Chapitre I: Structure des programmes

1. Introduction

Une idée trop répandue, tend à faire croire qu'un ordinateur est une machine capable de résoudre des problèmes et de prendre des décisions avec précision et rapidité. En fait, même le plus gros des ordinateurs ne peut exécuter qu'un nombre restreint d'opérations très élémentaires qui doivent lui être communiquées de façon précise dans un langage (langage de programmation) qui lui est compréhensible.
Le problème principal de l'utilisateur d'un ordinateur est donc de parvenir à lui décrire correctement et le plus simplement possible la suite des actions élémentaires permettant d'obtenir, à partir des données fournies, les résultats escomptés.
La base même de la programmation, et ce quel que soit le langage utilisé, consistera donc à la résolution de problèmes de façon logique, claire et structurée. Cette logique permettra d'établir des résolutions de problèmes communes à tous langages et compréhensibles par tous. Cette façon de travailler évite le fait que chacun puisse programmer 'à sa propre sauce' et que les programmes deviennent illisibles pour une personne extérieure. Ces différentes règles de logique de programmation à respecter permettront de concevoir par la suite des programmes de façon structurée.

2. Notions d'algorithme

Un *algorithme* est une suite logique d'actions élémentaires à effectuer pas à pas qui constitue une action complexe permettant de résoudre un problème.

Un *algorithme* doit présenter les caractéristiques suivantes :

- Lorsqu'il est mis en oeuvre, un algorithme doit conduire à l'exécution d'un nombre fini d'opérations. En pratique, il est utile que ce nombre soit suffisamment petit pour que la durée d'exécution du programme associé soit raisonnable.
- Chaque action doit être définie avec précision, sans aucune ambiguïté, tout comme devront l'être le passage entre ces étapes successives.
- Chaque action doit, en outre, être exécutable.
- Un algorithme doit pouvoir utiliser des données dont les valeurs ne sont spécifiées qu'au moment de l'exécution.
- Un algorithme doit produire des résultats.

N.B. : Une fois le problème résolu grâce à l'algorithme, il restera à traduire celui-ci dans le langage de programmation compris par la machine**,** soit le Basic, Pascal, C, ...

3. Elaboration d'un algorithme

Voici les étapes à suivre :

- Nommer l'algorithme en question.
- Déclarer ensuite les constantes éventuelles, les variables utilisées, ainsi que les différents types utilisés.
- Selon le langage utilisé, on peut aussi avoir recours à des Procédures et des Fonctions.
- Ecrire le corps du programme qui sera délimité par les marques « Début » et « Fin ».

4. Représentations des algorithmes

Il existe plusieurs manières de représenter les algorithmes, mais toutes ces représentations ont pour but

- d'être précises, claires et non ambiguës ;
- de faciliter la traduction en un langage de programmation ;
- de rester relativement indépendantes de ces langages et des machines utilisées.

Les représentations d'algorithmes peuvent se classer en 2 catégories :

1) les représentations sous forme d'un texte descriptif,
2) les représentations graphiques.

5. Construction d'un programme

Les étapes de construction d'un programme sont :

❶ Analyser le problème à résoudre, définir les variables utilisées ainsi que les valeurs qu'elles peuvent prendre, les données, les résultats que l'on désire obtenir, les cas particuliers possibles...
❷ Ecrire l'algorithme.
❸ Coder l'algorithme dans le langage de programmation adéquat.

1. Bref historique

Cette annexe n'a pas de but de faire l'historique de l'informatique ou des langages de programmations mais doit vous permettre d'avoir une idée de l'apparition des langages dans le temps. De plus, lorsque c'est possible, un bref détail (inventeur, pourquoi) sera établi.

1840 : premier programme créé par Ada Lovelace.

1943 : COLOSSUS 1, calculateur électronique conçu par Max Newman avec un premier langage de programmation binaire inventé par Alan Turing. Durant cette même année Von Neumann introduit les branchements conditionnels (GOTO vient de naître).

1947 : Amélioration du traitement binaire : le langage Assembleur, inventé par Maurice V. Wilkes.

1951 : Apparition du langage LOGO (tortue)

1954 : FORTRAN (FORmula TRANslator) crée par J. Backus qui travaille pour IBM
But: problèmes numériques
Contributions: compilation rapide et séparée, sous-programme
Remarques: A subit beaucoup de révisions: Fortran II, IV, 66, 77, 90, HP.
Encore en vogue dans le milieu scientifique.

1956 : IBM crée le langage APL (A programming Language)

1958 : Première version de l'ALGOL (Algol 58) (détail ci-dessous).
Niklaus Wirth, lance ALGOL W, le futur PASCAL.

1960 : ALGOL 60 (ALGOrithmic Language)
But: problèmes numériques
Contributions: structures en blocs, récursivité
Remarques: A influencé les générations de langages après 1960, ancêtre de Pascal, Modula, Ada, etc.

COBOL 1960 (Common Business Oriented Language)
But: Administration et applications commerciales
Contributions: enregistrements, descripteurs de fichiers
Remarques: A subi plusieurs révisions dont la dernière est Object COBOL.

1962 : FORTRAN IV

LISP J. crée par McCarthy (LISt Processing)
But : calculs symboliques
Contributions : programmation fonctionnelle: il n'y a plus de notion de variable.

APL K. crée par Kenneth Iverson (Harvard et IBM) (A Programming Language)
But : calculs numériques basés sur un ensemble très complet d'opérateurs vectoriels et matriciels
Contribution : typage dynamique

MAD (Michigan Algorithm Decoder) crée par Arden.

SIMULA (SIMUlation LAnguage), pionner de la programmation objet.

1965 : BASIC crée par J. Kemeny et T. Kurtz, (Beginner's All-purpose Symbolic Instruction Language)
But : rendre la programmation accessible à tous
Contribution : 1er à introduire des outils de développement interactifs
Remarques : simplification des structures et extrêmement populaire parmi les utilisateurs de PC

1966 : SNOBOL 4 crée par Griswold (String Oriented symBolic Language)
But : traitement de chaînes de caractères
Contributions : reconnaissance et identification de chaînes
Remarque : ancêtre de Icon

1967 : PL/1 (IBM) 1967 (Programming Language 1)
But : Combiner les langages FORTRAN, COBOL, ALGOL 60 et

LISP pour obtenir un langage d'utilisation générale
Contributions : traitement d'exceptions et notion de tâches concurrentes
Remarque : langage complexe avec un succès limité

SIMULA 67 crée par O.J. Dahl et K. Nygaard (SIMUlation Language)
But : conçu pour la simulation et la modélisation de systèmes
Contributions : notions de classe et de coroutine
Remarque : précurseur des langages orientés objets

1968 : ALGOL 68
But : révision de ALGOL 60
Contributions : orthogonalité, tentative de décrire formellement le langage

1969 : PASCAL crée par N. Wirth
But : apprentissage de la programmation par raffinements successifs
Remarque : a obtenu un succès au-delà de toute prédiction

1970 : Création du système d'exploitation UNIX et création du langage C par Dennis Ritchie. Le nom provient à la base du langage CPL crée à partir de l'ALGOL 60 par Barron et Stracchey. Ce premier a subi une amélioration qui s'est nommée BCPL par Martin Richards. Le nom C vient du fait de la troisième version et de l'élimination de CPL

1971 : FORTH (contraction de fourth : $4^{ème}$ génération) crée par Charles Moore à des fins de calculs scientifiques (astronomie).

1972 : Apparition du premier traitement de textes : WANG

Création du langage SMALLTALK par A. Kay (Xerox), premier véritable langage orienté objet
But : Modéliser des objets communicants
Contributions : hiérarchisation de classes instanciées dynamiquement
Remarques : a donné naissance à Eiffel, C++, Object Pascal, Java (1997) et C#

Création du BASIC français (LES : Langage Symbolique d'Enseignement)

1973 : PROLOG A. Colmerauer (PROgramming in LOGic)
But : programmer en logique de 1er ordre
Contributions : raisonner sur les problèmes, décrire les règles d'un programme au lieu de son algorithme.
Remarques : a donné lieu aux langages Gödel et Mercury, et au paradigme de la programmation par contraintes: CHIP, CLP(R) et Oz (1998).

1974 : MESA crée par Xerox
But : programmation concurrente et répartie
Remarque : a influencé la partie concurrente de Java

1975 : Ted Nelson introduit le concept Hypertexte qui sera repris par Apple.

1977 : FORTRAN 77 évolution de FORTRAN IV.

CLU crée par B. Liskov
But : expérimenter la généricité et les types abstraits
Contributions : itérateur, peaufine la notion de traitement d'exception
Remarque : Le langage ne comporte pas l'énoncé goto.

EUCLID
But : Vérification de programmes
Contribution : notions d'assertion sous la forme de prédicat

MODULA 2 crée par N. Wirth>
But : programmation de systèmes
Contribution : séparation d'un module en deux parties : sa spécification et son implémentation

Sortie du premier tableur : VISICALC

1979 : ADA crée par Jean-David Ichbiah
But : maîtriser le cycle de développement
Contributions : notion de rendez-vous et d'environnement intégré

VAL crée par J. Dennis
But : Exploiter le parallélisme inhérent des programmes de manière implicite
Contribution : a donné naissance au paradigme de la programmation par flots de données.
Remarques : a donné naissance à Id et SISAL (1985)

1981 : Création du MS/DOS

1982 : Microsoft lance MULTIPLAN, successeur de VISICALC, ancêtre d'Excel.

Création du langage POSTSCRIPT par John Warnock, héritier de FORTH.

1983 : Microsoft lance WORD

Création du C++ et du TURBO PASACL

1984 : Création du langage CLIPPER

1985 : Première version WINDOWS de Microsoft.

Création de PARADOX (base de données) par BORLAND)

Sortie de QUICKBASIC

1987 : Création du langage PERL par Larry Wall

1988 : DBASE IV sort après des évolutions.

1989 : PASCAL OBJECT par Borland

1990 : HASKELL

1991 : VISUAL BASIC qui subira des changements, pas tous compatibles entre-eux), pour donner la version .NET

LINUX par Linus Torvalds

PYTHON par Van Rossum

1993 : Développement du langage HTML.

1995 : ADA 95(objet) dernière version en cours de l'ADA.

DELPHI par Borland

JAVA par SUN

JAVASCRIPT par Netscape

1997 : PHP

2000 : Création du langage D par Walter Bright successeur du C.

C# par Microsoft

KYLIX par Borland pour remplacer DELPHI

2003 : création de S3 qui provient du PERL et du C++

Vers 1967, on dénombrait cent vingt langages dont quinze seulement étaient vraiment utilisés. En 1978, parmi les treize langages retenus pour la première conférence on trouve Algol, APL, Basic, Cobol, Fortran, Jovial, Lisp, PL/1, Simula et Snobol. En 1993, sur des critères semblables, la seconde conférence en retint quatorze dont Ada, Algol 68, C, C++, Forth, Icon, Lisp, Pascal, Prolog et Smalltalk.
Les principaux langages actuels sont sans doute Ada, C, C++, Cobol, Fortran, Haskell, Java, JavaScript, Lisp, ML, Perl, PHP, PostScript, Prolog, Python, Smalltalk et VisualBasic.

2. Les différents types de langages

Pour être exécuté sur un ordinateur, un programme informatique doit être écrit en langage machine c'est à dire qu'il ne peut contenir que des instructions compréhensibles par le processeur. Un programme exécutable est totalement dépendant de la machine pour laquelle il est écrit. Ainsi, les programmes pour IBM PC doivent être compatibles avec le jeu d'instructions des microprocesseurs Intel de la famille 80x86 qui ne présente aucune compatibilité avec les microprocesseurs de la famille 68000 de Motorola qui équipent le McIntosh de Apple.
Quel que soit son degré d'évolution, un processeur n'est capable d'exécuter qu'un nombre limité d'instructions "plus ou moins" rudimentaires (lecture/écriture de la mémoire, opérations arithmétiques et logiques élémentaires, utilisation des registres, gestion d'interruptions programmées ...). Une action "complexe" (opérations sur des nombres décimaux, affichage, impression, ...) doit être décomposée jusqu'au niveau du processeur.
Le langage de programmation de plus "bas niveau" est le langage machine, suite ésotérique de nombres binaires. Un ton au dessus se situe le langage d'assemblage. Il utilise des abréviations (mnémoniques) qui correspondent aux instructions élémentaires mais sont plus claires pour le programmeur. Le programme source obtenu doit être ensuite traduit (assemblé) dans le langage machine par un programme (assembleur) adapté au processeur afin d'obtenir le programme objet exécutable.
Dans les langages de plus haut niveau les instructions élémentaires contiennent de nombreuses instructions du langage machine. Le programme source est plus rapide à écrire (dans la mesure où les mots utilisés sont pris dans le langage naturel), plus facile à modifier et indépendant de la machine à laquelle il est destiné. La traduction en langage exécutable se fera à l'aide d'un programme adapté à l'ordinateur : compilateur ou interpréteur.
Programme compilé : s'il ne rencontre pas d'erreur de syntaxe dans le code source, le compilateur crée un programme objet (indépendant du langage dans lequel il a été écrit) puis un programme exécutable après édition de liens. Pour le modifier, il faudra intervenir sur le programme source et refaire une compilation.
Programme interprété : l'interpréteur ne traduit le programme source en langage machine qu'au moment de son exécution. Celle-ci est moins rapide que dans le cas d'un programme compilé et le programme "plante" sur les erreurs de syntaxe.
Programme pré-compilé : un "compilateur virtuel" fabrique un p-code (commun à toutes les machines). Ce code est ensuite interprété par chaque ordinateur pour son processeur.

Eléments d'un langage de programmation	
Types d'Instructions	Contenus
Entrées et Sorties	Entrées : Clavier, Souris, Lecture de fichiers, ... Sorties : Ecran, Imprimante, Ecriture de fichiers, ...
Calculs	Numériques : Arithmétiques, algébriques, trigonométriques, ... Alphanumériques : Concaténations, extractions de chaînes, ...
Structures de contrôle	Alternatives (conditionnelles) Itératives (répétitives)
Manipulations de données	Recherches, Ajouts, Mouvements, Suppressions, ...

