Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Ghardaia
Faculté des Sciences et de la Technologie
Département de Mathématiques et de l'Informatique

Polycopié de cours

Algorithmique et Structure de Données : Partie 1

Dr. Slimane BELLAOUAR

Édition 2020

لا يزال المرء عالما ما دام في طلب العلم، فإذا ظن أنه قد علم فقد بدأ جهله.

ابن قتيبة(٩٨٨ م)، أديب فقيه محدث مؤرخ عربي

Table des matières

Avant-propos

Une idée très répandue, à tort, dans les milieux publiques, industrielles et même académiques, que l'ordinateur est une machine capable de résoudre les problèmes car elle est dotée d'une certaine intelligence.

En effet, cette machine n'est capable de rien, sauf si une entité (par exemple un programmeur) lui avait montré les étapes à suivre, sans aucune croyance, que cette machine peut donner la moindre interprétation de ces étapes. Autrement dit, la machine exécute les étapes aveuglement. Cette philosophie, de montrer à la machine les étapes à suivre pour résoudre un problème, porte le nom d'*algorithme*.

Une enquête menée auprès des diplômés (de cinq ans d'études) de l'université de Stanford a demandé quels cours utilisaient-ils dans leur travail ? Le module introduction à la programmation a pris la première place. Viennent ensuite les cours de niveau logiciel couvrant, essentiellement, les structures de données de base et les algorithmes.

Dans cette optique, cet ouvrage intitulé «Algorithmique et Structure de Données : Partie 1» est dédié aux étudiants première année licence du domaine Mathématiques et Informatique. Il respecte les consignes des derniers canevas mis en vigueur pour les spécialités Systèmes Informatiques (SI) et Ingénierie des Systèmes d'Information et du Logiciel (ISIL) depuis l'année universitaire 2018/2019. Néanmoins, il peut être utilisé par d'autres filières techniques ou d'autres institutions dispensant les modules algorithmique, structure de données et programmation.

Par ailleurs, les ouvrages d'algorithmique sont en abondance dans la littérature. Cependant, ils sont plus ou moins accessibles pour les étudiants novices. Notre objectif consiste à mettre à la portée de l'étudiant un ouvrage simple à lire et à comprendre, tant au niveau conceptuel que technique. Pour ce faire :

- Nous épuisons toute notre expérience acquise, pendant plus de dix ans, d'enseignement de l'algorithmique, dans l'université de Ghardaia et dans d'autres institutions.

- Nous accompagnons l'explication des nouveaux concepts et des nouvelles techniques par des exemples et des illustrations.
- Chaque chapitre est suivi par un certain nombre d'exercices de divers degrés de difficultés, bien conçus, pour mettre en oeuvre les acquis théoriques. Ces exercices se proposent pour couvrir des situations de la vie professionnelle (calcul mathématique, gestion, . . .).
- En annexe, nous incluons une série des examens partiels, finals et de rattrapage avec des corrigés types des années passées afin de permettre à l'étudiant une auto-évaluation.

Pour le formalisme algorithmique, nous avons utilisé, principalement, les conventions du langage Pascal. Ce formalisme, facilite aux étudiants l'implémentation des algorithmes dans un langage de haut niveau.

Afin de mettre en oeuvre les solution algorithmiques, dans le présent ouvrage, nous avons choisi d'utiliser deux langages de programmation : C et Pascal. Le langage C est choisi parce qu'il est prescrit dans le canevas de mise en conformité. Alors que le langage Pascal est choisi pour aider les étudiants à apprendre la programmation. D'ailleurs, le langage Pascal s'est développé dans les universités pour un but pédagogique.

Pour être conforme aux canevas de mise en conformité, cet ouvrage est la première partie d'une série de trois parties qui couvrent l'enseignement du module algorithmique et structure de données pour les trois premiers semestres de la filière Informatique du domaine Mathématiques et Informatique.

La première partie, «Algorithmique et Structure de Données 1» couvre les éléments suivants :

1. Introduction à l'informatique.
2. Éléments de base de l'algorithmique.
3. Présentation du formalisme algorithmique.
4. Tableaux et chaînes de caractères.
5. Types personnalisés.

Dans une perspective de faciliter et d'élargir l'accès à cette série d'ouvrages, nous envisageons produire des versions en arabe et en anglais. Nous envisageons aussi d'inclure le langage de programmation *Python* dans les versions futures.

Au terme de ce travail, je tiens tout d'abord à remercier Dieu le tout puissant et miséricordieux, qui m'a donné la force et la patience d'accomplir ce modeste travail.

Mes vifs remerciements vont à Messieurs Slimane Oulad-Naoui, Abdelkader Bouhani, Chaker Abdelaziz Kerrache, Khaled Kechida et Mesdemoiselles Habiba Benabderrahmane et Wassila Belfardi, enseignants à l'université de Ghardaia, qui m'ont accompagné tout au long ces années dans l'enseignement du module *Algorithmique*

et structures de données. Leurs empreintes sont visiblement apparentes sur le contenu du présent polycopié.

J'adresse aussi mes remerciements à Monsieur Nasreddine Lagraa, Professeur à l'université Amar Thelidji-Laghouat, Madame Fatima Zohra Laallam, Professeur à l'Université Kasdi Merbah-Ouargla, Monsieur Mohammed Omari, Professeur à l'Université Ahmed Draia-Adrar d'avoir accepté d'évaluer le présent ouvrage et pour l'effort fourni. Leurs corrections, remarques, recommandations et suggestions sont inestimables et ont influencé grandement la qualité du document.

Je tiens aussi à remercier chaleureusement Monsieur Redouane Sadouni, enseignant à l'université de Ghardaia, pour toutes les facilités qui présente en matière de communication et d'interaction. Un tel geste nous encourage d'opter pour le mieux dans notre noble mission.

Enfin, mes remerciements vont également à tous ceux qui ont, d'une manière ou d'une autre, de pré ou de loin, apporté leur appui.

1. Introduction à l'informatique

Ce chapitre, introduction à l'informatique, est en tête du présent manuscrit, malgré que son contenu, en effet, ne fait pas partie de l'algorithmique et des structures de données. Néanmoins, nous l'avons inclus dans ce document pour deux raisons essentielles. D'abord pour être conforme au canevas de la mise en conformité du domaine mathématiques et informatique (MI). De plus, nous jugeons très utile, pour un nouveau étudiant qui veut affranchir les portes de l'algorithmique et de structures de données et de la programmation de connaître, grossièrement, la discipline informatique en générale, et l'ordinateur et ses composants en particulier.

Il s'agit, dans ce chapitre, dans un premier temps de définir le mot informatique et de cerner, ensuite, la discipline informatique. En outre, le chapitre introduit la définition d'un ordinateur, ses fonctions, le lien entre logiciel et matériel, les types des ordinateurs et enfin, les différents composants d'un ordinateur selon l'architecture von Neumann.

Dans le présent chapitre, nous avons décidé de ne pas présenter la partie représentation de l'information car selon le nouveau canevas, elle sera traitée, en détail, dans le module structure machine 1 *qui sera dispensé dans le même semestre (semestre 1) que le module* ASD1.

Les étudiants désirant plus amples informations sur le lien entre les concepts ordinateur, langage machine et langages évoluées peuvent consulter le chapitre 1. de l'ouvrage "S'initier à la programmation, Avec des exemples en C, C++, C#, Java et PHP" [Delannoy, 2008].

1.1 Qu'est ce que l'informatique ?

Le mot *informatique* est la contraction de deux mots : *information* et *automatique*. Le mot information fait référence à la science de l'information, alors que le mot

automatique est assimilé à l'art d'effectuer automatiquement et rationnellement des actions. D'où la Définition 1.1 s'en découle :

Définition 1.1 (Informatique) L'informatique est la science de traitement automatique et rationnel de l'information en utilisant une machine (ordinateur)

Suite à cette définition, nous pouvons déduire que la discipline informatique peut être appliquée dans plusieurs domaines, qu'ils soient scientifiques, techniques, économiques ou sociaux.

Ce qu'il faut retenir, encore, que l'informatique n'est pas le résultat d'une synthèse de plusieurs disciplines. Plutôt, elle est une discipline à part entière. Elle comporte, essentiellement, un volet fondamental et un autre appliqué. Dans le volet fondamental, nous pouvons étudier la théorie des langages, la calculabilité, la logique, ... Alors que dans le volet pratique, nous pouvons aborder les sujets de la programmation, génie logiciel, système d'exploitation, langage de programmation, compilation, ...

Les anglophones utilisent le terme «*computer science*» pour exprimer la discipline informatique. Mais parfois, le terme *informatics* est aussi utilisé.

1.2 Ordinateur

Définition 1.2 (Ordinateur) Un ordinateur est une machine électronique programmable capable de réaliser des calculs logiques sur des nombres binaires.

Le mot «*ordinateur*» fut introduit en 1955 par IBM comme un nom français pour la nouvelle machine IBM 650, tout en évitant la traduction directe du mot anglais *computer* (calculateur).

1.2.1 Fonctions d'un ordinateur

Un ordinateur doit assurer les quatre fonctions de base suivantes :

1. Acquérir l'information (acquisition).
2. Conserver l'information (stockage).
3. Traiter l'information (calcul).
4. Restituer des résultats (restitution).

L'acquisition des données est la fonction d'*entrée*. Elle se manifeste par la réception des données ou des instructions par l'ordinateur qui dispose des supports de stockage pour *mémoriser* de l'information lorsqu'il est en marche et lorsqu'il est désactivé. La fonction de *calcul*, ou d'une manière générale, le traitement se traduit par l'exécution de certaines opérations de base sur les données. Bien-sûre, l'ordinateur, doit pouvoir

produire et/ou communiquer les résultats. C'est ce qu'on appelle la fonction de *sortie*.

Sur la base des quatre fonctions de base, il est possible pour un utilisateur d'effectuer de nombreuses tâches : calculs scientifiques, traitements bureautiques, messagerie électronique, applications de gestion, conception assistée par ordinateur, jeux, ...

1.2.2 Matériel et logiciel

Si nous revenons à la Définition 1.2, nous pouvons constater qu'elle comporte trois parties : une machine électronique, une machine programmable et un calcul logique sur des nombres binaires.

La première partie, machine électronique fait référence au terme *matériel* (hardware) qui est un assemblage d'équipements (processeur, mémoires, périphériques, ...) dont le fonctionnement est soumis aux lois de la physique.

La seconde partie, machine programmable attire notre attention que cette machine électronique n'est capable d'effectuer des tâches différentes que par le biais des instructions qui lui sont adressées. Ces instructions assemblées sous forme de programmes portent le nom de *logiciel* (software).

La troisième partie implique que l'ordinateur peut traiter différents types d'informations (textes, nombres, sons, images, vidéos, instructions) sous forme binaire.

1.2.3 Types des ordinateurs

Il existe plusieurs taxonomies des types d'ordinateurs. Nous retenons, ici, celle reposant sur la taille et la puissance :

1. *Superordinateur* : Un superordinateur (supercomputer) est un mot générique qui désigne l'ordinateur de plus hautes performances possibles, en particulier le plus rapide lors de sa conception. Les superordinateurs sont très coûteux et sont utilisés pour des applications spécialisées nécessitant d'immenses quantités de calculs mathématiques. À titre d'exemple, nous pouvons citer les applications suivantes : prévision météo, simulation scientifique, calculs de dynamique des fluides, recherches sur l'énergie nucléaire, conception électronique, analyse de données géologiques, ...

   La discipline qui s'intéresse aux superordinateurs est appelée «calcul de haute performance» (High-Performance Computing ou HPC).

   Les premiers superordinateurs ont vu le jour en 1961. Ils portent le nom IBM Strech ou IBM 7030. De nos jours, certains superordinateurs adoptent la technique des systèmes massivement parallèles et utilisent des microprocesseurs de type RISC (Reduced instruction set computer).

   Cray Research (fondée par Seymour Cray après son départ de Control Data Corporation (CDC)) est le fabricant de superordinateur le plus connu.

2. *Mainframe* : Mainframe est un terme se rapportant à l'armoire contenant l'unité centrale de grande puissance de traitement reliée à un réseau de terminaux. Ces types d'ordinateurs volumineux et coûteux sont capables de prendre en charge simultanément des centaines, voire des milliers d'utilisateurs. Ainsi, ils sont principalement utilisés par les gouvernements et les grandes organisations pour le traitement de données en bloc, les applications critiques, le traitement de transactions, les statistiques de recensement, ...

   Parmi les fabricants des mainframes, nous pouvons compter la compagnie Bull avec les DPS/6 à DPS/8 sous système GCOS (General Comprehensive Operating System) .

3. *Mini-ordinateur* : Les mini-ordinateurs sont des ordinateurs de taille moyenne. Un mini-ordinateur est un système multitraitement capable de prendre en charge simultanément jusqu'à 200 utilisateurs.

   À titre non exhaustif, nous pouvons citer la série PDP-7, PDP-8 et PDP-11 de la gamme *Programmed Data Processor* du constructeur *Digital Equipment Corporation* comme des mini-ordinateurs typiques.

4. *Ordinateur personnel* : Un ordinateur personnel (Personnal Computer ou PC) est conçu pour être utilisé par une seule personne à la fois. Les PCs sont basés sur la technologie de microprocesseur qui permet aux fabricants de mettre la totalité de l'unité de traitement centrale sur une puce.

   Les premiers ordinateurs personnels grand public sont apparus à la fin des années 1970. En 1977, Apple Computer a introduit *Aplle II*, un des plus populaires PCs. Plus tard, en 1981, IBM a construit son premier ordinateur personnel connu sous le nom de *IBM PC*.

   Ces dernières années, le terme PC s'applique à tout ordinateur basé sur un microprocesseur Intel ou sur un microprocesseur compatible Intel.

   Les ordinateurs personnels se répartissent en plusieurs catégories :

   - *Un ordinateur de bureau* est conçu pour être utilisé à un bureau et est rarement déplacé. Il consiste en une boîte appelée unité centrale qui contient la plupart des composants essentiels (Figure 1.1). Le moniteur, le clavier et la souris se connectent tous à l'aide de câbles (ou, dans certains cas, d'une technologie sans fil). Les ordinateurs de bureau sont flexibles, car nous pouvons y connecter le moniteur, le clavier et la souris de notre choix, ainsi que l'installation de disques de stockage, de mémoires et de cartes d'extension supplémentaires.
   - *Un ordinateur portable (portatif)*, parfois dit laptop ou notebook, est un petit ordinateur personnel, compact et léger. Aussi, il dispose de sa propre alimentation sous forme d'une batterie. Par conséquent, il est très utile dans le cas de mobilité. Il est composé d'un boîtier dépliable, sa couverture s'ouvre pour révéler un écran, un clavier et un périphérique de pointage intégré qui se substitue à la souris (Figure 1.2).

FIGURE 1.1 – Ordinateur de bureau

FIGURE 1.2 – Ordinateur portable

- *Netbook* est un abrégé pour Internet notebook. C'est un ordinateur portable plus petit et moins puissant, conçu principalement pour accéder à Internet. Un netbook est généralement moins cher qu'un ordinateur portable, mais plus léger et plus pratique à transporter. Cependant, il n'est pas assez puissant (en terme de processeur et mémoire) pour exécuter toutes les applications de bureau. De plus, le disque dur est remplacé par une mémoire flash, afin de réduire la consommation électrique et le coût.
- *Une tablette* est un ordinateur portable constitué d'un écran tactile et un ensemble de périphériques intégrés. Elle ne dispose pas de clavier ou de périphérique de pointage ; un clavier logiciel et le glissement de doigt sur l'écran font foi. Les tablettes ont des capacités de mémoire et de stockage limitées.
- *Un smart phone* est un téléphone mobile capable d'exécuter des applications et il est doté de capacités Internet. Les smart phones disposent généralement des écrans tactiles (Figure 1.3). Ils ont une variété d'applications sensibles à la localisation, telles que le système de positionnement global (GPS), les programmes de cartographie et les guides commerciaux locaux.

1.2.4 Architecture d'un ordinateur

Nous nous intéressons, ici, à l'architecture dite de von Neumann en référence au mathématicien John von Neumann né en 1903 à Budapest (Hongrie). Cette architec-

FIGURE 1.3 – Smart phone

ture est également appelée modèle de von Neumann ou architecture de Princeton.

Définition 1.3 (Architecture de von Neumann) L'architecture de von Neumann est basée sur le concept d'ordinateur à programme enregistré, dans lequel les données et les instructions sont stockées dans la même mémoire.

Dans l'architecture de von Neumann (Figure 1.4), nous pouvons distinguer quatre parties :

1. l'unité centrale,
2. la mémoire,
3. les dispositifs d'entrée,
4. les dispositifs de sortie.

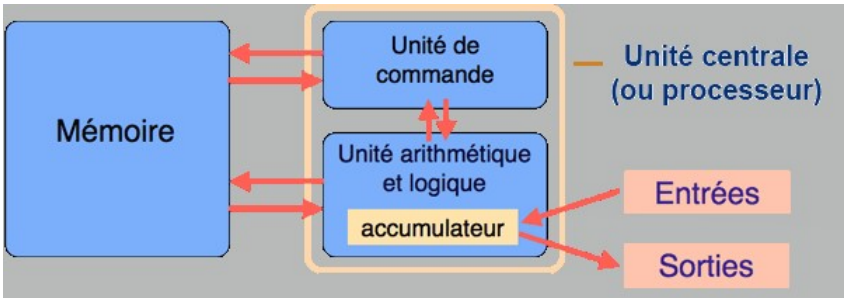


FIGURE 1.4 – Architecture de von Neumann

Unité centrale

L'unité centrale, ou unité centrale de traitement (UCT) est un appareil qui interprète et exécute les commandes que vous donnez à un ordinateur. L'UCT est également connue sous le nom de processeur. Elle détermine la vitesse de l'ordinateur qui est mesurée en Hz (KHz, MHz, GHz).

L'UCT (processeur) est constituée de circuits électroniques qui peuvent exécuter des actions primitives (câblées) constituant le jeu d'instructions du processeur appelée encore «langage machine».

Dans le modèle de von Neumann, les instructions sont exécutées les unes après les autres (principe d'exécution séquentielle).

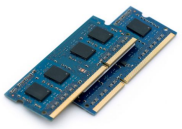
L'UCT contient, principalement, l'unité de contrôle (UC) et l'unité arithmétique et logique (UAL).

L'UC contrôle le fonctionnement de l'UAL, de la mémoire et des périphériques d'entrée/sortie de l'ordinateur, en leur indiquant comment répondre aux instructions du programme qu'elle vient de lire et d'interpréter à partir de la mémoire. L'UC fournit, également, les signaux de synchronisation et de contrôle requis par d'autres composants de l'ordinateur.

L'UAL permet d'effectuer des opérations arithmétiques (addition, soustraction, ...) et logiques (ET, OU, NON, ...).

Mémoire

Définition 1.4 (Mémoire) une mémoire est un dispositif capable d'enregistrer une information, de la conserver et de la restituer.



Dans l'architecture de von Neumann, la mémoire fait référence à la mémoire principale ou centrale. Contrairement à un disque dur (mémoire secondaire), cette mémoire est rapide et également directement accessible par l'UCT.

La mémoire est divisée en un certains nombres d'emplacements (mots mémoires). Chaque emplacement est identifié d'une manière unique par une adresse. Seul le processeur peut accéder aux emplacements mémoires soit en écriture soit en lecture. En écriture, le processeur spécifie une adresse et un contenu, et la mémoire range le contenu dans l'emplacement indiqué par l'adresse. En lecture, le processeur spécifie une adresse donnée et récupère son contenu qui demeure inchangé.

Deux types de mémoires couramment utilisés sont la mémoire à accès aléatoire (Random Access Memory, RAM) ou mémoire vive et la mémoire morte (Read Only Memory, ROM).

La RAM est caractérisée par un stockage temporaire des instructions et des données. Elles risquent d'être altérées par un défaut d'alimentation électrique, ainsi, la RAM est qualifiée de volatile.

La ROM pouvant être lue mais pas (ou peu de fois) écrite. Ce type de mémoires est non volatil ; il contient des programmes nécessaires au fonctionnement du matériel, notamment, lors du chargement du système d'exploitation.

L'inscription des informations dans les ROMs s'appelle programmation, qui est généralement réalisée par le fabricant selon une certaine méthode suivant le type de ROM :

- ROM.
- PROM (Programmable ROM) : pouvant être écrite une seule fois par l'utilisateur.
- REPROM (REProgrammable ROM) : pouvant être écrite un certain nombre de fois par l'utilisateur :
  - E-PROM (Erasable PROM) : effacement par exposition aux ultraviolets.
  - EA-PROM (Electrically Alterable PROM) : modification par tension électrique.
  - EE-PROM (Electrically Erasable PROM) : effacement par tension électrique.
  - FLASH EPROM : EE-PROM, mais effacement par bloc (typiquement 512 octets ou plus) et non pas octet pat octet.

Une des caractéristiques principales des mémoires est la capacité (Définition 1.5).

Définition 1.5 (Capacité) La capacité (taille) d'une mémoire est le nombre (quantité) d'information que peut enregistrer cette mémoire.

La capacité peut être exprimée en :

- bit : Le terme bit est une contraction des mots «binary» et «digit». Un bit ne peut prendre que deux valeurs, désignées le plus souvent par les chiffres 0 et 1. Il représente la quantité minimale de l'information. Ainsi, il constitue l'unité de mesure de l'information dans l'informatique.
- Octet : 1 Octet = 8 bits.
- kilo-octets (KO) : 1 kilo-octets (KO) = 1024 octets = 2^{10} octets.
- Méga-octets (MO) : 1 Méga-octets (MO) = 1024 KO = 2^{20} octets.
- Géga-octets (GO) : 1 Géga-octets (GO) = 1024 MO = 2^{30} octets.
- Téra-octets (TO) : 1 Téra-octets (TO) = 1024 GO = 2^{40} octets.

Par ailleurs, dans la littérature, on parle aussi de *cache* et de *registre* comme des types de mémoires avec une vitesse d'accès considérable qui correspond à la vitesse de l'UCT.

Dans ce contexte, nous profitons de l'occasion pour introduire les mémoires secondaires qui sont des dispositifs de stockage.

Dispositifs de stockage

Dans cette section, nous nous intéressons aux dispositifs de stockage ou mémoires secondaires. Ce type de mémoires conserve les informations indéfiniment, alors, il est qualifié de non-volatil. Autrement dit, l'information persiste même après la coupure de l'alimentation. Il existe de nombreux types de mémoires secondaires, nous en présentons les plus répandus :

Disques magnétiques

Un disque magnétique est un dispositif de stockage d'informations prenant en charge une densité de stockage élevée et un temps d'accès relativement rapide. Il est constitué d'une surface rigide recouverte d'un revêtement magnétique.

Dans le cas d'un disque dur disposant de plusieurs têtes de lectures, une seule tête est utilisée à tout moment pour lire ou écrire ; les données sont donc stockées en série, même si les têtes peuvent en principe être utilisées pour lire ou écrire plusieurs bits en parallèle. Les têtes écrivent les données en magnétisant le matériau magnétique lorsqu'il passe sous les têtes et lisent les données en détectant les champs magnétiques. La capacité actuelle des disques durs est de l'ordre de TO.

Une *disquette* (disque souple) contient un plateau en plastique souple recouvert d'un matériau magnétique tel que l'oxyde de fer. Le temps d'accès pour les disquettes est généralement plus lent qu'un disque dur. Les capacités des disquettes peuvent atteindre 1.44 MO. Cependant, des lecteurs disques de haute capacité ont fait leur apparition plus tard. Par exemple le *Iomega Zip* a une capacité de 100 MO alors que la capacité de *Iomega Jaz* est de 2 GO.

Disques optiques

Les *disques optiques* utilisent la technologie des rayons laser pour stocker et récupérer des données.

CD ROM : Un CD ROM (Compact Disk Read Only Memory) est un disque optique de 12 cm de diamètre et de 1.2 mm d'épaisseur pouvant être écrit une seule fois et lu à volonté (Write Once Read Many). Les CD ROM sont utilisés pour les données informatique suite au développement de la technologie des CD introduite en 1983 et appliquée sur l'audio. Un CD ROM est composé de plastique recouvert d'aluminium, qui réfléchit la lumière différemment pour les plats (en anglais lands) ou les creux (en anglais pits), qui sont des zones créés lors du processus de gravure.

Un CD ROM permet de stocker environ 650 MO de données informatiques ou 74 mn de données audio. Les données sont inscrites sur le disque sous forme d'une piste en spirale qui fait près de 5 km de longueur, du centre vers l'extérieur et compte 22188 tours, de la manière suivante :

- la taille d'un bit sur le CD est normalisée et correspond à la distance de $0,278\ \mu m$.
- les "1" sont inscrits sous forme d'une transition (bord de cuvette)
- les "0" sont inscrits sous forme d'une zone plate (fond de cuvette ou plat)

DVD : Le disque numérique polyvalent, ou DVD pour Digital Versatile Disc, est une version plus récente du stockage sur disque optique. Il existe des normes industrielles pour le stockage de données sur DVD-Audio, DVD-Video, DVD-ROM et DVD-RAM. Lorsqu'un seul côté du DVD est utilisé, sa capacité de stockage peut atteindre 4,7 GO. Les normes DVD incluent également la possibilité de stocker des données des deux faces dans deux couches de chaque face, pour une capacité totale de 17 GO. La technologie DVD est une avancée progressive par rapport au CD, et non une technologie entièrement nouvelle. En fait, le lecteur de DVD est rétrocompatible. Il peut être utilisé pour lire des CD et des CD-ROM ainsi que des DVD.

Autres : Un CD R (CD Recordable) est un CD initialement vierge permettant à l'utilisateur d'enregistrer ses données à l'aide d'un graveur CD. Mais, une fois gravé, le CD R s'utilise comme un simple CD ROM.

Un *DVD-R* a le même format que pour un DVD-ROM mais il n'est plus possible d'utiliser plusieurs couches par face. On obtient donc une capacité de 4,7 GO par face.

Un *CD RW (CD ReWritable)* est un CD ré-inscriptible pouvant être effacé et écrit à l'aide d'un graveur CD un très grand nombre de fois. Il peut être lu par un lecteur CD.

Un *disque Blue-Ray* est un disque optique qui fait appel à une diode laser bleue mise au point en 1996 par Shuji Nakamura. L'utilisation de ce type de diode permet de diminuer la taille du spot et donc d'augmenter la densité des données sur le disque. Par conséquent, les disques Blu-Ray peuvent stocker jusqu'à 27Go sur une seule couche, et peuvent donc atteindre 100 GO sur 4 couches.

Périphériques

Définition 1.6 (Périphérique) Un périphérique est un dispositif matériel qu'on peut attacher à un ordinateur et qui assure les échanges d'informations en entrée et/ou en sortie entre l'ordinateur et l'environnement extérieur.

Nous distinguons trois types de périphériques, à savoir, d'entrée de sortie et mixtes. Toutefois, il faut noter que certains ouvrages incluent les dispositifs de stockage comme un type de périphérique.

Périphériques d'entrée Un périphérique d'entrée et un équipement permettant d'entrer l'information à l'ordinateur :

- Clavier : Le clavier est un périphérique qui contient plusieurs touches pour écrire et saisir les différents caractères (lettres, chiffres, ponctuation, ...) ainsi que des touches spéciales et des touches de fonction. La distribution des lettres d'une langue dans le clavier est une fonction de la fréquence d'utilisation de ces lettres. Le principe consiste à placer les lettres les plus fréquentes sur les touches les plus accessibles du clavier. Il existe de nombreuses dispositions des touches : AZERTY, QWERTY, QWERTZ, Dvorak, ...

- Souris : La souris est un périphérique de pointage servant à déplacer un curseur sur l'écran et permettant de sélectionner, déplacer, manipuler des objets en cliquant sur un bouton. Il existe plusieurs familles de souris (mécaniques, opto-mécaniques, optiques, sans fil).

- Microphone : Le microphone sert à faire entrer du son (voix) à l'ordinateur.

- Webcam : Une webcam est une caméra qui sert pour filmer et produire une vidéo comme une entrée à l'ordinateur. Elle est généralement située sur le bandeau haut de l'écran. Elle peut être externe ou interne.

- Scanner : Le scanner est un accessoire qui sert à numériser un documents (texte ou image) et le faire entrer à l'ordinateur comme une image numérique. Il est possible de transformer cette image en texte en utilisant la technique d'OCR (Optical Character Recognition). Dans l'entreprise, un scanner est généralement utilisé conjointement avec un GED (Gestion Électronique des Documents).

- Autres : Lecteur d'étiquettes codes barres, lecteur de QR (Quick Response) code, pointeuses biométriques à empreinte et à puce, lecteur de cartes de crédit, ...

Périphériques de sortie

- Écran : Un écran ou moniteur est un périphérique qui permet d'afficher les informations saisies ou réclamées par l'utilisateur. En effet, il affiche les images générées par la carte graphique de l'ordinateur. Il existe deux familles d'écrans : Les écrans à tube cathodique (notés CRT pour Cathod Ray Tube) et les écrans plats qui sont des panneaux à cristaux liquides ou électroluminescents.

- Imprimante : Une imprimante est un périphérique permettant de transférer des textes ou des images (depuis l'ordinateur) sur du papier ou sur des feuilles de transparent. Dans le marché, il existe essentiellement trois modèles d'imprimantes : matricielles, à jet d'encre et laser.

- Table traçante : Une table traçante ou traceur est un périphérique d'impression en mode trait. Ce périphérique est utilisé dans la CAO (Conception Assistée par Ordinateur) et dans le DAO (Dessin Assisté par Ordinateur). Depuis les années 1980, les tables traçantes ont été remplacées par des imprimantes laser et à jet d'encre de grand format.

- Haut parleur/casque : Un haut parleur est un périphérique qui permet d'émettre les sons provenant de l'ordinateur. Certains écrans disposent de haut-parleurs intégrés.

Périphériques mixtes

- Modem : Un modem (modulateur démodulateur) est un petit boîtier qui permet de se connecter à Internet. Aussi, on utilise un modem entre deux ordinateurs pour échanger des données à travers le réseau téléphonique. La modulation est la fonction de sortie, elle permet de convertir un signal numérique en un signal analogique. La démodulation constitue la fonction d'entrée.
- Unité multifonctions : Une unité multifonctions est un périphérique capable d'effectuer plusieurs tâches d'entrée/sortie : impression, numérisation, envoyer des faxes, recevoir des faxes, ...

2. Éléments de base de l'algorithmique

Avant d'étudier une nouvelle discipline, habituellement, on commence par ses éléments de base. Ce chapitre prend en charge cette tâche et se propose d'introduire les éléments de base d'un algorithme.

En effet, nous commençons par la définition d'un algorithme et l'étude de ses propriétés. Ensuite et dans une perspective de donner une vision claire du travail du concepteur des algorithmes, nous faisons un tour d'horizon complet des étapes nécessaires pour passer d'un problème au résultat.

J'invite les étudiants de jeter un coup d'oeil sur le livre "Algorithmique Raisonner pour concevoir" [Haro, 2015]. Son premier chapitre est une bonne introduction de la notion algorithmique.

2.1 C'est quoi un algorithme ?

Afin de répondre à la question «c'est quoi un algorithme ?», nous commençons par un certains nombres de questions de la vie quotidienne :

- Savez vous comment démarrer une voiture jusqu'à la faire rouler ?
- Savez vous calculer les racines d'un polynôme du seconde degré : $ax^2 + bx + c = 0$, avec $a \neq 0$?
- Avez vous installé une imprimante ?

Si la réponse est oui, sans le savoir, vous avez déjà exécuté un algorithme !

Encore d'autres questions :

- Avez vous indiqué la route pour un égaré ?
- Avez vous montré à ta voisine une recette de cuisine ?

Si la réponse est oui, sans le savoir, vous avez déjà (construit) et faire exécuter un algorithme !

À travers ce petit interrogatoire, nous pouvons déduire que les algorithmes font partie de notre vie quotidienne que l'on sache ou non.

Toujours, dans notre chemin vers une définition du mot algorithme, donnons des exemples d'illustration (Exemples 2.1 et 2.2) qui essaient de répondre à quelques questions vues précédemment :

Exemple 2.1 (Démarrer une voiture jusqu'à la faire rouler)

1. Ouvrir la portière de la voiture,
2. s'asseoir (où il y a le volant),
3. mettre la ceinture de sécurité,
4. adapter le siège,
5. régler les rétroviseurs,
6. vérifier que c'est au point mort,
7. mettre la clé dans le démarreur,
8. tourner la clé,
9. enfoncer la pédale d'embrayage (celle de gauche),
10. passer la 1ère vitesse,
11. relâcher doucement la pédale d'embrayage,
12. n'oublier pas de retirer le frein à main.

Exemple 2.2 (Résolution de l'équation $ax^2 + bx + c = 0$, avec $a \neq 0$)

1. Saisir les valeurs de (a, b, c),
2. calculer $\Delta = b^2 - 4ac$,
3. si $\Delta < 0$ alors pas de racines dans $\mathbb{R}$
   si $\Delta = 0$ alors racine double : $x = -\frac{b}{2a}$
   si $\Delta > 0$ alors deux racines :
   $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$.

Maintenant, suite à l'interrogatoire et aux Exemples 2.1 et 2.2, nous pouvons aboutir à une première définition (Définition 2.1) du mot algorithme :

Définition 2.1 (Algorithme) Un algorithme est une *suite d'actions* qui une fois *exécutée* correctement, conduit à un *résultat* donné.

