPEAT...UNTIL... · 76  
 Répéter ... Jusqu'à ... · 75  
 RESET · 249, 251  
 REWRITE · 249, 251  
 RGD · 357  
 RMDIR · 251  
 ROUND · 31  
 rubrique · 242

---

## S

schtroumpf · 114  
 SEEK · 250, 252  
 SEEKEOL · 251  
 SEEKOF · 251  
 séquence · 44, 77  
 SET OF · 226  
 SETTEXTBUF · 251  
 SHORTINT · 29  
 Si ... Alors ... · 44  
 Si ... Alors ... Sinon ... · 45  
 SIN · 31  
 SINGLE · 29  
 SOFTWARE · 12  
 sommet · 352, 378  
 sondage linéaire · 397, 399  
 sondage quadratique · 398  
 sous-programme · 166, 173  
 sous-programme appelant · 166  
 sous-programme appelé · 166  
 SQR · 31, 172  
 SQRT · 31, 172  
 STRING · 107

structure · 227, 271  
structure conditionnelle composée · 45  
structure conditionnelle multiple · 46  
structure conditionnelle simple · 44  
structures conditionnelles · 44  
structures de contrôle · 22, 44  
SUCC · 32, 225  
système d'exploitation · 13

---

## T

table de hachage · 395  
tableau · 103  
tableau à deux dimensions · 105  
tableau circulaire · 322  
taille d'un arbre · 353, 363  
Tant que ... Faire ... · 74  
TAS · 270  
technique de Flag · 110, 115, 219  
test d'arrêt · 184  
TEXT · 243  
TO · 78  
tour de Hanoï · 188, 221  
traitement · 10, 19  
transposée d'une matrice · 119  
tri à bulles · 105, 115  
tri d'un tableau · 104, 115, 121  
tri par sélection · 105, 115, 219  
triangle de Pascal · 117  
TRUE · 32  
TRUNC · 31  
TRUNCATE · 252  
type · 19, 20, 166  
TYPE · 224

type de base · 227  
type structuré · 106  
types définis · 224  
types numériques · 20  
types simples · 20  
types structurés · 20, 103  
types symboliques · 20

---

## U

UPCASE · 108

---

## V

valeur · 19, 20  
valeur de hachage · 395  
VAR · 27, 178  
variable de contrôle · 76, 77  
variable pointée · 269  
variables · 19, 166  
variables globales · 173  
variables locales · 173  
vecteur · 106  
VRAI · 20, 32, 44

---

## W

WHILE ... DO... · 75  
WITH...DO... · 229  
WORD · 29  
WRITE · 27, 28, 249, 251, 252  
WRITELN · 27, 28, 251

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

**More  
Books!**



yes  
**I want morebooks!**

Buy your books fast and straightforward online - at one of world's fastest growing online book stores! Environmentally sound due to Print-on-Demand technologies.

Buy your books online at  
**[www.morebooks.shop](http://www.morebooks.shop)**

Achetez vos livres en ligne, vite et bien, sur l'une des librairies en ligne les plus performantes au monde!

En protégeant nos ressources et notre environnement grâce à l'impression à la demande.

La librairie en ligne pour acheter plus vite  
**[www.morebooks.shop](http://www.morebooks.shop)**

KS OmniScriptum Publishing  
Brivibas gatve 197  
LV-1039 Riga, Latvia  
Telefax: +371 686 204 55

[info@omniscryptum.com](mailto:info@omniscryptum.com)  
[www.omniscryptum.com](http://www.omniscryptum.com)



FOR AUTHOR USE ONLY