Chapitre II : Les langages

Un langage général doit être complet c'est à dire qu'il doit permettre d'exprimer tous les algorithmes calculables.
Ne sont pas considérés comme langages de programmation :
les langages de description de pages comme HTML ou PostScript
les langages de 4ème génération souvent destinés aux bases de données relationnelles comme SQL ou Oracle
les systèmes d'exploitation comme UNIX ou Linux
les applications comme Excel

Classes de langages	
Algorithmique ou Impératif ou Procédural	Les programmes décrivent séquentiellement une suite d'actions que la machine doit effectuer. Une procédure est une portion de programme qui accomplit une tâche spécifique. Certaines procédures appartiennent au langage (intrinsèques), d'autres sont écrites par le programmeur. Ex : Fortran, Algol, Cobol, Ada, Basic, Pascal
Déclaratif fonctionnel ou Déclaratif logique	Les programmes sont construits sur la déclaration de fonctions et/ou de relations. Ces langages sont utilisés en Intelligence artificielle. Ex : Lisp, ML, C, Prolog
Orienté objets	Les programmes manipulent des objets eux mêmes issus de classes (un objet est une instance d'une classe) qui possèdent leurs propres attributs et méthodes. Ces langages pratiquent l'encapsulation (interdiction d'accéder au contenu d'un objet), l'héritage (une classe hérite de la classe dont elle dérive) et le polymorphisme (une méthode pourra agir différemment suivant la classe à laquelle elle s'applique). De nombreux langages ne sont pas purement orientés objets (hybrides). Ex : Smalltalk, Eiffel, C++, Java

3. Un peu de détails

Ada	(Hommage à Ada Augusta Byron comtesse de Lovelace) Né en 1979, Ansi en 1983, Iso en 1987, Objet en 1995. Langage de programmation compilé conçu par Jean Ichbiach de CII Honeywell Bull après un concours du Département de la Défense qui en a fait son langage standard. Caractérisé par un fort typage des données, il est arrivé trop tard pour les "main frames" et trop tôt pour la micro. Concurrencé par C, il reste présent sur les systèmes embarqués de l'US Air Force. Dérivé de Pascal. Souche de Eiffel, Oak.
Algol	(Algorithmic Language) Né en 1958, modifié en 1960,1968. Langage de programmation compilé conçu par un consortium européen pour concurrencer Fortran trop proche d'IBM. Très théorique et indépendant de la machine, il permet les calculs sur des variables en mémoire et autorise la récursivité. Dérivé de Fortran. Souche de CPL, Pl/1, Simula, Mesa.
APL	(A Programming Langage) Né en 1960, APL2 en 1984, objet en 1996. Langage de programmation interprété conçu par Ken Iverson d'IBM pour être aux nombres ce que LISP était aux caractères. Il utilise des symboles mathématiques non disponibles au clavier. Il échoua devant Basic et les tableurs dérivés. Souche de J.
Basic	(Beginner's All-purpose Symbolic Instruction Code) Né en 1964, MS Basic 2.0 en 1975, Visual Basic 1.0 en 1991, Visual Basic 6.0 en 1998, Visual Basic Net en 2002. Langage de programmation interprété conçu par John Kemeny et Thomas Kurtz à Darthmouth. Il fut diffusé en plusieurs dialectes et étendu au graphique et au multimédia sur les premiers micros (Bill Gates créa un Basic pour Altaïr). Faiblement typé et peu structuré, il est compilé et orienté objet dans l'environnement Visual Basic de Microsoft.
Bug	Le 09/09/1945 à 15 h 45 Grace Murray Hopper (en faisant le ménage ?) découvre un papillon dans le Relais 70 du Panneau F du prototype de Mark-II et note "First actual case of bug being found".
C	(Après B ?) Né en 1971, K&R en 1978, Ansi en 1989, Iso en 1990, 1996, 1999. Langage de programmation compilé conçu par Brian Kernighan et Dennis Ritchie au Bell Laboratories à partir des langages du MIT. C'est le langage système utilisé pour le développement d'Unix. Moyennement typé, très structuré, il utilise une syntaxe particulière. Ce langage de haut niveau permet d'utiliser, comme l'assembleur, des instructions de bas niveau. Héritier de CPL, BCPL, B. Souche de Awk, Csh, C with Classes, Concurent C, Objective C, Python, Cmm.

C++	(de C) Né en 1983, Ansi/Iso en 1998. Langage de programmation compilé conçu par Bjarne Stroustrup au Bell Lab en utilisant le compilateur C. Il appartient à la chaîne GNU du MIT. Langage objet avec héritage multiple, pointeurs et surcharge d'opérateurs. Visual C++ conserve sa puissance dans un environnement plus convivial. Héritier de C with Classes. Souche de Cmm, C#
Cobol	(Common Business Oriented Language) Né en 1959, modifié en 1961, Extended en 1962, modifié en 1965, Ansi en 1968, 1974, Osi/Ansi en 1985, Objet en 1997. Langage de programmation compilé conçu par Grace Hopper et un comité de normalisation des applications de gestion pour un consortium américain dont le Département de la Défense. Il est spécialisé dans les fichiers et le traitement de données. Verbeux, sa syntaxe et son vocabulaire sont proches de l'anglais courant. Héritier de B-O, Flow-Matic. Souche de PL/1.
CP/M-DOS	(Control Platform for MicroComputer - Disk Operating System) Système d'exploitation pour micro-ordinateur conçu en 1976 par Gary Kildall pour Digital Research. Adaptation simplifiée (très !) d'Unix à une seule machine.
dBase	Logiciel de base de données relationnelle pour micro-ordinateur, développé par Wayne Ratliff pour Ashton Tate, muni d'un langage de création et gestion de type SQL, et commercialisé en 1980 sous le nom de dBase II. Les versions III et IV ont occupé 70% du marché et leur format de fichiers s'est imposé comme standard de base de données sur micros. Incapable de s'adapter à Windows, la société a été rachetée en 1991 par Borland. L'environnement Clipper (Nantucket) lui apportait un langage de programmation complet et un compilateur.
Forth	(Pour Fourth) Né en 1969, Objet en 1987. Langage de programmation interprété conçu par Charles Moore astronome. Il fournit un langage exécutable pour machine virtuelle. Il manipule deux piles (évaluation et contrôle), un dictionnaire (modifiable par ajout ou modification de mots), la récursivité et la notation Polonaise Postfixée. Utilisé comme système d'exploitation sur certains micro-ordinateurs, il a été utilisé dans plusieurs satellites pour la gestion de processus en temps réel. Le système est adaptable à tous les processeurs (du microcontrôleur 8 bits au Cray) et à tous les systèmes d'exploitation. Souche de PostScript
Fortran	(Formula Translator) Né en 1954, I en 1956, II en 1957, III en 1958, IV en 1962, Ansi en 1966, 1978, Iso en 1991, 1997, modifié en 2002. Langage de programmation compilé conçu par John Backus pour un IBM 704. Il devient dès 1957 le standard pour le calcul mathématique. En dépit de ses faiblesses pour le traitement des chaînes et des ensembles de données, l'existence de très importantes bibliothèques font de ce vétéran un langage encore très demandé par les scientifiques. Souche de Algol, PL/1.

HTML	(HyperText Markup Langage) Né en 1990, 1.0 en 1992, 2.0 en 1995, 3.2 en 1997, XML 1.0 en 1998, 4.0en 1999, XHTML en 2000. Langage informatique pour la description de pages sur l'internet conçu par Tim Berners-Lee du CERN. Il utilise des marques sous la forme de balises (Tags) pour décrire les pages et les caractères qui les composent, insérer le multimédia et utiliser les liens hypertextes. Héritier de SGML.
Java	(Kawa !) Oak en 1991, I en 1995, 1.2 en 1998, 1.3 en 2000, 1.4 en 2002, 1.5 en 2004. Langage de programmation précompilé conçu par James Gosling pour Sun MicroSystems. Après une pré-compilation unique en byte-code ou p-code pour une machine virtuelle, les applets sont interprétés en temps réel par chaque machine. Moyennement typé, très structuré, c'est un langage objet avec garbage collection, héritage simple et surcharge des méthodes. Héritier de Objective C, Ada, C++, Cedar, Smalltalk, Scheme. Souche de NetRexx, C#.
JavaScript	(de Java) Cmm en 1992, LiveScript puis JavaScript en 1995, 1.5 en 1998 Langage de programmation interprété à base de syntaxe Java. C'est le plus utilisé des langages de scripts sur l'internet. Il est à l'origine d'un certain nombre de dialectes. Héritier de C, C++. Souche de JScript, ECMAScript.
Linux	(Linus et Unix) Système d'exploitation pour micro-ordinateur conçu en 1991 par Linus Torvalds pour le microprocesseur Intel 386. Le noyau est actuellement disponible et gratuit pour tous les systèmes. Accompagné des logiciels libres du GNU, il est une alternative aux systèmes propriétaires (Windows entre autres).
Lisp	(List Processing) Né en 1958, I en 1959, 1.5 en 1962, Common en 1984, Ansi en 1994, Langage de programmation compilé conçu par John Mac Carthy pour le MIT. Premier langage de programmation fonctionnel. Il est à l'origine d'un certain nombre de dialectes (dont Scheme). C'est un langage automodifiable basé sur le traitement de listes et de symboles, récursif avec garbage collection. Souche de Logo, Smalltalk, Scheme, Clos.
Logo	(du grec Logos - Raison ?) Né en 1968, Object en 1986. Langage de programmation conçu par une équipe dirigée par Wally Fuerzeig pour Bolt & Newmans. Il ajoute des primitives graphiques aux bases de Lisp. S'appuyant sur les théories de Piaget, Seymour Papert développe autour de la "tortue" une pédagogie de la découverte. Héritier de Lisp.
Lotus 1.2.3	Développé en 1983 par Mitch Kapor pour Lotus. Il intégrait tableur, grapheur et base de données dans le même environnement. Son succès fut comparable à celui de VisiCalc sur Apple II.

MacOS	(Operating System for Mac Intosh) Système d'exploitation installé par Apple sur Mac Intosh depuis 1993. Il gère le multitâche et le multimédia. MacOS8 pour le PowerPC a supprimé toute compatibilité avec les processeurs 68xxx.
Maple	Système algébrique sur ordinateur conçu par B.W. Char, K.O. Geddes, W.M. Gentleman et G.h. Gonnet pour l'Université de Waterloo en 1980. Ecrit en B puis en C, en perpétuelle évolution, il reste un très puissant outil dans le monde des mathématiques.
Modula	(Modular Language) Né en 1975, 2 en 1979, 3 en 1988, Iso 1997. Langage de programmation compilé conçu par Niklaus Wirth en se rapprochant de C pour combler les lacunes de Pascal et donner accès au hardware. Le programme est découpé en modules contenant du code et des données visibles localement. Les modules sont compilés séparément et réutilisables. Peu utilisé hors du cadre universitaire. Héritier de Pascal, Mesa. Souche de Oberon, Python.
MS-DOS	(MicroSoft - Disk Operating System) Système d'exploitation pour micro-ordinateur conçu en 1980 par Tim Paterson pour Microsoft et livré avec le Personal Computer d'IBM sous le nom de PC-DOS puis distribué sur une multitude de clones sous le nom de MS-DOS. La communication passe par un interpréteur de commandes (Command.com). Il reste jusqu'à Windows 95 la plate forme de l'Interface Utilisateur Graphique (GUI).
Pascal	(Hommage à Blaise Pascal) Né en 1970, Afnor en 1983, Object en 1985, Borland Object en 1989, Delphi en 1995, 5 en 1999, 6 en 2001, 7 en 2002. Langage de programmation compilé conçu par Niklaus Wirth à l'ETH de Zurich. Le premier compilateur (P4) générait du p-code pour une machine virtuelle. Le passage sur micro-ordinateur fut réalisé avec l'USCD par Kenneth Bowles de l'Université de Californie à San Diégo puis avec le Turbo-Pascal de Philippe Kahn et Niels Jansen pour la société Borland. Il est largement utilisé comme premier langage. Moyennement typé, fortement structuré (programmation modulaire), sa syntaxe est simple et rigoureuse. Il en existe aujourd'hui de nombreux dialectes dont celui de Borland dans l'environnement Delphi. Héritier de Algol. Souche de CLU, Mesa, Modula, Ada
Perl	(Practical Extracting and Report Language) Né en 1987, 2 en 1988, 3 en 1989, 4 en 1991, 5 en 1994, 5 en 1998, 5.6 en 2000, 5.8 en 2002. Langage de programmation interprété conçu par Larry Wall comme langage de ligne de commande pour Unix. En dépit de la lourdeur de sa syntaxe, il permet d'analyser et de traiter efficacement du texte structuré. Utilisable sur PC (Windows) ou Mac Intosh, il est très utilisé pour créer des scripts CGI. Héritier de SH, NAWK. Souche de Ruby, PHP.

PHP	(Personal Home Page Hypertext Preprocessor) Né en 1995, 2 en 1997, 3 en 1998, 4 en 2000, 4.1 en 2001, 4.2 en 2002, 4.3.1 en 2003 Langage de programmation interprété conçu par Rasmus Lerdorf. Le code intégré dans une page HTML est transformé en HTML pur par le serveur. Non typé, il utilise une syntaxe proche du C. Son succès est du à son statut de logiciel libre associé à un serveur web Apache et à un moteur de base de données MySQL. Héritier de Perl.
PL/1	(Programming Language One) Né en 1964, Ansi en 1976 Langage de programmation à vocation universelle conçu par George Radin pour IBM en reprenant le meilleur des langages existants. Très typé, il manipule nombres complexes, chaînes de bits, listes et tableaux dynamiques, utilise pointeurs, récursivité et primitives de synchronisation pour gérer le multitâche, et offre un macro-générateur permettant d'étendre le langage ainsi qu'une bonne gestion des fichiers et de l'édition. Il avait contre lui d'être exigeant en mémoire à une époque où celle-ci était rare ... et d'être la propriété d'IBM. Souvent utilisé, avec une pincée d'assembleur, comme langage système sous diverses déclinaisons (EPL pour Multics, PL/6, HPL pour Honeywell-Bull, PL/M pour CP/M, PL/S pour IBM ...) jusqu'à l'apparition de C (beaucoup moins élégant), il est toujours largement utilisé chez IBM. Héritier de Algol, Fortran, Cobol. Souche de PL/M.
Prolog	(Programmation en logique) Né en 1970, II en 1982, III en 1984, IV en 1997. Langage de programmation compilé conçu à Marseille par Alain Colmeraurer de la Faculté des Sciences de Lumigny et Philippe Roussel d'Elsa Software, et à l'Université d'Edimbourg par Robert Kowalski. Après la modélisation d'un problème (faits, règles, but à atteindre) à l'aide de formules (Clauses de Horn), le moteur d'inférence est capable de tirer des conclusions en utilisant des déductions logiques (arbres et backtracking). En dépit de la débâcle japonaise des ordinateurs de la Vème génération, il reste un bon langage pour réaliser rapidement des maquettes en intelligence artificielle.
PostScript	Né en 1982, level 2 en 1992, level 3 en 1996. Langage de description de pages mis au point par Adobe qui permet de combiner et de stocker des éléments de provenances diverses (fontes décrites par des courbes de Bézier, graphisme vectoriel ou BitMap) destinés à être imprimés. Il utilise des piles et un dictionnaire. Héritier de Forth.
Scheme	(de Schemer) Né en 1975, MIT en 1978, modifié en 1984, IEEE en 1990, R5RS en 1998. Langage de programmation (dialecte de Lisp) compilé conçu par Gerald Sussman et Guy Steele au MIT. Symbolique et fonctionnel, permettant la représentation directe de données complexes, il est très utilisé en enseignement et en recherche sur la programmation. Héritier de Lisp, Algol. Souche de Haskell, Oak.

Simula	(Simulation) Né en 1964, modifié en 1967. Langage de programmation conçu par Ole Johan Dahl et Krysten Nygard au Centre Informatique Norvégien d'Oslo pour la simulation de systèmes réels. Il définit des classes, des sous classes et des objets (instances de classes) possédant des méthodes. A l'origine de la programmation orientée objet. Héritier de Algol. Souche de Smalltalk, C with Classes, Eiffel.
Smalltalk	Né en 1969, modifié en 1972, 1974, 1976, 1978, 1980. Langage de programmation conçu par Alan Kay et Adele Goldberg pour Xerox dans un environnement graphique avec fenêtres et souris. Il est totalement orienté objet avec constructeurs et destructeurs, communication par messages, héritage simple, surcharge d'opérateurs et garbage collection. Héritier de Lisp, Simula. Souche de Objective C, Self, Oak, Ruby.
SQL	(Structured Query Language) Quel en 1972, Sequel en 1974, SQL en 1982, Ansi en 1986, Iso en 1989, SQL2 en 1992. Langage de création et d'interrogation de bases de données conçu pour IBM sur les idées de Ted Codd et Larry Ellison en relation avec les Systèmes de Gestion de Bases de Données Relationnelles (SGBDR). De type déclaratif, il n'utilise ni variables, ni procédures, ni fonctions.
Unix	(Opposition à Multics) Né en 1969, Version C en 1973, BSD en 1977, III en 1982, V en 1983. Système d'exploitation en temps partagé, multitâches et multi utilisateurs conçu par Ken Thompson et Dennis Ritchie au Bell Lab. Constitué d'un noyau primitif, d'un shell, de quelques utilitaires et d'un éditeur, il est totalement écrit en langage C. AT&T, Berkeley, Western Electric et Novell ont participé à son évolution. Il en existe de nombreuses variantes avec ou sans interface graphique (X-Windows, Solaris, Linux).
VisiCalc	(Visible Calculator) Tableur écrit en Pascal UCSD par Dan Bricklin et Bob Franston (qui n'en déposèrent pas le brevet) en 1979. Il fit la gloire de l'Apple II.
Windows	Née en 1985, l'interface utilisateur graphique de Microsoft (très inspirée de celle du MacIntosh) trop lente ne décolle pas. En 1990, la version 3 est une réussite (échange interactif de données, réseau poste à poste), Apple ne gagne pas son procès pour plagiat et Windows s'impose sur tous les micro-ordinateurs. Ce n'est qu'en 1995 avec Windows 95 et Windows NT (réseau client-serveur) qu'il deviendra un véritable système d'exploitation multitâches.
WordStar	Traitement de texte écrit par John Barnaby pour MicroPro en 1979. Ses combinaisons de touches sont entrées dans la légende.