Nous essayons de discuter quelques points dans la Définition 2.1 pour fixer quelques idées :

- Une *action* est une étape de l'algorithme. Elle peut être composée de plusieurs *actions primitives*. Une *action primitive* peut être exécutée sans aucune information complémentaire.
- Une action est *exécutée* par un *processeur* qui peut être défini comme étant une entité en mesure de comprendre et d'exécuter les actions constituant un algorithme en manipulant un ensemble d'objets (éléments) appelé *environnement*. Un processeur peut être :
  - Une personne,
  - un dispositif électronique,
  - un dispositif mécanique,
  - un *ordinateur*.

Dans l'optique de la discussion ci-dessus, nous pouvons étendre la Définition 2.1 comme suit :

Définition 2.2 (Algorithme) Un algorithme est une séquence (suite) d'actions primitives qui, exécutées par un processeur bien défini, réalisera un travail bien précis.

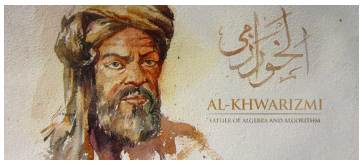
2.2 Propriétés d'un algorithme

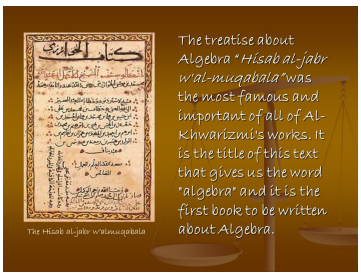
Après avoir introduit la définition du terme algorithme (Définition 2.2), il est recommandé de présenter les propriétés qu'un algorithme doit avoir :

- *Généralité* : Il doit tenir compte de tous les cas possibles. Il traite le cas général et les cas particuliers.
- *Finitude* : Il doit toujours s'arrêter au bout d'un temps fini.
- *Répétetivité* : Il est en général répétitif (il contient un traitement qui se répète).
- Il est indépendant des langages de programmation et des matériels informatiques.
- Dans des conditions d'exécution similaires (avec des données identiques), il fournit toujours le même résultat.

2.3 Origine du mot algorithme

Nous pensons à un algorithme comme quelque chose de nouveau, mais le terme remonte, en fait, à environ 900 ans. Le mot algorithme vient du nom du génie mathématicien Perse Arabophone **Muhammad ibn Musa al-Kharezmi**. Il est né autour de 780 après JC dans la région qui lui a donné son surnom, le Kharezm, aujourd'hui en Ouzbékistan. Sous l'ère du Calif abbasside Abdallah al-Mamun, il était membre de la «maison de sagesse» à Bagdad, un institut où les esprits savants effectuaient des recherches scientifiques. Il a apporté des contributions novatrices aux mathématiques, à l'astronomie, à la géographie et à la cartographie. Il a écrit un livre influent intitulé «Concernant l'art Hindou de calcul». Trois cents ans plus tard, le livre a été redécouvert et traduit en latin. Il introduit des chiffres hindou-arabes à l'ouest, tout en remplaçant éventuellement ceux trop lourds de Romains. Le système de numération hindou-arabe, ainsi que le point décimal, décrits à la fois dans le livre d'Al-khawarizmi, constituent la base des nombres que nous utilisons aujourd'hui dans le monde entier. Le nom d'Al-khawarizmi, une fois latinisé dans le titre du livre, est devenu *algorithmi* et ceci est à l'origine du mot *algorithme*.

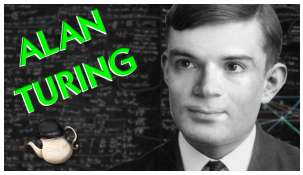




Nous devons également remercier Al-khawarizmi pour le mot *algèbre* qui vient d'un autre traité, intitulé *Hisab al-jabr w'al-muqabala*. Ses ouvrages ont révolutionné les mathématiques en occident, montrant comment des problèmes complexes peuvent être décomposés en parties plus simples et résolus.

Le mot algorithme a été forgé au moyen-âge pour désigner simplement les techniques de calcul liées au système de numération de position, autrement dit le calcul sur les chiffres, introduit en Europe par les Arabes. Au 13ème siècle, il était devenu un mot anglais. Mais ce n'est qu'à la fin du 19ème siècle que le mot algorithme est venu pour signifier des actions exécutées étape par étape pour résoudre un problème.

Au début du XXe siècle, Alan Turing, mathématicien et informaticien britannique, expliqua comment, en théorie, une machine pouvait suivre des instructions algorithmiques et résoudre des problèmes mathématiques complexes. C'était la naissance de l'ère informatique.



Pendant la seconde guerre mondiale, Alan Turing construisit une machine appelée *Bombe* qui utilisait des algorithmes pour casser les codes allemands d'*Enigma* (une machine électromécanique portable servant au chiffrement et au déchiffrement de l'information).

Les algorithmes sont maintenant partout pour nous aider à passer d'un point A à un point B, faire des recherches sur Internet, faire des recommandations sur ce que nous pouvons acheter, regarder ou

partager et prédire.

Ce petit mot qui tire son origine du moyen-âge Arabe transforme peu à peu nos vies.

2.4 Du problème au Résultat

Dans une perspective de donner une vue générale du travail de l'informaticien programmeur, dans la présente section, nous essayons de faire un tour d'horizon sur les étapes de résolution d'un problème, ceci depuis le contact avec le problème jusqu'au l'obtention des résultats.

Pour aider l'étudiant à nous suivre aisément dans notre promenade, nous adoptons une ingénierie inverse qui commence par quelque chose de plus concret pour l'étudiant, les application informatiques, ensuite, nous rebroussons chemin tout en introduisant les ingrédients nécessaires pour arriver au point de départ, bien sûr, le problème à résoudre.

2.4.1 Applications de l'ordinateur

Depuis les années 1950, Les applications de l'ordinateur ne cessent de se développer et de se diversifier. Elles deviennent un outil indispensable dans notre vie professionnelle et privée. À titre non exhaustif, nous pouvons citer les applications suivantes :

- Accès à Internet,
- envoi de courrier électronique,
- création de sites web,
- archivage et retouche de photos,
- jeux vidéo,
- bureautique : traitement de texte, tableur, gestion de bases de données, . . . ,
- gestion et comptabilité : facturation, paye, stocks, . . . ,
- calcul scientifique,
- prévisions météorologiques,
- aide à la conception électronique (CAO) ou graphique (DAO),
- pilotage de satellites, d'expériences,. . . .

Mais la question qui se pose, pourquoi une telle machine, l'ordinateur, peut-il effectuer plusieurs tâches aussi variées ?

La réponse est assez simple, car il est possible de programmer l'ordinateur. Nous entendons dire par programmer que nous devons fournir à l'ordinateur l'ensemble des opérations à exécuter.

2.4.2 Programme

Nous commençons par la Définition 2.3.

Définition 2.3 (Programme) Un *programme* est constitué d'un ensemble de directives nommées *instructions*.

Les instructions spécifient (à un ordinateur) :

- les opérations élémentaires à exécuter,
- la manière dont elles s'enchaînent.

Mais il faut noter qu'un ordinateur possède un répertoire limité d'*opérations élémentaires* qu'il sait exécuter très rapidement. De même, il sait faire des choix ainsi que des répétitions.

La Figure 2.1 illustre la relation entre un algorithme et un programme.

FIGURE 2.1 – Traduction d'un algorithme en un programme.

En effet, un algorithme est écrit, dans un papier, décrivant formellement une méthode de résolution d'un problème. Cette description doit être dans un langage précis et compréhensible par l'être humain.

Par contre, un programme est une représentation d'une suite d'instructions en mémoire d'une machine. Il s'agit d'une représentation dans un langage de programmation qui puisse être interprétée par la machine.

Donc, pour passer d'un algorithme à un programme, nous aurons besoin d'une traduction, en générale, réalisée manuellement par le développeur de l'application.

Suite à cette illustration, la Définition 2.4 s'en découle.

Définition 2.4 (Programme) Un programme est une *séquence d'instructions* écrites dans un *langage de programmation* traduisant un algorithme.

2.4.3 Langages de programmation

Dans la Section 2.4.2, nous avons introduit le concept langage de programmation, peut être, sans donner de raison. Dans la présente section, nous discutons la raison pour laquelle nous utilisons un langage de programmation. Par la suite, nous réalisons une investigation de quelques langages de programmation couramment utilisés.

Traduction

Un ordinateur ne connaît que le système de numération binaire. Le seul langage que l'ordinateur puisse comprendre est une longue suite de bits (0 et 1) souvent traités par mots (groupes de bits de 8, 16, 32 ou 64). Alors, toute information (donnée ou instruction) traitée par l'ordinateur doit être codée en binaire.

À titre d'exemple, supposons que l'instruction machine 0101010011011010 signifie additionner (code opération 0101) le contenu du registre accumulateur *AX* (010011) avec une valeur située à l'adresse 011010 et replacer le résultat dans l'accumulateur.

Il est clair, qu'il est difficile, voire impossible, pour un être humain, d'écrire et de comprendre des programmes en langage machine.

Une solution consiste à utiliser :

1. un langage sous forme accessible pour exprimer le programme. Le langage utilisé est dit *langage de programmation* et le programme écrit est dit *programme source* ou *code source*.

2. un *traducteur* du langage utilisé. Le traducteur se manifeste sous plusieurs formes, à savoir, *assembleur*, *interpréteur* ou *compilateur*, ceci selon le type du langage de programmation.

Ce scénario est illustré à l'aide de la Figure 2.2.

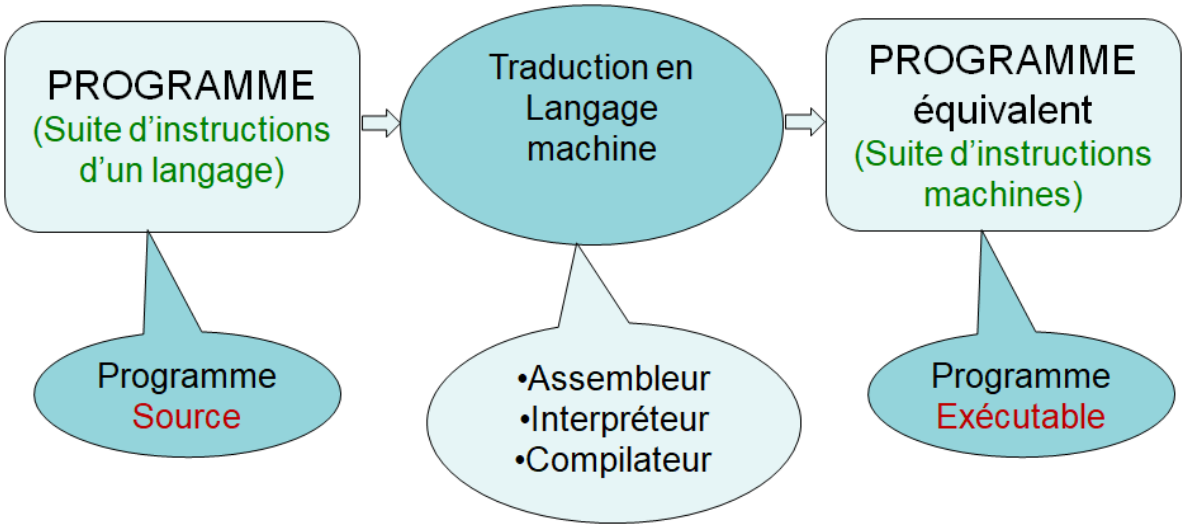


FIGURE 2.2 – Traduction d'un programme source en un programme exécutable.

Quel langage ?

Si nous réalisons une petite recherche dans Internet sur les langages de programmation, nous pouvons découvrir des centaines ! Alors, quel langage apprendre ?

Pour prendre une décision, nous devons répondre à deux questions :

1. Quel langage sur le marché du travail ?

2. Quel langage est le plus simple à apprendre ?

Pour se faire une idée de ce que le marché de travail demande, il suffit de fouiller sur les sites de demande d'emploi (usajobs, careerbuilder, indeed, monster, . . .). Voici une sélection des Top 4 des meilleurs langages de programmation de 2019 pour décrocher un job :

4. *Swift*, pour les applis IOS.

3. *Java*, pour les applis Android, développement web back-end, . . .

2. *Python*, principalement pour la Data Science et le développement web back-end.

1. *JavaScript* pour quasiment tout, notamment, pour le développement web.

La deuxième question s'intéresse à la facilité d'apprentissage. En effet, la conviction d'apprendre en premier lieu un langage simple repose sur le fait q'un tel apprentissage vous permet de bien comprendre et de se familiariser avec les concepts fondamentaux de la programmation, sans se préoccuper des éléments complexes inutiles dès le départ.

Dans notre contexte, l'objectif du module ASD1 est la conception des algorithmes. Entre autres, l'apprentissage de la programmation n'est qu'un objectif secondaire. Par ailleurs, le cursus licence Informatique, inclut d'autres modules (Outils de programmation pour les mathématiques, Algorithmique et structure de données 3, Programmation orientée objet, Développement des applications Web, Génie logiciel, Applications Mobiles, . . .) permettant, ultérieurement, d'apprendre d'autres langages (C, Matlab, Java, Python, JavaScript, . . .).

Pour ces raisons, conjointement au langage C, notre choix s'est focalisé, d'abord, sur la faciliter d'apprentissage. D'où le choix d'un langage non pas seulement simple, mais aussi pédagogique. C'est bien le langage de haut niveau Pascal.

Avant de clôturer cette section, nous essayons de faire un tour d'horizon sur les langages de programmation à travers un petit exemple qui sert à afficher le message "Hello World !".

A-Langage Assembleur (1950)

Le langage assembleur est un langage de bas niveau inventé à partir de 1950. Les séquences binaires du langage machine sont remplacées par une notation symbolique (mnémonique). La traduction de ces instructions (mnémoniques) se fait à l'aide d'un programme assembleur.
Le programme de Listing 2.1 affiche en assembleur le message "Hello World !".

```
Cseg segment
assume cs:cseg, ds:cseg
org 100h
main proc
jmp debut
mess db   Hello   world! $
debut:
mov dx, offset mess
mov ah, 9
int 21h
ret
main endp
cseg ends
end main
```

Listing 2.1 – exemple de langage assembleur *x*86 sous DOS

B-Langages de programmation procédurale

La programmation procédurale est un paradigme de programmation recommandé pour un nouveau développeur. Ce paradigme utilise une approche linéaire descendante et traite les données et les procédures comme deux entités différentes. Sur la base du concept d'appel de procédure, la programmation procédurale divise le programme en procédures, appelées également routines ou fonctions, contenant simplement une séquence d'étapes à exécuter.

Autrement dit, la programmation procédurale consiste à écrire une liste d'instructions indiquant à l'ordinateur ce qu'il doit faire étape par étape pour achever la tâche.

À titre d'exemple, dans ce qui suit, nous mentionnons quelques langages de la famille procédurale

Langage COBOL

Le langage *COBOL* pour **CO**mmon **B**usiness **O**riented **L**anguage, crée en 1959, est un langage orienté gestion.

Le Listing 2.2 est un code en COBOL pour afficher le message "Hello World!".

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
DISPLAY "Hello world!".
STOP RUN.
```

Listing 2.2 – exemple de langage COBOL

Langage Basic

Le *BASIC* est l'acronyme de **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode, crée en 1964 pour être utilisé par des non informaticiens.

Le code Basic du Listing 2.3 affiche le message "Hello World !".

```
10 PRINT "Hello world!"
20 END
```

Listing 2.3 – exemple de langage Basic

Langage Pascal

Pascal est un langage de programmation crée en 1969 à l'école polytechnique de ZURICH par N. WIRTH. Il est conçu pour enseigner la programmation comme une science. Il se caractérise par une syntaxe claire, rigoureuse et facilitant la structuration des programmes. De plus, il peut être utilisé en industrie.

Le Listing 2.4 illustre un programme Pascal qui affiche le message "Hello World !".

```
program Bonjour;
begin
  Writeln('Hello world!');
end.
```

Listing 2.4 – exemple de langage Pascal

Langage C

Le langage *C* est un langage de bas niveau crée au cours de l'année 1972 dans les laboratoires Bell par *Dennis Ritchie* et *Ken Thompson*.

Le programme C de Listing 2.5 montre un exemple d'affichage d'un message "Hello World !".

```
#include <stdio.h>
int main(int argc, char **argv)
{
printf("Hello world!\n");
return 0;
}
```

Listing 2.5 – exemple de langage C

C- Langages Orientés objets

Les langages orientés objets utilisent le paradigme de la programmation orientée objet (POO). Le paradigme POO est basé sur l'écriture, l'interaction et la réutilisation de briques logicielles appelées objets.

Le langage *Simula*-67, dans les années 1960, annonça les prémices de la POO. Cependant, la définition réelle des concepts de base de la POO (objet, message, encapsulation, polymorphisme, héritage, redéfinition, ...) est posée avec Smalltalk 72 puis Smalltalk 80.

Langage C++

Le langage *C++* (C avec des classes) est crée en 1983. Il permet l'utilisation de l'ensemble des bibliothèques C existantes. Il est très utilisé dans l'industrie.

Le Listing 2.6 montre un programme écrit en C++ qui affiche le message "Hello World!".

```
#include <iostream>
int main()
{
std::cout << "Hello world!" << std::endl;
return 0;
}
```

Listing 2.6 – exemple de langage C++

Langage Ada

Les origines du langage *Ada* remontent au début des années 1980. Ada est utilisé pour développer des applications scientifiques et de gestion. Une des caractéristiques particulières du langage Ada est le trait temps réel intégré au langage.

Le Listing 2.7 est un code en Ada pour afficher le message "Hello World!".

```
with Text_IO; use Text_IO;
procedure hello is
begin
   Put_Line("Hello world!");
end hello;
```

Listing 2.7 – exemple de langage Ada

Langage Java

Le langage *Java* est crée en 1995 par *Sun Microsystems* rachetée en 2009 par la société *Oracle* qui le maintient désormais.

Java est conçu pour développer des applications fonctionnant sur tout type de processeur et système d'exploitation. Il est bien adapté pour les applications web.

L'exemple d'affichage d'un message "Hello World!" du Listing 2.8 montre la syntaxe de Java.

```
public class HelloWorld {
  public static void main(String[] args) {
```

```
3       System.out.println("Hello world!");
4     }
5   }
```

Listing 2.8 – exemple de langage Java

D- Langages de programmation événementielle

Les langages de programmation événementielle suivent un paradigme de programmation qui repose sur le concept d'événement. Les composants de l'application communiquent entre eux via des évènements (click sur un bouton, saisie dans une zone de texte, choix dans une case d'option, chargement d'une fiche, fermeture d'une fenêtre, ...).

Ci-après une liste non exhaustive des langages de programmation événementielle :

- Visual basic
- Visual C++
- Visual j++
- C # (C Sharp)
- Delphi
- C++ builder
- J++ builder

E- Autres Langages

Langages de script

Un langage de script permet d'écrire rapidement de petits programmes interprétés dont l'utilisation principale est l'automatisation de certaines petites tâches et dont les fonctionnalités sont peu nombreuses.

À titre d'exemple, les langages suivants sont des langages de scripts :

- JavaScript
- AppleScript
- VBScript

Langages à balises

Un langage à balises est un langage d'enrichissement de l'information textuelle. Il utilise des unités syntaxiques pour délimiter une séquence de caractères ou marquer une position précise à l'intérieur d'un flux de caractères. Ces délimiteurs syntaxiques s'appellent *balises*.

Les langages suivants constituent des exemples de langages à balises :

- PS (Postscript)
- LaTeX
- HTML (Hyper Text Markup Language)
- XML (eXtensible Markup Language)

Langages de programmation web

Les langages de programmation web permettent d'éditer des sites web qui peuvent être statiques ou dynamiques.

En plus des langages de programmation déjà mentionnés comme des langages de programmation web, nous pouvons rajouter les suivants :

- PHP (Hypertext Preprocessor) est un langage de script interprété (dérivé de C et de Perl) côté serveur. Il est possible d'intégrer du code PHP dans du HTML.
- ASP (Active Server Pages) est un langage interprété défini par Microsoft en 1996 pour créer des pages web dynamiques pour Windows. Il travaille côté serveur. ASP utilise la technologie ADO (ActiveX Data Object) pour se connecter à une base de données.

2.4.4 Un algorithme pour un problème

Jusqu'à présent, nous avons décrit le chemin qui nous emmène de l'algorithme à l'application tout en dévoilant les concepts application, programme source, langage de programmation et traduction d'un programme source à un programme exécutable. Pour compléter l'image, il nous reste de discuter la démarche qui nous permet d'obtenir un algorithme pour un problème donné. C'est bien l'objet de la présente section.

Définition du problème

Le problème est posé par un client qu'on appel *instigateur* de problème. En général, le problème est posé sans détails et sans précisions. Le rôle du concepteur de l'algorithme, à ce stade, consiste à spécifier les détails du problème pour obtenir un énoncé précis :

1. Lire attentivement l'énoncé du problème pour le comprendre.
2. Spécifier toutes les informations disponibles (les données).
3. Spécifier les résultats escomptés et leurs formats, éventuellement d'une manière formelle.

À titre d'exemple, si l'énoncé du problème posé par l'instigateur est «Trouver la liste des diviseurs d'un nombre», le concepteur d'algorithme doit redéfinir le problème en précisant les données (entrées) et les résultats (sorties) du problème pour obtenir un énoncé précis «Étant donné un nombre entier N, construire une solution informatique qui nous permet d'obtenir la liste de ses diviseurs».

Analyse du problème

L'analyse du problème consiste à trouver le moyen (algorithme) de passer des données aux résultats. Dans certains cas on peut être amené à faire une étude théorique.

Dans la littérature, il existe plusieurs approches pour analyser un problème. L'analyse ascendante et l'analyse descendante font partie.

Si nous revenons au problème de calcul des diviseurs d'un nombre entier, nous pouvons proposer le résultat de l'analyse (algorithme) suivant :

- Diviser successivement N par $i = 1, 2, 3, \ldots, N/2$
  - à chaque fois que le reste de la division de N par i est égale à 0, (alors i est un diviseur)
    - imprimer i.

D'une manière plus sophistiquée, nous pouvons proposer l'Algorithme *diviseurs*.

Algorithme diviseurs

```
Variable N, i : Entier
début
    Lire(N)
    pour i de 1 à (N div 2) faire
    début
        si (N Mod i) = 0 alors
        début
            Ecrire (i)
        fin
    fin
fin
```

2.4.5 Étapes (du problème aux résultats)

Cette section se propose de faire une récapitulation de la démarche de passage du problème aux résultats. La Figure 2.3 illustre une telle démarche.

Les premières étapes, définition du problème et son analyse constituent la phase de conception qui se fait sans recours à une machine.

Ensuite, pour concrétiser l'algorithme conçu nous devons le traduire en un langage de programmation. Par exemple, l'Algorithme *diviseurs* est traduit dans le langage Pascal pour donner un programme source (div.pas) tel que montré dans la Figure 2.4.

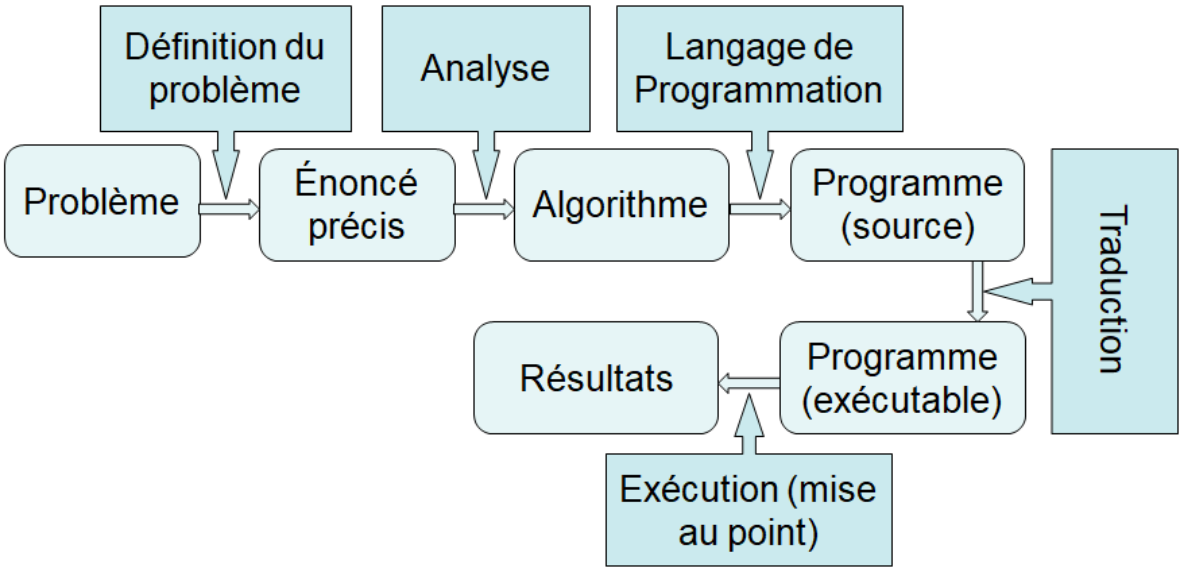


FIGURE 2.3 – Étapes de mise en oeuvre d'une application.

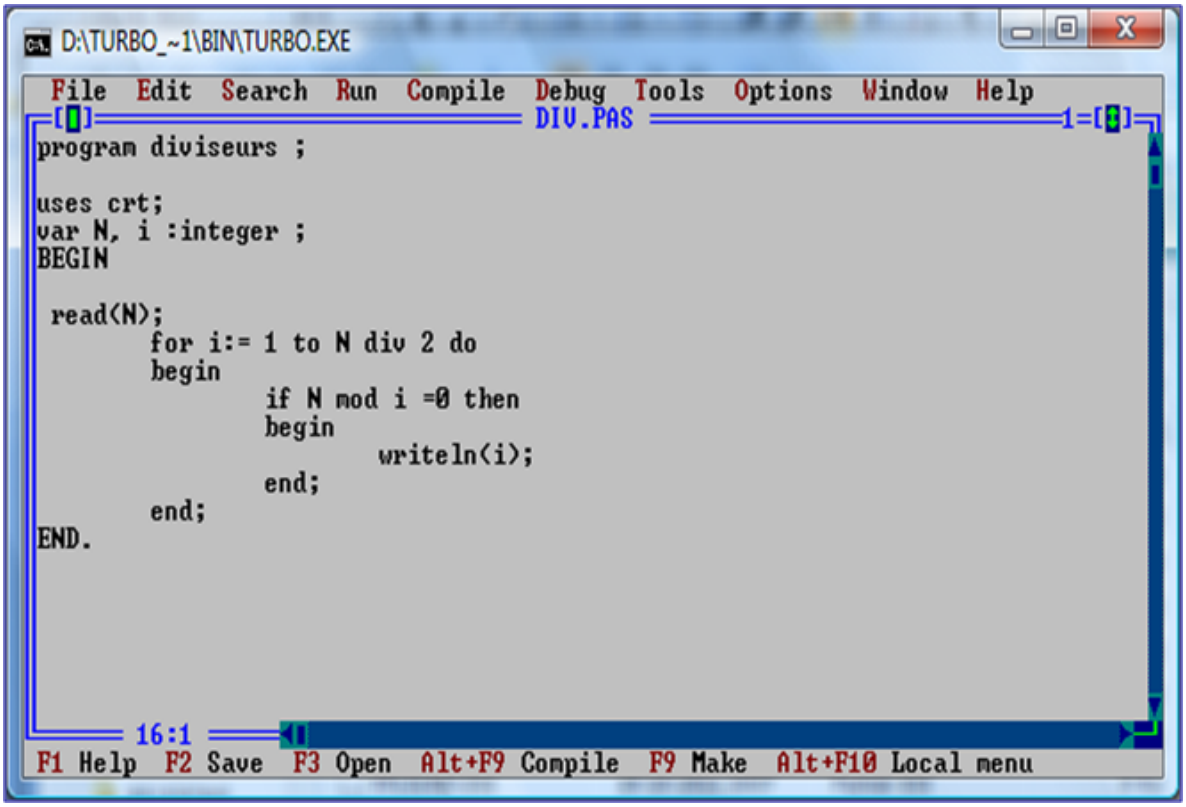


FIGURE 2.4 – Programme source (langage Pascal) associé à l'Algorithme diviseurs.

Une seconde traduction est nécessaire, cette fois ci, le traducteur est un compilateur Pascal qui sert d'abord à lire et à traiter le programme source pour le convertir en langage machine directement compréhensible par la machine. Le fichier obtenu

(programme exécutable) peut être lancé comme n'importe quel autre programme en langage machine.

Après avoir obtenu le programme exécutable, on doit achever le processus de mise en oeuvre d'une application par la phase de mise au point qui consiste, dans un premier temps, d'exécuter le code obtenu et de voir les résultats. La Figure 2.5 montre le résultat de l'exécution du code machine pour le cas de calcul des diviseurs d'un nombre entier.

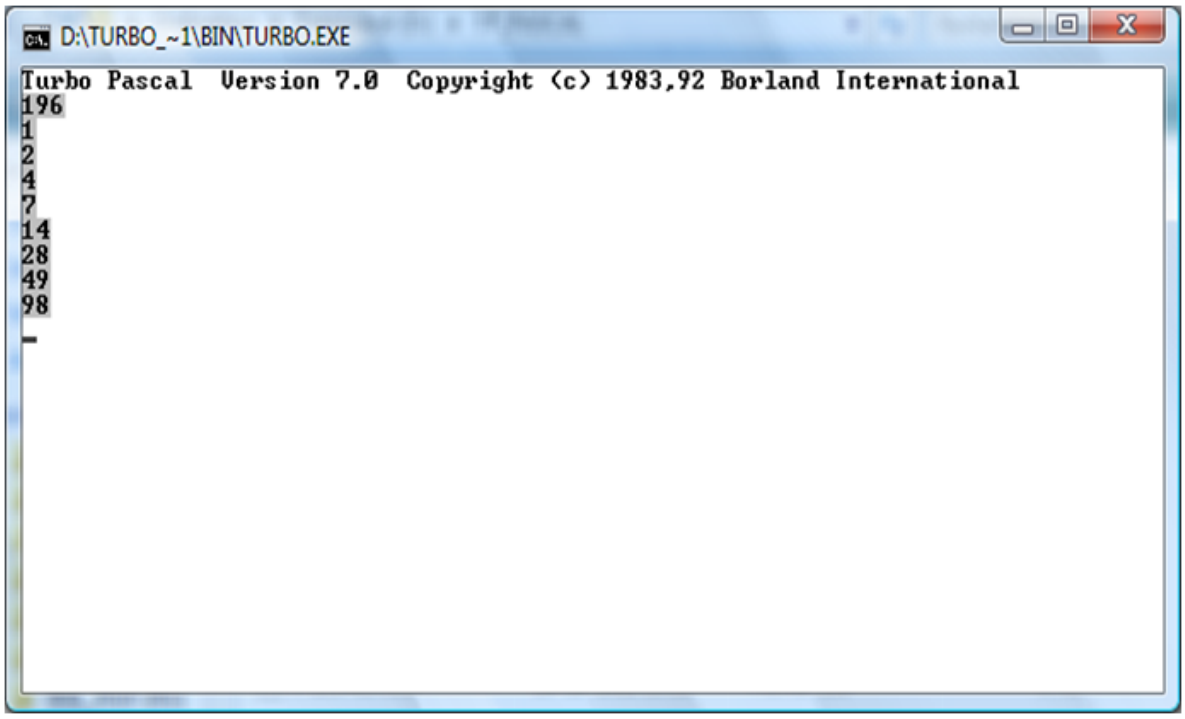


Figure 2.5 – Exécution du code exécutable associé au programme de calcul des diviseurs d'un nombre entier.

Une fois les résultats obtenus, il faut les comparer avec ceux attendus. S'il y a une correspondance, alors le processus est terminé, sinon on doit passer au stade de correction des erreurs qui s'étale de la correction de la traduction de l'algorithme jusqu'à l'analyse du problème.

La phase de correction peut être exprimée à l'aide du fragment de l'Algorithme *correction*.

Algorithme correction