Chapitre III: Les objets

1. Déclarations des objets

Un programme manipule des objets que ce soit des données ou des résultats, et effectue des actions sur ces objets.
Nous pouvons distinguer 2 catégories d'objets: les *constantes et* les *variables.*

Une ***constante*** est un objet dont l'état restera inchangé durant toute l'exécution du programme, c'est un simple élément d'information.

Exemple : *Considérons une constante Pi à laquelle on affecte la valeur 3.1416...*

Chaque fois que le programme utilise Pi, il travaille avec sa valeur 3.1416...

Une ***variable*** est un objet pouvant représenter un entier, un réel, un caractère, un booléen, ... et dont l'état peut être modifié à tout moment dans le programme.

Exemple : *Considérons une variable compteur à laquelle on attribue la valeur 0 au début. Si l'on augmente le compteur d'une unité plusieurs fois, la variable compteur se verra attribuée successivement les différentes valeurs 0, 1, 2, 3, ...*

2. Types des objets

Le fait d'attribuer un type bien précis à un objet, définit parfaitement celui-ci et caractérise l'ensemble des valeurs pouvant être assignées à ces objets.
On peut dès lors distinguer facilement les objets entre eux.

Exemple : *Un objet déclaré de type entier ne pourra contenir des caractères.*

Voici les différents types d'objets:

2.1. Type entier

Les différentes opérations possibles sur 2 entiers X et Y donnant un entier comme résultat sont :

Opération	***Signification***	***Exemple***
X + Y	Somme	
X – Y	Soustraction	
X . Y	Produit	
X Div Y	Partie entière de la division de X par Y[1]	7 Div 2 = 3
X Mod Y	Reste de la division de X par Y[2]	7 Mod 2= 1
Abs (X)	Valeur absolue de X	Abs (–7) = 7

2.2. Type réel

Les différentes opérations possibles sur *2* réels X et Y donnant un réel comme résultat sont :

[1] Si Y < > 0.
[2] Si Y < > 0.

Opération	***Signification***	***Exemple***
X + Y	Somme	
X – Y	Soustraction	
X . Y	Produit	
X / Y	Division	
Sqrt (X)	Racine carrée de X	Sqrt (16) = 4
Abs (X)	Valeur absolue de X	Abs (–2,4) = 2,4

2.3. Type caractère

Il existent différentes fonctions manipulant les caractères :

Opération	***Signification***	***Exemple***
Ord ()	Code ASCII du caractère	Ord (‘^’) = 94
Chr ()	Caractère correspondant au code ASCII	Chr (94) = ^

2.4. Type chaîne de caractères

Ce type permet d’attribuer à une variable une suite de caractères juxtaposés.

Il existe, selon les programmes, différentes fonctions manipulant les chaînes de caractères.

Exemple *: La fonction Longueur qui donne le nombre de caractères que comporte la chaîne. Longueur (‘BONJOUR’) = 7.*

2.5. Type booléen

Un objet de type booléen représente une vérité; soit vrai, soit faux.
Les différentes opérations logiques possibles et donnant un résultat booléen sont :

NON	
Vrai	Faux
Faux	Vrai

ET	**Vrai**	**Faux**
Vrai	Vrai	Faux
Faux	Faux	Faux

OU	**Vrai**	**Faux**
Vrai	Vrai	Vrai
Faux	Vrai	Faux

2.6. Type énuméré

On peut définir un type sous forme d’une liste ordonnée.

Exemple : *Création d'un type Semaine ne comprenant que les jours de la semaine; les variables de ce type ne pourront avoir comme valeur que les jours lundi, mardi, ... si ces noms ont été considérés comme faisant partie du type semaine.*
Variable Jour: Semaine

2.7. Type structuré

Un type structuré permet de décrire des variables qui sont en fait un ensemble de types différents.

Exemple : *Considérons la structure Personne composée de 3 types différents; un type chaîne de caractère pour le nom de la personne, un type entier pour son âge et un type caractère pour représenter son sexe (Mou F).*

```
Type Structure Personne
Nom: Chaîne de caractère
Age: Entier
Sexe: Caractère
Fin
```

On peut ensuite déclarer une variable appartenant à ce type.

```
Variable Membre: Personne
```

On peut alors dans le programme affecter différentes valeurs à la variable Membre.

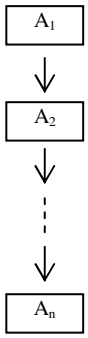
```
Membre.Nom = 'Pierre'
Membre.Age = 30
Membre.Sexe = 'M'
```

1. Introduction

Les schémas de contrôle permettent l'écriture de programme de manière structurée. Cette programmation structurée est basée sur le fait que tout programme unidimensionnel (ne possédant qu'une seule entrée et une seule sortie) est composé à l'aide de 3 schémas logiques de base :

- Le schéma Séquentiel
- Le schéma de Choix
- Le schéma Itératif

2. Schéma Séquentiel



Explication :

A_1, A_2, ... A_n : actions

Exécution d'une action A_1 suivie d'une action A_2, ... jusqu'une action A_n.

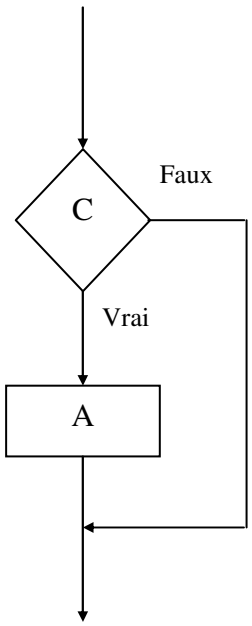
Ce schéma exécute donc dans un programme une suite d'opérations dans un ordre bien défini.

Exemple :

Crée un algorithme qui permette l'échange de deux nombres entiers.

3. Schéma de Choix

3.1. Schéma de choix simple



Explication :

C : expression logique ou test logique
A : action

L'expression C est évaluée. Si elle est vraie, l'action A sera alors exécutée. Dans le cas contraire, le programme ne tient pas en considération l'action A et se place juste après celle-ci.

Si C alors A (pour A une action simple)

ou

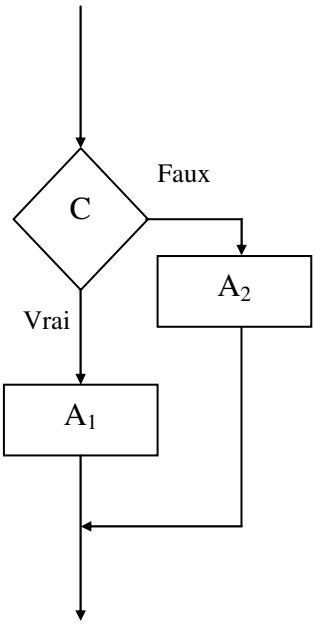
Si C alors Début (pour A un ensemble d'actions)
A
Fin

N.B. L'expression C est un test et ne modifie donc en rien l'état du programme.

Exemple :

Crée un algorithme qui permette de trouver le maximum entre deux nombres entiers.

3.2. Schéma de choix alternatif



Explication :

C : expression logique ou test logique
A_1, A_2 : actions

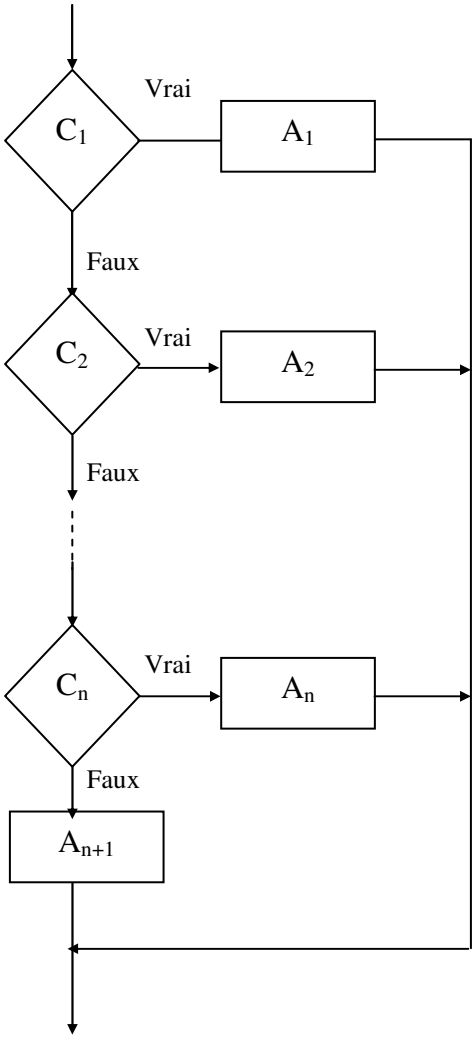
L'expression C est évaluée. Si elle est vraie, l'action A_1 sera alors exécutée. Dans le cas contraire, c'est l'action A_2 qui sera exécutée.

```
Si C  alors A1          (A1 action simple)
      sinon A2          (A2 action simple)
ou
Si C  alors Début       (A1 un ensemble d'actions)
            A1
            Fin
      sinon Début
            A2
            Fin
```

Exemple :

Crée un algorithme qui, en fonction du choix de l'utilisateur, permette de calculer la masse ou le poids dans la formule : P = m . g (g = 9,81)

3.3. Schéma de choix multiple



Explication :

$C_1, C_2, \ldots, C_n$: expressions logiques ou tests logiques
A_1, A_2, A_n : actions

Dans ce cas, nous avons plusieurs conditions logiques avec leurs actions respectives. Ces conditions C_i seront évaluées de manière séquentielle et lorsqu'une d'elles sera satisfaite, l'action correspondante A_i sera exécutée.
Dans le cas où aucune condition n'est satisfaite, l'action A_{n+1} sera exécutée.

Si C_1 alors A_1 (pour A_i une action simple)
sinon si C_2 alors A_2
sinon si C_3 alors A_3
sinon si C_n alors A_n
sinon A_{n+1}

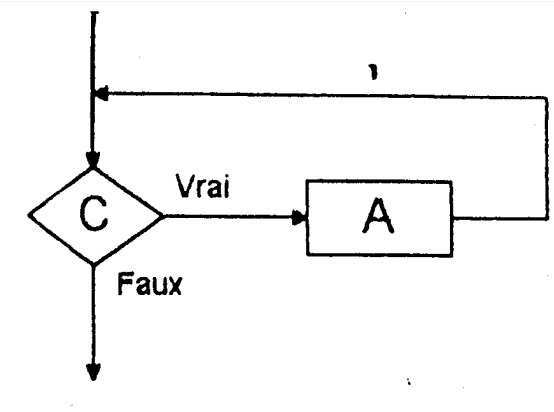
Si A_i est un ensemble d'actions, il faut remplacer A_i par :
Début
A_i
Fin

Exemple :

Crée un algorithme qui, en fonction du choix de l'utilisateur, permette de calculer la somme, la différence, le produit ou le quotient de deux nombres réels. Attention au quotient. L'opération s'effectue toujours le premier nombre opération le second nombre.

4. Schéma Itératif

4.1. Schéma Tant que (While)



Explication :

C: expression logique ou test logique
A: action (ou ensemble d'actions)

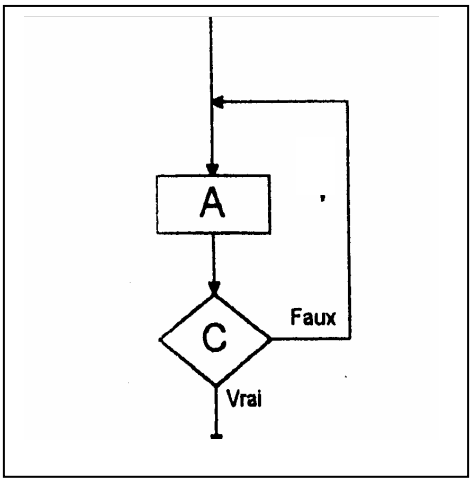
Il s'agit ici de répéter une action (ou un ensemble d'actions) A tant que la condition C est respectée.

Tant que C Faire
A
Fin Faire

Exemple :

Crée un programme qui permette de trouver le PGCD de deux nombres entiers.

4.2. Schéma Répéter (Repeat)



Explication :

C : expression logique ou test logique
A : action (ou ensemble d'actions)

Il s'agit ici de répéter une action (ou un ensemble d'actions) A jusqu'à ce que la condition C soit respectée.

Répéter A
jusque C

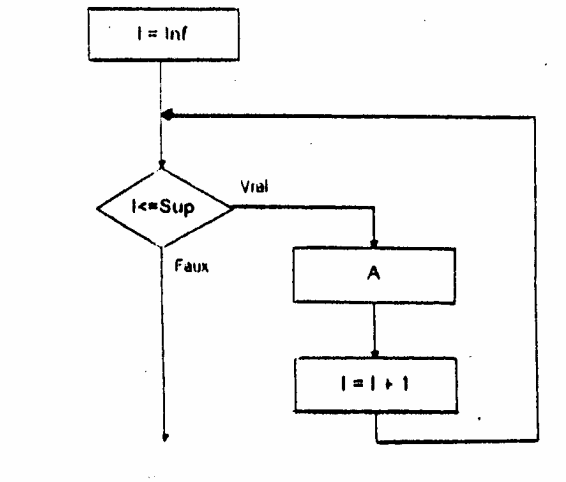
N.B. : A sera au minimum exécutée une fois puisque la condition logique est évaluée en fin de boucle (contrairement au schéma précédent).

Exemple :

Etant donnés deux nombres entiers M et N, vérifier si la valeur entière X, introduite par l'utilisateur, est dans l'intervalle [M, N]. L'algorithme s'arrête lorsque X est dans l'intervalle. Celui-ci nous indique le nombre d'essai pour y parvenir.

4.3. Schéma Pour

1.1.1. Boucle Pour croissante



Explication :
I : Variable (compteur)
Inf : Valeur limite inférieure
Sup : Valeur limite supérieure
A : action (ou ensemble d'actions)

Il s'agit ici de répéter une action (ou un ensemble d'actions) un nombre de fois égal à (Sup - Inf). L'indice I étant initialisé à la limite inférieure, il sera incrémenté, à chaque passage dans la boucle, de 1 unité jusqu'à ce qu'il soit égal à Sup. Lorsque I a atteint la valeur Sup, c'est alors le dernier passage dans la boucle et donc la dernière exécution de A.

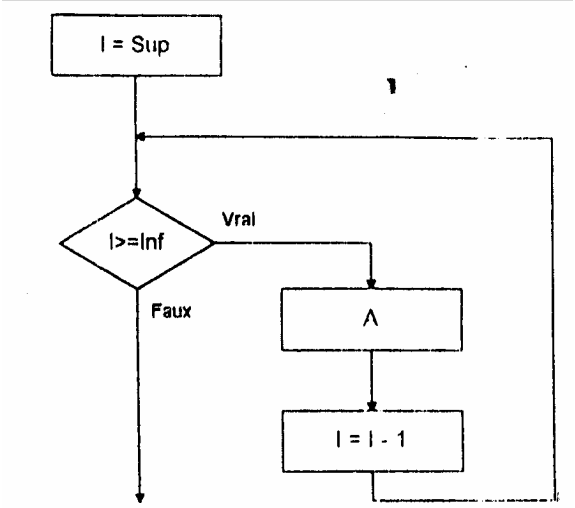
Pour I = Inf à Sup Faire
A
I = I + 1
Fin Faire

N.B. : Ce schéma est utilisé lorsqu'on connaît le nombre d'incrémentations à effectuer avant de commencer celles-ci.

Exemple :

Créer un algorithme qui permette de calculer le produiut x . y, de deux entiers, par additions successives.

1.1.2. Boucle Pour décroissante



Explication :

I : Variable (compteur)
Inf : Valeur limite inférieure
Sup : Valeur limite supérieure
A : action (ou ensemble d'actions)

Il s'agit ici de répéter une action (ou un ensemble d'actions) un nombre de fois égal à (Sup - Inf). L'indice I étant initialisé à la limite supérieure, il sera décrémenté, à chaque passage dans la boucle, de 1 unité jusqu'à ce qu'il soit égal à Inf.
Lorsque I a atteint la valeur Inf, c'est alors le dernier passage dans la boucle et donc la dernière exécution de A.

Pour I = Sup à Inf Faire
A
I = I - 1
Fin Faire

N.B. : Ce schéma est utilisé lorsqu'on connaît le nombre de décrémentations à effectuer avant de commencer celles-ci.

Exemple :

Crée un algorithme qui permet de calculer n ! = 1 . 2 . 3 . 4 . 5 . … . (n – 1) . n (n naturel ≠ 0)

5. Remarque

Lorsqu'on ne respecte pas la logique qui a été décrite dans les différents schémas de contrôle, on obtient des programmes non structurés. Ce manque de structure se remarquera directement lors du passage de l'algorithme au programme si le langage est un langage structuré. Toutefois, un langage permettant des branchements (ex.: le Basic avec des Goto) permettra de traduire ces algorithmes. Les programmes tourneront mais lors de la conception de programmes plus complexes et plus longs, cela devient fastidieux à comprendre et à modifier. La difficulté résidant dans un organigramme où existent de nombreuses lignes qui s'entrecroisent et occasionnent de nombreux renvois nuisant à la clarté et à la lisibilité de l'algorithme.

Exemple de programme non structuré et de sa modification en un programme structuré:

Programme non structuré	Programme structuré

Il est donc conseillé d'éviter les programmes non structurés.

Un programme est **dit structuré** s'il suit les 3 conditions suivantes :
- il peut être représenté par un diagramme possédant une seule entrée et une seule sortie;
- il existe un chemin menant de l'entrée vers tout sommet et de tout sommet vers la sortie;

il est représenté uniquement par une combinaison des schémas logiques de base.

1. Introduction

Les algorithmes que nous avons décrits jusqu'à présent ne font intervenir que peu d'instructions. Les problèmes que l'on rencontre en pratique sont plus complexes et leur résolution est beaucoup plus difficile, parce qu'il faut tenir compte de nombreux éléments qui interagissent.

Une approche efficace consiste à décomposer le problème en sous problèmes qui sont alors étudiés séparément. D'autre part, certains algorithmes interviennent dans de nombreux contextes différents. On souhaite disposer de techniques qui permettent de les mettre au point une fois et de les utiliser sans devoir les adapter à chaque nouveau contexte. Enfin, on est amené dans certains algorithmes à utiliser à plusieurs reprises une même séquence d'instructions, on souhaite alors ne pas devoir recopier systématiquement ces instructions.

Les ***fonctions*** et les ***procédures*** sont des outils qui ont été développés pour résoudre les problèmes que nous venons d'évoquer.

On utilise parfois le terme de sous-programme ou routine pour représenter indifféremment une fonction ou une procédure.

2. Les fonctions

Une fonction est une séquence d'instructions dont l'exécution dépend des valeurs prises par certains paramètres et qui a pour effet de calculer une valeur.
La première étape consiste à déclarer ces fonctions dans le programme appelant celles-ci et ce au même endroit où se font les autres déclarations de type et de variables.

La déclaration de la fonction comprend 2 parties: l'en-tête et le corps de la fonction.

- En-tête : *Fonction Nom (Liste des paramètres formels : Type) : Type.*
- Corps : *Partie déclarative des variables utiles à la fonction.*
  Séquence d'instructions c-à-d, ensemble des actions constituant le traitement de la fonction.

Exemple :

```
Algorithme Maximum
Var a, b, c : Entier
Fonction Max (x, y: Entier): Entier
Début
Si x >= y  alors Max = x
           sinon Max = y
Fin
Début
Lire (a)
Lire (b)
c = Max (a,b)
Ecrire ('Le maximum est', c)
Fin.
```

Les paramètres a et b envoyés vers la fonction sont appelés paramètres effectifs. Ils sont ensuite convertis dans la fonction de façon locale en paramètres formels x et y qui seront utilisés dans la fonction. Cette façon de travailler permet de garder intact les variables effectives alors que l'on pourrait les utiliser dans la fonction. De plus, le fait d'utiliser cette fonction dans plusieurs programmes, les variables seront probablement différentes d'un programme à l'autre.

Remarque :

Les variables qui seront déclarées normalement dans la partie déclarative du programme sont des variables dites globales car elles sont connues par tout le programme, y compris les sous-programmes comme les fonctions et les procédures.
A l'inverse, les variables qui sont déclarées dans la fonction ne seront reconnues que par cette fonction. On dit alors que ces variables sont locales car elles ne 'vivent' que durant l'exécution de la fonction.

3. Les procédures

Les fonctions sont parfois d'un usage trop restrictif parce qu'elles ont pour but de calculer une valeur unique. Les procédures jouent un rôle semblable à celui des fonctions dans les autres cas ; la différence entre une fonction et une procédure résidant simplement dans le fait que la procédure ne voit pas le résultat de ses actions assigné à son nom.
De la même façon que les fonctions, les procédures doivent être déclarées dans la partie déclarative de programme. Elles comportent aussi l'en-tête et le corps de la procédure.

- En-tête : *Procédure Nom (Liste des paramètres formels (avec leur catégorie correspondante) : Type).*

En effet, on distingue 3 types de paramètres formels pour les procédures, selon que leur valeur est utilisée ou non, que leur valeur est calculée ou non, dans le corps de la procédure.

1) Les *paramètres formels d'entrée* (= Données) sont ceux dont la valeur doit être connues pour que les instructions de la procédure puissent être exécutées mais qui ne doivent pas subir de modification.
2) *Les paramètres formels de sortie* (= Résultats) sont les paramètres pour lesquels la procédure détermine une valeur.
3) *Les paramètres formels mixtes* (= Données Résultats) sont les paramètres qui jouent à la fois le rôle de paramètres d'entrée et de sortie.

- Corps : *Partie déclarative des variables utiles à la procédure.*
  Séquence d'instructions c-à-d ensemble des actions constituant le traitement de la procédure.

Exemples :

1°) Algorithme **Permuter**
Var a, b : Entier
Procédure Echanger (Données Résultats x, y Entier)
Var z : Entier
Début
z = x
x =y
y = z
Ecrire (x,y)
Fin
Début
Lire (a)
Lire (b)
Echanger (a,b)
Fin.

Ce programme envoie vers une procédure d'échange (= programme appelé) les valeurs a et b qui sont converties en x et y dans la procédure où elles sont échangées. Cette procédure renvoie ensuite les valeurs échangées vers a, b du programme appelant via x, y.

2°) Algorithme **Div-Mod**
Var a, b, q, r : Entier
Procédure Division (Données x, y: Entier; Résultats q, r Entier)
Début
r = x
q = 0
Tant que r >= y Faire
q = q + 1
r = r – y
Fin Faire
Ecrire (x, y, q, r)
Fin
Début
Lire (a)
Lire (b)
Division (a, b, q, r)
Fin.

Les paramètres a, b sont transmis en données et restent inchangés et les paramètres q, r sont transmis en mode résultat.

2. Exercice

Crée une calculatrice dont les opérations et fonctions mathématiques font appel à des fonctions ou procédures. Les opérations : + , – , * , / , les fonction mathématiques : factorielle, sinus, cosinus, tangente, racine carrée, puissance et modulo.

3. Déclaration et syntaxe

Un caractère est une variable de type « char » qui prend un octet en mémoire : *Var z : char ;*

Une chaîne de caractère est une variable de type « string » qui peut comporter au maximum 255 caractères.
Si rien n'est spécifié, la variable de type string est par défaut de 255 caractères, par contre s'il y a un nombre ou une constante entre crochet, la variable aura cette dimension.

Var z : string ;
Var z : string[36] ;

4. Affectation

Lors de l'affectation, la valeur doit être placée entre ´ ´ (quote). Si cette valeur contient une apostrophe, le quote doit être doublé.
Les caractères qui peuvent être utilisés sont les caractères ASCII :

Code ASCII	Caractère DOS	Code ASCII	Caractère DOS	Code ASCII	Caractère DOS	Code ASCII	Caractère DOS
0	NUL	32	(Espace)	64	@	96	`
1	☺	33	!	65	A	97	a
2	☻	34	"	66	B	98	b
3	♥	35	#	67	C	99	c
4	♦	36	$	68	D	100	d
5	♣	37	%	69	E	101	e
6	♠	38	&	70	F	102	f
7	•	39	'	71	G	103	g
8	◘	40	(	72	H	104	h
9	○	41	)	73	I	105	i
10	◙	42	*	74	J	106	j
11	♂	43	+	75	K	107	k
12	♀	44	,	76	L	108	l
13	♪	45	-	77	M	109	m
14	♫	46	.	78	N	110	n
15	☼	47	/	79	O	111	o
16	►	48	0	80	P	112	p
17	◄	49	1	81	Q	113	q
18	↕	50	2	82	R	114	r
19	‼	51	3	83	S	115	s
20	¶	52	4	84	T	116	t
21	§	53	5	85	U	117	u
22	▬	54	6	86	V	118	v
23	↨	55	7	87	W	119	w
24	↑	56	8	88	X	120	x
25	↓	57	9	89	Y	121	y
26	→	58	:	90	Z	122	z
27	←	59	;	91	[	123	{
28	∟	60	<	92	\	124	|
29	↔	61	=	93	]	125	}
30	▲	62	>	94	^	126	~
31	▼	63	?	95	_	127	DEL

Chapitre VI : Les chaînes de caractères et les caractères

Code ASCII	Caractère DOS	Code ASCII	Caractère DOS	Code ASCII	Caractère DOS	Code ASCII	Caractère DOS
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	-
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	*ƒ*	191	┐	223	▀	255	

Les 23 premiers caractères sont utilisés par MS-DOS à des fonctions spécifiques (suppr, end, insert, enter, esc, tab, shift, retour ligne, …)
Le caractère 9 ASCII équivaut à « tabulation »
Le caractère 10 ASCII équivaut à « nouvelle ligne »
Le caractère 13 ASCII équivaut à « fin de ligne »

5. Fonctions

5.1. Modification

Le type « string » est en fait un tableau de caractères à une dimension dont l'élément d'indice zéro contient une variable de type « char » et dont le rang dans la table ASCII correspond à la longueur de la chaîne. Il est donc possible de modifier un seul caractère.

Exemple :

```
Program ex01 ;
Uses crt ;
Var chaine : string ;
Begin
chaine := ´bed´
writeln(chaine) ;
chaine[2] := ´a´ ;
writeln(chaine) ;
readkey ;
end.
```

Pour la modification pour la variable de type « char », il suffit d'affecter une nouvelle donnée.

```
Var z : char ;
z := ´a´
z := ´b´
```

5.2. Concaténation

Il n'est évidemment pas possible de concaténer des variables de types « char » du fait qu'elles ne seront plus composées d'un seul caractère mais de plusieurs.
Cette fonction ne s'applique donc qu'au variable de type « string ».
Il faut tout de même faire attention, lorsque celle-ci est dimensionnée de ne pas dépasser cette dimension sous peine de la voir tronquée.

Exemples :

```
Program concatener;
uses crt;
var y, z:string;
var x :string[20];
begin
z:='il faut voir que tout est ';
z:=z+' possible';
x:= 'il faut voir que tout est ';
y:=concat(z,x);
writeln(z);writeln(x);writeln(concat(z,x));writeln(y);
readkey;
end.
```

5.3. **Manipulation**

```
Program copy_delete;
uses crt;
var z,test:string;
begin
z:='anticonstitutionnelement';
writeln(z);
test:=copy(z,5,12);
writeln(test);
test:=z;
delete(test,1,4);
writeln(test);
insert('anti',test,1);
writeln(test);
readkey;
end.
```

La fonction *copy(z,i,j)* qui doit être associée à une variable de type « string » renvoie la copie à partir de la i^{ème} position pour j caractères du contenu de la variable z.

La fonction *delete(test,i,j)* ne doit pas être associée à une variable car elle modifie la variable introduite (dans ce cas : test), en lui supprimant à partir de la i^{ème} position j caractères.

La fonction *insert(texte,test,i)* ne doit pas être associée à une variable car elle insère dans la variable introduite (dans ce cas : test), la chaîne ou le contenu de la variable texte à partir de la i^{ème} position.

5.4. **Autres fonctions**

```
Program conversion_et_autres_fonctions;
uses crt;
var z,test:string;
var num : byte;
var r : real;
var n,error : integer;
begin
z:='anticonstitutionnelement';
writeln(z);
num := pos('constitut',z);
writeln(num);
r:=3.1415;
str(r,test);
writeln(test);
n:=17;
str(n,test);
writeln(test);
```

```
val(test,r,error);
writeln(r);
val(test,n,error);
writeln(n);
readkey;
end.
```

La fonction *pos(chaine1,chaine2)* qui doit être associée à une variable de type « byte » (0 à 255) renvoie la première position de la variable chaine1 dans la variable chaine2.

La fonction *str(r,test)* ne doit pas être associée à une variable car elle convertit la variable introduite (dans ce cas : r), dans la variable test de type « string ».

La fonction *val(test,n,error)* ne doit pas être associée à une variable car elle convertit lorsque c'est possible la variable introduite (dans ce cas : test) dans la variable n de type « integer » ou « real » ou si c'est impossible 0 dans la variable error.

```
Program dernieres_fonctions;
uses crt;
var z: array[1..256] of char;
var i :integer;
var num : byte;
begin
z:='bonjour';
writeln(z);
z[1]:=upcase(z[1]);
writeln(z);
for i:=1 to length(z) do z[i]:=upcase(z[i]);
writeln(z);
z[3]:=chr(47);
writeln(z);
num := ord(z[3]);
writeln(num);
readkey;
end.
```

La fonction *upcase(char)* qui doit être associée à une variable de type « char » renvoie la majuscule si nécessaire de la variable char.

La fonction *chr(num)* qui doit être associée à une variable de type « char » renvoie le caractère du code ASCII introduit avec la variable num.

La fonction *ord(char)* qui doit être associée à une variable de type « byte » renvoie le numéro de code ASCII introduit avec la variable char.

6. Exercice

Créer le programme qui permette de jouer au PENDU.

1. Introduction

Jusqu'à présent, nous avons associé un nom de variable particulier à chacune des quantités manipulées par un algorithme. Nous allons maintenant utiliser des variables représentant un ensemble de valeurs : les tableaux. Un *tableau* est constitué d'une suite d'éléments de même type mais se distinguant les uns des autres par un indice. Un *indice* est un entier donnant la position relative de l'élément dans la suite. A priori, l'indice peut varier de 1 (premier élément du tableau) à n (nombre d'éléments du tableau).

2. Les vecteurs

Un vecteur est un tableau à une dimension. C'est-à-dire que chaque élément du tableau est repéré par un indice.
Puisque l'utilisation des variables indicées est permise en programmation, il convient de les déclarer comme telles dans la partie déclarative de nos algorithmes.
Type Vecteur = Tableau ['Inf'[3]*...'Sup'*[4]*] De 'Type élément'*

Exemple :

Calculer la moyenne (M) des éléments (entiers) d'un vecteur V de dimension n.

```
Algorithme Moyenne
Type Vecteur : Tableau[1 ... n] De Entier
Var V : Vecteur
Var M : Réel
Var n, i : Entier
Début
M = 0
Pour i = 1 à n faire
        M = M + V[i]
        i = i + 1
        fin faire
M = M/n
Ecrire ('La moyenne est ', M)
Fin
```

[3] = Valeur minimale de l'indice.
[4] = Valeur maximale de l'indice.

3. Les matrices

Une matrice est un tableau à 2 dimensions. C'est-à-dire que chaque élément du tableau est repéré par un couple d'indices. La déclaration se fait de la même façon que pour les vecteurs.