```
début
    si Résultats obtenus = Résultats attendus alors
    début
        Exécution terminée
    fin
    sinon
    début
        Il existe des erreurs logiques
        Revoir la traduction de l'algorithme
        ou
        Revoir l'analyse du problème
    fin
fin
```

2.5 Exercices

Exercise 2.1 Disposant d'un processeur, qui est en fait un automate composé d'un bras mobile supportant une plume et d'une tablette sur laquelle on place du papier quadrillé, et sachant que cet automate ne comprend et n'exécute que deux types d'ordres :

- abaisser ou lever la plume,
- déplacer la plume dans une direction donnée et sur une distance délimitée.

(Chaque action primitive est composée de deux caractères : le premier caractère indique si la plume est abaissée ou non, il prend la valeur **A** s'il faut abaisser la plume ou la valeur **L** s'il faut lever la plume ; le deuxième caractère indique la direction et la longueur d'avancement de la plume comme indiqué dans la Figure 2.6 :
Exemple : si l'on veut tracer un carré la suite d'ordres est : (A3, A5, A7, A1).
On vous demande d'analyser puis d'écrire la suite d'ordres permettant à l'automate d'exécuter le dessin de la Figure 2.7, mais il faudra les réaliser sans lever la plume, sans repasser sur une ligne déjà tracée et sans la couper.
NOTA : La flèche indique la position initiale du bras mobile.
Quel est le problème posé par ce formalisme ? Comment l'améliorer ?

Indications Lors de la discussion du problème de formalisme proposé par cet exercice, pensez aux propriétés du langage algorithmique.

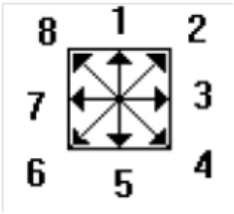


FIGURE 2.6 – Direction d'avancement de la plume.

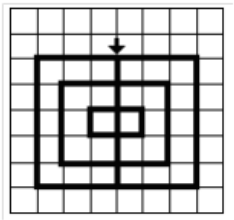


FIGURE 2.7 – Dessin à réaliser.

Exercise 2.2 Deux fellahs remplissent à chaque fois un bidon de 24 litres de lait après la traite de leurs vaches, cependant ils disposent de deux bidons de 15 litres et de 9 litres de contenances respectives. Pouvez vous les aider en leur donnant la solution qui leur permet de répartir ce lait en 2 parts égales ?
Une fois l'analyse faite, il vous est demandé de construire l'algorithme correspondant en utilisant le formalisme suivant :
Soient A, B et C les bidons de contenances respectives 24, 15 et 9 litres, les actions primitives seront de la forme $A \longrightarrow B$:

- Par exemple, si on prend une partie de A pour remplir B, on notera ceci par l'action primitive $A \longrightarrow B^*$;
- par contre, si on transvase A dans B sans le remplir on notera $A \longrightarrow B$.

Indications Essayez de trouvez plusieurs solutions et les comparer en matière de nombres d'étapes.

Exercise 2.3 Quelque temps plus tard ils veulent offrir à un parent 4 litres d'huile d'olives mais ils n'ont que 2 pots de contenances respectives 5 et 3 litres, comment vont-ils faire ? (Utilisez pour l'écriture de l'algorithme le même formalisme que celui de l'Exercice 2.2).

Exercise 2.4 Comment remplir le triangle de la Figure 2.8 de manière à ce que le nombre écrit dans une case soit égal à la somme des 2 nombres qui se trouvent dans les 2 cases du dessous

Indications Formalisez le problème mathématiquement sous forme d'un système d'équations.

Exercise 2.5 Vous disposez de 12 pièces apparemment identiques mais vous êtes sûrs que l'une d'entre elles est fausse, car elle n'a pas le même poids que toutes les autres. Comment ferez-vous, à l'aide d'une balance à double plateau et en 3 pesées maximum, pour retrouver la pièce fausse et savoir si elle est plus

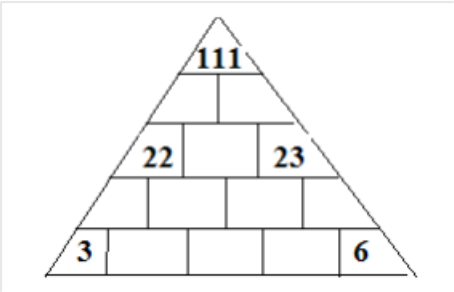


FIGURE 2.8 – Triangle à remplir..

lourde ou plus légère ? Construisez l'analyse de ce problème.

Exercise 2.6 On a 10 piles de 10 pièces qui se ressemblent mais on sait qu'une pile est composée entièrement de pièces fausses. Sachant qu'une bonne pièce pèse 5 grammes et qu'une pièce fausse pèse 6 grammes. Donner l'analyse qui nous permet de retrouver la pile de pièces fausses en une seule pesée et ensuite d'écrire l'algorithme correspondant ?

Exercise 2.7 Trois personnes, portant chacune un chapeau sur la tête, sont alignées l'une derrière l'autre de façon à ce que chacune ne peut voir que devant elle. Sachant qu'au départ on dispose de 5 chapeaux, 3 de couleur rouge et 2 de couleur verte et que chacune des personnes peut voir les chapeaux restants, comment la personne, qui se trouve devant, fera t'elle pour connaître la couleur de son chapeau alors qu'elle n'a le droit, si elle le souhaite, de poser qu'une seule question aux 2 autres personnes et les concernant personnellement ? (Donner son analyse)

Exercise 2.8 Comment 2036 : 4 = 10 ?

Exercise 2.9 Soit la suite : 1, 11, 21, 1112, 3112, . . . Quel est l'élément suivant ?

Exercise 2.10 Comment peut-on obtenir 1000 à partir d'une addition qui ne contient que des 8 ?

3. Présentation du formalisme algorithmique

Le présent chapitre se fixe comme objectif la définition du formalisme algorithmique *qui sert à exprimer d'une manière claire et non ambiguë des solutions proposées à des problèmes analysés.*

Nous commençons d'abord par la présentation de la structure d'un algorithme qui comprend une en-tête, une partie déclaration et un corps d'algorithme.

Ensuite, nous abordons la partie traitement à travers les différentes structures de contrôle.

Pour compléter le chapitre, nous discutons la partie données via l'introduction de la notion de variable et les différentes actions associées, ainsi qu'une description plus ou moins détaillée de ce qu'on appelle l'environnement d'un algorithme (partie déclaration).

En vue de compléter leurs connaissances, les étudiants sont encouragés de piocher les deux ouvrages [Zegour, 2013a, Zegour, 2013b]. Ils traitent quatre aspects : algorithmique, donnée, méthodologie et programmation. En outre, ils renferment un recueil de sujets d'examens corrigées ainsi que des exercices de programmation en langage PASCAL. En outre, Je trouve que le livre "Algorithmes et exercices corrigés" [Tormento and Boutin, 1989] est très accessibles en matière de compréhension pour les étudiants novices.

Pour les étudiants ayant plus de curiosité, il n'ont qu'à passer des nuits avec les ouvrages de niveau intermédiaire et qui traitent des sujets au delà de la porté du présent ouvrage [Cormen, 2013, Cormen et al., 2010, Sedgewick and Wayne, 2011].

Conformément au proverbe "C'est en forgeant qu'on devient forgeron", au delà des exercices proposés dans chaque chapitre, les étudiants sont invités à discuter, analyser et proposer des solutions aux problèmes posés, à titre indicatif, dans les livres [Bessaa, 2018, Laurent and Ayel, 1985, Richard and Richard, 1985]. Ils contiennent des exercices corrigés couvrant les éléments de base, les structures de contrôles, les tableaux et chaînes de caractères.

3.1 Préambule

Nous commençons ce chapitre par un petit exercice :

Exercise 3.1 Exprimer en langage naturel la solution de l'équation du second degré $ax^2 + bx + c = 0$.

Si nous demandons aux étudiants d'une classe du module algorithmique de proposer une solutions à l'Exercice 3.1, nous pourrons avoir autant de propositions (en matière d'expression) que le nombre d'étudiants.

À titre d'illustration, nous proposons deux solutions :

Proposition de solution 1

$a, b, c \longleftarrow$ les données
Calculer le discriminant
Cas où il est> 0 : $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$, $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$
Cas où il est nul : $x_1 = x_2 = -\frac{b}{2a}$
Cas où il est négatif : pas de solution

Proposition de solution 2

1. Saisir les valeurs de (a, b, c)
2. calculer $\Delta = b^2 – 4ac$
3. Si $\Delta < 0$ alors pas de racines dans $\mathbb{R}$
   Si $\Delta = 0$ alors racine double
   Si $\Delta > 0$ alors deux racines

Il n'est pas difficile de conclure que la rédaction, en langage naturel, d'une solution algorithmique d'un problème donné est souvent ambiguë. Cette ambiguïté mène généralement à des interprétations différentes. D'où la nécessité de l'utilisation d'un langage commun qu'on appel *formalisme algorithmique* (Définition 3.1).

Définition 3.1 Un formalisme algorithmique est un ensemble de conventions (règles) pour décrire un algorithme.

L'objet du présent chapitre consiste à présenter ce formalisme algorithmique d'une manière détaillée.

3.2 Structure d'un algorithme

Un algorithme est composé de trois parties principales tel que illustré par la Figure 3.1. L'*en-tête* d'un algorithme sert à donner un nom à l'algorithme en utilisant

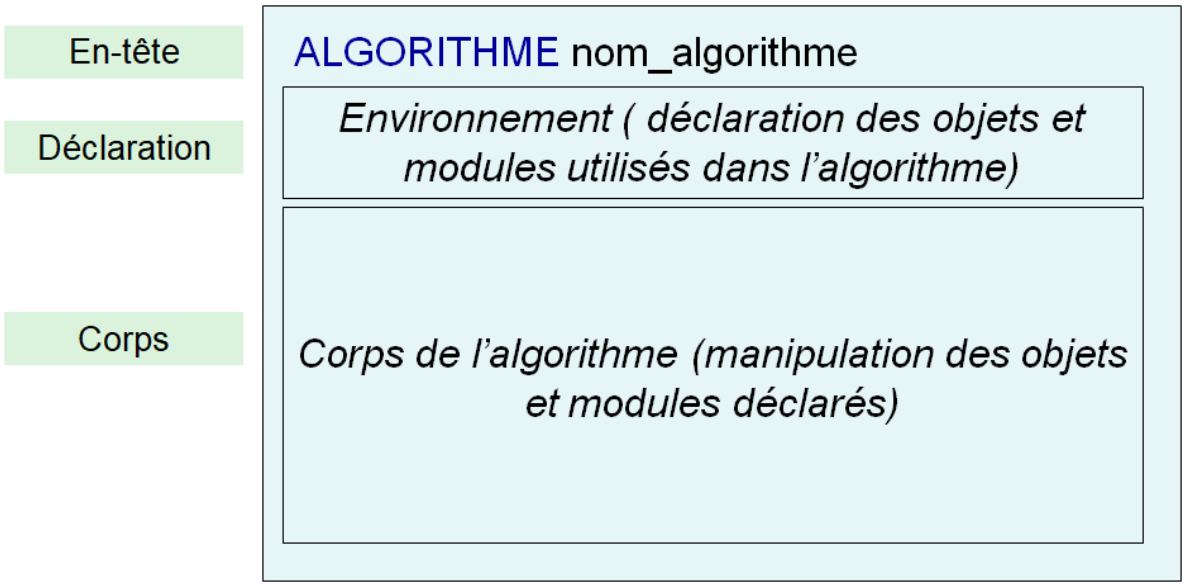


FIGURE 3.1 – Structure d'un algorithme

le mot réservé **Algorithme** avec la syntaxe suivante :

Algorithme nom_algorithme

La partie en-tête n'a aucune influence sur le déroulement de l'algorithme. Le nom sert à identifier l'algorithme parmi d'autres. Comme pour les identificateurs d'autres entités de l'algorithme (constantes, types, variables, procédures, fonctions, ...), le nom de l'algorithme est une suite de caractères alphanumériques dont le premier caractère doit être obligatoirement une lettre. Le seul caractère spécial autorisé est le caractère souligné `_` (underscore). Comme les mots réservés, les identificateurs peuvent être écrits indifféremment en majuscules ou en minuscules et peuvent avoir une taille quelconque.

La partie déclaration ou environnement sert à déclarer tous les objets (constantes, types, variables, ...) ou modules (procédures, fonctions) manipulés dans la partie corps de l'algorithme.

Le corps de l'algorithme contient les actions que l'algorithme doit effectuer pour réaliser le travail demandé. Cette partie est délimitée par les mots réservés **début** et **fin**. Elle contient des structures de contrôles qui décrivent l'enchaînement des actions.

3.3 Structures de contrôle

La description de l'enchaînement des actions au sein du corps de l'algorithme s'effectue en utilisant des structures de contrôle qui permettent des traitements séquentiels, conditionnels ou répétitifs.

3.3.1 Séquencement

Le séquencement est la structure de contrôle la plus simple. Elle donne la possibilité de la mise en séquence d'un ensemble d'actions selon la syntaxe suivante :

Action 1
Action 2
Action 3
...
Action n

L'exécution de la séquence est réalisée linéairement selon l'ordre d'apparition des actions dans l'algorithme. Autrement dit, on commence d'abord par l'exécution de l'*Action* 1, puis l'*Action* 2 jusqu'à l'*Action n*.

Exemple 3.1 Séquencement

```
Lire (A)
Lire(B)
C ← A*B
Ecrire (C)
```

L'Exemple 3.1 illustre ce principe de séquencement. Nous commençons par la lecture de la variable *A*, puis la lecture de la variable *B*, ensuite le résultat de la multiplication de *A* et de *B* est assigné à la variable *C*. Enfin, nous affichons le contenu de la variable *C*.

3.3.2 Structure conditionnelle

La structure *conditionnelle* donne la possibilité d'effectuer un choix d'un traitement selon qu'une condition soit vérifiée ou non. Alors la structure conditionnelle nous autorise à concevoir un algorithme qui n'exécutera pas certaines actions.

La syntaxe est la suivante :

si *condition* **alors**
début
| bloc
fin

La *condition* est une expression booléenne (logique) qu'on peut lui associer la valeur booléenne **vrai** ou **faux**. Un *bloc* est composé d'une ou plusieurs actions. Il

commence par le mot réservé **début** et se termine par **fin**. Dans le cas où un bloc est composé d'une seule action, les délimiteurs **début** et **fin** du bloc sont facultatifs.

L'exécution de la conditionnelle se déroule comme suit :

- évaluation de la *condition* ;
- si la *condition* est vraie, le bloc est exécuté puis le contrôle passe à la suite de la conditionnelle ;
- si la *condition* est fausse, le contrôle passe à la suite de la conditionnelle, sans exécuter le bloc.

Exemple 3.2 conditionnelle avec un bloc d'une seule action

```
Lire (A)
positif ← A
si (positif < 0) alors
début
    positif ← −1 ∗ positif
fin
Ecrire ('la valeur positive est : ' , positif)
```

Comme le bloc de la conditionnelle de l'Exemple 3.2 contient une seule action, nous pouvons omettre les mots réservés **début** et **fin** pour obtenir le fragment du code suivant :

```
Lire (A)
positif ← A
si (positif < 0) alors
    positif ← −1 ∗ positif
Ecrire ('la valeur positive est : ' , positif)
```

Dans le cas de l'Exemple 3.3, il n'est pas possible d'omettre **début** et **fin** du bloc.

Exemple 3.3 conditionnelle avec un bloc de plus d'une action

```
Lire (A)
positif ← A
si (positif < 0) alors
début
    positif ← −1 ∗ positif
    Ecrire ('la valeur positive est : ' , positif)
fin
```

3.3.3 Structure alternative

La structure *alternative* est une autre forme de la structure conditionnelle. Elle permet d'exprimer un choix entre deux traitements selon la valeur d'une condition.

La syntaxe de cette structure est :

si *condition* **alors**
début
| bloc 1 /* bloc 1 exécuté si condition égale Vrai */
fin
sinon
début
| bloc 2 /* bloc 2 exécuté si condition égale Faux */
fin

Si la *condition* est vérifiée alors *bloc 1* est exécuté, si elle n'est pas vérifiée c'est *bloc 2* qui est exécuté. Dans tous les cas, l'exécution continuera à la suite de la structure alternative.

L'Exemple 3.4 montre l'utilisation de la structure alternative.

Exemple 3.4 alternative

Lire (A)
Lire (B)
si $A > B$ **alors**
début
| $max \leftarrow A$
| $A \leftarrow A + B$
fin
sinon
début
| $max \leftarrow B$
| $B \leftarrow A + B$
fin
Ecrire ('le maximum est :', max)

Il est à noter que l'indentation (le décalage) dans l'écriture d'un algorithme sont nécessaire à sa bonne lisibilité. la bonne présentation d'un algorithme montre qu'on a compris son exécution.

La règle d'une bonne présentation est assez simple. Elle peut être introduite à travers les deux conditionnelles suivantes :

si *début de bloc rencontré* **alors**
début
| décaler (d'une tabulation) vers la droite
fin
si *fin de bloc rencontré* **alors**
début
| décaler (d'une tabulation) vers la gauche
fin

À titre d'illustration, il est évident de distinguer en terme de visibilité des blocs

d'actions entre les deux Algorithmes *avec_indentation* et *sans_indentation* qui sont en effet identiques :

Algorithme avec_indentation

```
début
    Lire (A)
    Lire (B)
    si (A > B) alors
        max ← A
    sinon
        max ← B
    Ecrire ('le maximum est :', max)
fin
```

Algorithme sans_indentation

```
début
Lire (A)
Lire (B)
si (A > B) alors
max ← A
sinon
max ← B
Ecrire ('le maximum est :', max)
fin
```

3.3.4 Conditionnelles imbriquées

Il existe des situations où nous aurons besoin qu'une partie de structure conditionnelle contienne à son tour une autre structure conditionnelle. Dans une telle situation, nous parlons d'*imbrication* de structures conditionnelles les unes dans les autres.

Il est à préciser que cette propriété d'imbrication ne se limite pas seulement aux structures conditionnelles, mais elle peut être généralisée pour toutes les structures de contrôle que nous rencontrons dans le formalisme algorithmique.

Exemple 3.5 Écrire un corps d'algorithme qui lit une note et qui affiche le commentaire associé comme suit :

- note de 0 à 8 inclus : insuffisant
- note de 8 à 12 inclus : moyen
- note de 12 à 16 inclus : bien
- note de 16 à 20 inclus : très bien

Une solution possible au problème de l'Exemple 3.5 consiste à faire une succession de choix :

```
Lire(note)
si (note ≤ 8) alors
    Écrire ('insuffisant')
si ((note > 8) et (note ≤ 12)) alors
    Écrire ('moyen')
si ((note > 12) et (note ≤ 16)) alors
    Écrire ('bien')
si ((note > 16) et (note ≤ 20)) alors
    Écrire ('très bien')
```

Dans cette solution, à chaque note saisie par l'utilisateur, quatre tests sont réalisés.

Nous pouvons proposer une autre solution qui utilise des imbrications :

```
Lire(note)
si (note ≤ 8) alors
    Écrire ('insuffisant')
sinon   si (note ≤ 12) alors
            Écrire ('moyen')
        sinon si (note ≤ 16) alors
                Écrire ('bien')
            sinon si (note ≤ 20) alors
                    Écrire ('très bien')
```

Dans les blocs **sinon**, il est inutile de faire les tests $note > 8$, $note > 12$ et $note > 16$ car nous sommes certains que la note est strictement supérieure à 8,12 et 16 respectivement.

Dans cette solution, on est amené à faire, dans le pire des cas, quatre tests.

3.3.5 Structures répétitives

Nous avons déjà vu dans la Section 2.2 que parmi les propriétés d'un algorithme est qu'il est, en général, répétitif.

La *structure répétitive* appelée, aussi, *boucle* permet d'exécuter plusieurs fois consécutifs un bloc d'actions. Elle dispose de trois formes :

- tant que ;
- répéter ;
- pour.

Structure tant que

La structure *tant que* est l'action de répétition la plus classique, sa syntaxe est comme suit :

tant que *condition* **faire**
début
| bloc
fin

Elle permet l'exécution d'un bloc d'actions plusieurs fois tant que la condition (de continuité) est vérifiée. Le déroulement de l'exécution se fait comme suit :

1. évaluation de la *condition* ;
2. si la *condition* est vraie, on exécute le *bloc* d'action et on recommence en 1.
3. si la *condition* est fausse, on sort de la boucle et on exécute sa suite.

Exemple 3.6 (boucle tant que)

compteur ← 1
tant que *(compteur <= 4)* **faire**
début
| Ecrire (compteur)
| compteur ← compteur +1
fin

La Table 3.1 montre le déroulement du fragment d'un algorithme de l'Exemple 3.6 :

Table 3.1 – Déroulement d'une structure *tant que*

compteur	**condition (compteur** <= 4	**continuer (oui/non)**
1	Initialisation (avant d'entrer dans la boucle)	
1	(compteur <= 4) = vrai	Oui (entrer dans la boucle)
2	(compteur <= 4) = vrai	Oui
3	(compteur <= 4) = vrai	Oui
4	(compteur <= 4) = vrai	Oui
5	(compteur <= 4) = faux	Non (sortir de la boucle)

Nous pouvons constater que la boucle peut être exécutée 0, 1 ou plusieurs fois, voire une infinité de fois. Par conséquent, pour écrire une boucles, trois consignes sont à soigneusement respecter :

1. Initialiser correctement la (les) variable (s) de la condition avant d'entrer dans la boucle (**compteur** ← **1**).

2. Écrire une condition correcte (**compteur** <= **4**).

3. La modification de(s) variable(s) de la condition (**compteur** ← **compteur + 1**).

La situation où le concepteur ne respecte pas ces consignes se résume par la conditionnelle suivante :

```
si (consignes non respectées ) alors
début
    risque de ne pas entrer dans la boucle
    Ou
    de ne pas pouvoir en sortir                    /* boucle infinie */
fin
```

Du moment qu'il n'existe pas une recette bien définie pour respecter les consignes d'écriture d'une boucle, il en découle que nous pouvons obtenir des algorithmes équivalents avec des boucles différentes :

```
compteur ← 1
tant que (compteur <= 4 ) faire
début
    Ecrire (compteur)
    compteur ← compteur +1
fin
```

```
compteur ← 1
tant que (compteur <> 5 ) faire
début
    Ecrire (compteur)
    compteur ← compteur +1
fin
```

```
compteur ← 1
tant que (compteur < 5 ) faire
début
    Ecrire (compteur)
    compteur ← compteur +1
fin
```

```
compteur ← 0
tant que (compteur < 4 ) faire
début
    compteur ← compteur +1
    Ecrire (compteur)
fin
```

Structure répéter

La structure *répéter* est une autre forme de structures répétitives. Elle se note ainsi :

répéter
| bloc
jusqu'à *condition*

Le déroulement de la boucle *répéter* peut être décrit comme suit :

1. exécution du *bloc d'actions* ;
2. évaluation de la *condition* (d'arrêt) ;
3. si la condition est fausse, on recommence à 1 ;
4. si la condition est vraie, on sort de la boucle et on exécute sa suite.

L'Exemple 3.7 illustre l'utilisation de la structure répéter.

Exemple 3.7 (boucle répéter)

compteur ← 1
répéter
| Ecrire (compteur)
| compteur ← compteur +1
jusqu'à *(compteur* > 4*)*

Il est à retenir que :

- Nous utilisons les formes *tant que* et *répéter* lorsque le nombre des itérations est inconnue.
- Le nombre des itérations est égal au moins une fois pour la forme *répéter*, alors qu'il peut être nul pour la forme *tant que*.

Structure pour

La boucle *pour* est utilisée pour exécuter un bloc d'actions un certain nombre de fois connu à l'avance.

La boucle *pour* dispose de la syntaxe suivante :

pour *variable_contrôle* ***de*** *valeur_initiale* ***à*** *valeur_finale* **faire**
début
| bloc
fin

Le *bloc* est exécuté à chaque fois que la valeur de la *variable_contrôle* est comprise entre *valeur_initiale* et *valeur_finale*.

Le déroulement de la boucle *pour* peut être décrit comme suit :

1. initialiser la *variable_contrôle* avec la *valeur_initiale* ;
2. tester si la *variable_contrôle* est comprise ou non dans l'intervalle [*valeur_initiale*, *valeur_finale*] ;
3. si le test est vrai alors
   - exécuter le *bloc* d'actions de la boucle *pour* ;
   - incrémenter la *variable_contrôle* (à chaque parcours la *variable_contrôle* passe automatiquement à la valeur suivante dans son domaine) ;
   - recommencer à 2 ;
4. si le test est faux alors sortir de la boucle et exécuter l'action qui suit la fin du *bloc pour*.

Exemple 3.8 (boucle pour)

```
pour compteur de 1 à 5 faire
début
    Ecrire (compteur * 5)
fin
```

La boucle *pour* de l'Exemple 3.8 affiche les valeurs 5, 10, 15, 20, et 25.

Il est à noter que l'évolution de la *variable_contrôle* peut être décroissante en utilisant la syntaxe suivante :