Type Matrice = Tableau ['lnf1'[5]*...'Sup1'*[6]*, 'Inf2'*[7]*...'Sup2'*[8]*] De 'Type élément'*

Exemple :

Calculer la trace[9] d'une matrice A de dimension n x n.

```
Algorithme Trace
Type Matrice : Tableau[ 1 ... n, 1...n] De Entier
Var A : Matrice
Var T, i, n : Entier
Début
T = O
Pour i = 1 à n faire
        T = T + A[i , i]
        i = i + 1
        fin faire
Ecrire ('La trace vaut ', T)
Fin.
```

[5] = Valeur minimale du premier indice.
[6] = Valeur maximale du premier indice.
[7] = Valeur minimale du second indice.
[8] = Valeur maximale du second indice.
[9] = Somme des éléments diagonaux de la matrice.

4. Recherche d'un élément dans un vecteur

Recherchons un élément x dans un tableau de n éléments entiers.
Le résultat sera de type booléen et l'algorithme se termine dès que x a été trouvé ou que l'on a parcouru tout le tableau.

1ère version (= Recherche Linéaire)

```
Algorithme Recherchex
Type Vecteur : Tableau [1 ... n] De Entier
Var V : Vecteur
Var x : Entier
Var i, n : Entier
Var Trouve : Booléen
Début
Ecrire ('Introduire x')
Lire (x)
Trouve = Faux
i = 1
Tant que (i <= n) et (Trouve = Faux) faire
        Si V[i] = x alors Trouve = Vrai
        i = i + 1
        fin faire
Si Trouve = Vrai alors écrire ('x appartient au tableau')
                  sinon écrire ('x n'appartient pas au tableau')
Fin.
```

2ème version (Recherche dichotomique = Recherche Binaire)

Pour cette version, nous supposerons que le tableau est ordonné c'est-à-dire que V[i] ≤ V[i+1] pour tout indice i.

```
Algorithme Dicho
Type Vecteur : Tableau [1 ... n] De Entier
Fonction Trouve (T: Vecteur, x: Entier) : Booléen
Var Inf, Sup, Milieu : Entier
Var T : Vecteur
   Début
   Inf = 1
   Sup = n
   Trouve = Faux
   Milieu = (Inf + Sup) Div 2
   Si T[Sup] = x alors Trouve = Vrai
   Répéter
        Début
        Si T[Milieu] = x alors Trouve Vrai sinon
                Début
                Si T[Milieu] < x alors Inf = Milieu
                        sinon
                        Sup = Milieu
                Fin
        Fin
   Milieu = (Inf + Sup) Div 2
   Jusque Trouve = Vrai ou Inf = Milieu
   Fin
Début
Lire (x)
Si Trouve (T, x) = Vrai alors Ecrire (x, 'appartient au tableau')
                          sinon Ecrire (x, 'n'appartient pas au tableau')
Fin.
```

5. Recherche d'un élément dans une matrice

Recherchons un élément y dans un tableau de n x m éléments entiers.
Le résultat sera de type booléen et l'algorithme se termine dès que y a été trouvé ou que l'on a parcouru tout le tableau.

```
Algorithme Recherchey
Type Matrice : Tableau [1... n, 1 ... m] De Entier
Var A : Matrice
Var y : Entier
Var i, j : Entier
Var Trouve : Booléen
Début
Ecrire ('Introduire y')
Lire (y)
Trouve = Faux
i = 1
Tant que (i <= n) et (Trouve Faux) faire
        j = 1
        Tant que (j <= m) et (Trouve = Faux) faire
                        Si A[i, j] = y alors Trouve = Vrai
                        j = j + 1
        fin faire
        i = i + 1
fin faire
Si Trouve = Vrai alors écrire (y, 'appartient au tableau')
                  sinon écrire (y, 'n'appartient pas au tableau')
Fin.
```

Dans ce chapitre, nous nous consacrons à effectuer des tris par différentes méthodes.
De la méthode la plus simple et la plus intuitive à une méthode complexe mais très rapide.
Il est possible dans chacune des méthodes d'apporter des améliorations qui permettront de rendre le programme plus rapide donc moins gourmand en mémoire.
Cinq méthodes seront étudiées : « Le tri par recherche du minimum », « Le tri par bulle », « Le tri par sélection », « Le tri par insertion » et « Le tri par Sedgewick ».

1. Le tri par recherche du minimum

Cette méthode est la méthode la plus intuitive qu'il soit.

1.1. Spécification abstraite

Pour trier une liste, il suffit de rechercher l'élément le plus petit de cette liste, le stocker dans un tableau, de le supprimer de la liste et de reproduire cette démarche jusqu'au moment où il n'y a plus qu'un seul élément.

1.2. Algorithme

```
Algorithme TriParMinimum
variable : i, j, k, n, pos, min : Entiers naturels
variable : Tab,Sol : Tableau d'Entiers naturels de 1 à n éléments
Début
Pour j = 1 à n faire
        début
        min ← 64256
        Pour i = 1 à n – j + 1faire
                début
                Si T[i] < min alors
                        début
                        min ← T[i]
                        pos ← i
                        Fin Si
                Fin faire
        Pour k = pos +1 à n – j faire T[k – 1] ← T[k]
        Sol[j] ← min
        Fin Faire
Fin.
```

1.3. La complexité

Choisissons comme opération élémentaire la comparaison de deux cellules du tableau.
Comme pour la recherche du minimum, le programme pour toutes les valeurs du tableau en revue. La complexité est donc un nombre dépendant de la longueur n du tableau.
Celle-ci vaut $n + (n – 1) + (n – 2) + … + 3 + 2 + 1 = n . (n – 1)/2$.
La complexité en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.
En plus de cela, il faudra y ajouter les échanges à partir de la position qui, elle aussi, au pire donnera du n^2. Donc la complexité globale sera de $2n^2$, que l'on écrit $O(2n^2)$.

1.4. Programme pascal

```
program TriParMinimum;
const N = 10;
type TTab = array [1..N] of integer;
var Tab, Sol : TTab ;

procedure TriMinimum (var Tab:TTab ; var Sol :TTab) ;
var i, j, k, pos, min : integer;
begin
   for j := 1 to N do
   begin
   min : = 64256 ;
      for i := 1 to n – j + 1 do
         begin
            if Tab[i] < min then
                  begin
                  min := Tab[i];
                  pos := i;
                  end ;
         end;
      for k := pos + 1 to N – j + 1 do
      Tab[k – 1] := Tab[k] ;
      Sol[j] :=min ;
   end ;
end;
procedure Initialisation(var Tab:TTab) ;
var i : integer;
begin
   randomize;
   for i := 1 to N do
   Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
var i : integer;
begin
   writeln('------------------------');
   for i:= 1 to N do
   write(Tab[i] : 3, ' | ');
   writeln;
end;
```

```
begin
   Initialisation(Tab);
   writeln('TRI PAR MINIMUM');
   writeln;
   Impression(Tab);
   TriMinimum(Tab,Sol);
   Impression(Sol);
   writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR MINIMUM

-------------------------------------------------------
  1 :  57 :  34 :  96 :  66 :  91 :   5 :   3 :  77 :  20 :
-------------------------------------------------------
  1 :   3 :   5 :  20 :  34 :  57 :  66 :  77 :  91 :  96 :
-------------------------------------------------------
```

2. Le tri par bulle

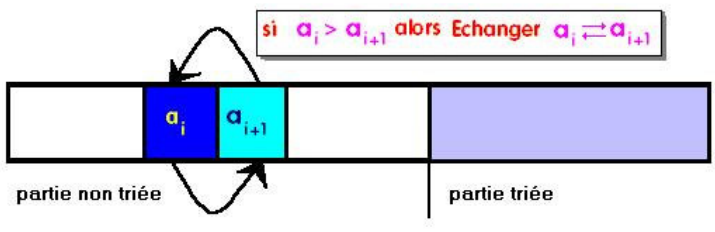
C'est le moins performant de la catégorie des tris par échange ou sélection, mais comme c'est un algorithme simple, il est intéressant à utiliser pédagogiquement.

2.1. Spécification abstraite

Son principe est de parcourir la liste (a_1, a_2, ..., a_n) en intervertissant toute paire d'éléments consécutifs (a_{i-1}, a_i) non ordonnés. Ainsi après le premier parcours, l'élément maximum se retrouve en a_n.
On recommence l'opération avec la nouvelle sous suite (a_1, a_2, ..., a_{n-1}), et ainsi de suite jusqu'à épuisement de toutes les sous suites (la dernière est un couple).
Le nom de tri par bulle vient donc de ce qu'à la fin de chaque itération interne, les plus grands nombres de chaque sous suite se déplacent vers la droite successivement comme des bulles de la gauche vers la droite.

2.2. Spécification concrète

La suite (a_1, a_2, ..., a_n) est rangée dans un tableau T[...] en mémoire centrale. Le tableau contient une partie triée (en grisé à droite) et une partie non triée (en blanc à gauche). On effectue plusieurs fois le parcours du tableau à trier ; le principe de base étant de réordonner les couples (a_{i-1}, a_i) non classés (en inversion de rang soit $a_{i-1} > a_i$) dans la partie non triée du tableau, puis à déplacer la frontière (le maximum de la sous suite (a_1, a_2, ... , a_{n-1})) d'une position :



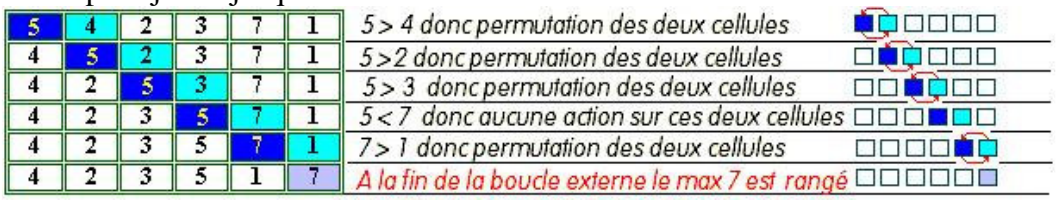
Tant que la partie non triée n'est pas vide, on permute les couples non ordonnés (a_{i-1}, a_i) tels que $a_{i-1} > a_i$ pour obtenir le maximum de celle-ci à l'élément frontière. C'est-à-dire qu'au premier passage c'est l'extremum global qui est bien classé, au second passage le second extremum, etc...

2.3. **Algorithme**

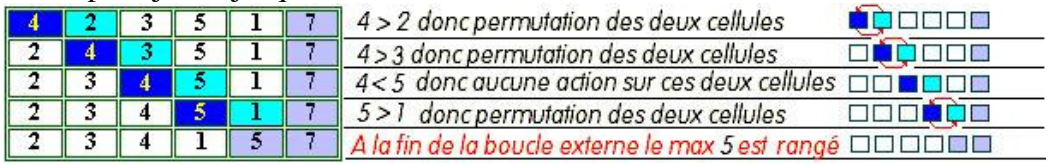
```
Algorithme Tri_par_Bulles
variable : i, j, n, temp : Entiers naturels
variable : Tab : Tableau d'Entiers naturels de 1 à n éléments
début
  pour i de n jusqu'à 1 faire
  pour j de 2 jusqu'à i faire
       si Tab[j – 1] > Tab[j] alors
       temp ← Tab[j – 1]
       Tab[j – 1] ← Tab[j]
       Tab[j] ← temp
       Fin si
  Fin Pour
  Fin Pour
Fin
```

Exemple : soit la liste (5, 4, 2, 3, 7, 1), appliquons le tri par bulles sur cette liste d'entiers. Visualisons les différents états de la liste pour chaque itération externe contrôlée par l'indice i.

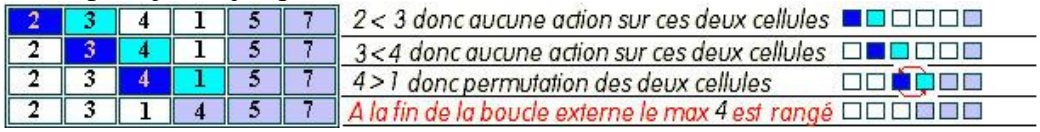
i = 6 / pour j de 2 jusqu'à 6 faire



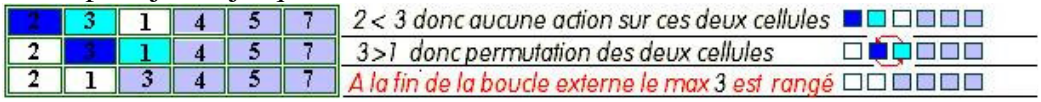
i = 5 / pour j de 2 jusqu'à 5 faire



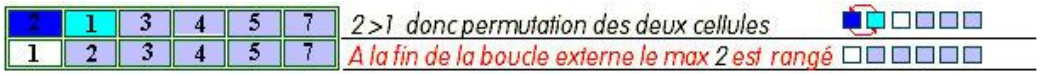
i = 4 / pour j de 2 jusqu'à 4 faire



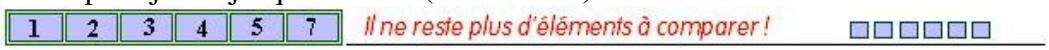
i = 3 / pour j de 2 jusqu'à 3 faire



i = 2 / pour j de 2 jusqu'à 2 faire



i = 1 / pour j de 2 jusqu'à 1 faire (boucle vide)



2.4. **Complexité :**

Choisissons comme opération élémentaire la comparaison de deux cellules du tableau.
Le nombre de comparaisons est une valeur qui ne dépend que de la longueur n de la liste (n est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle s'exécute n fois (donc une somme de n termes) et qu'à chaque fois la seconde boucle exécute (i – 2) + 1 fois.
La complexité en nombre de comparaison est égale à la somme des n termes suivants (i = n, i = n – 1, ...)
C = (n – 2) + 1 + ([n –1] – 2) + 1 +.....+ 1 + 0 = (n – 1)+(n – 2) + ... + 1 = n(n – 1)/2 (c'est la somme des n – 1 premiers entiers).
La complexité en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choisissons maintenant comme opération élémentaire l'échange de deux cellules du tableau.
Calculons par dénombrement du nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échanges). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse et donc chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.
La complexité au pire en nombre d'échanges est de l'ordre de n^2, que l'on écrit $O(n^2)$.

2.5. **Programme pascal**

```
program TriParBulle;
const N = 10;
type TTab = array [1..N] of integer;
var Tab : TTab ;

procedure TriBulle (var Tab:TTab) ;
var i, j, t : integer;
begin
   for i := N downto 1 do
   for j := 2 to i do
      if Tab[j – 1] > Tab[j] then
      begin
         t := Tab[j – 1];
         Tab[j – 1] := Tab[j];
         Tab[j] := t;
      end;
end;
procedure Initialisation(var Tab:TTab) ;
```

```
var i : integer;
begin
   randomize;
   for i := 1 to N do
   Tab[i] := random(100);
end;
```

```
procedure Impression(Tab:TTab) ;
var i : integer;
begin
   writeln('------------------------');
   for i:= 1 to N do write(Tab[i] : 3, ' | ');
   writeln;
end;
```

```
begin
   Initialisation(Tab);
   writeln('TRI PAR BULLE');
   writeln;
   Impression(Tab);
   TriBulle(Tab);
   Impression(Tab);
   writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR BULLE

------------------------------------------------------------
 32 |  60 |  60 |  54 |  70 |  53 |  64 |  91 |  69 |  81 |
------------------------------------------------------------
 32 |  53 |  54 |  60 |  60 |  64 |  69 |  70 |  81 |  91 |
------------------------------------------------------------
```

3. Le tri par bulle amélioré

3.1. Spécification abstraite

L'amélioration se situe sur le fait que si le tableau est ordonné à partir d'un instant, il n'est plus nécessaire de continuer. Comment dès lors savoir que le tableau est bien ordonné ?
En incorporant dans la procédure de test, une variable compteur qui permettent de savoir combien de permutations ont été effectué lors du passage de la boucle i.
Si cette variable compteur est nulle, cela veut dire qu'aucune permutation n'a été effectuée et donc le tableau est ordonné. Il est clair que les complexités (en terme de changement et en terme de test) n'ont pas changé. Le seul gain se situe dans le nombre de test. Dans le premier cas, le nombre est fixe à n . (n – 1)/2, alors que dans ce cas, il est au maximum de n . (n – 1)/2.

3.2. **Algorithme**

```
Algorithme Tri_par_Bulles
variable : i, j, n, temp, compteur : Entiers naturels
variable : Tab : Tableau d'Entiers naturels de 1 à n éléments
début
i = n
compteur = 1
  Tant que (i > 0) Et (compteur <>0) faire
      compteur = 0
      pour j de 2 jusqu'à i faire
        si Tab[j – 1] > Tab[j] alors
        temp ← Tab[j – 1]
        Tab[j – 1] ← Tab[j]
        Tab[j] ← temp
        compteur ← compteur + 1
        Fin si
      Fin Pour
  Fin Tant que
Fin
```

3.3. **Programme pascal**

Nous allons dans le programme pascal, placer les deux procédures : la méthode classique et la méthode améliorée pour pouvoir comparer.

```
program TriParBulleAmeliore;
const N = 10;
type TTab = array [1..N] of integer;
var Tab, Tab2 : TTab ;

procedure TriBulle (var Tab:TTab) ;
var i, j, t, total : integer;
begin
total := 0 ;
   for i := N downto 1 do
   for j := 2 to i do
      total :=total + 1 ;
      if Tab[j – 1] > Tab[j] then
      begin
         t := Tab[j – 1];
         Tab[j – 1] := Tab[j];
         Tab[j] := t;
      end;
writeln('Le nombre de test est de ',total);
end;
```

```
procedure TriBulleAmeliore (var Tab2:TTab) ;
var i, j, t, compteur : integer;
begin
   total := 0;
   i :=N;
   compteur:=1;
   while (i >0) And (compteur <>0) do
   for j := 2 to i do
      total := total + 1;
      if Tab2[j – 1] > Tab2[j] then
      begin
         t := Tab2[j – 1];
         Tab2[j – 1] := Tab2[j];
         Tab2[j] := t;
         compteur := compteur + 1;
      end;
writeln('Le nombre de test est de ',total);
end;

procedure Initialisation(var Tab:TTab) ;
var i : integer;
begin
   randomize;
   for i := 1 to N do
   begin
   Tab[i] := random(100);
   Tab2[i] := Tab[i];
   end ;
end;

procedure Impression(Tab:TTab) ;
var i : integer;
begin
   writeln('------------------------');
   for i:= 1 to N do write(Tab[i] : 3, ' | ');
   writeln;
end;

begin
   Initialisation(Tab,Tab2);
   writeln('TRI PAR BULLE');
   writeln;
   writeln('Le tableau de depart');
   Impression(Tab);
   writeln('------------------------');writeln;
   writeln('Par la methode simple');
```

```
TriBullesimple(Tab2);
Impression(Tab2);
writeln('------------------------');writeln;
writeln('Par la methode amelioree');
TriBulle(Tab);
Impression(Tab);
writeln('------------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR BULLE

Le tableau de depart
---------------------------------------
 71| 71| 13| 67| 36| 53| 59| 37| 39| 54|
---------------------------------------

Par la methode simple
Le nombre de tests est de 45
---------------------------------------
 13| 36| 37| 39| 53| 54| 59| 67| 71| 71|
---------------------------------------

Par la methode amelioree
Le nombre de tests est de 39
---------------------------------------
 13| 36| 37| 39| 53| 54| 59| 67| 71| 71|
---------------------------------------
```

4. Le tri par sélection (Version 1)

C'est une version volontairement inefficace de la catégorie des tris par sélection, une amélioration sera apportée dans le point suivant.

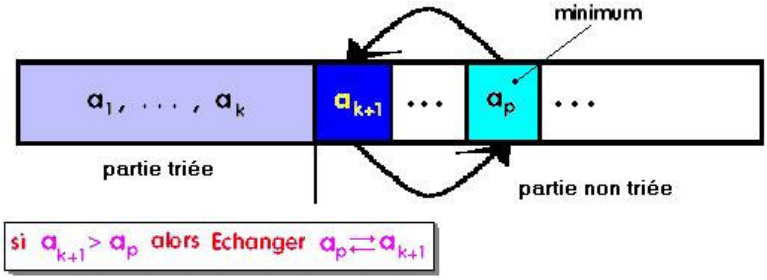
4.1. Spécification abstraite

La liste (a_1, a_2,... , a_n) est décomposée en deux parties : une partie triée (a_1, a_2, ... , a_k) et une partie non triée (a_{k+1}, a_{k+2}, ... , a_n); l'élément a_{k+1} est appelé élément frontière (c'est le premier élément non trié).
Le principe est de parcourir la partie non triée de la liste (a_{k+1}, a_{k+2}, ... , a_n) en cherchant l'élément minimum, puis en l'échangeant avec l'élément frontière a_{k+1}, ensuite à déplacer la frontière d'une position. Il s'agit d'une récurrence sur les minima successifs.
On recommence l'opération avec la nouvelle sous suite (a_{k+2}, ... , a_n) , et ainsi de suite jusqu'à ce que la dernière soit vide.

4.2. Spécification concrète

La suite (a_1, a_2, ... , a_n) est rangée dans un tableau T[...] en mémoire centrale. Le tableau contient une partie triée (en grisé à gauche) et une partie non triée (en blanc à droite). On recopie le minimum de la partie non triée du tableau dans la cellule frontière (le premier élément de cette partie).
Si $a_{k+1} > a_p$, alors $a_{k+1} \leftarrow a_p$ et l'on obtient ainsi à la fin de l'examen de la sous liste (a_{k+1}, a_{k+2}, ... , a_n) la valeur minimale de (a_{k+1}, a_{k+2}, ... , a_n) est stockée dans la cellule a_{k+1}. La sous suite (a_1, a_2, ... , a_k, a_{k+1}) est maintenant triée et l'on recommence la boucle de recherche du minimum sur la nouvelle sous liste (a_{k+2}, a_{k+3}, ... , a_n).

Tant que la partie non triée n'est pas vide, on range le minimum de la partie non triée dans l'élément frontière.



4.3. **Algorithme**

Voici une version maladroite mais exacte. Elle échange physiquement et systématiquement l'élément frontière Tab[i] avec un élément Tab[j] dont la valeur est la plus petite.

```
Algorithme Tri_Selection1
variable : m, i , j , n, temp : Entiers naturels
variable : Tab : Tableau d'Entiers naturels de 1 à n éléments
début
pour i de 1 jusqu'à n – 1faire
   début
   m ← i
   pour j de i+1 jusqu'à n faire
     début
     si Tab[j] <  Tab[m] alors
        début
        m ← j
        temp ← Tab[m]
        Tab[m] ← Tab[i]
        Tab[i]  ← temp
         m ← i
        Fin si
     fin pour
   fin pour
Fin Tri_Selection
```

Voici une version correcte et améliorée du précédent (nous allons voir avec la notion de complexité comment appuyer cette intuition d'amélioration), dans laquelle l'on sort l'échange a_i et a_j de la boucle interne j, pour le déporter à la fin de cette boucle.
Au lieu de travailler sur les contenus des cellules de la table, nous travaillons sur les indices, ainsi lorsque a_j est plus petit que a_i nous mémorisons l'indice j du minimum dans une variable m plutôt que le minimum lui-même. A la fin de la boucle interne j, la variable m contient l'indice de minimum de $(a_{i+1}, a_{k+2}, \ldots, a_n)$ et l'on permute l'élément concerné (d'indice m) avec l'élément frontière a_i.

```
Algorithme Tri_Selection2
variable : m, i , j , n, temp : Entiers naturels
variable : Tab : Tableau d'Entiers naturels de 1 à n éléments
début
pour i de 1 jusqu'à n – 1faire
  début
  m ← i
  pour j de i + 1 jusqu'à n faire
    début
    si Tab[j] <  Tab[m] alors
      m ← j
    Fin si
    Fin pour;
  temp ← Tab[ m ] ;
  Tab[ m ] ← Tab[ i ] ;
  Tab[ i ]  ← temp
  Fin pour
Fin Tri_Selection
```

4.4. **Complexité**

Choisissons comme opération élémentaire la comparaison de deux cellules du tableau.

Pour les deux versions 1 et 2 :

Le nombre de comparaisons est une valeur qui ne dépend que de la longueur n de la liste (n est le nombre d'éléments du tableau), ce nombre est égal au de fois que les itérations s'exécutent, le comptage montre que la boucle i s'exécute n – 1 fois, donc une somme de n – 1 termes et qu'à chaque fois la boucle j exécute (n – (i + 1) +1) fois la comparaison.

La complexité en nombre de comparaisons est égale à la somme des n – 1 termes suivants (i = 1, ... i = n – 1)

C = (n – 2) + 1 + (n – 3) + 1 +. .. + 1 + 0 = (n – 1) + (n – 2) + ... + 1 = n.(n – 1)/2 (c'est la somme des n – 1 premiers entiers).

La complexité en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choisissons maintenant comme opération élémentaire l'échange de deux cellules du tableau.

Calculons par dénombrement du nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échange). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse.

Pour la version 1

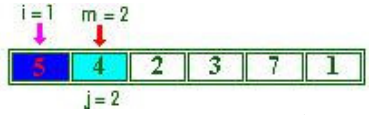
> Au pire, chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.
> La complexité au pire en nombre d'échanges de la version 1 est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Pour la version 2

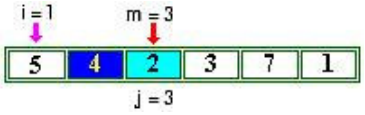
> L'échange a lieu systématiquement dans la boucle principale i qui s'exécute n – 1 fois.

La complexité en nombre d'échanges de cellules de la version 2 est de l'ordre de n, que l'on écrit O(n).
Un échange valant 3 transferts (affectation) la complexité en transfert est O(3n) = O(n)

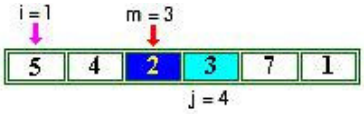
Toutefois cette complexité en nombre d'échanges de cellules n'apparaît pas comme significative du tri, outre le nombre de comparaison, c'est le nombre d'affectations d'indice qui représente une opération fondamentale et là les deux versions ont exactement la même complexité $O(n^2)$.
Exemple : soit la liste à 6 éléments (5 , 4 , 2 , 3 , 7 ,1), appliquons la version 2 du tri par sélection sur cette liste d'entiers. Visualisons les différents états de la liste pour la première itération externe contrôlée par i (i = 1) et pour les itérations internes contrôlées par l'indice j (de j = 2 ... à ... j = 6) :



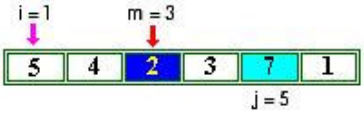
comme 4 < 5 on mémorise dans m



comme 2< 4 on mémorise dans m



comme 2>3 on ne mémorise pas

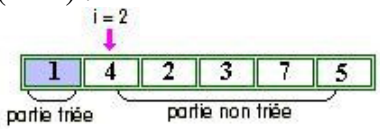


comme 2>7 on ne mémorise pas

comme 1< 2 on mémorise dans m

on échange T[1] et T[6]

L'algorithme ayant terminé l'échange de T[1] et de T[6], il passe à l'itération externe suivante (i = 2) :



etc....

4.5. **Programme pascal**

Seul le programme de la version 2 sera écrit.
Le programme de la version 1 sera fait en exercice à domicile.

```
program TriParSelection2;
const N = 10;
type TTab = array [1..N] of integer;
var Tab : TTab ;

procedure TriSelection (var Tab:TTab) ;
var i, j, t, m : integer;
begin
  for i := 1 to N-1 do
  begin
     m := i;
```

```
    for j := i+1 to N do
    if Tab[ j ] < Tab[ m ] then m := j;
    t := Tab[m];
    Tab[m] := Tab[i];
    Tab[i] := t;
  end;
end;

procedure Initialisation(var Tab:TTab) ;
var i : integer;
begin
  randomize;
  for i := 1 to N do Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
  writeln;
end;

begin
  Initialisation(Tab);
  writeln('TRI PAR SELECTION'); writeln;
  Impression(Tab);
  TriSelection(Tab);
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR SELECTION

------------------------------------------------------------
 28 |  51 |  86 |  43 |  32 |   6 |  52 |  51 |  79 |  42 |
------------------------------------------------------------
  6 |  28 |  32 |  42 |  43 |  51 |  51 |  52 |  79 |  86 |
------------------------------------------------------------
```

5. Le tri par insertion

C'est un tri en général un peu plus coûteux, en particulier en nombre de transferts à effectuer qu'un tri par sélection.

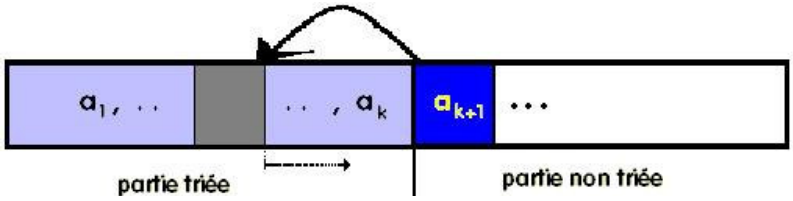
5.1. **Spécification abstraite**

Son principe est de parcourir la liste non triée (a_1, a_2, ... , a_n) en la décomposant en deux parties, une déjà triée et une non triée. La méthode est identique à celle que l'on utilise pour ranger des cartes que l'on tient dans sa main : on insère dans le paquet de cartes déjà rangées

une nouvelle carte au bon endroit. L'opération de base consiste à prendre l'élément frontière dans la partie non triée, puis à l'insérer à sa place dans la partie triée (place que l'on recherchera séquentiellement), puis à déplacer la frontière d'une position vers la droite. Ces insertions s'effectuent tant qu'il reste un élément à ranger dans la partie non triée. L'insertion de l'élément frontière est effectuée par décalages successifs d'une cellule.
La liste (a_1, a_2, ... , a_n) est décomposée en deux parties : une partie triée (a_1, a_2, ... , a_k) et une partie non triée (a_{k+1}, a_{k+2}, ... , a_n); l'élément a_{k+1} est appelé élément frontière (c'est le premier élément non trié).

5.2. Spécification concrète itérative

La suite (a_1, a_2, ... , a_n) est rangée dans un tableau T[...] en mémoire centrale. Le tableau contient une partie triée ((a_1, a_2, ... , a_k) en grisé à gauche) et une partie non triée ((a_{k+1}, a_{k+2}, ... , a_n) en blanc à droite).



En faisant varier j de k jusqu'à 2, afin de balayer toute la partie (a_1, a_2, ... , a_k) déjà rangée, on décale d'une place les éléments plus grands que l'élément frontière :

Tant que $a_{j-1} > a_{k+1}$ *faire*
    décaler a_{j-1} *en* a_j
    passer au j précédent
fin tant que

La boucle s'arrête lorsque $a_{j-1} < a_{k+1}$, ce qui veut dire que l'on vient de trouver au rang j – 1 un élément a_{j-1} plus petit que l'élément frontière a_{k+1}, donc a_{k+1} doit être placé au rang j.

5.3. Algorithme

```
Algorithme Tri_Insertion
variable: i , j , n, v : Entiers naturels
variable: Tab : Tableau d'Entiers naturels de 0 à n éléments
début
  pour i de 2 jusqu' à n faire
  début
     v ← Tab[i]
     j ← i
     Tant que Tab[j – 1] > v faire
     Début
       Tab[j]  ← Tab[j –1]
       j ← j – 1
     Fin Tant que
     Tab[j]  ← v
  Fin pour
Fin Tri_Insertion
```

Sans la sentinelle en T[0], nous aurions une comparaison sur j à l'intérieur de la boucle :

Tant que Tab[j – 1] > v faire
Début
Tab[j] ← Tab[j – 1
j ← j –
si j = 0 alors Sortir de la boucle fin si
Fin Tant

Une proposition serait d'intégrer directement l'imposition j > 0 dans le « tant que » comme suit :

Tant que (Tab[j – 1] > v) et (j >0) faire
Tab[j] ← Tab[j – 1]
j ← j – 1
FinTant

Il y a des problèmes de dépassement d'indice de tableau lors de l'implémentation du programme, essayez d'analyser l'origine du problème en notant que la présence d'une sentinelle élimine le problème.