```
pour variable_contrôle de valeur_finale à valeur_initiale (pas =
  valeur_décrémentation) faire
début
    bloc
fin
```

Dans ce cas, la *variable_contrôle* est décrémentée après chaque parcours.

Par ailleurs, les boucles *répéter* et *pour* n'augmentent pas le pouvoir expressif du langage algorithmique en matière de structures répétitives. La boucle *tant que* suffit pour exprimer tous les traitements répétitifs, d'ailleurs c'est la forme la plus naturelle et celle qu'on utilise systématiquement dans les algorithmes. Néanmoins, les autres formes (*répéter* et *pour*) peuvent être utilisées pour alléger l'écriture des algorithmes.

3.4 Partie données

Dans la Section 3.3, nous avons introduit les structures de contrôles qui servent à décrire l'enchaînement des actions. Ces actions manipules des objets qui contiennent des données.

Dans notre contexte, nous distinguons deux types d'objets :

- des objets qui ne peuvent pas varier lors de déroulement d'un algorithme : **Constantes** ;
- des objets qui peuvent varier durant le déroulement d'un algorithme : **Variables** (ardoises).

Dans la suite de la présente section, nous nous intéressons au concept *variable* ainsi qu'aux actions simples associées (affectation, opérations mathématiques, entrées et sorties).

3.4.1 Variables

Définition 3.2 Une variable désigne un emplacement mémoire qui permet de stocker une valeur. Elle est définie par :

- un *nom* unique qui la désigne. Le nom d'une variable doit respecter les propriétés des identificateurs annoncées dans la Section 3.2 ;
- un *type* (domaine de définition) indiquant l'ensemble de valeurs que la variable peut prendre ;
- une *valeur* attribuée et modifiée au cours de déroulement de l'algorithme.

La Définition 3.2 introduit le concept de *variable* qui est comparable à celui d'*ardoise* disposant de la possibilité de contenir une information et d'être modifiée continuellement.

3.4.2 Affectation

Définition 3.3 L'affectation consiste à attribuer (affecter) une valeur à une variable. Elle est notée par le signe $\leftarrow$.

La valeur affectée à une variable peut être :

- une constante : $X \leftarrow 0$, $nombre \leftarrow 7$;
- la valeur d'une autre variable : $X \leftarrow Y$, $Nombre \leftarrow Resultat$;
- la valeur d'une expression : $X \leftarrow Y + 1$, $Nombre \leftarrow Resultat * 2$.

L'action $X \leftarrow 0$ se lit : la variable X reçoit la valeur 0. Si X avait une valeur auparavant, cette ancienne valeur sera écrasée par la nouvelle valeur 0.

Dans l'opération d'affectation, on doit veiller à ce que la valeur ou le contenu de la variable ou le résultat de l'expression à droite du signe d'affectation doit être de même type ou de type compatible avec celui de la variable à gauche.

3.4.3 Expressions

Définition 3.4 Une expression est une suite d'opérandes (variables, constantes, fonctions) combinées par un ensemble d'opérateurs avec éventuellement un jeux de parenthèses ouvrantes et fermantes.

La Définition 3.4 est illustrée à l'aide de la Figure 3.2

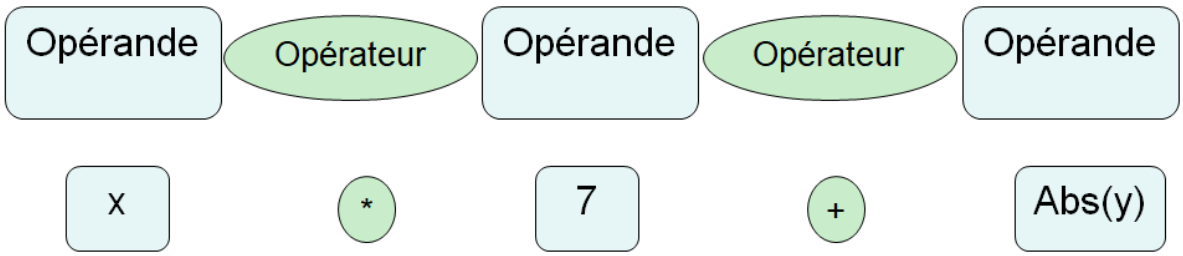


Figure 3.2 – Forme générale d'une expression.

Types des expressions

Dans le langage algorithmique et les langages de programmation, il existe plusieurs types d'expressions, à savoir, arithmétiques, logiques, relationnelles, chaînes de caractères, ensembles et mixtes.

Dans cette phase d'apprentissage d'algorithmique, nous nous limitons aux expressions arithmétiques, logiques, relationnelles et mixtes tel que montré dans la Figure 3.3. Les expressions chaînes de caractères sont traitées dans le Chapitre 4 du présent ouvrage, alors que les expressions ensembles vont être étudiées dans le Chapitre 5.

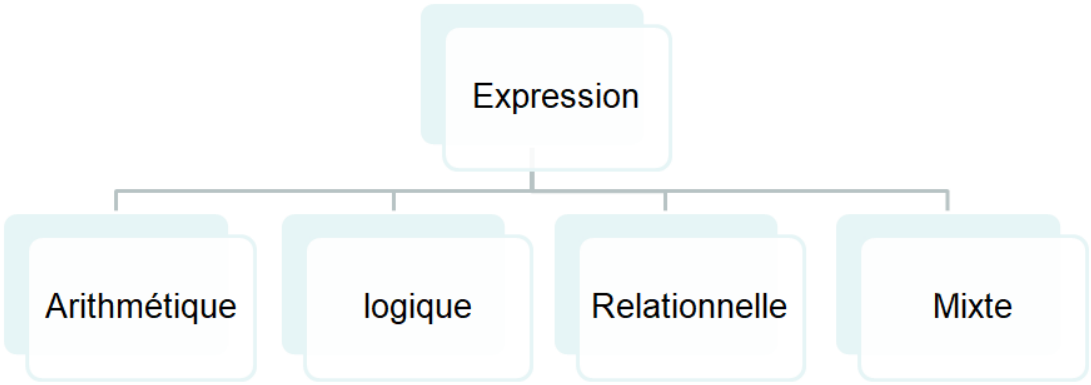


Figure 3.3 – Types des expressions.

Expressions arithmétiques

Les Tables 3.2 et 3.3 résument les types des opérandes et les résultats des opérations arithmétiques *binaires* et *unaires*. Une opération binaire utilise un opérateur qui mis en oeuvre deux opérandes. Les opérateurs à un opérande sont dits opérateurs unaires.

Table 3.2 – Opérations arithmétiques binaires

Opérateur	Opération	Types des opérandes	Type des résultats
+	addition	entier/réel	entier/réel
−	soustraction	entier/réel	entier/réel
*	multiplication	entier/réel	entier/réel
/	divison	entier/réel	réel/réel
div	divison entière	entier	entier
mod	modulo (reste)	entier	entier

Table 3.3 – Opérations arithmétiques unaires

Opérateur	Opération	Types des opérandes	Type des résultats
+	identité de signe	entier/réel	entier/réel
−	changement de signe	entier/réel	entier/réel

- Si les opérandes d'un opérateur binaire sont tous deux d'un type entier, le résultat est de type entier.
- Si au moins l'un des opérandes des opérateurs binaires $+, -, *$ est de type réel, le résultat est de type réel.
- La valeur X/Y (avec $Y \neq 0$) est toujours réel quelque soit le type des opérandes. Le contenu de Z après l'exécution de l'action $Z \leftarrow 26/4$ est la valeur réelle 6.5.
- La valeur de I **div** $J = \lfloor I$ **div** $J \rfloor$ (avec $J \neq 0$) est la partie entière inférieure. Le contenu de la variable L après l'exécution de l'action $L \leftarrow 26$**div**4 est la valeur entière 6.
- L'opérateur **mod** renvoie le reste de la division de ses deux opérandes : I **mod** $J = I - (I$ **div** $J) * J$ (avec $J \neq 0$).
- Le signe du résultat de I **mod** J est celui de I.
- Le type de résultat d'un opérateur uniare est identique à celui de l'opérande.

Expressions logiques

La Table 3.4 montre les types des opérandes et les résultats des opérations logiques.

Table 3.4 – Opérations logiques

Opérateur	Opération	Types des opérandes	Type des résultats
non	négation logique (unaire)	booléen	booléen
et	et logique (conjonction)	booléen	booléen
ou	ou logique (disjonction)	booléen	booléen

Le type des résultats des opération logiques est régit par la logique booléenne conventionnelle.

L'Exemple 3.9 suggère quelques formes d'utilisation des expressions booléennes :

Exemple 3.9
$X \leftarrow Vrai$, l'expression ici est une constante (la valeur Vrai).
$Y \leftarrow non\ X$, l'expression est une simple négation logique.
$Z \leftarrow (X\ et\ Y)\ ou\ (non\ X)$, l'expression est composée d'une conjonction, d'une disjonction et d'une négation.

Il est à préciser que dans le langage algorithmique que l'évaluation des expressions logiques se fait d'une manière complète. Autrement dit, chaque opérande des expressions logiques est évalué même si le résultat de l'expression résultante est déjà connu.

Toutefois, dans la plupart des langages de programmation, il est possible de choisir entre deux modèles d'évaluation des expressions logiques : évaluation *complète* et évaluation *optimisée*, parfois dite évaluation *court-circuit*.

Pour le modèle optimisé, l'évaluation se fait de gauche à droite et s'interrompt dès que le résultat devient évident. C'est-à-dire que l'évaluation d'un opérande ne se fait que si sa valeur est indispensable pour la connaissance du résultat de l'expression.

Expressions relationnelles

La Table 3.5 montre les types des opérandes et les résultats des opérations relationnelles.

Par type simple, nous entendons les types entretenus par les expressions arithmétiques et logiques déjà étudiées. Pour plus de détail, voir Section 3.5.2.

Expressions mixtes

Une expression mixte est une expression qui combine des opérandes quelconques avec des opérateurs quelconques.

Table 3.5 – Opérations relationnelles

Opérateur	Opération	Types des opérandes	Type des résultats
=	égal	simple	booléen
>	supérieur	simple	booléen
>=	supérieur ou égal	simple	booléen
<	inférieur	simple	booléen
<=	inférieur ou égal	simple	booléen
<>	différent	simple	booléen

Exemple 3.10
$x/y < z$ ou x div $y = 0$ ou a et non b et c mod $d > e$

L'expression de l'Exemple 3.10 est complexe, elle peut induire des confusions lors de son évaluation. Une solution à ce problème peut intervenir deux astuces :

1. Définition d'une hiérarchie (priorité) des opérateurs.
2. Utilisation des parenthèses.

Hiérarchie (priorité) des opérateurs

La hiérarchie des opérateurs est définie dans la Table 3.6.

Table 3.6 – Hiérarchie (priorité) des opérateurs

Opérateurs	Priorité	Catégorie
not	Première (haute)	opérateur unaire
*, /, div, mod, et	deuxième	opérateurs de multiplication
+, -, ou	troisième	opérateurs d'addition
=, <>, <, >, <=, >=	quatrième (basse)	opérateurs relationnelles

En plus de la priorité des opérateurs définie dans la Table 3.6, il est nécessaire de respecter les règles suivantes lors de l'évaluation d'une expression :

1. Un opérande placé entre deux opérateurs de priorités différentes sera lié à l'opérateur de priorité la plus élevée. Par exemple, dans l'expression $X * \mathbf{a} + b$ l'opérande **a** sera lié à l'opérateur $*$.
2. Un opérande placé entre deux opérateurs de même priorité sera lié à l'opérateur se trouvant à gauche. Lors de l'évaluation de l'expression $X + \mathbf{b} - c$, l'opérande **b** sera lié à l'opérateur +.

3. Les expressions contenues entre parenthèses sont évaluées séparément, puis leur résultat est traité comme un seul opérande. Pour évaluer l'expression $(X + b) * c$, on commence par l'évaluation de l'addition $X + b$, puis la multiplication $(*)$ de c par le résultat de $X + b$.

Exemple 3.11
Évaluer l'expression mixte $a + b * c / d - e * f + g - 10$

L'évaluation de l'expression de l'Exemple 3.11 s'effectue en appliquant les règles de priorités des opérateurs comme illustrer dans la Figure 3.4.

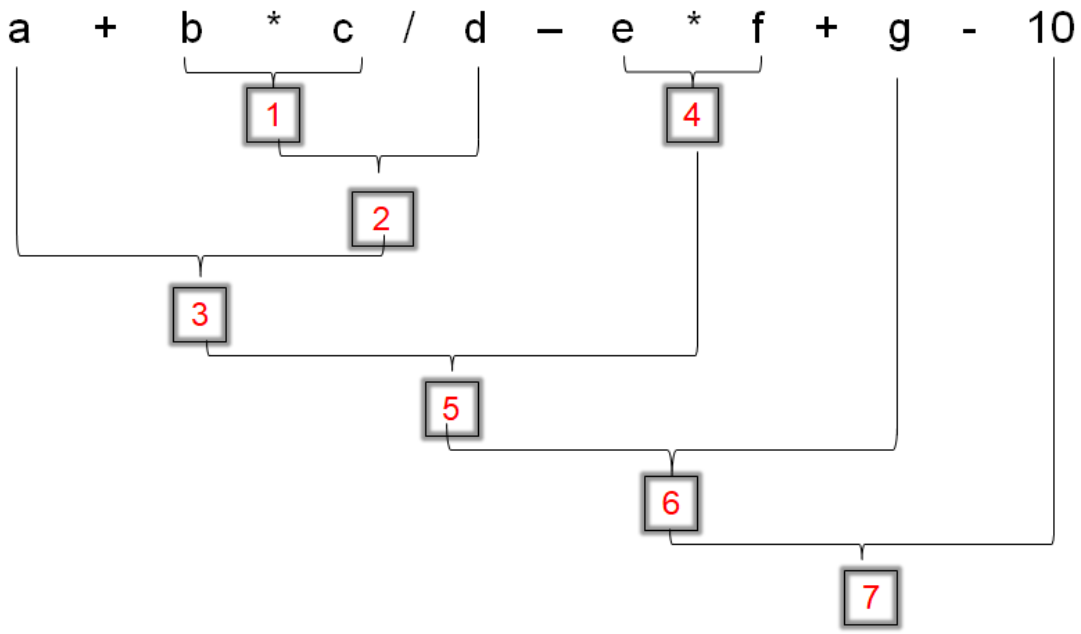


FIGURE 3.4 – Évaluation d'une expression mixte.

Utilisation des parenthèses

L'utilisation des parenthèses permet d'outrepasser les règles de priorité des opérateurs sus-citées. Les expressions entre parenthèses sont évaluées en premier et en commençant par les parenthèses les plus intérieures. De cette manière, les parenthèses peuvent changer l'ordre d'évaluation de l'expression.

De plus, les parenthèses peuvent également être utilisées pour assurer une meilleur lisibilité d'une expression. On peut utiliser un nombre excessif de parenthèses à condition de garder l'évaluation de l'expression valable, comme illustré dans l'Exemple 3.12.

Exemple 3.12
Soit l'expression mathématiques $\frac{a*b*c}{\frac{c*d+5}{f-2}+g}$.
En utilisant le formalisme algorithmique, cette expression peut être écrite

ainsi : $a * b * c/((c * d + 5)/(f - 2) + g)$.
Pour une bonne lisibilité de l'expression, on peut utiliser un nombre excessif de parenthèses : $(a * b * c)/(((c * d + 5)/(f - 2)) + g)$.

3.4.4 Action de lecture

L'action de *lecture* permet d'introduire les données, venant de l'extérieur, dans des variables. Plus précisément, cette action va chercher des valeurs sur un périphérique standard d'entrée (le clavier) et les attribue aux variables, en général appelées *paramètres* selon la syntaxe suivante :

Lire $(P1, P2, \ldots)$

L'exécution de cette action consiste à :

1. demander à l'utilisateur de saisir les valeurs sur le périphérique d'entrée (clavier) ;
2. modifier les variables (paramètres) passées entre parenthèses dans l'ordre et avec prise en compte de la compatibilité des types.

Cette action de lecture est équivalente à :

$P1 \leftarrow$ première donnée
$P2 \leftarrow$ deuxième donnée
...

Notez que l'action de lecture ne peut porter que sur des variables (conteneurs de données). *Lire* une expression ou une constante n'aurait aucun sens.

3.4.5 Action d'écriture

L'action d'*écriture* permet de restituer les résultats d'un algorithme. Plus précisément, cette action permet d'afficher sur un périphérique standard de sortie (écran) les valeurs d'une ou plusieurs expressions.
La syntaxe de l'action d'écriture est comme suit :

Écrire $(E1, E2, \ldots)$

---

les expressions E, $E2$, ... sont évaluées et les résultats sont affichés sur l'écran.
Une expression Ei peut être :

- Une variable.
- Un libellé : chaîne de caractères entre apostrophes.
- Une expression.

Exemple 3.13
Écrire ('les résultats de l''équation sont ', X1, ' et ', (-b-sqrt(delta))/2*a)).

Dans l'Exemple 3.13, le premier paramètre et le troisième sont des chaînes de caractères. Le deuxième paramètre est une variable. Alors que le quatrième paramètre est une expression arithmétique.

Si le caractère apostrophe est un caractère à afficher comme dans la chaîne *'les résultats de l'équation sont '*, il faut l'incorporer dans la chaîne en doublant les apostrophes comme suit *'les résultats de l''équation sont '*.

3.5 Environnement d'un algorithme

Tous les objets (données) manipulés dans le corps de l'algorithme et éventuellement leurs types doivent être déclarés dans la partie environnement d'un algorithme.

Dans cette section nous nous focalisons sur la déclaration des constantes, des types et des variables. La déclaration des modules sera traitée dans l'ouvrage *Algorithmique et structure de données : Partie 2*.

3.5.1 Déclaration des constantes

Nous commençons par la définition (Définition 3.5) des constantes, puis nous passons à leur déclaration.

Définition 3.5 Une constante est un objet élémentaire particulier dont la valeur ne peut pas varier durant l'exécution de l'algorithme.

Le type de la constante est sous-entendu. Il peut être un entier, un réel, un booléen, un caractère ou une chaîne de caractères.

La déclaration d'une constante se fait comme suit :

Constante nom_de_la_constante = valeur

Exemple 3.14
Constante Pi = 3.14
Constante message = 'mémoire insuffisante!'
Constante max = 100
Constante trouve = Vrai

Dans l'Exemple 3.14, nous avons utilisé le mot réservé **Constante** pour chaque déclaration. Toutefois, il est possible d'utiliser ce mot réservé une seule fois et qui porte sur toutes les constantes qui suivent (Exemple 3.15) :

Exemple 3.15
Constante Pi = 3.14
message = 'mémoire insuffisante!'
max = 100
trouve = Vrai

Notez qu'à chaque fois que la constante est rencontrée dans un algorithme, elle est remplacée par sa valeur.

3.5.2 Déclaration des types

Définition 3.6 Un type est un domaine de définition qui détermine l'ensemble des valeurs qu'un objet peut prendre ainsi qu'un ensemble d'opérateurs sur ces valeurs.

Pour introduire les types de données, nous adoptons la taxonomie du langage pascal montrée dans la Figure 3.5.

Dans cette section, nous nous intéressons aux types standards selon la classification de la Figure 3.6.

Les types structurés et non standards sont traités dans le Chapitre 5.

Avant de se plonger dans les détails, il est à retenir que tous les types simples autre que le type réel sont des types scalaires. Les valeurs possibles d'un type scalaire forment un ensemble ordonné et à chaque valeur est associée un rang. Dans tous les types scalaires, chaque élément autre que le premier possède un prédécesseur et chaque élément autre que le dernier possède un successeur.

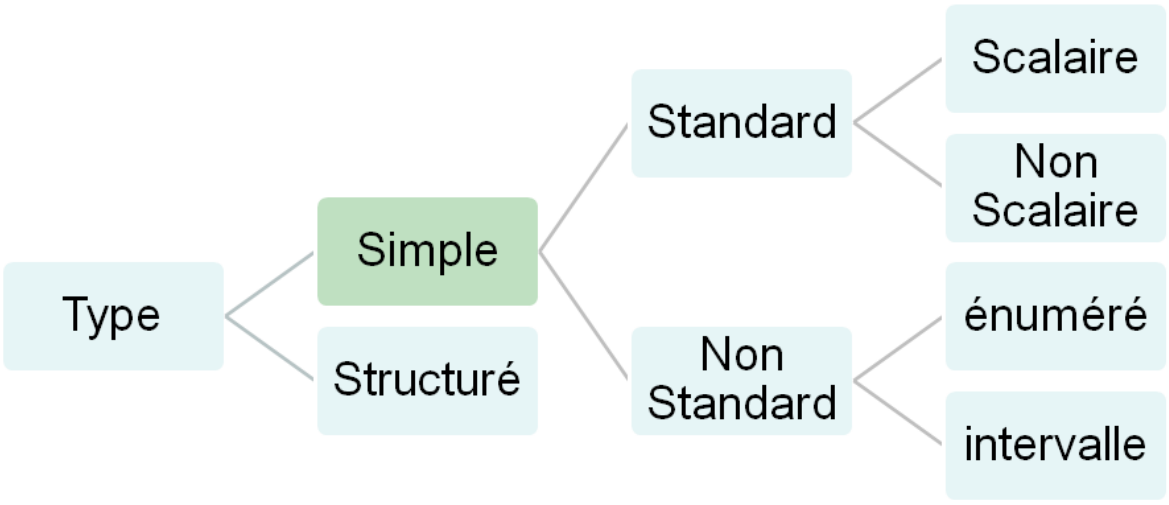


Figure 3.5 – Taxonomie des types.

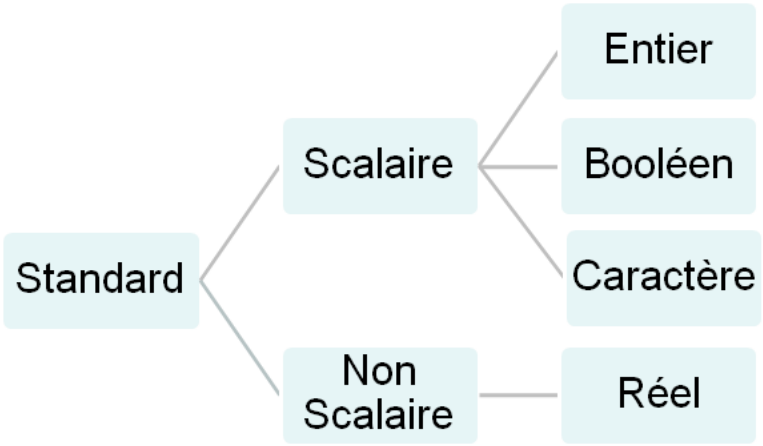


FIGURE 3.6 – Taxonomie des types standards.

Type entier

Définition 3.7 Le type *entier* est l'ensemble des entiers relatifs, cependant il faudra signaler que si en mathématiques cet ensemble est infini, sur un ordinateur les valeurs sont limitées par la longueur des mots machines.

À titre d'exemple, la Table 3.7 montre les intervalles des types *Integer* et *Word* du langage Pascal.

TABLE 3.7 – Types Integer et Word du langage Pascal.

Type	Intervalle	Format
Integer	–32768..32767	16 bits signés
Word	0..65536	16 bits non signés

Type booléen (logique)

Définition 3.8 Le type booléen constitue l'ensemble des valeurs {vrai, faux}.

Ce type n'existe pas dans tous les langages de programmation car il n'est pas aussi indispensable que les autres types.

Type caractère

Définition 3.9 Le type caractère est l'ensemble des caractères alphanumériques (alphabétiques et numériques, signes spéciaux et caractère blanc (ou espace)). Il correspond à un seul caractère.

Plus de détail pour le type caractère est discuté dans la Section 4.7.1.

Type réel

Définition 3.10 Le type réel est l'ensemble des nombres ayant une partie fractionnelle. Cet ensemble est aussi limité, mais les limites sont plus larges et dépendent de la représentation interne (en machine).

la Table 3.8 montre les intervalles des types *Real* et *Double* du langage Pascal.

TABLE 3.8 – Types Real et Double du langage Pascal.

Type	Intervalle	Taille en octet
Real	$2.9x10^{-39}..1.7x10^{38}$	6
Double	$5.0x10^{-324}..1.7x10^{308}$	8

Indications Les types standards sont des types prédéfinis, ils n'ont pas besoin d'être déclarés.

Par ailleurs, il est possible de déclarer un nouveau type selon les besoins du concepteur d'algorithmes. D'une manière générale la déclaration d'un nouveau type suit la syntaxe ci-dessous :

Type nom_de_type : **Type_base** ;

Plus de détail sur la création de types personnalisés est discuté dans le Chapitre 5.

3.5.3 Déclaration des variables

Nous rappelons que la notion de variable sert à repérer un emplacement dans la mémoire. Sa déclaration suit la syntaxe suivante :

Variable nom_de_la_variable : Type

L'Exemple 3.26 rend plus claire la déclaration des variables.

Exemple 3.16
Variable Compteur : entier
X1 : réel
lettre : caractère
trouve : booléen

lorsque plusieurs objets sont de même type, nous pouvons les regrouper dans une même et seule déclaration (Exemple 3.17).

Exemple 3.17
Variable Compteur1, compteur2 : entier
X1 , X2, X3 : réel

3.5.4 Commentaires

Définition 3.11 Un commentaire est un texte libre qui peut être étendu sur plusieurs lignes et encadré par les séquences de caractères ”/*” et ”*/”.

L'Exemple 3.18 est un commentaire.

Exemple 3.18 /* Ceci est un commentaire */

Les commentaires sont utilisés pour permettre une interprétation aisée de l'algorithme. Cette interprétation est importante à la fois pour permettre aux lecteurs de l'algorithme de comprendre l'approche proposée, mais aussi pour le concepteur de l'algorithme, au cas où il ne se souvient plus de sa démarche.

Attention, un commentaire n'est pas pris en compte à la compilation. Autrement dit, il n'a aucune influence sur l'exécution de l'algorithme.

3.6 Exemple récapitulatif

Dans cet exemple récapitulatif, nous reprenons l'écriture d'un algorithme pour la résolution d'une équation de seconde degré en respectant le formalisme algorithmique qu'on s'est mis d'accord. Cet exemple rassemble la plupart des concepts (déclaration des variables, opération de lecture et d'écriture, action alternative, affectation, expressions, commentaires, ...) que nous avons évoqué dans le présent chapitre.

```
Algorithme Equation_2_degre
  Variable a, b, c, delta, x1, x2 : réel                    /* partie déclaration */


  début
  |                                                  /* Début corps de l'algorithme */
  |  Écrire ('saisissez les paramètres a, b et c de l"équation ax²+bx+c=0 : ')
  |  Lire (a, b, c)                                  /* lecture des paramètres */
  |  si a=0 alors
  |  début
  |  |  Écrire ('équation du premier degré')
  |  fin
  |  sinon
  |  début
  |  |  delta ← b*b-4*a*c
  |  |  si delta > 0 alors
  |  |  début
  |  |  |  x1 ← (-b-sqrt(delta))/(2*a)
  |  |  |  x2 ← (-b+sqrt(delta))/(2*a)
  |  |  |  Écrire ('les racines de l"équation sont : ', x1, x2)
  |  |  fin
  |  |  sinon
  |  |  début
  |  |  |                                                        /* delta <= 0 */
  |  |  |  si delta = 0 alors
  |  |  |  début
  |  |  |  |  x1 ← -b/(2*a)
  |  |  |  |  Écrire ('la racine double de l"équation est : ', x1)
  |  |  |  fin
  |  |  |  sinon
  |  |  |  début
  |  |  |  |                                                     /* delta < 0 */
  |  |  |  |  Écrire ('Pas de racines réelles de l"équation ')
  |  |  |  fin
  |  |  fin
  |  fin
  |                                                  /* Fin de l'algorithme */
  fin
```

3.7 Coin langage Pascal

La section coin langage n'est pas un substitut d'un cours complet sur les langages de programmation, mais juste un résumé des concepts déjà introduits pour le langage algorithmique. Ce résumé permet à l'étudiant d'être capable de traduire facilement en langage de programmation les algorithmes conçus.

Les ouvrages [Borland, 1991a, Borland, 1991b, Delannoy, 2007] couvrent d'une manière quasi totale tous les ingrédients du langage de programmation Turbo Pascal 7.0.

3.7.1 Structure d'un programme Pascal

Le Listing 3.1 montre l'anatomie d'un programme Pascal.

```
1  PROGRAM ProgramName;
2
3  CONST
4      (* déclarations globales des Constantes *)
5
6  TYPE
7      (* déclarations globales des Types *)
8
9  VAR
10     (* déclarations globales des Variables  *)
11
12     (* définitions des sous programmes *)
13
14 BEGIN
15     (* corps du programme *)
16 END.
```

Listing 3.1 – Anatomie d'un programme Pascal

Notez que le nom d'un programme source en Pascal se termine par une extension *PAS* (exemple : bonjour.PAS). Cependant, celui de l'exécutable se termine par *EXE* (exemple : bonjour.EXE).

3.7.2 Mots réservés de Turbo Pascal

Par convention, les mots réservés en Pascal s'écrivent en majuscule. Cependant, le compilateur Turbo Pascal ne distingue pas entre majuscule et minuscule.

- AND - ARRAY - ASM
- BEGIN
- CASE - CONST - CONSTRUCTOR
- DESTRUCTOR - DIV - DO - DOWNTO
- ELSE - END - EXPORTS
- FILE - FOR - FUNCTION

- GOTO
- IF - IMPLEMENTATION - IN - INHERITED - INLINE - INTERFACE
- LABEL - LIBRARY
- MOD
- NIL - NOT
- OBJECT - OF - OR
- PACKED - PROCEDURE - PROGRAM
- RECORD - REPEAT
- SET - SHL - SHR - STRING
- THEN - TO - TYPE
- UNIT - UNTIL - USES
- VAR
- WHILE - WITH
- XOR

3.7.3 Identificateurs

Un identificateur doit avoir au maximum 63 caractères et doit obligatoirement commencer par une lettre de l'alphabet. La liste des caractères acceptés est :

- a à z
- A à Z
- 0 à 9
- _

Les identificateurs doivent être impérativement différents des mots réservés et des noms de procédures et fonctions définis par le langage Pascal.

3.7.4 Commentaires

Les commentaires sont mis entre accolades { et }, ou entre parenthèses et astérisques *(** et ***)*. On ne doit pas mélanger entre les deux symboles.

3.7.5 Déclarations

La section déclaration est réservée à la déclaration des types, des constantes et des variables. Chaque déclaration se termine par un point-virgule.

Déclaration des types

Dans Turbo Pascal, il existe des types prédéfinis. Ils n'ont pas besoin d'être déclarés :

- Types entiers : *Shortint*, *Integer*, *Longint*, *Byte* et *Word*.
- Types booléens : *Boolean*, *Bytebool*, *WordBool* et *LongBool*.
- Type caractère : *Char*.
- Types réels : *Real*, *Single*, *Double*, *Extended*, *Comp*.

La déclaration des types personnalisés est traitée dans le Chapitre 5.

Déclaration des constantes

Nous distinguons deux sortes de constantes : les constantes non typées et les constantes typées.

La déclaration des *constantes non typées* se fait comme suit :

```
Const identificateur = constante;
```

Voici quelques exemples de déclarations des constantes non typées (Exemple 3.19) :

Exemple 3.19

```
Const   min = 0;
        max = 1000;
        milieu = (max - min) div 2;
        message = 'insufficient memory';
        nombredesemaines = 365 div 7;
        lettrealpha = chr(224);
```

Les *constantes typées* définissent le type et la valeur. De cette manière, ils sont comparables à des variables initialisées. Leurs déclaration est comme suit :

```
Const identificateur : Type = constante;
```

Les exemples suivants montrent quelques déclaration de constantes typées (Exemple 3.20) :

Exemple 3.20

```
Const   min : Integer = 0;
        max : Integer = 1000;
        pi : Real = 3.1416;
        message :String [18]= 'insufficient memory';
        lettrealpha : Char = chr(224);
```

Déclaration des variables

La synatxe de déclaration des variables est comme suit :

```
Var identificateur1, identificateur2, ... : Type;
```

Voici quelques exemples de déclaration des variables (Exemple 3.21) :

Exemple 3.21

```
Var X, Y, Z : Real;
    i, j, k  : Integer;
    Fin, correcte : Boolean;
```

3.7.6 Instructions

Dans cette section, nous examinons les instructions suivantes : affectation, If, While, Repeat et For.

Affectation

Le signe `:=` est utilisé pour exprimer une instruction d'affectation.
Voici quelques exemples (Exemple 3.22) :

Exemple 3.22

```
Var X, Y, Z : Real;
    i, j, k  : Integer;
    Fin, correcte : Boolean;
Begin
    ...
    X := Y + Z;
    i := j - k + chiffre;
```

```
8           Fin := not correcte;
9           correcte := false;
10          ...
11      End.
```

Instruction if

L'instruction *if* prend deux formes :

```
1       if condition then
2           bloc 1;
```

```
1       if condition then
2           bloc 1
3       else
4           bloc 2;
```

condition est une expression booléenne simple ou composée. *bloc 1* et *bloc 2* sont des blocs d'instructions.

Les blocs d'instructions, s'ils comportent plus d'une instruction doivent être encadrés par les mots réservés **Begin** et **End**.

Lors de l'utilisation de l'instruction *if* il faut respecter les règles suivantes :

1. Il n'est pas permis de placer un point-virgule avant la clause **else**.
2. La clause **else** se rapporte toujours au **if** le plus proche qui ne soit déjà associé à un autre **else**.

Instruction while

L'instruction *while* doit respecter la syntaxe suivante :

```
1       while condition do
2           bloc;
```

condition est une expression booléenne simple ou composée. *bloc* est un ensemble d'instructions.

Les variables de *condition* doivent être initialisées avant l'instruction *while*, pour qu'au premier passage, *condition* puisse être évaluée.

Les deux Exemples (3.23 et 3.24) suivants illustrent l'utilisation de la boucle *while* :

Exemple 3.23

```
1
```

```
    while i <> j do
        i := i + 1;
```

Exemple 3.24

```

    while i < 100 do
    Begin
        write(i);
        i := i + 1
    End;
```

Instruction repeat

La syntaxe de l'instruction *repeat* est :

```
    repeat
        bloc
    until condition;
```

condition est une expression booléenne simple ou composée. *bloc* est un ensemble d'instructions.

Il est à noter que *bloc* n'est pas encadré par les mots réservés **Begin** et **End**. Ils sont implicitement remplacés par **repeat** et **until**.

Instruction for

La syntaxe de l'instruction *for* est :

```
    for v_controle := e_initiale to e_finale do
        bloc;
```

v_controle est la variable de contrôle. Elle doit être d'un type scalaire. *e_initiale* et *e_finale* sont des expressions initiales et finales. Elles doivent être compatibles avec ce type scalaire.

e_initiale et *e_finale* sont calculées une seule fois, à l'entrée dans l'instruction *for*.

Avec le mot réservé **to**, la variable de contrôle est incrémentée de 1 à chaque répétition. Si *e_initiale* et supérieure à *e_finale*, l'instruction *for* n'est pas exécutée.

Notez qu'il existe une autre forme de l'instruction for :

```
    for v_controle := e_finale downto e_initiale    do
        bloc;
```

Avec le mot réservé **downto**, la variable de contrôle est décrémentée de 1 à chaque répétition. Si *e_finale* et inférieure à *e_initiale*, l'instruction *for* n'est pas exécutée.

Instruction de lecture

La syntaxe de lecture au clavier se réalise selon plusieurs formes :

```
1  read(P1, P2, ...);
2
3  readln(P1, P2, ...);
4
5  readln;
```

où $P1, P2, \ldots$ constituent une liste de variables pouvant avoir différents types : *integer*, *longint*, *real*, *char*, *string*, *array*[] of *char* et éventuellement d'autres types que nous n'avons pas encore vu.

Instruction d'écriture

La syntaxe d'affichage à l'écran se réalise selon plusieurs formes :

```
1  write(E1, E2, ...);
2
3  writeln(E1, E2, ...);
4
5  writeln;
```

où $E1, E2, \ldots$ constituent une liste d'expressions pouvant avoir différents types : *integer*, *longint*, *real*, *char*, *boolean*, constante chaîne, *string*, *array*[] of *char* et éventuellement d'autres types que nous n'avons pas encore vu.

Par ailleurs, il est possible d'imposer un format d'affichage pour une expression en utilisant la notation *Ei :g* (g est une expression entière). Ceci demande l'affichage de l'expression *Ei* sur g caractères.

De plus, si *Ei* est de type réel, nous pouvons utiliser le gabarit *Ei :g :d* (g et d, étant des expressions entières) pour imposer un affichage en *point fixe* sur un total de g caractères avec d caractères après le point décimal.

3.8 Coin langage C

La présente section présente grossièrement les éléments de base du langage C. Pour plus de détail, je recommande les deux ouvrages, qui traitent le langage C, d'une manière pédagogique [Delannoy, 2009, Kernighan and Ritchie, 1978].

3.8.1 Structure d'un programme C

Le listing 3.2 montre la structure d'un programme C.

```
1   // directives au préprocesseur qui commencent par #
2
3   // déclaration des variables externes
4
5   // déclaration des fonctions secondaires
6
7   main ( )
8   {
9
10      // corps du programme
11
12  }
```

Listing 3.2 – Anatomie d'un programme C

Les directives au préprocesseur permettent d'introduire (avant compilation) les fonctions déjà crées. Elles permettent aussi, les déclaration d'équivalences sur des constantes.

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale.

La fonction *main()* constitue le programme principal. C'est la partie obligatoire dans un programme C.

3.8.2 Mot réservés de l'ANSI C

l'ANSI (American National Standards Institute) C contient 32 mots réservés (mots clefs) :

- auto
- break
- const, continue, case, char
- double, default, do
- else, enum, extern
- float, for
- goto
- int, if
- long
- register, return
- short, struct, signed, switch, sizeof, static

- typedef
- unsigned, union
- void, volatile
- while

3.8.3 Identificateurs

Un identificateur a pour rôle de donner un nom à une entité du programme (variable, fonction, type défini par *typedef*, structure, union, enum, étiquette).

Comme dans la plupart des langages de programmation, un identificateur est une suite de caractères alphanumériques dont le premier caractère doit être alphabétique. Le blanc souligné `_` est considéré comme une lettre. Ainsi, il peut apparaître en début d'un identificateur.

Contrairement au langage Pascal, les lettres majuscules et minuscules sont distinctes.

Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur. Mais les compilateurs modernes peuvent reconnaître au moins 31 caractères.

Il est recommandé d'utiliser des identificateurs entièrement en majuscules pour les variables du préprocesseur.

3.8.4 Commentaires

En C, les commentaires sont mis entre `/*` et `*/`. Ils peuvent apparaître n'importe où dans le programme pour expliquer brièvement que fait-il. Ils ne doivent pas être imbriqués.

3.8.5 Déclarations

Déclaration des types

Il existe un nombre réduit de types prédéfinis en langage C (Table 3.9) :

Table 3.9 – Types prédéfinis en langage C.

Type	**description**
char	Caractère
int	entier
float	réel simple précision
double	réel double précision

Les qualificatifs *short* et *long* permettent d'influencer la taille mémoire des entiers et réels.

Les qualificatifs *signed* et *unsigned* peuvent s'appliquer aux types caractères et entiers pour indiquer si le bit de signe du poids fort doit être considéré ou non comme un bit de signe.

Ceci étant dit, la déclaration des types personnalisés est traitée dans le Chapitre 5.

Déclaration des constantes

Une constante peut être déclarée sous la forme :

```
const Type Identificateur = Valeur;
```

Elle peut aussi être définie en utilisant la directive *#define* :

```
#define Identifcateur  Valeur
```

La directive *#define* demande au préprocesseur de substituer toute occurrence de *Identificateur* par *Valeur* dans la suite du fichier source.

L'Exemple 3.25 illustre les deux moyens pour définir une constante.

Exemple 3.25

```
#define pi 3.14
main()
{
    const float pii = 3.14;
    ...
}
```

Déclaration des variables

La clause de déclaration d'une variable spécifie son identificateur et son type.