5.4. **Complexité**

Choisissons comme opération élémentaire la comparaison de deux cellules du tableau.
Dans le pire des cas, le nombre de comparaisons du « tant que » est une valeur qui ne dépend que de la longueur i de la partie (a_1, a_2, ... , a_i) déjà rangée. Il y a donc au pire i comparaisons pour chaque i variant de 2 à n.
La complexité au pire en nombre de comparaison est donc égale à la somme des n termes suivants (i = 2, i = 3, ... i = n)
C = 2 + 3 + 4 +...+ n = n(n + 1)/2 – 1 comparaisons au maximum. (c'est la somme des n premiers entiers moins 1).
La complexité au pire en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choisissons maintenant comme opération élémentaire, le transfert d'une cellule du tableau.
Calculons par dénombrement du nombre de transferts dans le pire des cas. Il y a autant de transferts dans la boucle « tant que » qu'il y a de comparaisons, il faut ajouter 2 transferts par boucle i, soit au total dans le pire des cas : C = n(n + 1)/2 + 2(n – 1) = (n^2 + 5n – 4)/2
La complexité au pire en nombre de transferts est de l'ordre de n^2, que l'on écrit $O(n^2)$.

5.5. **Programme pascal**

```
program TriParInsertion;
const N = 10;
type TTab = array [0..N]of integer;
var Tab : TTab ;
procedure TriInsertion (var Tab:TTab) ;
var i, j, v : integer;
begin
   for i := 2 to N do
   begin
      v := Tab[i];
      j := i ;
      while Tab[j – 1] > v do
```

```
        begin
        Tab[j] := Tab[j – 1] ;
        j := j – 1 ;
        end;
      Tab[j] := v ;
   end
end;

procedure Initialisation(var Tab:TTab) ;
var i : integer;
begin
   randomize;
   for i := 1 to N do
     Tab[i] := random(100);
   Tab[0]:= –Maxint ;
end;

procedure Impression(Tab:TTab) ;
var i : integer;
begin
   writeln('------------------------');
   for i:= 1 to N do write(Tab[i] : 3, ' | ');
   writeln;
end;

begin
   Initialisation(Tab);
   writeln('TRI PAR INSERTION');
   writeln;
   Impression(Tab);
   TriInsertion(Tab);
   Impression(Tab);
   writeln('----------------------');
   end ;
```

Résultat de l'exécution du programme précédent :

```
TRI PAR INSERTION

------------------------------------------------------------
 62 |  15 |  34 |   3 |  25 |  22 |  63 |   3 |  66 |  17 |
------------------------------------------------------------
  3 |   3 |  15 |  17 |  22 |  25 |  34 |  62 |  63 |  66 |
------------------------------------------------------------
```

6. Le tri rapide par Sedgewick

C'est le plus performant des tris en table qui est certainement celui qui est le plus employé dans les programmes. Ce tri a été trouvé par C.A.Hoare, on le réfère à Robert Sedgewick qui a travaillé dans les années 70 sur ce tri et l'a amélioré On va voir les principes de ce tri et sa complexité en moyenne et au pire.

6.1. **Spécification abstraite**

Son principe est de parcourir la liste L = (a_1, a_2, ... , a_n) en la divisant systématiquement en deux sous listes L1 et L2. L'une est telle que tous ses éléments sont inférieurs à tous ceux de l'autre liste et en travaillant séparément sur chacune des deux sous listes en réappliquant la même division à chacune des deux sous listes jusqu'à obtenir uniquement des sous listes à un seul élément.
C'est un algorithme dichotomique qui divise donc le problème en deux sous problèmes dont les résultats sont réutilisés par recombinaison, il est donc de complexité O(n.log(n)).
Pour partitionner une liste L en deux sous listes L1 et L2 : on choisit une valeur quelconque dans la liste L (la dernière par exemple) que l'on dénomme pivot, puis on construit la sous liste L1 comme comprenant tous les éléments de L dont la valeur est inférieure ou égale au pivot, et l'on construit la sous liste L2 comme constituée de tous les éléments dont la valeur est supérieure au pivot.

L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16] prenons comme pivot la dernière valeur pivot = 16.
Nous obtenons par exemple :
L1 = [4, 14, 3, 2]
L2 = [23, 45, 18, 38, 42]
A cette étape voici l'arrangement de L :
L = L1 + pivot + L2 = [4, 14, 3, 2, 16, 23, 45, 18, 38, 42].
En effet, en travaillant sur la table elle-même par réarrangement des valeurs, le pivot 16 est placé au bon endroit directement : [4<16, 14<16, 3<16, 2<16, 16, 23>16, 45>16, 18>16, 38>16, 42>16].
En appliquant la même démarche au deux sous listes : L1 (pivot = 2) et L2 (pivot = 42)
[4, 14, 3, 2, 16, 23, 45, 18, 38, 42] nous obtenons :
L11=[ ] liste vide
L12=[3, 4, 14]
L1=L11 + pivot + L12 = (2, 3, 4, 14)
L21=[23, 38, 18]
L22=[45]
L2=L21 + pivot + L22 = (23, 38, 18, 42, 45)
A cette étape voici le nouvel arrangement de L :
L = [(2, 3, 4, 14), 16, (23, 38, 18, 42, 45)]

Ainsi de proche en proche en subdivisant le problème en deux sous problèmes, à chaque étape on obtient un pivot bien placé.

6.2. **Spécification concrète**

La suite (a_1, a_2, ... , a_n) est rangée dans un tableau T[...] en mémoire centrale.
Le processus de partitionnement décrit ci-haut (appelé aussi segmentation) est le point central du tri rapide, on construit une fonction Partition réalisant cette action. Comme l'on réapplique la même action sur les deux sous listes obtenues après partition, la méthode est donc récursive, le tri rapide est alors une procédure récursive.

6.2.1. Voici une spécification générale de la procédure de tri rapide

```
Tri Rapide sur [a...b]
Partition [a...b] renvoie pivot & [a...b] = [x ... pivot'] + [pivot] + [pivot" ... y]
Tri Rapide sur [pivot" ... y]
Tri Rapide sur [x ... pivot']
```

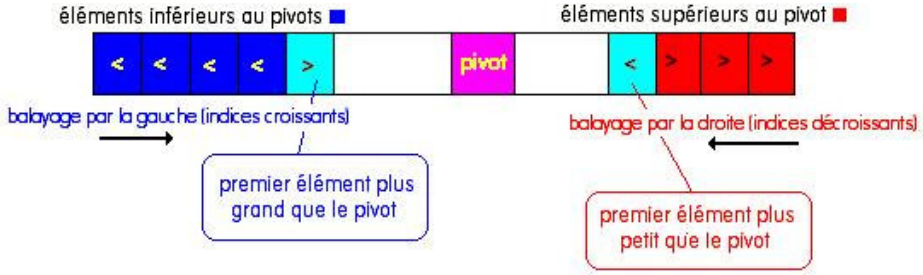
6.2.2. Voici une spécification générale de la fonction de partitionnement

La fonction de partitionnement d'une liste [a...b] doit répondre aux deux conditions suivantes :
renvoyer la valeur de l'indice noté i d'un élément appelé pivot qui est bien placé définitivement : pivot = T[i], établir un réarrangement de la liste [a...b] autour du pivot tel que :
[a...b] = [x ... pivot'] + [pivot] + [pivot" ... y]
[x ... pivot'] = T[G], ... ,T[i – 1] (où : x = T[G] et pivot' = T[i – 1]) tels que les T[G], ... ,T[i – 1] sont tous inférieurs à T[i], [pivot" ... y] = T[i + 1] , ... ,T[D] (où : y = T[D] et pivot" = T[i + 1]) tels que les T[i + 1], ... ,T[D] sont tous supérieurs à T[i].

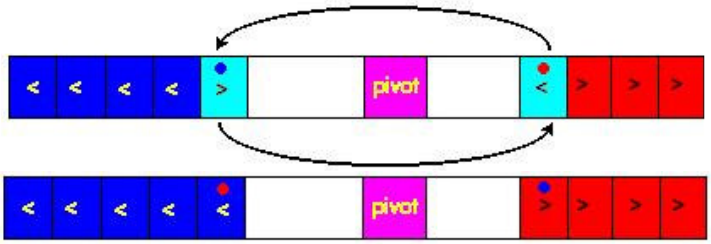
Il est proposé de choisir arbitrairement le pivot que l'on cherche à placer, puis ensuite de balayer la liste à réarranger dans les deux sens (par la gauche et par la droite) en construisant une sous liste à gauche dont les éléments ont une valeur inférieure à celle du pivot et une sous liste à droite dont les éléments ont une valeur supérieure à celle du pivot.

1°) Dans le balayage par la gauche, on ne touche pas à un élément si sa valeur est inférieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus grande que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous liste mais plutôt dans l'autre sous liste

2°) Dans le balayage par la droite, on ne touche pas à un élément si sa valeur est supérieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus petite que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous liste mais plutôt dans l'autre sous liste

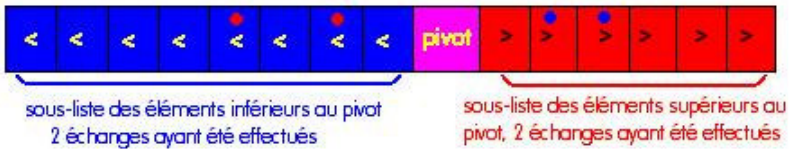


3°) On procède à l'échange des deux éléments mal placés dans chacune des sous listes :



4°) On continue le balayage par la gauche et le balayage par la droite tant que les éléments sont bien placés (valeur inférieure par la gauche et valeur supérieure par la droite), en échangeant à chaque fois les éléments mal placés.

5°) La construction des deux sous listes est terminée dès que l'on atteint (ou dépasse) le pivot.



Appliquons cette démarche à l'exemple précédent : L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]
Choix arbitraire du pivot : l'élément le plus à droite ici 16

a) Balayage à gauche :
   4 < 16 ⇒ il est dans la bonne sous liste, on continue
   liste en cours de construction : [4, 16]
   23 > 16 ⇒ il est mal placé il n'est pas dans la bonne sous liste, on arrête le balayage gauche,
   liste en cours de construction :[4, 23, 16]

b) Balayage à droite :
   38 > 16 ⇒ il est dans la bonne sous liste, on continue
   liste en cours de construction : [4, 23, 16, 38]
   18 > 16 ⇒ il est dans la bonne sous liste, on continue
   liste en cours de construction : [4, 23, 16, 18, 38]
   45 > 16 ⇒ il est dans la bonne sous liste, on continue
   liste en cours de construction : [4, 23, 16, 45, 18, 38]
   14 < 16 ⇒ il est mal placé il n'est pas dans la bonne sous liste, on arrête le balayage droit,
   liste en cours de construction : [4, 23, 16, 14, 45, 18, 38]
   Echange des deux éléments mal placés :
   [4, 23, 16, 14, 45, 18, 38] [4, 14, 16, 23, 45, 18, 38]

c) On reprend le balayage gauche à l'endroit où l'on s'était arrêté :
----------↓----------
[4, 14, 3, 42, 2, 23, 45, 18, 38, 16]
3 < 16 ⇒ il est dans la bonne sous liste, on continue
liste en cours de construction : [4, 14, 3 ,16, 23, 45, 18, 38]
42 > 16 ⇒ il est mal placé il n'est pas dans la bonne sous liste, on arrête de nouveau le balayage gauche,
liste en cours de construction : [4, 14, 3, 42, 16, 23, 45, 18, 38]

d) On reprend le balayage droit à l'endroit où l'on s'était arrêté :
----------↓----------
[4, 14, 3, 42, 2, 23, 45, 18, 38, 16]
2 < 16 ⇒ il est mal placé il n'est pas dans la bonne sous liste, on arrête le balayage droit,
liste en cours de construction : [4, 14, 3, 42, 16, 2, 23, 45, 18, 38]
On procède à l'échange : [4, 14, 3, 2, 16, 42, 23, 45, 18, 38]

e) On arrête la construction puisque nous sommes arrivés au pivot, la fonction partition a terminé son travail elle a évalué :

- le pivot : 16
- la sous liste de gauche : L1 = [4, 14, 3, 2]
- la sous liste de droite : L2 = [23, 45, 18, 38, 42]
- la liste réarrangée : [4, 14, 3, 2, 16, 42, 23, 45, 18, 38]

f) Il reste à recommencer les mêmes opérations sur les parties L1 et L2 jusqu'à ce que les partitions ne contiennent plus qu'un seul élément.

6.3. **Algorithme**

```
Variable : Tab[min..max] tableau d'entier

fonction Partition(G ,D : entier ) résultat : entier
variable : i , j , piv , temp : entier
début
  piv ← Tab[D]
  i ← G – 1
  j ← D
  répéter
     répéter i ← i + 1 jusqu'à Tab[i] >= piv
     répéter j ← j-1 jusqu'à Tab[j] <= piv
     temp ← Tab[i]
     Tab[i] ← Tab[j]
     Tab[j] ← temp
  jusqu'à j <= i
  Tab[j] ← Tab[i]
  Tab[i] ← Tab[d]
  Tab[d] ← temp
  résultat ← i
FinPartition
```

```
Procédure TriRapide(G, D : entier );
Variable :  i : entier
début
  si D > G alors
    début
    i ← Partition(G ,D)
    TriRapide(G , i – 1 )
    TriRapide(i+1, D)
  Finsi
FinTRiRapide
```

On suppose avoir mis une sentinelle dans le tableau, dans la première cellule la plus à gauche, avec une valeur plus petite que n'importe qu'elle autre valeur du tableau.
Cette sentinelle est utile lorsque le pivot choisi aléatoirement se trouve être le plus petit élément de la table

Comme nous avons : ∀j, Tab[j] > piv, alors la boucle sur j pourrait ne pas s'arrêter ou bien s'arrêter sur un message d'erreur.
La sentinelle étant plus petite que tous les éléments y compris le pivot arrêtera la boucle et encore une fois évite de programmer le cas particulier du pivot = min (a_1, a_2, ... , a_n).

6.4. **Complexité**

Seuls les résultats classiques et connus mathématiquement, pour les démonstrations il est possibles d'aller voir dans les ouvrages de R. Sedgewick & Aho-Ullman.

L'opération élémentaire choisie est la comparaison de deux cellules du tableau.
Comme tous les algorithmes qui divisent et traite le problème en sous problèmes le nombre moyen de comparaisons est en O(n.log(n)) que l'on nomme complexité moyenne.
L'expérience pratique montre que cette complexité moyenne en O(n.log(n)) n'est atteinte que lorsque les pivots successifs divisent la liste en deux sous listes de taille à peu près équivalente. Dans le pire des cas (par exemple le pivot choisi est systématiquement à chaque fois la plus grande valeur) on montre que la complexité est en $O(n^2)$.