```
Type Identificateur1, Identificateur2, ...,
Identificateurn;
```

En langage C, contrairement au langage Pascal, il n'existe pas une partie dédiée à la déclaration des variables. Toutefois, toute variable doit faire l'objet d'une déclaration avant d'être utilisée. Par ailleurs, une valeur initiale peut être affectée à une variable lors de sa déclaration.

L'Exemple 3.26 présente une diversité de déclaration de variables :

Exemple 3.26

```
#include <stdio.h>
```

```
    unsigned char c;
    long v;
    main()
    {
        unsigned int i= 1, j;
        short size;
        unsigned long tmp;
        double x = 2.33e5, y = 3.01, z;
        char car = 'B';
        char sep = '\'';  /* sep reçoit un apostrophe*/
        char c1 = '\x041';/* c1 reçoit 41 comme code
   ASCII en hexadécimal de A */
        ...
    }
```

3.8.6 Instructions

Dans cette section, nous examinons les instructions suivantes : affectation, *if*, *while*, *repeat* et *for*.

Affectation

L'affectation s'effectue en utilisant le signe `=` selon la syntaxe suivante :

```
variable = expression;
```

Voici quelques exemples (Exemple 3.27) :

Exemple 3.27

```
main()
{
    int i,j, k;
    i = 4;        /* i reçoit 4 */
    j = (i*3)+1; /* j reçoit 13 résultat de (4*3)+1 */
    k = j = 9;   /* j reçoit d'abord 9 puis k reçoit la
   valeur de j (9) */
}
```

L'affectation en langage C peut effectuer une conversion de type implicite, la valeur de l'expression du terme de droite est convertie dans le type du terme de gauche tel que illustré par l'Exemple 3.28.

Exemple 3.28

```
main()
{
    int i;          /* i entier */
    float x = 2.5; /* x réel initialisée à 2.5 */
    i = x;          /* i reçoit 2, le résultat est tronqué
   */
    char c ='B';    /* c caractère initialiséé à 'B' */
    i = c;          /* i reçoit 66 le code ASCII de 'B' */
}
```

Instructions conditionelles

Il existe deux formes d'instructions conditionnelles, *if-else* et *else-if*.

La forme *if-else* est utilisée pour exprimer des décisions selon la syntaxe suivante :

```
if (condition)
    bloc 1
else
    bloc 2
```

La partie *else* est optionnelle et chaque bloc peut comporter une ou plusieurs instructions.

Exemple 3.29

```
if (n < 0)
    if (a == b)
        a = - a;
    else
        b = - b;
```

Dans l'Exemple 3.29, le premier *if* est sans *else* et le *else* se rapporte au deuxième *if*. La règle générale stipule que *else* se rapporte toujours à *if* le plus interne.

La forme *else-if* est utilisée pour exprimer des décisions multi directionnelles selon la syntaxe suivante :

```
if (condition 1)
    bloc 1
else if (condition 2)
    bloc 2
else if (condition 3)
```

```
        bloc 3
    ...

    else
        bloc n
```

Les conditions sont évaluées dans l'ordre. Si n'importe qu'elle condition est vérifiée alors le bloc associé est exécuté. Dans le cas où aucune condition n'est satisfaite alors le dernier bloc associé à *else* est exécuté. Sachant que le dernier *else* est optionnel.

L'exemple de la recherche binaire illustre l'utilisation de la forme *else-if* (Exemple 3.30).

Exemple 3.30

```
/* chercher x dans un tableau trié v de taille n */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;
    low = 0;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x < v[mid])
            high = mid -1;
        else if (x > v[mid])
            low = mid +1;
        else
            return mid; /* x existe dans v */
    }
    return -1; /* x n'existe pas dans v */
}
```

Instructions répétitives

Dans cette section, nous présentons les instructions répétitives *while*, *for* et *do-while* ainsi que les instructions de sortie de boucles *break* et *continue*.

- **Instruction while**

  syntaxe

```
while (condition)
    bloc
```

Exemple 3.31

```
int n = 10, s = 0;
while (n > 0)
{
      s = s + n;
      n--;
}
```

- **Instruction for**

  syntaxe

```
for (initialisations ; condition ; finalisations)
    bloc
```

Cette structure commence par les instructions *initialisations* qui seront exécutées une seule fois, au début de l'exécution de la boucle.

La *condition* est évaluée et testée avant d'entrer dans le *bloc* de la boucle.

Les instructions *finalisations* sont exécutées à la fin de chaque itération.

La structure *for* est équivalente à :

```
initialisations
while (condition)
{
    bloc
    finalisations
}
```

Exemple 3.32

```
int n, i, fact;
...
for (i=1, fact=1 ; i <= n ; i++)
      fact *= i;
 ...
```

- **Instruction do-while**

  syntaxe

```
do
    bloc
while (condition);
```

Exemple 3.33

```
int n = 10, s = 0;
 do
{
      s = s + n;
      n--;
}
while (n = 0);
```

Notez que la forme *do-while* du langage C est comparable à *repeat-until* du langage Pascal.

- **Instructions break et continue**

  L'instruction *break* fournit une sortie anticipée de *while*, *do-while* et *for*.

  Tandis que l'instruction *continue* permet de passer directement à l'itération suivante de la boucle encours. Dans les boucles *while* et *do-while* le contrôle passe, immédiatement après *continue*, à la *condition* qui sera évaluée et testée. Alors que dans la boucle *for*, le contrôle passe aux instructions *finalisations*.

Exemple 3.34

```
int i;
for (i = 0 ; i < 10 ; i++)
{
      ...
      if (a[i] == 0) /*ignorer les éléments nuls*/
          continue;
      printf("encore");
      ...
      if (a[i] < 0) /*sortir si éléments négatif*/
          break;
      ...
}
```

Instruction d'écriture

Le langage C utilise la fonction *printf* pour afficher des informations à l'écran :

```
1   int printf(format, E1, E2, ...);
```

format est une chaîne de caractères qui peut contenir :

- texte à afficher,
- spécificateurs de conversion.

Les spécificateurs de conversion indiquent le format d'affichage des expressions *Ei*. Ces spécificateurs commencent toujours par `%` et se terminent par un ou deux caractères de conversion. La fonction *printf* renvoie le nombre de caractère affichés.
Entre le signe `%` et le caractère de conversion, il peut y avoir, dans l'ordre :

- Le signe moins `-` qui spécifie un alignement à gauche de l'expression correspondante.
- Un nombre qui spécifie la largeur minimale du champ.
- Un point `.` qui sépare la largeur du champ de la précision.
- Un nombre, une précision qui spécifie le nombre maximum de caractères, à imprimer, d'une chaîne de caractères, ou le nombre de chiffres après le point décimal dans une valeur flottante, ou le nombre minimum de chiffres d'un entier.
- un *h* si l'entier à imprimer sous format *short*, ou *l* sous format *long*.

La Table 3.10 présente les spécificateurs de conversion de base.

Table 3.10 – Spécificateurs de conversion de base de *printf*

Format	Type expression	Type d'affichage
%d, %i	int	décimal signé
%o	unsigned int	octale non signée
%x, %X	unsigned int	héxadécimal non signé
%u	unsigned int	décimal non signé
%c	unsigned char	un caractère
%s	char *	chaîne de caractères
%f	double	décimal virgule fixe
%e, %E	double	décimal notation exponentielle
%g, %G	double	décimal, représentation la plus courte parmi %f et %e

L'Exemple 3.35 montre différentes utilisations de la fonction *printf*.

Exemple 3.35

```c
    int i = 123, j = 45;
    printf("%i x %i = %li\n", i, j, (long)i*j);
    /* affiche 123 x 45 = 5535 */
    char c = 'B';

    printf("Le caractère %c a le code %i\n", c,c);
    /* affiche Le caractère B a le code 66 */

    long N = 15000000;
    printf("%d, %d", N);
    /* affiche -7328, 35 */
    /* La première valeur est tronquée, la seconde est
   relativement aléatoire */

    printf("%ld, %lx",N, N)
    /* affiche 1500000, 16e360 */

    float X=12.1234;
    double Y = 12.123456789;
    long double Z = 15.5;
    printf("%f", X); /* affiche 12.123400 */
    printf("%f", Y); /* affiche 12.123456 */
    printf("%e", X); /* affiche 1.212340e+01 */
    printf("%e", Y); /* affiche 1.212346e+01 */
    printf("%le", Z); /* affiche 1.550000e+01 */

    printf("%f", 123.456); /* affiche 123.456000 */
    printf("%12f", 123.456); /* affiche __123.456000 */
    printf("%.2f", 123.456); /* affiche 123.45 */
    printf("%5.0f", 123.456); /* affiche __123 */
    printf("%10.3f", 123.456); /* affiche ___123.456 */

    char *chaine ="algorithmique et SDD";
    printf("%s \t %10s", chaine, chaine);
    /* affiche algorithmique et SDD     algorithmi */
```

De plus, la fonction *sprintf* effectue les mêmes conversions que *printf*, mais elle sauvegarde les résultats dans une variable chaîne de caractères :

```c
    sprintf(output,format, E1, E2, ...);
```

avec *output* une variable chaîne de caractères qui enregistre le résultat.

Instruction de lecture

La fonction *scanf* est l'analogue d'entrée de *printf* fournissant les options de conversion dans le sens inverse :

```
1    scanf(format, P1, P2, ...);
```

La fonction *scanf* lit les caractères à partir de l'entrée standard, les interprète selon les spécificateurs de conversion dans *format* et sauvegarde les données d'entrée converties dans les paramètres *Pi* dont les noms doivent être précédé par le signe `&` .

scanf s'arrête quand elle épuise la chaîne de caractère *format* ou lorsque une entrée ne correspond pas au spécificateur de conversion. Elle retourne comme résultat le nombre de paramètres correctement reçus et affectés.

La Table 3.11 résume les spécificateurs de conversion de base.

Table 3.11 – Spécificateurs de conversion de base de *scanf*

Format	**Type du paramètre**	**Type de données**
%d	int*	décimal signé
%o	int*	octale
%u	unsigned int*	décimal non signé
%x	int*	héxadécimal
%c	char*	un caractère
%s	char*	chaîne de caractères
%f	float*	flottant virgule fixe
%e	float*	flottant virgule exponentielle
%g	float*	flottant virgule fixe ou virgule exponentielle

L'Exemple 3.36 illustre certains utilisations de la fonctions *scanf*.

Exemple 3.36

```
int i;
float x;
char c;

scanf("%d", &i);
/* lit une valeur entière et l'affecte à i  */

scanf("%d%f", &i, &x);
/* lit une valeur entière de i et réelle de x */

scanf("%c", &c)
/* lit un caractère et l'affecte à la variable c */
```

Finalement, notez bien que l'expression *c = getchar()* joue le même rôle que l'appel de la fonction *scanf("%c", c)*, mais elle est plus rapide.

3.8.7 Opérateurs

La présente section s'intéresse aux différents opérateurs du langage C, à savoir : arithmétiques, relationnels, logiques booléens, logiques bit à bit, affectation composée, incrémentation, décrémentation, séquentiel, conditionnel et cast.

Opérateurs arithmétiques

Les opérateurs arithmétiques sont présentés par la Table 3.12. la division entière tronque toute partie fractionnaire. L'opérateur `%` ne peut pas être appliqué sur des données de type *float* ou *double*.

Table 3.12 – Opérateurs arithmétiques

Opérateur	Opération
+	addition
–	soustraction
*	multiplication
/	divison
%	modulo (reste de la division entière)

Opérateurs relationnels

Les opérateurs relationnels sont présentés par la Table 3.13.

Table 3.13 – Opérateurs relationnels

Opérateur	Opération
>	strictement supérieur
>=	supérieur ou égal
<	strictement inférieur
<=	inférieur ou égal
==	égal
! =	différent

Opérateurs logiques booléens

La Table 3.14 résume les opérateurs logiques booléens. Les expressions connectées par les opérateurs `&&` et `||` sont évaluées de gauche à droite et l'évaluation s'arrête dès que la vérité ou la fausseté du résultat est connue.

Table 3.14 – Opérateurs logiques booléens

Opérateur	Opération
&&	et logique
||	ou logique
!	négation logique

Opérateurs logiques bit à bit

Le langage C fournit six opérateurs (Table 3.15) pour la manipulation des bits. Ils s'appliquent uniquement à des opérandes de type *char*, *short*, *int* et *long*, qu'ils soient signés (*signed*) ou non signés (*unsigned*).

Opérateurs d'affectation composée

La plupart des opérateurs binaires disposent d'un opérateur d'affectation composée correspondant *op*=, où *op* est l'opérateur binaire. La Table 3.16 présente les opérateurs d'affectation composée ainsi que leurs significations.

Table 3.15 – Opérateurs logiques bit à bit

Opérateur	Opération
&	et binaire
|	ou inclusif binaire
∧	ou exclusif binaire
<<	décalage à gauche
>>	décalage à droite
~	complément à 1

Table 3.16 – Opérateurs d'affectation composée

Opérateur	Écriture	Équivalence
+ =	x += y	x = x + y
− =	x -= y	x = x - y
* =	x *= y	x = x * y
/ =	x /= y	x = x / y
% =	x %= y	x = x % y
& =	x &= y	x = x & y
∧ =	x ∧ = y	x = x ∧y
| =	x | = y	x = x |y
≪=	x ≪= y	x = x ≪ y
≫=	x ≫= y	x = x ≫ y

Opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation et de décrémentation (Table 3.17) s'utilisent en notation suffixe ($x++$, $x--$) ou préfixe ($++x$,$--x$). Dans la notation suffixe, c'est l'ancienne valeur de x qui sera retournée, alors que dans la notation préfixe, c'est la nouvelle valeur, tel que illustré dans l'Exemple 3.37.

Table 3.17 – Opérateurs d'incrémentation et de décrémentation

Opérateur	Écriture	Équivalence
++	x++	x = x + 1
−−	x−−	x = x - 1

Exemple 3.37

```
int x, y, z = 2;
x = ++z; /* x vaut 3 */
z = 2;
y = z++; /* y vaut 2 */
```

Opérateur séquentiel (virgule)

L'opérateur séquentiel `,` (virgule) permet de regrouper des expressions successives au sein d'une même expression. Ces expressions sont évaluées en séquence de gauche à droite.

Exemple 3.38

```
x = ((y=3), (y+2)); /* x vaut 5 */
```

Opérateur conditionnel

L'opérateur conditionnel ternaire `? :` fournit une alternative pour exprimer une instruction conditionnelle.

syntaxe

```
condition ? expression_1 : expression_2;
```

Si la *condition* est satisfaite, *expression_1* est évaluée et son résultat devient la valeur de l'expression conditionnelle. Sinon, *expression_2* est évaluée et ça sera la valeur de l'expression conditionnelle.

Exemple 3.39

```
z = (x > y) ? x : y; /* z = max(x, y) */
```

Opérateur cast

Il est possible de forcer le type d'une expression en utilisant l'opérateur de conversion, appelé *cast*.

syntaxe

```
(type) expression;
```

Exemple 3.40

```
1    int n = 10, p = 3, x;
2    float y;
3    x = n/p; /* x vaut 3 */
4    y = (float) n/p /* y vaut 3.33333... */
```

Règles de priorité des opérateurs

La Table 3.18 résume les règles de priorité et d'associativité de tous les opérateurs du langage C y compris ceux que nous n'avons pas présenté dans la présente section (Une partie des opérateurs non présentés ici seront étudiés ultérieurement dans les chapitres correspondant aux tableaux et aux types personnalisés. Le reste sera étudié dans le deuxième ouvrage *Algorithmique et structures de données - Partie 2*).

Les opérateurs dans la même ligne ont la même priorité. Alors que les ligne sont en ordre décroissant de priorité.

Table 3.18 – Priorité et associativité des opérateurs

Catégorie	**Opérateur**	**Associativité**
référence	() [] –>	gauche à droite
unaire	+ - ++ – ! ~ * & (*cast*) *sizeof*	droite à gauche
arithmétique	* / %	gauche à droite
arithmétique	+ -	gauche à droite
décalage	≪ ≫	gauche à droite
relationnels	< <= > >=	gauche à droite
relationnels	== !=	gauche à droite
manipulation bits	&	gauche à droite
manipulation bits	∧	gauche à droite
manipulation bits	|	gauche à droite
logique	&&	gauche à droite
logique	||	gauche à droite
conditionnel	? :	gauche à droite
affectation	= += -= *= /= &= ∧= |= ≪= ≫=	droite à gauche
séquentiel	,	gauche à droite

3.9 Exercices

Les objectifs pédagogiques de cette série d'exercices sont :

- Analyse d'un problème (Construction d'une analyse)
- Conception des algorithmes
  - Passage d'une analyse à un algorithme.
  - Manipulation des objets (affectation + expressions).
  - Maitrise des structures de contrôle.
  - Déroulement d'un algorithme.

Nous avons volontairement inclus un nombre intéressant d'exercices à cette série, car nous jugeons qu'il est très important pour un concepteur d'algorithmes de maîtriser le formalisme algorithmique parfaitement afin qu'il soit capable, ultérieurement, de proposer des algorithmes efficaces, concis et élégants.

Exercise 3.2 Pour quelles valeurs des variables a et b cette expression renvoie vrai ?
$((a * b >= 15)\ et\ (\ non\ (a < 4)\ ou\ (b >= 4\ et\ b <= 7)))$

1. a = 5, b = 2
2. a = 6, b = 2
3. a = 5, b = 10
4. a = 9, b = 2

Indications Il est important de savoir évaluer des expressions mixtes. Elles vous aident à dérouler correctement vos algorithmes et d'apprendre à formuler des conditions. Tous simplement, il faut respecter les règles de priorité des opérateurs et l'utilisation des parenthèses.

Exercise 3.3 Comment écrire une condition **Si** qui renvoie vrai si la variable i est dans l'intervalle $[1, 10[\cup]20, 30]$?

1. Si ((i >= 1 et i < 10) et (i > 20 et i <= 30))
2. Si ((i >= 10 et i < 20) ou (i > 20 et i <= 30))
3. Si ((i >= 1 et i < 10) ou (i > 20 et i <= 30))
4. Si ((i>= 1 ou i < 10) ou (i > 20 et i <= 30))

Indications Essayez de concevoir la condition par votre propre soin, ensuite comparez la avec la liste des choix.

Exercise 3.4 Soit le code suivant, qu'affiche t-il ?

```
Algorithme test
Variable test1, test2 : booléen
début
    test1 ← faux
    test2 ← (vrai ou test1)
    Écrire ('Sol ')
    si (test2) alors
    début
        Écrire('Do ')
    fin
    sinon
    début
        si (test1 et test2) alors
            Écrire('Mi ')
        sinon
            Écrire('Fa ')
    fin
    si ((test1 et test2) ou (test1 ou Non(test2))) alors
        Écrire('La ')
    sinon
        Écrire('Do ')
fin
```

Indications On vous demande de dérouler (faire la trace) de l'algorithme. Travaillez le déroulement avec soin, car le déroulement d'un algorithme est un moyen très pédagogique pour maîtriser l'algorithmique.

Exercise 3.5 Soit le code suivant, qu'affiche t-il ?

```
Algorithme SISINON
  Variable x, y, z : entier

  début
    x ← 2
    y ← 5
    z ← 10
    si (x * y = z) alors
    début
      Écrire('1 ')
      x ← x+3
      z ← z-2
    fin
    sinon
    début
      Écrire('2 ')
    fin
    si (z MOD 2 = 0) alors
    début
      si (x + z = 10) alors
      début
        Écrire('3 ')
      fin
      sinon
      début
        Écrire('4 ')
      fin
    fin
    sinon
    début
      Écrire('5 ')
    fin
  fin
```

Indications On vous demande de dérouler (faire la trace) de l'algorithme. Travaillez le déroulement avec soin, car le déroulement d'un algorithme est un moyen très pédagogique pour maîtriser l'algorithmique.

Exercise 3.6 Qu'obtiendra-t-on dans les variables $a, b, n1$ *et* $n2$ après exécution des blocs d'actions de la Table 3.19 ?

Indications Après la résolution de l'exercice, essayez de réfléchir à ne garder que les actions indispensables.

Table 3.19 – Blocs d'actions.

Bloc A	Bloc B	Bloc C
$a \leftarrow 5$	$n1 \leftarrow 5$	$n1 \leftarrow 5$
$b \leftarrow a + 4$	$n2 \leftarrow 7$	$n2 \leftarrow 7$
$a \leftarrow a + 1$	$n1 \leftarrow n2$	$n2 \leftarrow n1$
$b \leftarrow a - 4$	$n2 \leftarrow n1$	

Exercise 3.7 Trouver les valeurs booléennes prises au cours de l'algorithme suivant :

Algorithme Eval_expression

Variable a, b : entier
b1, b2, b3, b4 : booléen

début
a ← 10
b ← 4
b1 ← (10 > 10) ET (5 = 5)
b2 ← (a = 10) OU (b = 5) OU (3 = 6)
b3 ← (a > b) ET ((5 =5) OU (b < a))
b4 ← (Faux) ET (Vrai) OU (a > b)
fin

Indications Vous pouvez utiliser la table de vérité pour évaluer les expressions booléennes.

Exercise 3.8 Supposons que les variables n, p et q sont de type entier et qu'elles contiennent respectivement les valeurs 8, 13 et 29. Déterminer les valeurs des expressions suivantes :
$n + p/q$, $n + q/p$, $(n + q)/p$, $n + p/n + p$

Indications Faites attention à la priorité des opérateurs.

Exercise 3.9 Écrire un algorithme qui permet d'échanger les valeurs de 2 variables réelles :

1. En utilisant une variable intermédiaire.
2. Sans utiliser une variable intermédiaire.

Indications La tâche d'échange des valeurs de deux variables est une tâche simple, mais elle est largement utilisée dans d'autres algorithmes, notamment ceux de tri.

Exercise 3.10 Soient les déclarations et les actions suivantes :

```
Variable n, p : entier
         x : réel

début
    n ← 10
    p ← 7
    x ← 2.5
fin
```

Donnez le type et la valeur des expressions suivantes :

1. x + n / p
2. (x + n) / p
3. 5. * n
4. (n + 1) / n
5. (n + 1.0) / n

Exercise 3.11 Écrire un algorithme qui donne des valeurs à deux variables entières nommées a et b, et qui en affiche les valeurs, le produit et la somme, sous cette forme :
$a = 3, b = 5$
$a * b = 15$
$a + b = 8$

Indications Attention, ici les valeurs 3 et 5 des variables a et b respectivement sont juste des exemples. Votre algorithme doit traiter le cas général. Par ailleurs, vous devez respecter le format d'affichage.

Exercise 3.12 Écrire un algorithme qui demande deux nombres entiers et qui fournit leur somme et leur produit. Le dialogue avec l'utilisateur se présentera ainsi :

```
donnez deux nombres
35      3
leur somme est 38
leur produit est 105
```

Indications Attention, ici les valeurs 35 et 3 sont juste des exemples. Votre algorithme doit traiter le cas général. Respectez le format d'affichage.

Exercise 3.13 Écrire un algorithme qui lit le prix hors taxe d'un article, le nombre d'articles et le taux de TVA et qui affiche le prix TTC correspondant. Le dialogue se présentera ainsi :

prix unitaire HT :
12.5
nombre articles :
8
taux TVA :
19.6
prix total HT : 100.
prix total TTC : 119.6

Indications Attention, les valeurs 12.5, 8 et 19.6 sont juste des exemples du prix unitaire hors taxe, le nombre d'article et le taux de TVA respectivement. Votre algorithme doit traiter le cas général.

Exercise 3.14 Une assurance propose trois tarifs (Vert, Orange et Rouge) selon l'âge et le nombre d'accidents des automobilistes (Table 3.20).
Écrire un algorithme qui affiche le tarif après avoir saisi l'âge et le nombre d'accidents d'un automobiliste.

TABLE 3.20 – Tarifs d'assurance.

Nombre d'accidents	**Moins de 25 ans**	**25 ans et plus**
0 accident	Orange	Vert
1 ou 2 accidents	Rouge	Orange
3 à 6 accidents	Pas assuré	Rouge
7 accidents ou plus	Pas assuré	Pas assuré

Indications Vous pouvez donner plusieurs variantes de cet algorithme selon que vous considériez d'abord l'âge ou le nombre d'accidents et même selon que vous utilisiez des conditions simples ou complexes.

Table 3.21 – Offres d'abonnement téléphonique.

Offre	Fixe	Prix à la minute
Telecom 1	200 DA	2 DA
Telecom 2	300 DA	1.50 DA

Exercise 3.15 Vous désirez comparer deux offres d'abonnement téléphonique (Table 3.21). La facture est calculée avec un fixe (somme à payer obligatoirement tous les mois) et une partie proportionnelle au temps passé à téléphoner (indiqué en minutes).

Exercise 3.16 Compléter la Table 3.22 représentant la correspondance entre les conditions de continuité et les conditions d'arrêt.

Table 3.22 – Correspondance entre les conditions de continuité et les conditions d'arrêt.

condition d'arrêt	ondition de continuité
(nb = 4) ET (age < 25)	
(de = 6) OU (nbcoup > 5)	
(de1 = 6 ET (de2 = 6) OU (nbcoup > 5)	
(de1 = 6) OU (de2 = 6)	

Indications C'est un exercice intéressant qui nous enseigne comment passer d'une condition de continuité à une condition d'arrêt. Autrement dit, de passer d'une boucle *tant que* à une boucle *répéter* et vice-versa.

Exercise 3.17 Écrire un algorithme qui lit deux nombres entiers et qui détermine s'ils sont rangés ou non par ordre croissant et, dans tous les cas, affiche leur différence (entre le plus grand et le plus petit).

Exercise 3.18 Écrire un algorithme qui lit trois nombres entiers et qui trouve s'ils sont ou non rangés dans l'ordre croissant strict, c'est-à-dire que, si a, b et c désignent ces nombres, ils devront vérifier l'égalité mathématique : $a < b < c$.

Exercise 3.19 Écrire un algorithme qui saisie un entier et indique s'il est pair ou impair.

Indications Vous pouvez utiliser l'opérateur modulo.

Exercise 3.20 Écrire un algorithme qui lit un prix hors taxe et qui calcule et affiche le prix TTC correspondant (avec un taux de TVA de 17%). Il établit ensuite une remise dont le taux est le suivant :

- 0% pour un montant TTC inférieur à 1000 DA,
- 1% pour un montant TTC supérieur ou égal à 1000 DA et inférieur à 2000 DA,
- 2% pour un montant TTC supérieur ou égal à 2000 DA et inférieur à 5000 DA,
- 5% pour un montant TTC supérieur ou égal à 5000 DA.
- On affichera la remise obtenue et le nouveau montant TTC.

Exercise 3.21 Écrire un algorithme qui lit un entier représentant un mois de l'année (1 pour janvier, 4 pour avril, ...) et qui affiche le nombre de jours de ce mois (on supposera qu'on n'est pas en présence d'une année bissextile). On tiendra compte du cas où l'utilisateur fournit un numéro incorrect, c'est-à-dire non compris entre 1 et 12.

Indications Une année bissextile est une année comportant 366 jours au lieu de 365 jours. c'est à dire une année comportant un 29 février.

Exercise 3.22 Soit le code suivant, qu'affiche t-il ?

```
Algorithme boucle_tantque
Variable k,m : entier

début
    k ← 11010
    m ← k MOD 2
    tant que ( k > 0) faire
    début
        m ← (m + k MOD 10) * 2
        k ← k DIV 10
    fin
    Écrire (m)
fin
```

Indications On vous demande de dérouler (faire la trace) de l'algorithme.

Exercise 3.23 Le code suivant est exécuté et affiche la sortie suivante :
15 16 17 18 19 2

Algorithme boucle_tantque

```
Variable m, p, n : entier
         test1, test2 : booléen

début
    p ← 4
    n ← 10
    test1 ← faux
    test2 ← (test1 et (p < n))
    pour m de (n DIV 2) à n-1 faire
    début
        si (test1 ou test2) alors
        début
            Écrire ((p + m), ' ')
            test2 ← faux;
        fin
        sinon
        début
            Écrire ( (n + m), ' ')
            (* code manquant *)
        fin
        si (test1 et test2) alors
        début
            Écrire( 1, ' ')
        fin
        sinon
        début
            Écrire( 2, ' ')
        fin
    fin
fin
```

Quel est le code manquant ?

1. test1 ← vrai
2. test1 ← non test1
3. test1 ← non test2
4. test1 ← test2

Indications On vous demande de dérouler (faire la trace) de l'algorithme.

Exercise 3.24 Écrire un algorithme qui demande à l'utilisateur de lui fournir un nombre entier positif et inférieur à 100 et ceci jusqu'à ce que la réponse soit satisfaisante. Le dialogue avec l'utilisateur se présentera ainsi :

donnez un entier positif inférieur à 100 :
453
donnez un entier positif inférieur à 100 :
25
merci pour le nombre 25

Indications C'est un exercice intéressant pour le contrôle de validité des données. Utilisez la boucle *répéter*.

Exercise 3.25 Dans l'Exercice 3.24, l'utilisateur se voit poser la même question, qu'il s'agisse d'une première demande ou d'une nouvelle demande suite à une réponse incorrecte. Améliorez-le de façon que le dialogue se présente ainsi :

donnez un entier positif inférieur à 100 :
453
SVP, inférieur à 100 :
0
SVP, positif :
25
merci pour le nombre 25

Indications C'est un exercice intéressant pour le contrôle de validité des données. Utilisez la boucle *répéter*.

Exercise 3.26 Réécrire l'algorithme demandé dans l'Exercice 3.24 en utilisant une répétition *tant que*.

Indications Utilisez les règles de passage des conditions d'arrêt aux conditions de continuité apprises dans l'Exercice 3.16.

Exercise 3.27 Réécrire l'algorithme demandé dans l'Exercice 3.25 en utilisant une répétition *tant que*.

Indications Utilisez les règles de passage des conditions d'arrêt aux conditions de continuité apprises dans l'Exercice 3.16.

Exercise 3.28 Écrire un algorithme qui affiche les carrés des nombres entiers de 7 à 20.

Exercise 3.29 Écrire un algorithme qui lit deux nombres entiers dans les variables nd et nf et qui écrit les doubles des nombres compris entre ces deux limites (incluses).

Exercise 3.30 Construire l'algorithme qui permet d'effectuer la multiplication de 2 nombres entiers en utilisant des additions successives.

Exercise 3.31 Construire un algorithme qui calcule le Nième (avec $N > 2$) terme de la suite de FIBONACCI qui est définie par :

$$\begin{cases} U_0 &= U_1 = 1 \\ U_n &= U_{n-1} + U_{n-2} \end{cases}$$

Suite de FIBONACCI : 1, 1, 2, 3, 5, 8, 13, ...

Exercise 3.32 Sachant qu'un nombre premier est un nombre qui n'accepte aucun diviseur excepté 1 et lui même. Construire un algorithme qui nous donne les N premiers nombres premiers.

Exercise 3.33 Sachant qu'il n'existe que 4 nombres compris entre 100 et 500 tels que la somme des cubes des chiffres les composant est égale au nombre lui-même. Construire un algorithme qui permet de retrouver ces 4 nombres. Exemple : $153 = 13 + 53 + 33$

Exercise 3.34 Un nombre parfait est un nombre qui est égal à la somme de tous ses diviseurs excepté lui même. Construire un algorithme de recherche de tous les nombres parfaits compris entre 1 et 1000.

Indications Un nombre parfait est un entier naturel n tel que $\sigma(n) = 2n$, où $\sigma(n)$ est la somme des diviseurs positifs de n. 6 est un nombre parfait car $2*6 = 12 = 1 + 2 + 3 + 6$, ou encore $6 = 1 + 2 + 3$.

Exercise 3.35 Construire un algorithme de recherche du PPCM (plus petit commun multiple) de 2 nombres A et B.

Indications Les solutions sont multiples suite à la multiplicité des méthodes de calcul du PPCM déjà étudiées pendant votre cursus.

Exercise 3.36 Soient 2 nombres donnés A et B, tels que $A_x = B_{10}$ (ou x et 10 sont les bases dans lesquelles sont écrits A et B). Construire un algorithme qui nous permet de savoir dans quelle base est écrit A.
Exemple : $A = 20405_x$ et $B = 8453_{10}$
La base dans laquelle est écrit le nombre A est la base 8.

Exercise 3.37 Les mathématiques sont une science fascinante mais elle demeure, quand même, une matière rebutante pour beaucoup. Cependant, un de leur domaine ne peut pas laisser indifférent tout esprit curieux. Il s'agit des curiosités mathématiques. Parmi celles-ci, on trouve «la bande des 9».

En effet, si vous prenez trois (3) nombres (A, B, C) composés chacun de trois (3) chiffres et tel que : A + B = C, et si les neufs (9) chiffres utilisés sont : 1, 2 ,3 ,4 ,5 ,6 ,7 ,8 ,9. Alors, la somme des chiffres constituant le résultat (soit C) est toujours égal à 18.

Exemples :
152 + 487 = 639 238 + 419 = 657 357 + 462 = 819 784 + 152 = 936

Construire une solution qui vous permettra de trouver tous les cas (c'est à dire A, B et C) qui respectent cette «bizarrerie», de même que leur nombre.

Exercise 3.38 Construire un algorithme qui nous permet d'obtenir un nombre N ayant les valeurs successives suivantes :
1
22
333
4444
55555
666666
7777777
88888888
999999999
De même que la somme des chiffres composant l'ensemble de ces nombres.

Exercise 3.39 Dans la théorie des nombres parfaits, EULER a démontré que l'expression : $2^{n-1}(2^n - 1)$ donne toujours un nombre parfait lorsque la quantité entre parenthèses est un nombre premier. Construire un algorithme qui nous permet d'obtenir les 5 premiers nombres parfaits.

Exemple : lorsque n = 2 , $2^n - 1$ est égal à 3 qui est premier, et $2^{n-1}(2^n - 1)$ est égal à 6 , et 6 est un nombre parfait.

4. Tableaux et chaînes de caractères

4.1 Exemple Introductif

Nous abordons ce chapitre par la discussion d'un petit problème.

Supposons qu'on veut traiter et conserver les notes d'une classe de 30 étudiants. Par exemple, cette tâche peut se manifester en :

- Classement des étudiants de la classe.
- Calcul du nombre d'étudiants ayant une note supérieure à 10.

Selon le formalisme algorithmique que nous avons appris, nous nous disposons, actuellement, que des variables de types simples. Alors, pour résoudre le problème suscité, il nous faut disposer simultanément des notes de 30 étudiants. Pour cela, nous pouvons toujours utiliser 30 variables différentes nommées par exemple : *note1*, *note2*, . . . , *note30*.

Par exemple, pour le calcul du nombre d'étudiants ayant une note supérieure à 10 (avec conservation des notes), on doit écrire :

1. 30 actions de lecture de notes :

Lire (note1)
Lire (note2)
. . .
Lire (note30)

---

2. 30 actions conditionnelles pour faire le calcul :

```
Nbr_etud ← 0
si note1 > 10 alors
   Nbr_etud ← Nbr_etud + 1;
si note2 > 10 alors
   Nbr_etud ← Nbr_etud + 1;
...
si note30 > 10 alors
   Nbr_etud ← Nbr_etud + 1;
```

Le problème qui se pose pour cette solution est la lourdeur d'écriture de l'algorithme. Cette écriture devient plus fastidieuse si le nombre des étudiants saute à 100 ou 1000.

Une alternative à ce problème consiste à penser à une seule structure de donnée composée pouvant rassembler plusieurs valeurs. Cette structure est appelée *tableau*. C'est bien l'objet de la première partie du présent chapitre.

Par ailleurs, la plupart des langages de programmation offrent le service de la structure de données tableau.

4.2 Tableaux à une dimension

Nous commençons par définir la notion de *tableaux à une dimension* ainsi que les concepts associés (Définitions 4.1, 4.2 et 4.3).

Définition 4.1 Un *tableau à une dimension*, appelé aussi *vecteur* est une structure de données désignée par un **identifcateur unique** permettant de stocker un nombre fini d'éléments de même type directement accessibles par leurs *indices*.

Définition 4.2 Un *indice* est une variable de type scalaire qui permet d'accéder aux éléments d'un tableau. Un indice indique le rang (position) de l'élément.

Définition 4.3 La *taille* d'un tableau, parfois dite dimension, est le nombre maximal de ses éléments.

La déclaration d'un tableau suit la syntaxe suivante :

```
Variable Nom_tableau : tableau [MinDim..MaxDim] de type_composant
```

Exemple 4.1

```
Variable notes : tableau [1..30] de réel
```

Dans l'Exemple 4.1, *notes* est le nom d'un tableau, c'est un identificateur unique pour toute la structure composée qui contient plusieurs éléments. ***tableau*** est le

mot réservé pour dire que la variable *notes* est de type ***tableau***. Les indices 1 et 30 sont respectivement les bornes inférieurs et supérieurs du tableau *notes*. Ces indices définissent la taille du tableau, ici 30 éléments de type *réel* exprimant le type du tableau. Autrement dit, le type des éléments d'un tableau est bien le type du tableau.

La Figure 4.1 illustre la notion de tableau déclaré dans l'Exemple 4.1.

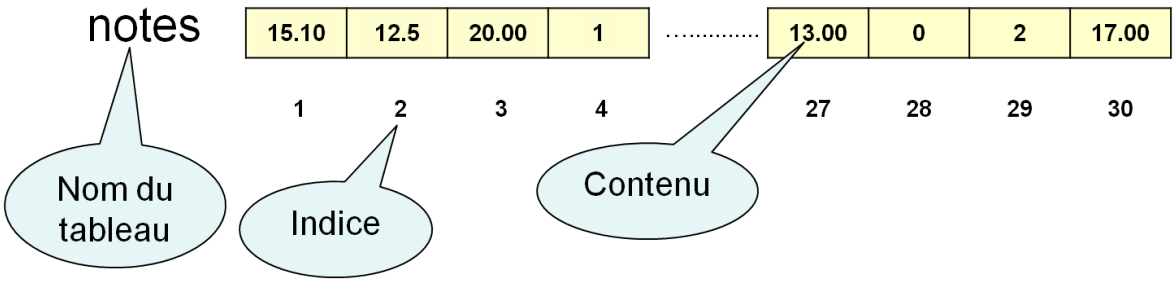


Figure 4.1 – Exemple d'une structure tableau

4.3 Utilisation d'un tableau à une dimension

L'intérêt des tableaux réside dans le faite qu'un tableau est une seule structure rassemblant plusieurs éléments et que chaque élément est accessible individuellement.

L'accès à un élément du tableau s'appuie sur la notion de *variable indicée* qui se matérialise par une opération d'indexation qui consiste à indiquer une valeur d'indice entre crochets après le nom du tableau. Par conséquent, le temps d'accès à un élément est constant quelque soit l'élément désiré.

Lors de la déclaration d'un tableau, on lui alloue un espace mémoire contigu qui doit être défini avant son utilisation et ne peut pas être changé.

La variable indicée (élément du tableau) s'utilise comme une variable simple. Par conséquent, elle peut :

1. Faire l'objet d'une affectation tel que exprimé dans l'Exemple 4.2.

Exemple 4.2

```
notes[1] ← 15.10
notes[27] ← 13.00
```

2. Figurer dans la liste de l'action de lecture (Exemple 4.3).

Exemple 4.3

```
Pour i de 1 à 30 faire
début
    Écrire ('saisir la note de rang ', i)
    Lire (notes[i])
Fin
```

3. Figurer dans la liste de l'action de l'écriture (Exemple 4.4).

Exemple 4.4

```
Pour i de 1 à 30 faire
début
    Écrire ('La note de rang ', i,' est égale à : ', notes[i])
Fin
```

4. Figurer dans une expression (Exemple 4.5).

Exemple 4.5

```
Moyenne ← 0
Pour i de 1 à 30 faire
début
    Moyenne ← Moyenne + notes[i]
Fin
Moyenne ← Moyenne / 30
```

Pour bien cerner l'importance et la nécessité d'utilisation des tableaux, revenons au problème d'ouverture du présent chapitre qui consiste à calculer le nombre d'étudiants ayant une note supérieure à 10. Cette fois ci, nous écrivons l'Algorithme *Note_Sup_10* en utilisant une structure de donnée *tableau*.

```
Algorithme Note_Sup_10
Constante max = 30
Variable i, Nbr_etud : entier
         notes : tableau[1..max] de réel

début
    Nbr_etud ← 0
    Pour i de 1 à max faire
    Début
        Écrire ('saisir la note de rang ', i)
        Lire (notes[i])
        si notes[i] >= 10 alors
            Nbr_etud ← Nbr_etud +1
    Fin
    Écrire ('Le nombre d''étudiants ayant une note supérieur à 10 = ',
     Nbr_etud)
fin
```

4.4 Algorithmes sur les tableaux

L'objectif de cette section est de présenter deux méthodes principales qui utilisent les tableaux, à savoir, une méthode de tri et celle de recherche d'une valeur dans un tableau.

Dans notre contexte, nous nous intéressons plus à l'utilisation des tableaux qu'aux différentes techniques de tri et de recherche.

4.4.1 Tri d'un tableau

Le tri consiste à réorganiser une séquence d'objets de manière à les mettre dans un ordre logique. Par défaut, le tri implique un ordre croissant. Dans le cas contraire, on doit préciser explicitement un ordre décroissant.

Il existe plusieurs raisons pour étudier les algorithmes de tri :

- Des techniques similaires à celles des méthodes de tri élégantes sont efficaces pour résoudre d'autres problèmes.
- Nous utilisons souvent des algorithmes de tri comme point de départ pour résoudre d'autres problèmes.

Et dans notre cas, la raison la plus importante est de maîtriser l'utilisation de la structure tableau.

Dans la littérature, il existe plusieurs méthodes de tri :

- Tri à bulle,
- tri par sélection,

- tri par insertion,
- tri par comptage,
- tri shell,
- tri rapide (quick sort),
- tri par fusion (merger sort).

Nous rappelons que notre objectif n'est pas une étude complète des méthodes de tri, plutôt c'est l'utilisation des tableaux. Pour cette raison, nous nous limitons à un seul algorithme, le tri à bulle.

Tri à bulle

Bien que la méthode de tri à bulle soit reconnue comme une méthode inefficace, nous la présentons ici à titre pédagogique en raison de sa simplicité.

Le tri à bulle se déroule dans le même tableau selon le principe suivant :

1. Parcourir le tableau en comparant les éléments consécutifs. S'ils sont mal ordonnés, alors échanger les.
2. Recommencer jusqu'à ce qu'il n'y a plus d'échanges à effectuer.

La concrétisation du principe sus-cité en utilisant le formalisme algorithmique donne l'Algorithme *Tri_bulle*.

Algorithme Tri_bulle

```
Constante n = 5
Variable notes = tableau[1..n] de réel
         echange : booléen /* est ce qu'un échange a été effectué */
         i : entier
         Inter : réel

début
   /* saisir les notes */
   Pour i de 1 à n faire
   début
        Écrire ('saisir la note de rang ', i)
        Lire (notes[i])
   fin
   /* Tri du tableau */
   répéter
        echange ← faux
        Pour i de 1 à n-1 faire
        début
             si notes[i] > notes[i+1] alors
             début
                  echange ← vrai
                  inter ← notes[i]
                  notes[i] ← notes[i+1]
                  notes[i+1] ← inter
             fin
        fin
   jusqu'à non echange
fin
```

Déroulement

Pour bien comprendre le fonctionnement d'un algorithme et d'en vérifier globalement sa validité, il est naturel de procéder à son déroulement (trace de l'algorithme).

Le déroulement consiste d'abord à dessiner un tableau où chaque colonne est réservée pour une variable. Ensuite, on commence par l'exécution de l'algorithme action par action et on note à chaque fois le changement des valeurs des variables dans les colonnes associées.

À titre d'échauffement, nous déroulons l'Algorithme *Tri_bulle*.

La première boucle sert à saisir les données dans le tableau *notes*. Supposons qu'à l'issue de cette première boucle, le contenu du tableau *notes* est montré dans la Figure 4.2.

La boucle *répéter* s'occupe de l'opération de tri du tableau *notes*. Un parcours de cette boucle consiste à aller de bout à l'autre du tableau en comparant les éléments

15.10	12.5	20.00	1	13.00

FIGURE 4.2 – Le contenu du tableau *notes* à trier.

voisins et en les échangeant s'ils sont mal ordonnés. L'échange est réalisée en utilisant une variable intermédiaire (*inter*). Cette échange remonte dans le tableau comme une bulle.

À l'issue du premier parcours (Figure 4.3), le tableau *notes* est partiellement trié. Les valeurs ne sont pas dans leurs positions définitives, donc il reste des paires d'éléments mal ordonnées. Le contenu de la variable booléenne *echange* (*echange* = Vrai) témoigne qu'on a procédé à une échange (permutation) entre au moins deux éléments voisins lors du parcours en cours. Par conséquent, on aura besoins d'un ou plusieurs parcours.

i	echange	notes				
1	vrai	15.10	12.5	20.00	1	13.00
2	vrai	12.5	15.10	20.00	1	13.00
3	Vrai	12.5	15.10	20.00	1	13.00
4	Vrai	12.5	15.10	1	20.00	13.00

FIGURE 4.3 – Premier parcours de tri à bulle.

Le déroulement de l'algorithme se poursuit avec le second et le troisième parcours (Figure 4.4 et Figure 4.5 respectivement) mais toujours avec la présence d'une ou plusieurs échanges. Cela implique qu'il est toujours nécessaire d'un nouveau parcours.

Dans le quatrième parcours (Figure 4.6), on constate que la variable *echange* demeure inchangée (*echange* = Faux) révélant, ainsi, qu'aucune échange n'a été effectuée. Alors le tableau *notes* est trié et l'algorithme doit s'arrêter.

4.4.2 Recherche d'un élément dans un tableau

La recherche d'un élément est parmi les tâches les plus fréquentes sur les tableaux. La recherche d'un élément dans un tableau consiste à déterminer si cet élément est présent ou non dans le tableau. Dans le cas où l'élément existe, on doit renvoyer sa position.

1	Faux	12.5	15.10	1	13.00	20.00
2	Vrai	12.5	15.10	1	13.00	20.00
3	Vrai	12.5	1	15.10	13.00	20.00
4	Vrai	12.5	1	13.00	15.10	20.00

Figure 4.4 – Second parcours de tri à bulle.

1	Faux	12.5	1	13.00	15.10	20.00
2	Vrai	1	12.5	13.00	15.10	20.00
3	Vrai	1	12.5	13.00	15.10	20.00
4	Vrai	1	12.5	13.00	15.10	20.00

Figure 4.5 – Troisième parcours de tri à bulle.

Deux techniques de recherche sont largement étudiées, la recherche séquentielle et la recherche binaire.

La recherche séquentielle consiste à faire un balayage du tableau jusqu'à trouver la valeur recherchée ou atteindre la fin du tableau. Ceci peut nuire à l'efficacité de l'algorithme, notamment, quand la taille du tableau est considérable.

Dans ce qui suit, nous étudions la recherche binaire. parfois dite recherche dichotomique. Cette méthode de recherche n'est applicable que si le tableau est trié.

Recherche binaire

La recherche binaire, parfois dite recherche dichotomique n'est applicable que si le tableau est déjà trié.

Soit *tab* un tableau trié contenant n éléments. Nous désirons écrire un algorithme qui cherche une valeur X dans *tab*. Si X existe dans *tab*, l'algorithme afffiche l'indice de son apparition dans *tab*. Sinon, l'algorithme affiche le message 'X n'existe pas dans le tableau'.

Le principe de la recherche binaire peut être décrit dans les étapes suivantes :

1. Au départ, la plage de recherche est tout le tableau, soit $tab[inf..sup]$, $inf = 1$, $sup = n$ et l'élément recherché = X.

2. À chaque itération on a :

   - une plage $[inf..sup]$;

1	Faux	1	12.5	13.00	15.10	20.00
2	Faux	1	12.5	13.00	15.10	20.00
3	Faux	1	12.5	13.00	15.10	20.00
4	Faux	1	12.5	13.00	15.10	20.00

FIGURE 4.6 – Dernier parcours de tri à bulle.

- son milieu $m = (inf + sup)\, div\, 2$;
- la subdivision : $[inf..m-1]$, $[m]$, $[m+1..sup]$.

3. Il existe trois cas possibles :

- Soit $tab[m] = X$, l'élément X est trouvé (fin de l'algorithme).
- Soit $tab[m] < X$, l'élément $X \notin [inf..m]$ et s'il existe, il se trouve dans la nouvelle plage $[m+1..sup]$.
- Soit $tab[m] > X$, l'élément $X \notin [m..sup]$ et s'il existe, il se trouve dans la nouvelle plage $[inf..m-1]$.

Le déroulement de l'Algorithme *recherche_binaire* pour les valeurs $X = 7$ et $X = 3$ est illustré respectivement dans les Figures 4.8 et 4.9. La recherche pour la valeur $X = 7$ est fructueuse et le message affiché est : "L'élément 7 existe dans le tableau et il est à la position : 4 ". Alors que La recherche pour la valeur $X = 3$ est infructueuse et le message 'l'élément , 3, n'existe pas dans le tableau'.

La recherche binaire consiste à itérer ce processus jusqu'à ce que l'on trouve X ou que la plage de recherche soit vide.

En suivant le principe ci-dessus, nous pouvons obtenir L'Algorithme *recherche_binaire*.

Algorithme recherche_binaire

```
Constante n = 5
Variable tab = tableau[1..n] de réel
         inf, sup, m : entier
         X : réel
         trouve : booléen

début
    /* saisir l'élément à rechercher */
    Écrire ('saisir l"élément à rechercher ')
    Lire (X)
    /* Recherche de l'élément */
    inf ← 1
    sup ← n
    trouve ← Faux
    tant que (inf <= sup ) ET (NON trouve) faire
    début
        m ← (inf + sup) div 2
        si tab[m] = X alors
        début
            trouve ← Vrai
        fin
        sinon
        début
            si tab[m] < X alors
            début
                inf ← m+1
            fin
            sinon
            début
                sup ← m-1
            fin
        fin
    fin
    si trouve alors
    début
        Écrire ('l"élément ', X, 'existe dans le tableau et il est à la position : ',
         m);
    fin
    sinon
    début
        Écrire ('l"élément ', X, 'n"existe pas dans le tableau');
    fin
fin
```

Pour comprendre le principe de la recherche binaire, prenons l'Exemple 4.6.

Exemple 4.6 Soit un tableau qui contient 5 éléments tel que montré dans la Figure 4.7. Rechercher les valeurs 7 et 3 dans ce tableau.

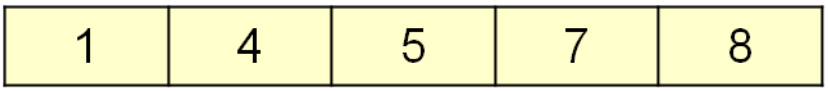


FIGURE 4.7 – Un tableau pour la recherche binaire.

Le déroulement de l'Algorithme *recherche_binaire* pour les valeurs $X = 7$ et $X = 3$ est illustré respectivement dans les Figures 4.8 et 4.9.

Lors de la recherche de la valeur $X = 7$ et à l'issue de la boucle *tant que*, la valeur de la variable booléenne *trouve* est *Vrai* attestant que la valeur $X = 7$ existe dans le tableau. Ainsi, la recherche est fructueuse et le message affiché est : "L'élément 7 existe dans le tableau et il est à la position : 4 ".

Alors que lors de la recherche de la valeur $X = 3$ et à l'issue de la boucle *tant que*, la valeur de la variable booléenne *trouve* est *Faux* attestant que la valeur $X = 3$ n'existe pas dans le tableau. Ainsi, la recherche est infructueuse et le message affiché est : "L'élément 3 n'existe pas dans le tableau".

X	inf	sup	m	trouve	tab
7	1	5	?	Faux	1 4 5 7 8
7	1	5	3	Faux	1 4 5 7 8 inf m sup
7	4	5	4	Vrai	1 4 5 7 8 inf m sup

L'élément 7 existe dans le tableau et il est à la position : 4

FIGURE 4.8 – Déroulement de l'Algorithme *recherche_binaire* pour la recherche de la valeur 7.

Avant de clôturer la section sur la recherche binaire, il est judicieux d'en faire une petite évaluation.

Nous rappelons que le principe de la recherche binaire consiste à diviser la plage de recherche par 2 à chaque itération. Donc, il n'est pas difficile d'observer que dans le pire des cas, on doit exécuter i itérations tel que $n = 2^i$. D'où $i = \log_2 n$. C'est un

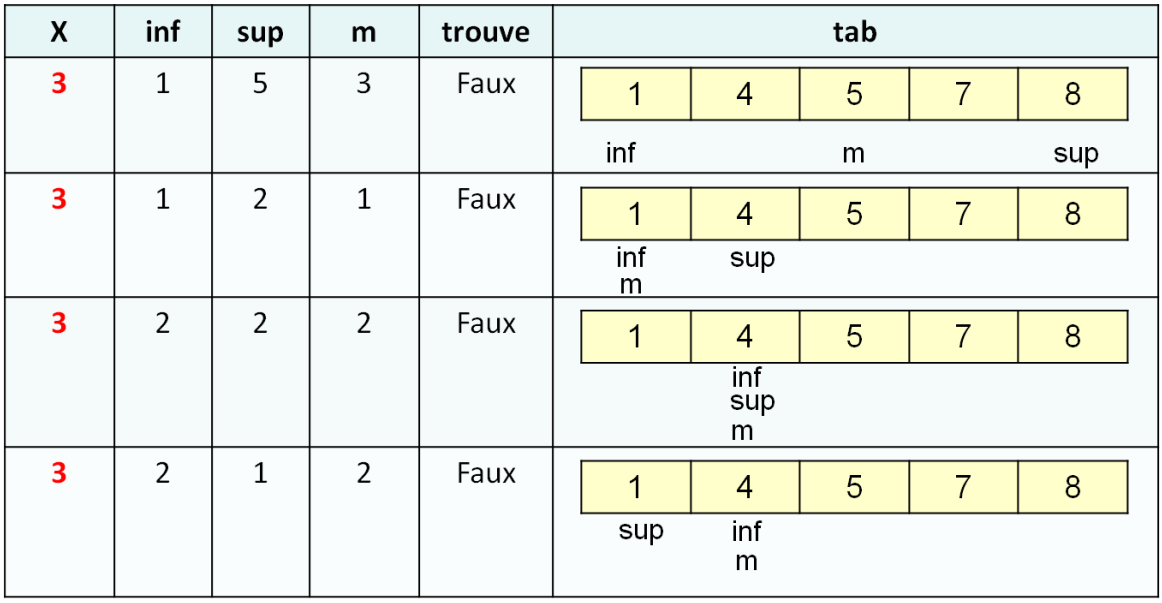


FIGURE 4.9 – Déroulement de l'Algorithme *recherche_binaire* pour la recherche de la valeur 3.

gain très appréciable en terme de nombre d'itérations effectuées en se comparant à la recherche séquentielle.

À titre d'exemple, si la taille du tableau est $2^{30} = 1073741824$. Dans le cas de la recherche séquentielle, il nous faut au plus 1073741824 itérations. Cependant, dans le cas de la recherche binaire, au plus, on est amené à effectuer 30 itérations. La différence est nettement visible !

4.5 Tableaux à deux dimensions

Si nous revenons au problème posé dans la Section 4.1, mais cette fois-ci, supposons qu'on veut traiter et conserver les notes de plusieurs matières d'une classe de 30 étudiants.

Devons-nous utiliser plusieurs tableaux en associant un tableau à chaque matière ? C'est possible, mais la manipulation de plusieurs tableaux pour le même problème peut paraître rigide !

Un autre angle de vision pour le problème posé consiste à penser dimensions. Autrement dit, nous pouvons ajouter une dimension pour les matières de la même manière que nous avons ajouté une dimension pour les étudiants.

Dans ce cas, nous parlons d'un *tableau à 2 dimensions* (ou tout simplement *matrice*), une pour les étudiants et une autre pour les matières.

Les tableaux à deux dimensions se déclarent sous cette forme :

Variable Nom_tableau : **tableau** [dimension1,dimension2] de **type_composant**

Du point de vue représentation, il est convenu que *dimension1* correspond aux lignes et *dimension2* correspond aux colonnes.

Voici un exemple d'illustration de déclaration d'un tableau à deux dimensions nommé *notes_mat* qui comporte 30 lignes pour les étudiants et 8 colonnes pour les matières (Exemple 4.7) :

Exemple 4.7
Variable notes_mat : **tableau** [1..30, 1..8] **de réel**

La Figure 4.10 est une représentation graphique de la déclaration de l'Exemple 4.7.

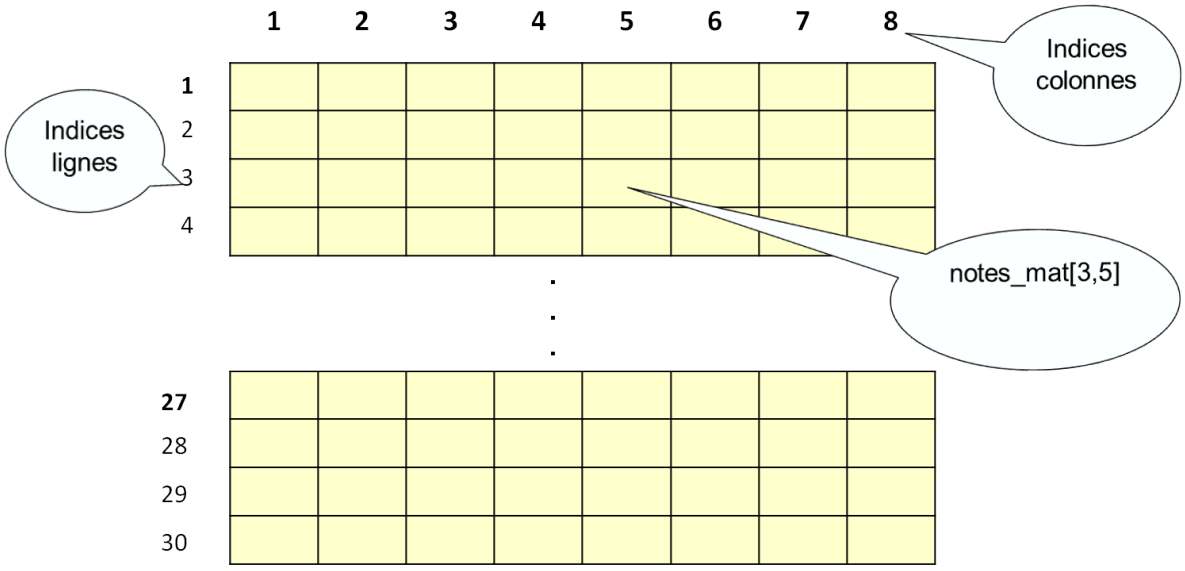


Figure 4.10 – Représentation graphique d'un tableau à 2 dimensions.

Chaque élément de la matrice est repéré par deux indices, un indice ligne et un autre colonne. Ainsi, *notes_mat[3, 5]* fiat référence à l'élément situé dans la 3ème ligne et la 5ème colonne.

La notation *notes_mat[i, j]* est une généralisation à 2 indices de la notion de variable indicée introduite dans la Section 4.3.

4.6 Utilisation d'un tableau à deux dimensions

Comme dans le cas d'un tableau à une dimension, l'accès à un élément d'un tableau à deux dimensions s'appuie sur la notion de *variable indicée*.

Nous rappelons qu'une variable indicée s'utilise comme une variable simple. Les exemples suivants montrent les différentes manipulations d'un tableaux à deux dimensions :

Affectation de valeurs

L'affectation d'une valeur à un élément d'un tableau à deux dimensions se fait en utilisant la notion de la variable indicée vue dans la Section 4.5. Cette opération est illustrée par l'Exemple 4.8.

Exemple 4.8

```
notes_mat[1,1] ← 15.10
notes_mat[27,8] ← 13.00
```

Lecture des élémnets

Afin de lire les éléments d'un tableau à deux dimensions, il est nécessaire d'utiliser deux boucles imbriquées correspondant chacune à une dimension. L'exemple 4.9 montre comment peut-on lire les éléments de la matrice *notes_mat*.

Constatez que la lecture dans l'Exemple 4.9 se fait ligne par ligne, alors qu'il est possible de faire le même traitement colonne par colonne. Il suffit d'interchanger l'ordre d'imbrication des boucles.

Exemple 4.9

```
Pour i de 1 à 30 faire
début
    Pour j de 1 à 8 faire
    début
        Écrire ('saisir une note')
        Lire (notes_mat[i, j])
    Fin
Fin
```

Écriture des éléments

Comme dans le cas de la lecture, l'écriture (le remplissage) d'un tableau à deux dimensions se fait en utilisant deux boucles imbriquées. L'opération d'écriture est montrée par l'Exemple 4.10

Exemple 4.10

```
Pour i de 1 à 30 faire
début
    Pour j de 1 à 8 faire
    début
        Écrire ('La note de rang [', i, ', ' , j, '] est égale à : ' ,
 notes_mat[i,j])
    Fin
Fin
```

4.7 Chaînes de caractères

Au cours de cette section, nous traitons d'abord le type caractère pour préparer le terrain pour l'étude des chaînes des caractères.

4.7.1 Type caractère

Un caractère est l'élément de base d'une chaîne de caractères. Il est utilisé pour désigner n'importe quel symbole pouvant apparaître dans un texte. Autrement dit, il s'agit du domaine qui regroupe :

- Les lettres alphabétiques (majuscule + minuscule) : 'A', 'c', ...
- Les caractères numériques (chiffres) : '3', '8', ...
- Les caractères spéciaux : ',', '?', '', '$', ...
- Le caractère espace (blanc) : ' '.

Les valeurs du type caractère sont ordonnées selon l'ordre des codes internes des caractères (ASCII, IBM-PC, ISO-xxx, ...).

Représentation des caractères

Rappelons que quelque soit la nature de l'information (image, son, texte, vidéo) traitée par un ordinateur est toujours représentée sous une forme de séquence d'éléments binaires (bits).

Dans les cas des caractères, on doit établir une correspondance entre les éléments de type caractère et une représentation binaire.

La Table 4.1 récapitule des exemples de standards de codage des données textuelles évolués au cours du temps.

À titre d'illustration, nous étudions le code ASCII (American Standard Code for Information Interchange) qui est l'un des plus anciens codes utilisés en informatique.

Table 4.1 – Exemples de standards de codage des caractères.

Nombre de bits	Nombre de caractères	Nom	Remarques
6	64	Display Code	Permet de représenter les 26 lettres majuscules latines, les chiffres 0..9, les symboles de ponctuation : , :; . ( ) .... Un codage suffisant pour écrire des programmes et imprimer des résultats.
7	128	ASCII	(American Standard Code for Information Interchange), adapté à la langue américaine : lettres majuscules et minuscules sans accents, chiffres, symboles de ponctuation. Le code ASCII est très utilisé sur les processeurs de la famille Intel.
8	256	EBCDIC	(Extended Binary Coded Decimal Interchange Code), très utilisé sur les gros systèmes, notamment les systèmes de la famille IBM tels que les architectures 370 et 390.
16	65536	UNICODE	(UNIversal CODE), un standard unique pour représenter tous les caractères utilisés dans le monde, y compris les idéogrammes. Avec ce standard, nous pouvons encoder des documents multilingues. Ce code est utilisé notamment sur les processeurs de type Pentium.

Codage ASCII

Le code ASCII de base est un code sur 7 bits définissant ainsi un jeu de 128 caractères (Figure 4.11). Il est utilisé dans la plupart des ordinateurs personnelles et des stations de travail.

Le code ASCII comprend :

- Les 33 caractères de contrôle (codes 0 à 31 et 127) sont des caractères non imprimables. Ils permettent de faire des actions telles que :
  - CR : Carriage Return (retour chariot ou retour à la ligne)
  - BEL : Bell (caractère d'appel) : bip sonore
  - ESC : Escape (échappement)
- Les codes 65 à 90 représentent les majuscules de l'alphabet latin.
- Les codes 97 à 122 représentent les minuscules de l'alphabet latin (Il suffit d'ajouter 32 au code ASCII en base décimale pour passer de majuscules aux minuscules).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

FIGURE 4.11 – Code ASCII

- Le code 32 représente l'espace ou blanc (SP : Space).
- Les codes de 48 à 57 représentent les chiffres.

Extension du codage ASCII

Notez que le code ASCII a été mis au point pour coder la langue anglaise. Il permet aussi d'écrire des textes techniques de toutes les langues utilisant l'alphabet latin. Cependant, il ne contient pas des caractères accentués, ni de caractères spécifiques à une langue. C'est pourquoi, dans les années 1990, la norme ISO 8859 a été crée. En effet la norme ISO 8859 n'est qu'une extension du code ASCII sur 8 bits.

- Les 128 premiers caractères sont ceux d'ASCII.
- Les 128 suivants sont spécifiques à une langue.

La multitude des langues a entraîné 16 versions désignées par ISO 8859 – n, où n est un entier compris entre 1 et 16. Par exemple l'ISO 8859 – 6 est utilisé pour la langue arabe, alors que l'ISO 8859 – 1 ou l'ISO 8859 – 15 sont utilisés pour avoir les caractères accentués du français.

Représentation des constantes de type caractère

Une constante caractère possédant un graphisme (code ASCII de 32 à 126) est représentée par un seul caractère mis entre quottes : 'a', 'D', '3, '*', '!', ... En effet cette écriture permet de distinguer une constante caractère, d'une part d'une variable et d'autre part d'une constante numérique.

En particulier, le caractère espace (code ASCII 32) s'écrit de la même façon : `' '`, entre autre c'est un caractère comme les autres. L'apostrophe s'écrit conventionnellement en la doublant `''''`, ce qui fait un total de quatre apostrophes.

Opérateurs sur le type caractère

Le type caractère est un type scalaire, ses éléments sont ordonnées selon l'ordre des codes internes des caractères (ASCII). Il est possible, alors, d'appliquer :

- les opérateurs de relation : <, >, <>, =, <=, >=.
- Les opérateurs de succession : *succ* et *pred* renvoient respectivement le caractère successeur et prédécesseur s'ils existent des caractères fournis comme argument.
  - *succ* ('A') = 'B'
  - *pred*('C') ='B'

De plus, il est possible d'appliquer les opérateurs de conversion *chr* et *ord* :

- La fonction *chr* renvoie le caractère associé au code décimal fourni comme argument de *chr* (*chr*(65) renvoie le caractère 'A').
- La fonction *ord* renvoie le code machine en décimal correspondant au caractère fourni comme argument de *ord* (*ord*('C') retourne 67).

Exemples de déclaration

L'Exemple 4.11 montre la déclaration d'une constante caractère (mon_choix) ainsi que la déclaration d'une variable (ton_choix) de type caractère.

Exemple 4.11

Constante mon_choix = 'o'

Variable ton_choix : **car**

4.7.2 Type chaîne de caractères

Dans la Section 4.7.1, nous avons déjà vu que le type *car* permet de manipuler des caractères singuliers. Dans plusieurs applications, on aura besoin de manipuler des suites de caractères qui peuvent représenter des entités comme un prénom, une adresse électronique ou même un document texte.

Une solution consiste à utiliser un tableau de caractères. Cependant, les tableaux présentent des limitations en matière de leur longueur statique ainsi que leur rigidité vis-à-vis la manipulation des suites de caractères. D'où la nécessité de concevoir un autre type de données qui outrepasse ces limitations. Un tel type est baptisé *chaîne de caractères*.

Le type chaîne de caractères est désigné par le mot réservé *chaîne* dont le domaine est formé de l'ensemble de toutes les séquences possibles des caractères (y compris la séquence vide). Ces séquences sont mises entre quottes et peuvent être constituées de 0, 1 ou plusieurs caractères tel que illustré dans l'Exemple 4.12.

Exemple 4.12

- 'Celle-ci est une chaîne de caractères'
- 'D'
- ''

Opérateurs sur le type chaîne de caractères

- Les opérateurs de relation (<, >, <>, =, <=, >=) : Comme pour les caractères, la comparaison des chaînes de caractères se base sur l'ordre des codes internes des caractères. Deux chaînes de caractères son égales lorsqu'elles ont la même longueur et qu'elles sont constituées des mêmes suites de caractères.
- La concaténation : la fonction *concat* sert à juxtaposer deux chaînes de caractères. À titre d'exemple, la fonction *concat*('Université de ','Ghardaia') fournit la chaîne de caractère 'Université de Ghardaia'.
- La longueur : la fonction *longueur* fournit le nombre de caractères d'une chaîne de caractères. Par exemple, *longueur*('Université de Ghardaia') renvoie la valeur entière 22.

Exemples de déclaration

L'Exemple 4.13 montre la déclaration d'une constante chaîne de caractère (message) ainsi que la déclaration d'une variable (nom) de type chaîne de caractères.

Exemple 4.13

Constante message = 'J''apprends la programmation'

Variable nom : **chaîne**

4.8 Coin langage Pascal

4.8.1 Tableaux à une dimension

Le mot réservé *array* permet de déclarer un tableau de données de même type. Pour déclarer des variables de type tableau, nous pouvons procéder comme suit :

1. Déclarer un type tableau,

```
type nom_type = array[type_indices] of type_composant;
```

où *type_indices* est un type ordinal quelconque et *type_composant* est le type des données rangées dans le tableau. Il peut être quelconque, y compris le type tableau (sauf le type fichier qui sera discuté dans la deuxième partie de la série d'ouvrages *Algorithmique et structures de données*).

Exemple 4.14

```
type vecteur = array[1..50] of real;
```

Dans l'exemple 4.14, *type_indices* = 1..50 est de type intervalle qui est un type ordinal.

2. Déclarer une variable de type tableau.

Dans cette deuxième étape, on utilise le type tableau, comme celui déclaré dans l'étape précédente.

```
var nom_var1, nom_var2, ... : type nom_type_tableau;
```

Exemple 4.15

```
var X, Y : vecteur;
```

Une autre manière de déclarer un tableau, consiste à procéder directement comme suit :

```
var nom_tableau : array[type_indices] of type_composant;
```

Exemple 4.16

```
var X, Y : array[1..50] of real;
```

L'Exemple 4.17 montre une déclaration d'un tableau dont *type_indices* comme type énuméré.

Exemple 4.17

```
type jour = (samedi, dimanche, lundi, mardi, mercredi,
   jeudi, vendredi)
var recette_jour : array[jour] of real;
```

Notez que l'espace mémoire réservé pour un tableau est constitué d'emplacements mémoire contigus. L'adresse la plus basse correspond au premier élément et l'adresse la plus haute au dernier élément.

4.8.2 Tableaux à deux dimensions

Les tableaux à deux dimensions peuvent être déclarés de deux façons :
Soit comme un tableau de tableau,

```
array[type_indice1] of array [type_indice2] of
type_composant
```

ou comme une matrice.

```
array[type_indice1, type_indice2] of type_composant
```

Ces deux façons de déclaration des tableau à deux dimensions sont illustrées par l'Exemple 4.18.

Exemple 4.18

```
var notes_mat1 : array[1..30] of array [1..8] of real;
var notes_mat2 : array[1..30, 1..8] of real;
```

Ainsi, il est possible d'accéder à un élément du tableau de deux façons :

```
nom_tableau [indice1] [indice2]
```

ou

```
nom_tableau [indice1, indice2]
```

Notez qu'il est possible de généraliser la notion des tableaux à deux dimensions pour des tableaux à multiples dimensions.

4.8.3 Chaînes de caractères (strings)

La déclaration d'une chaîne de caractères se fait à l'aide du mot réservé *string*.

```
var nom_var : string[n];
```

Dans ce cas on limite la taille de la chaîne à n caractères, avec n un entier entre 1 et 255.

Exemple 4.19

```
var nom, prenom : string[30] ;
```

La déclaration de l'Exemple 4.19 signifie que les variables chaînes de caractères *nom* et *prenom* peuvent comporter au maximum 30 caractères.

Une autre variante de déclaration des chaînes de caractères consiste à omettre la spécification de la taille maximale des chaînes de caractères .

```
1    var nom_var : string;
```

Dans une telle situation, la variable chaîne de caractères peut avoir une longueur par défaut de 255 caractères.

Fonctions chaînes de caractères

La Table 4.2 décrit les fonction prédéfinies en turbo pascal portant sur les chaînes de caractères.

Table 4.2 – Fonctions chaînes de caractères.

Fonction	Description	Exemple
length(S : string) : Integer	Renvoie la longueur effective d'une chaîne.	Writeln('Longueur = ', length(S));
concat(S1, [, S2, ..., SN] : string) : string	Concaténation de plusieurs chaînes de caractères.	S := concat('Info', 'ASD1');
pos(SS, S : string) : byte	Recherche la chaîne *SS* dans la chaîne *S* et renvoie la valeur entière représentant la position du premier caractère de *SS* dans *S*. Si *pos* ne trouve pas *SS*, elle renvoie zéro.	if pos(' ', S) > 0 then ... ;
copy(S : string, index : integer, count : integer) : string	Renvoie une partie d'une chaîne de caractères de longueur *count* à partir de la position *index* de la chaîne *S*.	S := copy(S, 2, 3);

Procédures chaînes de caractères

La Table 4.3 décrit les procédures prédéfinies en turbo pascal portant sur les chaînes de caractères.

Table 4.3 – Procédures chaînes de caractères.

Procédure	Description	Exemple
delete(var S : string ; index : integer ; count : integer)	supprime une partie d'une chaîne de caractères de longueur *count* à partir de la position *index* de la chaîne *S*. Si la chaîne résultante a plus de 255 caractères, elle est tronquée après le 255ème caractère.	Writeln(delete(S,4,5)) ;
insert(source : string ; var S : string ; index : integer)	Insère la sous chaîne *source* dans la chaîne *S* en position *index*.	insert('max ', S, 10)) ;
str(X [: taille [: nbdec]] ; var S : string) : string	Convertit une valeur numérique X (entier ou réel) en sa représentation sous forme de chaîne de caractères selon les paramètres *taille* (nombre total de caractères dans la chaîne retournée) et *nbdec* (nombre de décimales à retourner). L'effet est identique à la procédure *write* avec les mêmes paramètres, sauf que le résultat est rangé dans *S* au lieu d'être écrit dans un fichier texte.	str(425.12, S) ; str(425.12 :5 :2,S) ;
val (S : string ; var v ; var code : integer)	Convertit la chaîne de caractères *S* représentant un nombre en sa valeur numérique et range le résultat dans **v** (entier ou réel). Si la chaîne *S* est invalide, l'indice de caractère en cause est rangé dans *code* ; sinon, *code* est mis à zéro.	val ('12564', v, err) ; if (err > 0) then ...else ... ;

4.9 Coin langage C

4.9.1 Tableaux à une dimension

syntaxe

```
type Identificateur [taille];
```

où *taille* est une expression constante positive indiquant le nombre d'éléments du tableau. Un tableau est toujours indicé de 0 à *taille -1*.

Contrairement au langage Pascal, le langage C n'offre aucune possibilité d'affectation globale des tableaux.

Notez, Il est possible de procéder à une initialisation du tableau lors de sa déclaration comme suit :

```
type Identificateur [taille] = {V1, V2, ..., Vn};
```

L'exemple 4.20 présente différentes manières d'initialisations des tableaux.

Exemple 4.20

```
#define N 4
main()
{
    int T1[N] = {4 , 10, -1, 35};
    float T2[N] = {1.6};
    char T3[N] = {'A', , ,'D'};
    int T4[] = {12, 0, 4};
}
```

Tous les éléments du tableau *T1* sont initialisés. Seul le premier élément du tableau *T2* est initialisé. Pour le tableau *T3*, le premier et le dernier élément sont initialisés. En ce qui concerne *T4*, il s'agit d'une création et une initialisation d'un tableau de trois éléments.

La notation suivante est utilisée pour accéder à un élément d'un tableau à une dimension :

```
    T1[0] = T4[1];
    T2[2] = 3.0;
    ...
```

4.9.2 Tableaux à deux dimensions

Comme dans le cas de plusieurs langages de programmation, C permet de définir des tableaux à multiple dimensions. À titre d'exemple, la déclaration d'un tableau à deux dimensions se fait comme suit :

```
    type Identificateur [taille_lignes][taille_colones];
```

Il est aussi possible d'initialiser un tableau à plusieurs dimensions tel que illustré dans l'Exemple 4.21.

Exemple 4.21

```
#define N 4
#define M 4
int tab[N][M] ={{0,1,2,3},{4,5,6,7}, {8,9,10,11}};
```

Un élément d'un tableau est référencé en spécifiant les indices des dimensions comme indiqué dans les notations suivantes :

```
    tab[3][4] = 12;
    ...
    tab[i][j] = tab[0][0];
    ...
```

4.9.3 Chaînes de caractères

Le langage C ne dispose pas de d'un véritab type *chaîne de caractères*. En effet, Une chaîne de caractères est un tableau de caractères à une dimension qui bénéficie de certains traitements particuliers.

La syntaxe de déclaration et éventuellement d'initialisation d'une chaîne de caractères est comme suit :

```
    char Identificateur[taille] = "texte\0";
```

Exemple 4.22

```
char nom[20];
char prenom[20] = "texte\0";
```

Notez qu'il est également possible de déclarer une chaîne de caractères comme un *pointeur* sur *char* (Les pointeurs seront étudiés dans le deuxième ouvrage *Algorithmique et structure de données - Partie 2*).

L'accès à un caractère particulier d'une chaîne de caractères se fait de la même manière que l'accès à un élément d'un tableau. À titre d'exemple *prenom[0] = 'F'*.

Pour la manipulation des chaîne de caractères, nous avons déjà vu (Section 3.8.6) les fonctions *printf* et *scanf* ainsi que les spécificateurs de conversions.

En outre, le fichier en-tête *string.h* contient des déclarations permettant la manipulation des chaînes de caractères tel que illustré dans la Table 4.4.

Table 4.4 – Fonctions de base de manipulation des chaînes de caractères en C.

Fonction	Type résultat	Description
strlen(s)	entier	Renvoie la longueur de la chaîne.
strcpy(s, t)	chaîne	Copie la chaîne t dans la chaîne s et renvoie s.
strcat(s, t)	chaîne	Ajoute la chaîne t à la fin de la chaîne s et renvoie s.
strcmp(s, t)	entier	Compare les deux chaînes s et t. Renvoie 0 si $s = t$, une autre valeur sinon.
strncpy(s, t, n)	chaîne	Copie au plus n caractères de t vers s et renvoie s.
strncat(s, t, n)	chaîne	Ajoute au plus n caractères de t à la fin de s et renvoie s.

4.10 Exercices

Exercise 4.1 Quels résultats fournira cet algorithme ?

Algorithme Tab1

```
Variable nombre : tableau[1..5] de entier
         i : entier

début
    Pour i de 1 à 5 faire
        nombre[i] ← i * i
    Pour i de 1 à 5 faire
        Écrire (nombre[i])
fin
```

Indications Dans ce type de questions, on est demandé de faire la trace de l'algorithme.

Exercise 4.2 Quels résultats fournira cet algorithme lorsqu'on lui fournit en données les valeurs : 2, 5, 3, 10, 4 et 2 ?

```
Algorithme Tab2
  Variable c : tableau[1..6] de entier
           i : entier

  début
      Pour i de 1 à 6 faire
          Lire (c[i])
      Pour i de 1 à 6 faire
          c[i] ← c[i] * c[i]
      Pour i de 1 à 3 faire
          Écrire (c[i])
      Pour i de 4 à 6 faire
          Écrire (2 * c[i])
  fin
```

Indications Vous êtes demandés de faire la trace de l'algorithme.

Exercise 4.3 Que fournit cet algorithme ?

```
Algorithme Tab_Suite
  Variable suite : tableau[1..8] de entier
           i : entier

  début
      suite[1] ← 1
      suite[2] ← 1
      Pour i de 3 à 8 faire
          suite[i] ← suite[i-1] + suite[i-2]
      Pour i de 1 à 8 faire
          Écrire (suite[i])
  fin
```

Indications Dans ce type de questions, on est demandé de faire la trace de l'algorithme.

Exercise 4.4 Écrire un algorithme qui remplit un tableau aléatoirement, puis l'afficher.

Indications Vous pouvez supposer avoir une fonction *aléaoire(n)* qui fournit un entier compris dans l'intervalle $[0, n[$.

Exercise 4.5 Écrire un algorithme qui détermine la valeur maximale et sa position dans un tableau de taille N.

Exercise 4.6 Pour un tableau d'entiers de taille N, écrire un algorithme qui calcule les valeurs suivantes :

- Le nombre des nombres positifs dans le tableau.
- Le nombre des nombres négatifs.
- La moyenne des nombres positifs.
- La moyenne des nombres négatifs.

Indications Attention à la division par zéro.

Exercise 4.7 Écrire un algorithme qui déclare et remplisse un tableau contenant les six voyelles de l'alphabet latin.

Exercise 4.8 Écrire un algorithme qui lit un caractère et qui indique s'il s'agit d'une voyelle, en utilisant un tableau contenant les 6 voyelles de l'alphabet.

Indications C'est une suite de l'Exercice 4.7.

Exercise 4.9 Écrire un algorithme qui détermine le nombre d'occurrence d'un caractère donné dans un tableau de caractères.

Exercise 4.10 Écrivez un algorithme constituant un tableau (Table 4.7), à partir de deux tableaux (Tables 4.5 et 4.6) de même longueur préalablement saisis. Le nouveau tableau sera la somme des éléments des deux tableaux de départ.

Table 4.5 – Tableau 1.

5	**8**	**7**	**9**	**1**	**5**	**4**	**6**

Table 4.6 – Tableau 2.

7	**6**	**5**	**2**	**1**	**3**	**7**	**4**

Table 4.7 – Tableau à constituer.

11	**14**	**12**	**11**	**2**	**8**	**11**	**10**

Exercise 4.11 Quels seront les résultats fournis par ce programme ?

```
Algorithme mat1
   Variable mat : tableau[1..8 , 1..2] de entier
              k, m : entier

   début
          Pour k de 1 à 4 faire
               Pour m de 1 à 2 faire
                    mat[k, m] ← k + m
          Pour k de 1 à 4 faire
               Pour m de 1 à 2 faire
                    Écrire (mat[k, m])
   fin
```

Exercise 4.12 Soit la déclaration :

Variable x : **tableau**[1..2 , 1..3] **de entier**

Écrire un algorithme qui lit 6 valeurs pour le tableau **x**, en les demandant « ligne par ligne » et qui les réécrit, « colonne par colonne », comme dans :

donnez les valeurs de la ligne numéro 1
5 9 7
donnez les valeurs de la ligne numéro 2
8 10 3
voici la colonne numéro 1
5
8
voici la colonne numéro 2
9 10
voici la colonne numéro 3
7
3

Exercise 4.13 Écrire l'algorithme permettant de déterminer la position du plus grand élément d'un tableau à deux dimensions. Plus précisément, on s'arrangera pour obtenir, dans des variables entières nommées *imax* et *jmax*, les valeurs des deux indices permettant de repérer ce plus grand élément.

Exercise 4.14 Soit un tableau d'entiers, de N lignes et P colonnes. Écrire un algorithme qui permet de remplir ce tableau, et de calculer la moyenne de ses valeurs.

Exercise 4.15 Écrire un algorithme qui détermine la transposée d'une matrice $(N * P)$.

Indications La matrice transposée (ou la transposée) d'une matrice est la matrice obtenue en échangeant les lignes et les colonnes de la matrice d'entrée.

Exercise 4.16 Écrire un algorithme qui retire les blancs d'une phrase donnée

Exercise 4.17 Le mot miroir d'un mot donné est le mot obtenu en lisant le mot donné à partir de la fin (exemple : *emhtirogla* est le mot miroir de *algorithme*). Écrire un algorithme donnant le mot miroir d'un mot donné.

Exercise 4.18 Écrire un algorithme qui cherche un mot palindrome dans un tableau de caractères. Un mot palindrome est un mot qui se lit aussi bien à l'endroit qu'à l'envers. (Exemple : *laval*, *été*)

Exercise 4.19 Écrire un algorithme qui lit une phrase (caractère par caractère) se termine par un point et qui déterminer et affiche l'ensemble des caractères chiffres qui apparaissent dans cette phrase.

Exercise 4.20 Écrire un algorithme qui lit une phrase (caractère par caractère) se termine par un point et qui déterminer et affiche les lettres alphabétiques minuscules qui n'apparaissent pas dans cette phrase.

Exercise 4.21 Un texte peut cacher, dans l'ordre, les lettres d'un mot. Par exemple : *'Mounir est revenu ici'* contient le mot *'Merci'*. Écrire un algorithme qui détermine si un texte proposé contient ou non un mot donné.

5. Types personnalisés

Jusqu'à présent, dans les Chapitres 3 et 4, nous avons traité des types de données simples (entier, booléen, caractère et réel) et des types de données structurés (tableaux et chaînes de caractères).

Ces types de données étudiés ont deux facteurs communs. Le premier facteur est qu'ils sont prédéfinis. *Autrement dit, ils n'ont pas besoin d'être déclarés. Le second facteur commun est qu'ils sont* homogènes*; ils comportent un nombre fixes d'informations de même type.*

Cependant, dans les applications réelles, le concepteur d'algorithmes, outre les types prédéfinis, peut être amené à manipuler d'autres sortes d'informations, tels que les types énumérés, intervalles, enregistrements *et* ensembles. *La définition et la manipulation de ces types de données font l'objet du présent chapitre.*

Particulièrement, dans la Section 5.6, d'autres types de données spécifiques au langage C sont traités, à savoir, le type union *et les* champs de bits.

En plus des livres recommandés dans les chapitres précédents, l'étudiant peut trouver plus de détail sur les types personnalisés (énuméré, intervalle, enregistrement et ensemble) dans les ouvrages [Baba-Hamed and Hocine, 2006, Trigano, 1993].

5.1 Énumérations

Définition 5.1 Le type énuméré définit un ensemble ordonné de valeurs désignées par des identificateurs (de constantes) (256 au maximum). L'ordre est celui dans lequel les identificateurs ont été énumérés.

La déclaration d'un type énuméré suit la synatxe suivante :

Type nom_du_type = (élément_0, élément_1, . . . , élément_n-1)

Le rang d'une constante énumérée est déterminé par sa position dans la liste des identificateurs. Dans ce cas, *élément_0* est de rang 0.

Exemple 5.1
Type
Couleur = (Rouge, Vert, Bleu)
Jours = (Vendredi, Samedi, Dimanche, Lundi, Mardi, Mercredi, Jeudi)
Mois = (Janvier, Février, Mars, Avril, Mai, Juin, Aout, Septembre,
Octobre, Novembre, Décembre)

D'après les déclarations de l'Exemple 5.1, *Rouge*, *Vendredi* et *Janvier* sont des constantes respectivement de types *Couleur*, *Jours* et *Mois*.

La fonction *Ord* d'une constante de type énuméré renvoie le rang de cette constante. Par exemple *Ord*(Vert) renvoie 1.

5.2 Intervalles

Définition 5.2 Le type intervalle est une portion de l'intervalle des valeurs d'un type scalaire appelé type hôte (host type).
La définition d'un type intervalle consiste à spécifier les bornes de l'intervalle.

La déclaration d'un type intervalle se réalise comme suit :

Type nom_du_type = constante .. constante

Les deux constantes doivent être du même type scalaire , et constituent les bornes inférieures et supérieures de l'intervalle.

Exemple 5.2
Type
Lettre = 'a'..'t'
Jours_ouvrables = Dimanche..Jeudi
Num_mois = 1..12
Trimestre1 = Janvier..Mars

Dans les déclarations de l'Exemple 5.2, le type hôte du type intervalle *Lettre* est le type scalaire *caractère* et la constante 'a' est inférieure ou égale à 't'. De même, le type hôte du type intervalle *Jours_ouvrables* est le type scalaire *Mois* défini dans l'Exemple 5.1 et la constante *Janvier* est inférieure ou égale à *Mars*.

5.3 Enregistrements

Définition 5.3 Un enregistrement est un ensemble d'informations de types différents, appelés champs, accessibles individuellement ou collectivement en lecture et en écriture.

Exemple 5.3

Une date est composée :

- d'un jour (1 à 31),
- d'un mois (1 à 12),
- d'une année (0 à 9999).

Un employé est défini par un ensemble de renseignements :

- nom, prénom (chaîne de caractères),
- date naissance (date)
- sexe (caractère ou booléen),
- nombre d'enfants (entier).

La forme générale de déclaration d'un enregistrement est la suivante :

Type nom_du_type = **enregistrement**
  nom_champ$_1$: $type_1$
  nom_champ$_2$: $type_2$
  ⋮
  nom_champ$_m$: $type_m$
  fin

L'Exemple 5.4 concrétise la déclaration des types *date* et *employé* de l'Exemple 5.3.

Exemple 5.4

Type Tdate = **enregistrement**
  jour : 1..31
  mois : 1..12
  annee : 0.9999
  fin

```
Type Temploye = enregistrement
    nom, prenom : chaîne
    date_nais : Tdate
    sexe : booleen
    nbr_enfants : entier
    fin
```

Ainsi, nous pouvons définir les variables aujourd_hui, demain, hier de type Tdate :

```
Variable aujourd_hui, demain, hier : Tdate
```

De la même manière, nous pouvons déclarer les variables employe_titulaire et personne de type Temploye :

```
Variable employe_titulaire, personne : Temploye
```

Après avoir déclaré une variable de type enregistrement, nous pouvons accéder à un champ en précisant le nom de l'enregistrement suivi du nom du (ou des) champ(s) séparé(s) par l'opérateur `.` (Exemple 5.5).

Exemple 5.5

```
hier.jour ← 2
demain.mois ← aujourd_hui.mois
employe_titulaire.nom ← 'Foulane'
personne.date_nais.annee ← 2020
```

5.4 Ensembles

Définition 5.4 Un ensemble est une collection non ordonnée d'éléments de même type dont le nombre est fini, sur lesquels on peut effectuer les opérations et les relations mathématiques classiques : réunion, intersection, complémentation, égalité, inclusion, appartenance, ...

Déclaration d'un type ensemble

Un type ensemble peut être défini à partir d'un type de base de ses éléments de la manière suivante :

Ainsi, une variable *vens* de type ensemble peut être déclaré comme suit :

Type nom_type_ensemble = **ensemble** de **type de base**

Variable vens : **nom_type_ensemble**
ou
Variable vens : **ensemble** de **type de base**

Une variable de type **nom_type_ensemble** peut prendre comme valeurs toutes les parties de **type de base**.

L'Exemple 5.6 illustre la déclaration des variables et des types ensemble.

Exemple 5.6

```
Type couleur = (bleu, rouge, jaune)          /* type énuméré */
     T_ens_c = ensemble de couleur           /* type ensemble */

Variable vert, noire : T_ens_c
Variable blanc, violet : ensemble de couleur
```

Ici, le type **couleur** est un type énuméré, il sert comme un type de base pour le type ensemble **T_ens_c**. Les variables vert, noire, blanc, violet sont de même type malgré qu'ils sont déclarés différemment. Ils peuvent prendre comme valeurs toutes les parties de **type de base**. Dans l'Exemple 5.6, ceci correspond à l'ensemble des couleurs possibles : [], [*bleu*],b[*rouge*], [*jaune*], [*bleu*, *rouge*], [*bleu*, *jaune*], [*rouge*, *jaune*], [*bleu*, *rouge*, *jaune*]. L'Exemple 5.7 montre comment peut-on affecter des valeurs à des variables de type ensemble.

Exemple 5.7

```
noir ← [bleu, rouge, jaune]
blan ← []                        /* [] représente l'ensemble vide */
violet ← [bleu, rouge]
vert ← [bleu, jaune]
```

Représentation physique d'un ensemble

Le stockage en mémoire d'une variable de type ensemble s'effectue dans un tableau contigu dans lequel la présence de chaque élément est marqué à l'aide d'un seul bit (1 pour un élément présent, 0 sinon).

Pour illustrer la représentation physique d'un type ensemble, prenant L'Exemple 5.8.

Exemple 5.8

```
Type qualité = (actif, intelligent, sportif, courageux, habille)
     personnalité = ensemble de qualité
```

Variable A : **personnalité**

La variable A est de type ensemble. Si $A = [intelligent, habille]$, sa représentation physique est illustrée par la Table 5.1.

Table 5.1 – Représentation physique d'une variable de type ensemble.

actif	intelligent	sportif	courageux	habille
0	1	0	0	1

Opérations sur les ensembles

Soient les ensembles $A = [1,3]$ et $B = [3,6]$, la table 5.2 présente les opérations possibles sur les ensembles.

Table 5.2 – Opérations sur les ensembles.

Symbole mathématique	Symbole algorithmique	Description	Exemple
$\cup$	+	Union	$A + B = [1,3,6]$
$\cap$	*	Intersection	$A * B = [3]$
- ou \	-	Différence	$A - B = [1]$
=	=	Égalité	$A = [1,3]$
$\neq$	<>	Inégalité	$A <> B$
$\subset$	<	Inclusion stricte	$[3] < B$
$\subseteq$	<=	Inclusion	$[3,6] <= B$
$\supset$	>	Contenance stricte	$A > [3]$
$\supseteq$	>=	Contenance	$A >= [3,1]$
$\in$	dans ou in	Appartenance	3 in B

5.5 Coin langage Pascal

5.5.1 Création d'un nouveau type

D'une manière générale, la syntaxe de déclaration d'un nouveau type est la suivante :

```
1   TYPE identificateur_1 , identificateur_2 , ... : Nom_type;
```

Cette déclaration est utilisée pour déclarer les types de données *identificateur_1*, *identificateur_2*, ..., comme illustré dans l'Exemple 5.9.

Exemple 5.9

```
1    TYPE    int1, int2, int3 : Integer;
2            car, caractere : Char;
```

5.5.2 Type énuméré

La syntaxe de déclaration d'un type énuméré est :

```
1    TYPE identificateur_enumere = (item1, item2, ...);
```

Voici quelques exemples de déclarations de types énumérés (Exemple 5.10) :

Exemple 5.10

```
1    TYPE    COLORS = (Red, Green, Blue, Yellow,
                       Magenta, Cyan, Black, White);
2
3            TRANSPORT = (Bus, Train, Airplane, Ship);
```

5.5.3 Type intervalle

La syntaxe pour déclarer un type intervalle est la suivante :

```
1    TYPE identificateur_intervalle = borne_inferieure.. borne
     superieure;
```

Les exemples suivants montrent quelques déclarations de types intervalle (Exemple 5.11) :

Exemple 5.11

```
1    TYPE    nombre = 1 .. 100;
2            sub_color = Green .. Black;
3            sub_trans = Bus .. Airplane;
```

5.5.4 Enregistrements

La déclaration d'un type enregistrement se réalise à l'aide du mot clé *record* :

```
TYPE    identificateur_record = record
        liste_ident_1 : type_1;
        liste_ident_2 : type_2;
        ...
        liste_ident_n : type_n
end;
```

Où *liste_ident* représente un ou plusieurs identificateurs séparés par des virgules.

L'Exemple 5.12 présente une déclaration de type enregistrement :

Exemple 5.12

```
TYPE    exp = record
        X, Y : real;
        B : boolean;
        S : string[10];
        T : array [1..10] of integer
end;
```

Pour simplifier l'accès aux champs d'un enregistrement, nous pouvons utiliser l'instruction *with* tel que illustré dans l'Exemple 5.13 :

Exemple 5.13

```
with exp do begin
    writeln ('X = ', X);
    writeln ('Y = ', Y);
    if B then
        S := 'masculin'
    else
        S := 'feminin';
end;
```

5.5.5 Ensembles

Le type ensemble est défini par le biais du mot réservé *set* comme suit :

```
TYPE    identificateur_set = set of type_base;
```

type_base ne doit pas contenir plus de 256 valeurs possibles.

Les variables de type ensemble se déclarent de deux manières. Soit en utilisant un type ensemble déjà déclaré :

```
var v_set1, v_set2, ... : identificateur_set;
```

Soit en impliquant le type ensemble dans la clause *var* elle même :

```
var v_set1, v_set2, ... : set of type_base;
```

Voici un exemple de déclaration d'un type ensemble :

Exemple 5.14

```
Type lettres = set of char;
var lettres_maj : lettres;
```

La déclaration de l'Exemple 5.14 peut être remplacée par celle de l'Exemple 5.15.

Exemple 5.15

```
var lettres_maj : set of char;
```

Les opérations portants sur les ensembles sont présentés dans la Table 5.2.

5.6 Coin langage C

5.6.1 Création d'un nouveau type

En C, la définition d'un type s'effectue en utilisant le mot clé *typedef* de la manière suivante :

```
typedef int entier;
```

typedef consiste à créer un *alias (synonyme)* ; ici le nouveau type *entier* peut être utilisé dans les déclarations, transtypages, ... de la même manière que le type *int* peut être utilisé.

Exemple 5.16

```
typedef int entier;
entier main()
{
    entier i = 20;
    ...
```

```
6         return 10;
7     }
```

5.6.2 Structures

Une structure (enregistrement) est définie en utilisant le mot réservé *struct :*

```
1   struct nom_structure
2   {
3         type_1 champ_1;
4         type_2 champ_2;
5         ...
6         type_n champ_n;
7   };
```

Notez la présence obligatoire d'un point-virgule à la fin de la définition d'une structure.

La déclaration d'une variable de type structure se fait comme suit :

```
1   struct nom_structure nom_variable;
```

L'Exemple 5.17 illustre la manière de déclarer des structures et des variables associées.

Exemple 5.17

```
1   struct point /* déclaration d'une structure */
2   {
3    int x;
4    int y;
5   };
6   main()
7   {
8         ...
9         struct point pt; /* déclaration d'une variable de
     type structure */
10        ...
11  }
```

Il est aussi possible d'initialiser une structure lors de sa déclaration d'une manière séquentielle ou éventuellement sélective (Exemple 5.18).

Exemple 5.18

```
struct date /* déclaration d'une structure */
{
 int jour;
 int mois;
 int annee;
};
main()
{
      /* initialisation séquentielle */
      struct date date_nais = {1, 1, 2000};
      /* initialisation sélective */
      struct date date_rec = {.annee = 2009, .jour =
27, .mois=12};
      /* initialisation mélange : séquentielle + sé
lective, la valeur 11 est affactée au champ mois */
      struct date date_sout = {3, .annee=2018};
}
```

L'accès à un champ d'une structure se réalise à l'aide de l'opérateur `.` comme suit :

```
variable.champ;
```

L'Exemple 5.19 éclaire l'opération d'accès à un champ.

Exemple 5.19

```
struct date /* déclaration d'une structure */
{
 int jour;
 int mois;
 int annee;
};
main()
{
      struct date date_nais;
      date_nais.jour = 1;
      date_nais.mois = 3;
      date_nais.annee = 1969;
}
```

Par ailleurs, il est possible de définir des champs occupant un nombre précis de bits. Ceci est valable pour les champs de type entier (*int* ou *unsigned int*).

L'utilisation des *champs de bits* est beaucoup plus utile lors de la programmation système qui nécessite de manipuler des registres particuliers de la machine.

À titre d'exemple, le registre d'état du *MC* 68060 peut être décrit en utilisant une structure à champs de bit comme suit :

```
struct state_reg
{
   unsigned int trace : 2;
   unsigned int priv : 2;
   unsigned int : 1;              /*   inutilisé   */
   unsigned int masque : 3;
   unsigned int : 3;              /*   inutilisé   */
   unsigned int extend : 1;
   unsigned int negative : 1;
   unsigned int zero : 1;
   unsigned int overflow : 1;
   unsigned int carry : 1;
};
```

5.6.3 Unions

Une *union* est un regroupement d'objets de type différents mais qui ne peut contenir qu'un seul de ses membres à la fois. À l'instar des types *struct*, la déclaration de ce type se fait en utilisant le mot-clé *union* :

```
   union nom_union
   {
    type_1 membre_1;
    type_2 membre_2;
    ...
    type_n membre_n;
   };
```

Exemple 5.20

```
   union jour
   {
        char lettre;
        int num;
   };
   main()
   {
```

```
8          union jour hier, aujourdhui, demain;
9          hier.lettre = 'L',
10         aujourdhui.num = 5;
11         demain.num = (aujourdhui.num + 2) % 7;
12     }
```

L'accès à un membre d'une union se fait à l'aide de l'opérateur `.` (Exemple 5.20).

5.6.4 Énumérations

Une énumération se définit par le biais du mot-clé *enum* comme suit :

```
1  enum nom_enum {const_1, const_2, ..., const_n};
```

Exemple 5.21

```
1      enum naturel1 {Zero, Un, Deux, Trois, Quatre, Cinq};
2      enum naturel2 n = Deux;
3      printf("n = %d.\n", (int)n);
4      printf("Quatre = %d.\n", Quatre);
```

En effet, la déclaration de l'Exemple 5.21 (instruction 1) crée deux choses, un type énuméré appelé *naturel1* et des constantes énumérées *Zero*, *Un*, ... codées par des entiers 0, 1, ... Par conséquent, les instructions 3 et 4 de l'exemple affichent le résultat :

```
1      n = 2.
2      Quatre = 4.
```

Cependant, il est possible de modifier le(s) valeur(s) de(s) constante(s) lors de la déclaration de type :

```
1      enum naturel3 {Six=6, Sept, Huit, Neuf, Dix};
```

5.7 Exercices

Exercise 5.1 La feuille de soins médicaux regroupe les renseignements suivants concernant l'assuré : nom, prénom, date et lieu de naissance, adresse personnelle, nom et adresse de l'employeur et le mode de paiement.

- Décrire ce type.

Indications Attention, vous travaillez ici avec un type composé de plusieurs entités appelées champs. Veillez à ce que les valeurs des champs soient atomiques (non décomposables).

Exercise 5.2 Un nombre complexe est défini par : $a + ib$.

- Lire deux nombres complexes et afficher leur produit.
- Comparer deux complexes.

Indications

- Pour lire un enregistrement, il faut procéder champ par champ.
- Dans la comparaison de deux nombres complexes, essayer de définir une relation d'ordre.

Exercise 5.3 Dans une bibliothèque, on dispose de 1000 fiches. Les informations que l'on pourrait trouver dans la fiche d'un ouvrage de la bibliothèque sont les suivants : Code, auteur (nom et prénom), titre, éditeur et année d'édition.

- Afficher les auteurs qui ont édité en l'an 2011 chez l'éditeur *BERTI*.

Indications Vous pouvez utiliser un tableau d'enregistrements. Pour accéder aux champs d'un enregistrement, utilisez l'opérateur point `.`.

Exercise 5.4 Un étudiant sera identifié par : son numéro d'étudiant, son nom, son prénom, sa date de naissance (qui sera décomposée en jour mois et année de type entier) et un tableau contenant les moyennes aux 5 unités d'enseignement préparées (tel que illustré dans la Table 5.3).

1. Écrire en algorithmique, en C et en Pascal les structures de données nécessaires à la définition d'un étudiant et d'une structure permettant la gestion de 1500 étudiants. (La date de naissance pourra être stockée dans une structure à part)
2. Écrire en algorithmique, en C et en Pascal les primitives suivantes :
   - Saisie d'un étudiant (remplissage des différents champs de la structure)
   - Affichage des renseignements concernant un étudiant
   - Remplissage du tableau d'étudiants avec 1 étudiant. On fera appel à Saisie_Etudiant.

- Affichage de tous les étudiants présents dans le tableau. On fera appel à Affiche_Etudiant.
- Affichage d'un étudiant recherché par son nom dans le tableau.

Table 5.3 – Fiche d'un étudiant

Numéro d'étudiant	10601234				
Nom étudiant	FOULANE				
Prénom étudiant	Elfoulani				
Date de naissance - jour naissance - mois naissance - année naissance	29 02 1988				
Tableau de notes	12	15	17	13	14

Indications La question n° 1 s'intéresse aux données. Ici nous avons besoin d'un type composé. Donc nous utilisons les *enregistrements* et la technique d'imbrication des enregistrements.
La question n° 2 s'intéresse aux traitements (traitement d'un seul étudiant sans se préoccuper du tableau puis le traitement d'un ou plusieurs étudiants dans le tableau des étudiants.)

Exercise 5.5 Écrire une fonction qui détermine si un caractère est une lettre, un chiffre, un blanc (espace), un caractère de ponctuation, ou un autre type de caractère.

Indications Vous pouvez utiliser le type *ensemble*.

Exercise 5.6 Écrire une fonction qui détermine si une couleur donnée est contenue dans le drapeau national ou non, en testant l'appartenance à une suite de couleurs choisies.

Indications Vous pouvez utiliser le type *ensemble* et le type *énuméré*.

Exercise 5.7 Un groupe de personnes est numéroté de 1 à N. Écrire les déclarations des variables PERSONNES_AGES, HOMME, FUMEURS dont la valeur est un ensemble de personnes de ce groupe. En supposant que les valeurs appropriées ont été affectées à ces variables, écrire les expressions permettant d'obtenir des ensembles dont les valeurs sont :

- Le groupe complet des personnes.
- Les non fumeurs.
- Les hommes âgés qui fument.
- Toutes les personnes âgées plus les hommes fumeurs.

Indications Utilisez le type *ensemble* ainsi que les opérations sur les ensembles (union, intersection, différence, . . .).

Bibliographie

[Baba-Hamed and Hocine, 2006] Baba-Hamed, H. and Hocine, S. (2006). *Algorithmique et structures de données statiques : cours et exercices avec solutions*. Office des Publications Universitaires, Alger.

[Bessaa, 2018] Bessaa, B. (2018). *Exercices corrigés d'Algorithmique*. Les Fascicules du LMD. Pages Bleues, Alger.

[Borland, 1991a] Borland (1991a). *Turbo Pascal : Guide de référence*. Borland International Inc., France.

[Borland, 1991b] Borland (1991b). *Turbo Pascal : Guide du programmeur*. Borland International Inc., France.

[Cormen, 2013] Cormen, T. H. (2013). *Algorithmes : notions de bases*. DUNOD.

[Cormen et al., 2010] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Cazin, X., and Kocher, G.-L. (2010). *Algorithmique : cours avec 957 exercices et 158 problèmes*. Sciences sup. Dunod, Paris.

[Delannoy, 2007] Delannoy, C. (2007). *Programmer en Turbo Pascal 7*. Eyrolles.

[Delannoy, 2008] Delannoy, C. (2008). *S'initier à la programmation : Avec des exemples en C, C++, C#, Java et PHP*. Eyrolles.

[Delannoy, 2009] Delannoy, C. (2009). *Programmer en langage C : avec exercices corrigés / Claude Delannoy*. Eyrolles, Paris, 5e édition edition.

[Haro, 2015] Haro, C. (2015). *Algorithmique raisonner pour concevoir*. DataPro. ENI.

[Kernighan and Ritchie, 1978] Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*. Prentice-Hall, Inc., USA.

[Laurent and Ayel, 1985] Laurent, J.-P. and Ayel, J. (1985). *exercices commentés d'analyse et de programmation*. Dunod.

[Richard and Richard, 1985] Richard, C. and Richard, P. (1985). *initiation à l'ALGORITHMIQUE : 135 exercices corrigés.* Collection DIA. Belin.

[Sedgewick and Wayne, 2011] Sedgewick, R. and Wayne, K. (2011). *Algorithms, 4th Edition.* Addison-Wesley.

[Tormento and Boutin, 1989] Tormento, S. and Boutin, S. (1989). *Algorithmes cours et exercices*. BTS informatique de gestion. Bréal, Paris.

[Trigano, 1993] Trigano, P. (1993). *Méthodologie de la Programmation & Management des Informations*. BERTI, Alger.

[Zegour, 2013a] Zegour, D. (2013a). *Apprendre et enseigner l'algorithmique (Tome 1) : Cours et annexes*. Number vol. 1. Éditions Universitaires Européenes.

[Zegour, 2013b] Zegour, D. (2013b). *Apprendre et enseigner l'algorithmique (Tome2) : Sujets d'examen corrigés Exercices programmés en PASCAL*. Number vol. 2. Éditions Universitaires Européenes.

Annexes

A. Examens partiels avec corrigés types

A.1 Examen partiel 2015-2016 - Semestre 1

Exercice 1 : (6 points)

Concevoir un algorithme qui lit une valeur entière n, entrée par l'utilisateur, puis imprime le triangle suivant. Les différents nombres sont le résultat de la multiplication du numéro de ligne par le numéro de colonne, lorsque le numéro de colonne est inférieur ou égal au numéro de ligne.

1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
⋮
n $2n$ $3n$ $4n$ $5n$ $6n$ $7n$ ... n^2

Exercice 2 : (6 points)

Soit **I** = **[2,3[U]0,1] U [-10,-2]** dans l'ensemble des réels.

1. Écrivez une expression vérifiant l'appartenance d'une variable **x** à l'intervalle **I**.

2. Réécrire l'expression précédente en n'utilisant que les opérateurs relationnels < et =. Tous les opérateurs logiques sont, par contre, autorisés.

3. Écrivez un programme Pascal qui :

   - demande à l'utilisateur d'entrer un réel ;
   - enregistre la réponse de l'utilisateur dans une variable **x** de type réel ;
   - teste l'appartenance de **x** à l'ensemble **I**, tel que exprimé dans la $2^{ème}$ question, et affiche le message '**x** appartient à **I**' si c'est le cas, et '**x** n'appartient pas à **I**' dans le cas contraire.

Exercice 3 : (4 points)

Soit le code suivant :

```
Algorithme boucle_tantque
Variable k, m : entier

début
    k ← 11010
    m ← k mod 2
    tant que (k > 0) faire
    début
        m ← (m + k mod 10) * 2
        k ← k div 10
    fin
    Écrire (m)
fin
```

- Dérouler l'algorithme, qu'affiche t-il ?

Exercice 4 : (4 points)

Réaliser les conversions suivantes avec une précision de 4 :

$(101011101)_2 = (?)_{10}$

$(A9C)_{16} = (?)_{10}$

$(2454,46)_8 = (?)_{10}$

$(1000110011)_2 = (?)_{16}$

$(456)_8 = (?)_2 = (?)_{16}$

$(15.6)_{10} = (?)_2$

A.2 Examen partiel 2015-2016 - Semestre 1 : corrigé type

Exercice 1 : (6 points)

Algorithme imp_triangle

```
Variable i, j, n : entier

début
    Écrire ('Introduire un entier n : ')
    Lire (n)
    Pour i de 1 à n faire
    début
        Écrireln()
        Pour j de 1 à i faire
            Écrire (i*j, ' ')
    fin
fin
```

Exercice 2 : (6 points) 1.25 + 2.5 + 2.25

1. **I = (x >=2 et x <3) ou (x>0 et x <=1) ou (x >= -10 et x <= -2)**

2. **I = (non (x<2) et x <3) ou (non (x<0 ou x = 0) et (x <1 ou x = 1)) ou (non (x < -10) et (x <-2 ou x = -2))**

3.

```
1  Program  intervalle;
2  Var  x : Real;
3  begin
4    write ('SVP, Introduire un nombre réel :');
5    read(x);
6    if (non (x<2) et x <3) ou (non (x<0 ou x = 0) et (x <1 ou x
       =1)) ou (non (x< -10)  et (x <-2 ou x = -2)) then
7      write(x, '  appartient à I')
8    else
9      Ecrire('x n''appartient à I');
10 end./* fin du programme */
```

Exercice 3 : (4 points)

1. Dérouler l'algorithme, qu'affiche t-il ?

k	m	Affichage
11010	0	
1101	0	
110	2	
11	4	
1	10	
0	22	
		22

Exercice 4 : (4 points)

Réaliser les conversions suivantes avec une précision de 4 :

$(101011101)_2 = (349)_{10}$

$(A9C)_{16} = (2716)_{10}$

$(2454,46)_8 = (1324,59375)_{10}$

$(1000110011)_2 = (233)_{16}$

$(456)_8 = (100101110)_2 = (12E)_{16}$

$(15.6)_{10} = (1111,1001)_2$

A.3 Examen partiel 2019-2020 - Semestre 1

Exercice 1 : (Trouver les erreurs, 4 points)

Il a été demandé à un étudiant de la 1MI d'écrire un algorithme qui liste **les nombres naturels pairs inférieurs à** 1000. L'algorithme ci-dessous est fournit par le candidat.

Algorithme Liste de pairs

```
variable i : réel
         t : car
constante N = 1000
Pour i de 0 à N faire

début
    t := i mod 2 = 0
    si t alors Écrire ('i')
fin
```

1. Repérer le maximum d'erreurs dans cet algorithme et proposer les corrections nécessaires.

Exercice 2 : (Compter les répétitions, 6 points)

On désire compter le nombre de répétitions dans un flot d'entiers positifs lus au clavier. **Une répétition est à signaler (compter) lorsque deux entiers consécutifs lus sont égaux**. On admettra que la saisie est terminée à la lecture d'un entier négatif. **Exemples :**

- **1 3 2 2 5 9 9 7 -1 $\Rightarrow$ 2 répétitions**
- **14 6 6 6 6 4 -3 $\Rightarrow$ 3 répétitions**
- **1 2 3 -5 $\Rightarrow$ 0 répétitions**

1. Écrire l'algorithme permettant de compter le nombre de répétitions en supposant que l'utilisateur introduise toujours une suite correcte de nombres positifs suivis d'un nombre négatif à la fin (**pas de contrôle de saisie**).

Exercice 3 : (Estimer $cos(x)$, 10 points)

La formule donnant le *cosinus* d'un angle x en radians est la suivante :

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

1. Proposer un algorithme qui calcule le cosinus d'un angle réel x en radians fournit en entrée. **Le calcul s'arrête au terme numéro 100**.

   Assistance :

- La formule générale du *cos* est

$$cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

- Commencer par remarquer la différence entre deux termes successifs différents de 1.
- Les signes des termes alternent (+, -, +, -,...)
- $(n+2)! = (n+2)(n+1)n!$

A.4 Examen partiel 2019-2020 - Semestre 1 : corrigé type

Exercice 1 : (Trouver les erreurs, 4 points)

```
Algorithme Liste_de_pairs
constante N = 1000
variable i : entier
         t : booléen

début
    Pour i de 0 à N faire
    début
        t ← i mod 2 = 0
        si t alors Écrire (i)
    fin
fin
```

Exercice 2 : (Compter les répétitions, 6 points)

```
Algorithme N_répétitions
variable NB1, NB2, Cpt : entier

début
    Lire(NB1)
    Cpt ← 0
    tant que NB1 ¿= 0 faire
    début
        Lire(NB2)
        Si (NB1=NB2) Alors Cpt ← Cpt+1
        NB1 ← NB2
    fin
    Écrire(Cpt,'Répétitions')
fin
```

Exercice 3 : (Estimer $cos(x)$, 10 points)

Algorithme Cosinus

```
variable x, tmp, cos : réel
         i entier

début
    Lire(x)
    tmp ← 1
    cos ← 1
    Pour i de 2 à 198 pas 2 faire
    début
        tmp ← tmp * (−1) * x * x/((i − 1) * i)
        cos ← cos + tmp
    fin
    Écrire('cos( ',x,' )= ',cos)
fin
```

B. Examens finals avec corrigés types

B.1 Examen final 2014-2015 - Semestre 1

Exercice 1 : (2.5 points)

Que réalise les fragments de code suivants :

```
1 for i:=1 to 10 do;
2   Writeln('Essalamou aleikoum');
```

Listing B.1 – Fragment du code n°1.

```
1 for i=1 to 10 do
2   Writeln('Essalamou aleikoum');
```

Listing B.2 – Fragment du code n°2.

Exercice 2 : (4 points)

Soit l'algorithme suivant :

Algorithme mystère

```
Variable n,res : entier

début
        Lire (n)
        res ← 1
        Pour i de 2 à n faire
              res ← res * i
        Écrire (res)
fin
```

1. Dérouler l'algorithme pour $n = 3$ ensuite pour $n = 4$.

2. Que fait l'algorithme ?

Exercice 3 : (6 points)

On rappelle que la somme des **n** premiers nombres impairs est égale au carré de **n**. On en déduire un algorithme pour calculer et afficher la racine carrée entière d'un nombre entier positif donné. Exemple : ($1 + 3 + 5 = 9$, nous avons les 3 premiers nombres impairs dont la somme est égale à $3^2 = 9$).

Exercice 4 : (7.5 points)

1. Écrire un programme Pascal qui affiche les valeurs 1 à 9 en ligne, à l'aide d'une boucle **for** :

```
123456789
```

2. Modifier le programme pour qu'il affiche 9 lignes similaires, à l'aide de 2 boucles **for** :

```
123456789
123456789
...
123456789
```

3. Comment modifier le programme pour qu'il affiche un triangle ?

```
1
12
123
1234
12345
123456
```

```
1234567
12345678
123456789
```

4. Modifier une dernière fois le programme, pour qu'il affiche une pyramide inversée :

```
        1
       12
      123
     1234
    12345
   123456
  1234567
 12345678
123456789
```

B.2 Examen final 2014-2015 - Semestre 1 : Corrigé type

Exercice 1 : (2.5 points)

- Le fragment du code du listing B.1 boucle 10 fois puis affiche le message 'Es-salamou aleikoum' à l'écran.

- Pour le fragment du code du listing B.2, il existe une erreur à la compilation : l'opérateur = en place de l'affectation := dans l'instruction for.

Exercice 2 : (4 points)

1. Déroulement de l'algorithme mystère pour $n = 3$ ensuite pour $n = 4$:

n	**i**	**res**	**Affichage**
3	?	1	
3	2	2	
3	3	6	
			6

n	**i**	**res**	**Affichage**
4	?	1	
4	2	2	
4	3	6	
4	4	24	
			24

2. L'algorithme mystère calcule la factorielle de n.

Exercice 3 : (6 points)

Algorithme racine_entiere

```
Variable somme, racine , n : entier

début
    Écrire ('Introduire un nombre entier pour en calculer la racine entière :')
    somme ← 1
    racine ← 0
    tant que (somme <= n) faire
    début
        racine ← racine + 1
        somme ← somme + 2*racine + 1
    fin
    Écrire ('La racine entière de ' , n , 'est = ', racine)
fin
```

Exercice 4 : (7.5 points) 1 + 1.5 + 2.5 + 2.5

1. Écrire un programme Pascal qui affiche les valeurs 1 à 9 en ligne, à l'aide d'une boucle **for** :

```
123456789
```

```
for i:= 1 to 9 do
    write(i);
```

2. Modifier le programme pour qu'il affiche 9 lignes similaires, à l'aide de 2 boucles **for** :

```
123456789
123456789
...
123456789
```

```
for i:= 1 to 9 do
begin
    for j:= 1 to 9 do
        write(j);
    writeln;
end;
```

3. Comment modifier le programme pour qu'il affiche un triangle ?

```
1
```

```
12
123
1234
12345
123456
1234567
12345678
123456789
```

```
for i:= 1 to 9 do
begin
    for j:= 1 to i do
        write(j);
    writeln;
end;
```

4. Modifier une dernière fois le programme, pour qu'il affiche une pyramide inversée :

```
        1
       12
      123
     1234
    12345
   123456
  1234567
 12345678
123456789
```

```
for i:= 1 to 9 do
begin
    for j:= 9 downto 1 do
     begin
        if (i>=j) then
            write(i-j+1)
  else
      write(' ');
     end;
    writeln;
end;
```

ou bien

```
for i:= 9 downto 1 do
begin
```

```
3       for j:= 1 to 9 do
4        begin
5           if (j>=i) then
6              write(j-i+1)
7     else
8         write(' ');
9        end;
10      writeln;
11  end;
```

B.3 Examen final 2016-2017 - Semestre 1

Exercice 1 : (5 points)

Un photocopieur assure ses services aux prix de 5 DA la photocopie ; si le nombre de feuilles dépasse 30, il applique une réduction de 15% au prix total et si le nombre de feuilles dépasse 100, il applique une réduction de 25

1. Écrire un algorithme qui calcule le montant des photocopies pour un client qui a photocopié **NBF** feuilles. (**NBF** est une donnée introduite par l'utilisateur)
2. Réécrire l'algorithme pour **N** clients. (**N** est une donnée introduite par l'utilisateur)

Exercice 2 : (5 points)

Soit le programme Pascal suivant (pour des valeurs de x strictement positives) :

```
1  program test;
2  var  x,s,i : integer;
3    b : boolean;
4  begin
5    read(x);    (* x > 0*)
6    s := 0;
7    for i:= 1 to x div 2 do
8      if (x mod i = 0) then s := s+i;
9    if (x=s) then b := true
10   else b := false;
11   write(b);
12 end.
```

1. Dérouler ce programme pour $x = 6$ et $x = 9$.
2. À quoi correspondent les valeurs *true* et *false* de b et déduire ce que fait le programme.

Exercice 3 : (5 points)

Étant donné X et n deux entiers. Écrire un algorithme qui permet de calculer la somme suivante : $S = 1 - X + 2X^2 - 3X^3 + 4X^4 + \ldots + (-1)^n nX^n$

Exercice 4 : (5 points)

Écrire un algorithme qui :

1. remplit un tableau tab de deux dimensions (5x5) de valeurs entières.

2. puis affiche la valeur maximale ainsi que sa position (indice de ligne, indice de colonne).
3. ensuite, calcule la somme des éléments de la diagonale principale.

B.4 Examen final 2016-2017 - Semestre 1 : Corrigé type

Exercice 1 : (5 points)

1. Écrire un algorithme qui calcule le montant des photocopies pour un client qui a photocopié **NBF** feuilles. (**NBF** est une donnée introduite par l'utilisateur).

```
Algorithme mont_pcopie_1_client
Constante pu = 5
Variable BNF : entier
         mt_pcopie : réel

début
    Écrire ('svp, introduire le nombre de copies ')
    Lire (NBF)
    mt_pcopie ← NBF * pu
    si (NBF > 100) alors
    début
        mt_pcopie ← mt_pcopie - 0.25* mt_pcopie
    fin
    sinon
    début
        si (NBF > 30) alors
        début
            mt_pcopie ← mt_pcopie - 0.15* mt_pcopie
        fin
    fin
    Écrire (mt_pcopie)
fin
```

2. Réécrire l'algorithme pour **N** clients. (**N** est une donnée introduite par l'utilisateur).

```
Algorithme mont_pcopie_n_client
Constante pu = 5
Variable i, BNF, N : entier
        mt_pcopie : réel

début
    Écrire ('svp, introduire le nombre de clients ')
    Lire (N)
    Pour i de 1 à N faire
    début
       Écrire ('svp, introduire le nombre de copies ')
       Lire (NBF)
       mt_pcopie ← NBF * pu
       si (NBF > 100) alors
          mt_pcopie ← mt_pcopie - 0.25* mt_pcopie
       sinon si  (NBF > 30) alors
                mt_pcopie ← mt_pcopie - 0.15* mt_pcopie
       Écrire (mt_pcopie)
    fin
fin
```

Exercice 2 : (5 points)

1. Déroulement du programme **test** pour $x = 6$ et $x = 9$

- Pour x = 6

Table B.1 – Déroulement pour x = 6

x	s	i	b	Affichage
6	0	1	?	
6	1	2	?	
6	3	3	?	
6	6	1	true	true

- Pour x = 9

2. A quoi correspondent les valeurs *true* et *false* de *b* et déduire ce que fait le programme.

Table B.2 – Déroulement pour x = 9

x	s	i	b	Affichage
9	0	1	?	
9	1	2	?	
9	1	3	?	
9	4	4	?	
9	4	5	false	false

- *s* est la somme des diviseurs de *x*,
- si *x* est égale à la somme de ses diviseurs alors *b* prend la valeur *true*, sinon *b* prend la valeur *false*.
- Donc le programme *test* vérifie si un nombre est parfait ou non.

Exercice 3 : (5 points)

Étant donné X et n deux entiers. Écrire un algorithme qui permet de calculer la somme suivante : $S = 1 - X + 2X^2 - 3X^3 + 4X^4 + \ldots + (-1)^n nX^n$

Algorithme polynome

```
Variable P,X, n, i, signe, S : entier

début
    Lire (X, n)
    P ← X
    S ← 1
    signe ← -1
    Pour i de 1 à n faire
    début
        S ← S + signe * i * P
        P ← P * X
        signe ← - signe
    fin
        Écrire (S)
fin
```

Exercice 4 : (5 points)

Écrire un algorithme qui :

1. remplit un tableau tab de deux dimensions (5x5) de valeurs entières.
2. puis affiche la valeur maximale ainsi que sa position (indice de ligne, indice de colonne).
3. ensuite, calcule la somme des éléments de la diagonale principale.

Algorithme tab_2_dim

```
Constante n = 5 Variable tab : tableau [1..n, 1..n] de entier
    i,j, S, max, i_max, j_max : entier

début
  /* remplissage du tableau tab */
  Pour i de 1 à n faire
    Pour j de 1 à n faire
      Lire (tab[i,j])
  /* recherche de la valeur max */
  max ← tab[1, 1]
  i_max ← 1
  j_max← 1
  Pour i de 1 à n faire
    Pour j de 1 à n faire
      si (tab[i,j] > max) alors
      début
        max ← tab[i,j]
        i_max← i
        j_max← j
      fin
  Écrire ('la valeur max est = ', max, ' dans la ligne ', i_max , ' et dans la
   colonne ', j_max)
  /* somme diagonale */
  S ← 0
  Pour i de 1 à n faire
    S ← S + tab[i,i]
  Écrire (S)
fin
```

C. Examens de rattrapage avec corrigés types

C.1 Examen de rattrapage 2017-2018 - Semestre 1

Exercice 1 : (3.00 points)

Donner les étapes nécessaires (à l'aide d'un schéma) pour mettre en œuvre une application (du problème aux résultats).

Exercice 2 : (5.50 points)

1. Convertir le nombre décimal 52 en binaire.
2. Écrire un programme pascal qui permet de convertir en binaire (base deux) un nombre positif exprimé en base dix. Le programme doit répéter la saisie du nombre jusqu'à ce que ce dernier soit positif.

Vous pouvez utiliser un tableau pour sauvegarder les bits reste de la division.

Exercice 3 : (5.50 points)

Écrire un algorithme qui recherche une note dans un tableau trié, par ordre croissant, contenant N notes. La technique utilisée est celle par **dichotomie** dont le principe est le suivant :

1. Au départ, La plage de recherche est tout le tableau, soit **notes[inf..sup] (inf=1, sup=N)**, l'élément recherché = **X**.
2. À chaque itération on a :

(a) une plage [inf..sup],

(b) son milieu m = (inf+sup) div 2,

(c) la subdivision : [inf..m-1], [m], [m+1..sup].

3. Alors, soit :

(a) notes [m] = X ⇒ l'élément est trouvé (Fin de l'algorithme).

(b) notes [m] < X ⇒X $\notin$ [$inf..m$]⇒Nouvelle plage = [m+1..sup].

(c) notes[m] > X ⇒ X $\notin$ [$m..sup$]⇒ Nouvelle plage = [inf..m-1].

Exercice 4 : (6.00 points)

Soit l'algorithme suivant :

```
Algorithme Mystère
Constante N = 10
Variable t1, t2 : tableau [1..N] de chaîne
        i, j_haut, j_bas : entier
        haut_ou_bas : chaîne

début
   j_haut← 1
   j_bas ← N
   haut_ou_bas ← 'HAUT'
   Pour i de 1 à N faire
   début
      si (haut_ou_bas = 'HAUT') alors
      début
         t2[j_haut] ← t1[i]
         j_bas ← j_bas- 1
         haut_ou_bas ← 'HAUT'
      fin
      sinon
   début
         t2[j_bas]← t1[i]
         j_bas ← j_bas- 1
         haut_ou_bas ← 'HAUT'
      fin
   fin
fin
```

1. Quel sera le contenu du tableau **t2** si celui du tableau **t1** est le suivant ?

E	**O**	**X**	**G**	**A**	**L**	**M**	**A**	**E**	**N**

2. Déduisez que fait l'algorithme.

C.2 Examen de rattrapage 2017-2018 - Semestre 1 : corrigé type

Exercice 1 : (3.00 points)

Voire Figure 2.3 de la page 36.

Exercice 2 : (5.50 points)

1. Convertir le nombre décimal 52 en binaire. **(0.5 point)**
   $(52)_{10} = (110100)_2$

2. Écrire un programme pascal qui permet de convertir en binaire (base deux) un nombre positif exprimé en base dix. Le programme doit répéter la saisie du nombre jusqu'à ce que ce dernier soit positif. **(5 points)**

```
1  Program conv_b10_b2;
2  Const max = 520;
3  Var   reste : array[1..50] of 0..1;
4    n,I,j : integer;
5  begin
6    repeat
7      writeln('introduire un entier positif ');
8      readln(n);
9    until (n>0);
10   I := 1;
11   Repeat
12     reste[i] := n mod 2;
13     n := n div 2;
14     I := i+1;
15   Until (n=0);
16   For j = i-1 downto 1 do
17 write(reste[j]);
18 end.
```

Exercice 3 : (5.50 points)

Algorithme dichotomie

```
Constante n = 5
Variable notes = tableau[1..n] de réel
         inf, sup, m : entier
         X : réel
         trouve : booléen

début
    /* saisir l'élément à rechercher */
    Écrire ('saisir l''élément à rechercher ')
    Lire (X)
    /* Recherche de l'élément */
    inf ← 1
    sup ← n
    trouve ← Faux
    tant que (inf <= sup ) ET (NON trouve) faire
    début
        m ← (inf + sup) div 2
        si notes[m] = X alors
        début
            trouve ← Vrai
        fin
        sinon
        début
            si notes[m] < X alors
            début
                inf ← m+1
            fin
            sinon
            début
                sup ← m-1
            fin
        fin
    fin
    si trouve alors
    début
        Écrire ('l''élément ', X, 'existe dans le tableau et il est à la position : ',
         m);
    fin
    sinon
    début
        Écrire ('l''élément ', X, 'n''existe pas dans le tableau');
    fin
fin
```

Exercice 4 : (6.00 points)

1. Déroulement de l'algorithme **(4.50 points)**

i	**j_haut**	**j_bas**	**haut_ou_bas**	**t2**									
1	1	10	'HAUT'										
2	2	10	'BAS'	**E**									
3	2	9	'HAUT'	**E**									**O**
4	3	9	'BAS'	**E**	**X**								**O**
5	3	8	'HAUT'	**E**	**X**							**G**	**O**
6	4	8	'BAS'	**E**	**X**	**A**						**G**	**O**
7	4	7	'HAUT'	**E**	**X**	**A**					**L**	**G**	**O**
8	5	7	'BAS'	**E**	**X**	**A**	**M**				**L**	**G**	**O**
9	5	6	'HAUT'	**E**	**X**	**A**	**M**			**A**	**L**	**G**	**O**
10	6	6	'BAS'	**E**	**X**	**A**	**M**	**E**		**A**	**L**	**G**	**O**
11	6	5	'HAUT'	**E**	**X**	**A**	**M**	**E**	**N**	**A**	**L**	**G**	**O**

Le contenu de **t2** sera : **(1.50 points)**

E	**X**	**A**	**M**	**E**	**N**	**A**	**L**	**G**	**O**

C.3 Examen de rattrapage 2018-2019 - Semestre 1

Exercice 1 : (6 points)

Soit l'algorithme suivant :

Algorithme Mystère

```
Constante N = 10
Variable t1, t2 : tableau [1..N] de chaîne
          i, j_haut, j_bas : entier
          haut_ou_bas : chaîne

début
   j_haut← 1
   j_bas ← N
   haut_ou_bas ← 'HAUT'
   Pour i de 1 à N faire
   début
      si (haut_ou_bas = 'HAUT') alors
      début
         t2[j_haut] ← t1[i]
         j_bas ← j_bas- 1
         haut_ou_bas ← 'HAUT'
      fin
      sinon
   début
         t2[j_bas]← t1[i]
         j_bas ← j_bas- 1
         haut_ou_bas ← 'HAUT'
      fin
   fin
fin
```

1. Quel sera le contenu du tableau **t2** si celui du tableau **t1** est le suivant ?

C	**I**	**O**	**M**	**U**	**D**	**R**	**S**	**S**	**A**

2. Déduisez que fait l'algorithme.

Exercice 2 : (4 points)

Soit l'algorithme de trois boucles suivant :

```
Algorithme trois_boucles
Variable a,b,c,nb1,nb2 : entier

début
    Pour a de 1 à 10 faire
        Pour b de 1 à 10 faire
            Pour c de 1 à 10 faire
            début
                nb1 ← a*100+b*10+c
                nb2 ← a*a*a+b*b*b+c*c*c
                si (nb1 = nb2) alors
                    Écrire (nb1, a, b, c)
            fin
fin
```

Réécrire cet algorithme en utilisant la forme **TANT QUE** puis la forme **Répéter Jusqu'à**.

Exercice 3 : (5 points)

Idem que l'exercice n°2 de l'examen partiel 2015/2016 de la page 151.

Exercice 4 : (6 points)

Idem que l'exercice n°3 de l'examen final 2014/2015 de la page 160.

C.4 Examen de rattrapage 2018-2019 - Semestre 1 : corrigé type

Exercice 1 : (6 points)

1. Déroulement de l'algorithme

i	j_haut	j_bas	haut_ou_bas	t2									
1	1	10	'HAUT'										
2	2	10	'BAS'	**C**									
3	2	9	'HAUT'	**C**									**I**
4	3	9	'BAS'	**C**	**O**								**I**
5	3	8	'HAUT'	**C**	**O**							**M**	**I**
6	4	8	'BAS'	**C**	**O**	**U**						**M**	**I**
7	4	7	'HAUT'	**C**	**O**	**U**					**D**	**M**	**I**
8	5	7	'BAS'	**C**	**O**	**U**	**R**				**D**	**M**	**I**
9	5	6	'HAUT'	**C**	**O**	**U**	**R**			**S**	**D**	**M**	**I**
10	6	6	'BAS'	**C**	**O**	**U**	**R**	**S**		**S**	**D**	**M**	**I**
11	6	5	'HAUT'	**C**	**O**	**U**	**R**	**S**	**A**	**S**	**D**	**M**	**I**

Le contenu de **t2** sera :

C	**O**	**U**	**R**	**S**	**A**	**S**	**D**	**M**	**I**

Exercice 2 : (4 points)

1. Version avec Tant que

```
Algorithme trois_boucles
Variable a,b,c,nb1,nb2 : entier

début
    a ← 1
    tant que (a <= 10) faire
    début
        b ← 1
        tant que (b¡=10) faire
        début
            c ← 1
            tant que (c¡=10) faire
            début
                nb1 ← a*100+b*10+c
                nb2 ← a*a*a+b*b*b+c*c*c
                si (nb1 = nb2) alors
                    Écrire (nb1, a, b, c)
                c ← c + 1
            fin
            b ← b + 1
        fin
        a ← a + 1
    fin
fin
```

2. Version avec Répéter Jusqu'à

```
Algorithme trois_boucles
Variable a,b,c,nb1,nb2 : entier

    a ← 0
    répéter
        répéter
            répéter
                nb1 ← a*100+b*10+c
                nb2 ← a*a*a+b*b*b+c*c*c
                si (nb1 = nb2) alors
                    Écrire (nb1, a, b, c)
                c ← c + 1
            jusqu'à (c = 10)
            b ← b + 1
        jusqu'à (b = 10)
        a ← a + 1
    jusqu'à (a = 10)
```

Exercice 3 : (5 points)

Idem que la solution proposée pour l'exercice n°2 de l'examen partiel 2015/2016 de la page 153.

Exercice 4 : (6 points)

Idem que la solution proposée pour l'exercice n°3 de l'examen final 2014/2015 de la page 163.

Index