6.5. **Programme pascal**

```
program TriQuickSort;
const N = 10;
type TTab = array [0..N] of integer;
var Tab : TTab ;
function Partition ( G , D : integer) : integer;
var i , j : Integer;
var piv, temp  : integer;
begin
i := G – 1;
j := D;
piv := Tab[D];
```

```
repeat
   begin
   repeat i := i + 1 until Tab[i] >= piv;
   repeat j := j – 1 until Tab[j] <= piv;
   temp :=Tab[i];
   Tab[i] :=Tab[j];
   Tab[j] :=temp;
   end ;
until j <= i;
Tab[j] := Tab[i];
Tab[i] := Tab[d];
Tab[d] := temp;
result := i;
end ;
```

```
procedure Initialisation(var Tab:TTab) ;
var i : integer;
begin
randomize;
for i := 1 to N do Tab[i] := random(100);
Tab[0] := –Maxint ;
end;
```

```
procedure Impression(Tab:TTab) ;
var i : integer;
begin
writeln('------------------------');
for i:= 1 to N do write(Tab[i] : 3, ' | ');
writeln;
end;
```

```
begin
Initialisation(Tab);
writeln('TRI RAPIDE');
writeln;
Impression(Tab);
TriRapide( 1 , N );
Impression(Tab);
writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI RAPIDE

-------------------------------------------------------------
 17 |  32 |  14 |  45 |  54 |  50 |  60 |  10 |  68 |  12 |
-------------------------------------------------------------
 10 |  12 |  14 |  17 |  32 |  45 |  50 |  54 |  60 |  68 |
-------------------------------------------------------------
```

Chapitre IX : Les unités en Pascal

1. Introduction

Lorsque on écrit un programme dans un langage et en particulier en Pascal, on utilise un certain nombre d'instructions (procédures et fonction) qui sont définies dans des unités extérieures au programme (par exemple : sin, cos, mod, …).
L'unité qui est toujours associée automatiquement est l'unité *system*.
Il en existe d'autres qui sont fournies avec Pascal : *crt*, *dos*, *graph*, *strings, …*
Elles ne pourront être appelées que si les unités correspondantes sont déclarées dans le programme par la commande *uses*.
Il faut savoir qu'une unité s'écrit avec les commandes Pascal. Le fichier ainsi crée aura comme extension *.PAS* mais le fichier compilé aura l'extension *.TPU* ou *.PPW*.

2. Syntaxe

Quatre parties doivent composer une unité :

- déclaration de l'unité ;
- une partie déclarative qui catalogue le contenu de l'unité, elle doit commencer par le mot *INTERFACE* ;
- une partie fonctionnelle : procédures ou fonctions qui seront appelées par le programme principal ;
- bloc *Begin End.* qui ne doit RIEN contenir excepté les initialisations des variables.

3. Exemple

Nous prendrons comme exemple la fonction mathématique « tangente d'un angle » qui n'est pas définie comme fonction mathématique de base. Il est à noter que la fonction inverse par contre existe bien en Pascal. Par la mathématique, nous savons que $\text{tg}\,x = \frac{\sin x}{\cos x}$.

Dans la colonne de gauche, on trouve le code pour la définition de la tangente.
Par contre dans la colonne de droite, le code pour un programme qui calculera la tangente d'un angle quelconque.

```
Unit Tangt;
INTERFACE
Var a : Real ;
Function Tan ( a : Real ) : Real;
IMPLEMENTATION
Function Tan ( a : Real ) : Real;
Begin
Tan := Sin(a)/Cos(a);
End ;
BEGIN
END.
```

```
Program Tangente ;
Uses Tangt,crt ;
Var x,z : Real ;
var angle : integer;
BEGIN
write('Entrez un angle en degre : ') ;
readln(angle);
if (((angle mod 90) = 0) and (((angle div 90) mod 2) = 1)) then
   writeln('La tangente ne peut etre calculee')
   else
   begin
   x:=angle*pi/180;
   z:=tan(x);
   if(abs(z)<0.000001) then z:=abs(z);
   writeln('Voici sa tangente : ',z:6:3);
   end;
readkey;
END.
```

L'unité plus complète que la précédente qui est composée de la fonction tangente et le calcul de la puissance entière positive ou négative.
L'unité

```
Unit Mathe;
INTERFACE
Var a : Real ;
var n : integer;
Function exp ( a : Real;n :integer ) : Real ;
Function Tan ( a : Real ) : Real;
IMPLEMENTATION
Function exp ( a : Real;n: integer ) : Real ;
var i :integer;
var calc :real;
Begin
calc:=1.;
for i:=1 to abs(n) do calc:=calc*a;
if (n < 0) then calc:=1/calc;
exp:=calc;
End ;
Function Tan ( a : Real ) : Real ;
Begin
Tan := Sin(a)/Cos(a) ;
End ;
BEGIN
END.
```

Le programme utilisant l'unité « mathe ».

```
Program math;
uses crt,mathe;
var base,resultat,angle : real;
var n : integer;
begin
writeln('donner la valeur de la base');
readln(base);
writeln('donner l''exposant');
readln(n);
resultat:=exp(base,n);
writeln('le resultat est ',resultat:8:4);
writeln('Donner l''angle en degre pour la tangente');
readln(n);
angle:=n*pi/180;
resultat:=tan(angle);
writeln('La tangente vaut ',resultat:6:2);
readkey;
end.
```

Chapitre X : Gestions de fichiers

1. Introduction

Il est toujours intéressant de stocker des données dans un fichier pour pouvoir les analyser plus tranquillement. De plus, dans un fichier, nous ne sommes pas limités comme pour un écran au nombre de lignes.
De même, lorsqu'on lance un programme, nous devons souvent lui donner des entrées qui peuvent être parfois longues et fastidieuses, alors que si nous les prenons d'un fichier, cela nous économisera du temps.
Il est clair aussi, dans l'avenir, que les programmes donnent un résultat à ce moment mais dès l'arrêt de programme, les variables sont perdues. Voilà donc un grand intérêt des fichiers, qui permettront de continuer un programme là où nous l'avons arrêté pour ensuite ajouter, modifier ou supprimer des données.

2. Déclaration

Un fichier en Pascal est une suite linéaire d'informations dont un seul élément est immédiatement accessible. Seul cet élément est modifiable.
On peut insérer à la suite de ce dernier élément, mais il n'est pas possible d'insérer dans un endroit de la suite ou de supprimer un élément de la suite.
Pour cela, il faut réécrire toute la suite du fichier. Cela ressemble à un STRING.
Avec comme différence :

- le fichier n'a pas de dimension définie à l'avance ;
- chaque accès à un élément avance automatiquement la position actuelle à l'élément suivant ;
- la position actuelle peut dépasser de un le dernier élément présent, on dit alors qu'on est en fin de fichier (EOF). C'est à ce moment que l'on peut ajouter un nouvel élément ;
- le seul mouvement permis et même forcé est l'avance automatique d'un élément à chaque accès.

Ainsi, nous pouvons voir que les fichiers n'existent pas en tant que tel mais on peut associer un type à un fichier de l'OS.

Nous nous occuperons uniquement des fichiers de format texte (chaînes de caractères et nombres), et nous déclarerons cela comme « text ».

Var f : text ;

Dans ces fichiers au format texte, nous y retrouvons deux codes spéciaux : retour chariot (code ASCII 10) et passage à la ligne (code ASCII 13) à la fin de chaque ligne du fichier.

Le fichier lu pour être écrit à l'écran devra passer par une variable « classique » qui sera inévitablement de type « STRING ». De même, lors de l'écriture dans le fichier, nous utiliserons cette même variable string pour écrire.

3. Lecture, écriture

3.1. Affectation

Avant de travailler sur un fichier, il faut le déclarer en lui affectant une variable qui servira à désigner le fichier tout au long du programme.
Assign s'applique à tous les types de fichiers.

Assign (variable d'appel ,nom du fichier) ;
Assign (f , ´c:\essai.txt´) ;
Il va donc associer à la variable f, le fichier qui se trouvera à la racine du disque C qui se nomme essai.txt.

3.2. Initialisation

Il faut renvoyer le pointeur[10] (la position du curseur) au début du fichier pour pouvoir le lire ou écrire (toujours à partir du début).

Reset (variable d'appel) ;
Reset (f) ;

3.3. Création ou effacement

Si le fichier n'existe pas ou si on veut effacer le contenu pour le réécrire, nous utiliserons la commande *rewrite*.

Rewrite (variable d'appel) ;
Rewrite (f) ;

Attention, lorsque cette commande a été écrite, cela implique que le fichier ne pourra plus être lu, seul l'écriture sera possible.

3.4. Lecture et écriture

Pour lire ou écrire dans un fichier, les commandes *readln*, *writeln*, *read* et *write* seront utilisées et cela associé à des variables de types string.

Readln (variable d'appel , variable) ;
Readln (f , texte) ; (avec texte défini comme variable de type string)
Read (variable d'appel , variable) ;
Read (f , texte) ; (avec texte défini comme variable de type string)
Writeln (variable d'appel , variable) ;
Writeln (f , texte) ; (avec texte défini comme variable de type string)
Write (variable d'appel , variable) ;
Write (f , texte) ; (avec texte défini comme variable de type string)

3.5. Fermeture

Il est impératif de fermer les fichiers ouverts pendant un programme sous peine de voir les données inscrites perdues.

Close (variable d'appel) ;
Close (f) ;

[10] Nous verrons les pointeurs plus en détails dans le chapitre suivant.

4. Fonctions supplémentaires

4.1. Fin de fichier (EOF : End Of File)

La fonction *eof (variable d'appel)* renvoie une valeur booléenne. Cela permet de savoir si le pointeur est à la fin du fichier ou non.
La fonction qui est en fait une procédure de lecture, *seekeof (variable d'appel)* qui renvoie aussi un booléen mais qui en plus avance la position du pointeur.

4.2. Fin de ligne (EOLN : End Of LiNe)

La fonction *eoln (variable d'appel)* renvoie une valeur booléenne. Cela permet de savoir si le pointeur est à la fin de la ligne ou non.
La fonction qui est en fait une procédure de lecture, *seekeoln (variable d'appel)* qui renvoie aussi un booléen mais qui en plus avance la position du pointeur.

4.3. Effacer, renommer, tronquer un fichier

Erase (variable d'appel), Rename (variable d'appel, nouveau nom), Truncate (variable d'appel) permettent respectivement d'effacer le fichier, de le renommer ou encore de le tronquer, c'est-à-dire tout supprimer ce qui trouve après la position courante du pointeur. Pour les deux premiers, le fichier doit être fermé.

4.4. Appel d'un fichier exécutable

Il est possible d'appeler un fichier exécutable externe à partir d'un programme écrit en Pascal et de lui assigner des paramètres grâce à la commande *Exec*. Cette commande nécessite un commentaire de compilation : { $M $4000,0,0 }.
Swapvectors ;
Exec (nom+chemin, paramètres) ;
Swapvectors ;

Cette commande ne fonctionne que si le programme exécuté est d'extension .exe[11].

5. Exemples

5.1. **Application des différentes fonctions**

Cet exemple permet de voir presque toutes les fonctions écrites ci-dessus avec leurs résultats. La première ligne est nécessaire comme commentaire au compilateur pour la commande exec.

```
{ $M $4000,0,0 }
Program fichier;
uses crt,dos;
var f,g : text;
var i,param : string;
var zl,zf : boolean;
var k :integer;
begin
param:='/P';
```

[11] Il est aussi possible d'utiliser des fichiers d'extension .com et .bat en utilisant un programme qui convertit ces fichiers en extension .exe.

```
k:=0;
assign(f,'c:\essai.txt');
assign(g,'c:\copie.txt');
reset(f);
reset(g);
rewrite(g);

while not eof(f) do
begin
    readln(f,i);
    writeln(i,chr(20));
    writeln(g,i,chr(20));
end;
reset(f);
repeat
    writeln;
    writeln;
    writeln(g);
    writeln(g);
    readln(f,i);
    writeln(i);
    writeln(g,i);
until eof(f);

reset(f);
repeat
begin
    k:=k+1;
    read(f,i);
    if eoln(f) then
    begin
        zl:=eoln(f);
        zf:=eof(f);
        write('Fin de ligne ');
        writeln('zf :',zf,' zl :',zl);
    end
    else
        if eof(f) then
        begin
            zl:=eoln(f);
            zf:=eof(f);
            write('Fin de fichier ');
            writeln('zf :',zf,' zl :',zl);
        end
        else
            begin
                zl:=eoln(f);
                zf:=eof(f);
                write('Rien du tout ');
                writeln('zf :',zf,' zl :',zl);
                end;
    writeln('L''indice est de ',k);
    if (k mod 10 = 0) then readkey;
end;
```

```
until seekeof(f);
zf:=eof(f);
zl:=eoln(f);
if eof(f) then
begin
     zl:=eoln(f);
     zf:=eof(f);
     write('Fin de fichier ');
     writeln('zf :',zf,' zl :',zl);
end;
close(g);
rename(g,'c:\nouveau.txt');
swapvectors;
exec('C:\ESSAI.EXE',param);
swapvectors;
readkey;
erase(g);
readkey;
end.
```

5.2. **Ecriture et lecture d'un fichier**

Dans la colonne de gauche, la source pour l'encodage et le stockage des données dans un fichier appelé « data.txt ».
Par contre dans la colonne de droite, la source qui permet de lire à partir du fichier précédent et de placer les données dans les variables du programme.
Avec ce second programme, il sera possible d'effectuer des modifications dans les données sans oublier de les recopier dans le fichier « data.txt ».
Il est clair que les modifications apportées aux variables de ce second programme seront perdues dans l'arrêt de celui-ci.
Dans cet exemple, on a considéré 5 personnes avec 3 champs (nom, prénom et l'adresse).

Voici le code pour le programme d'encodage des données :

```
Program fichier_ecriture;
uses crt;
var f:text;
var j:string;
var i:integer;
begin
assign(f,'z:\data.txt');
rewrite(f);
for i:=1 to 5 do
       begin
       clrscr;
       writeln('Donnees concernant la personne ',i);
       writeln('Donner son nom');
       readln(j);
       write(f,j);
       write(f,chr(9));
       writeln('Donner son prenom');
       readln(j);
       write(f,j);
```

```
        write(f,chr(9));
        writeln('Donner son adresse');
        readln(j);
        write(f,j);
        writeln(f,chr(9));
        end;
close(f);
end.
```

Voici le code pour le programme de lecture des données :

```
Program fichier_lecture;
uses crt;
var donnee : array[1..6,1..3] of string;
var f:text;
var j:string;
var i,k,compteur : integer;
begin
donnee[6,1]:='Nom';
donnee[6,2]:='Prenom';
donnee[6,3]:='Adresse';
assign(f,'z:\data.txt');
reset(f);
compteur:=1;
while not eof(f) do
        begin
        readln(f,j);
        i:=1;
        for k:=1 to 3 do
            begin
            while (j[i] <> chr(9)) do
                begin
                donnee[compteur,k]:=donnee[compteur,k]+j[i];
                i:=i+1;
                end;
            i:=i+1;
            end;
        compteur:=compteur+1;
        end;
close(f);
for i:=1 to 5 do
        begin
        clrscr;
        writeln('Donnees de la personne ',i);
        for k:=1 to 3 do writeln(donnee[6,k],' : ',donnee[i,k]);
        readkey;
        end;
end
```

Chapitre XI : Graphisme et affichage en Pascal

1. Introduction

A l'aide des unités que l'on peut créer ou que l'on peut utiliser, il sera plus simple d'utiliser la partie graphique du Pascal (fenêtrage, couleur, souris, …).
Pour cela quelques unités doivent être déclarées au départ.
Nous déclarons déjà l'unité « crt » qui nous permet d'effacer de l'écran à l'aide de la fonction « clrscr » et la fonction « readkey » qui permet d'effectuer une pause.
Nous allons voir d'autres fonctions qui appartiennent à l'unité « dos » et « graph ».

2. L'unité « crt »

Pour déclarer cette unité, il faut utiliser la commande « uses » et placer à la suite « crt ».

2.1. ClrScr

Cette fonction permet d'effacer tout l'écran et de placer le curseur en haut à gauche de l'écran. Très utile au démarrage d'un programme ou lorsqu'il faut une nouvelle page.

2.2. DelLine

Cette fonction permet d'effacer la ligne courante, c'est-à-dire celle qui contient le curseur.

2.3. InsLine

Cette fonction permet d'insérer une ligne vide à la position courante du curseur.

2.4. ClrEol

Cette fonction permet d'effacer une ligne à partir de la position courante du curseur.
La position du curseur n'est pas modifiée.

2.5. TextBackgound (x)

Cette fonction qui demande un paramètre x (valeur comprise entre 0 et 15), permet de choisir une couleur de fond pour le texte.
Pour que la couleur de fond de l'écran soit celle voulue, il faut écrire :
Textbackground(4) ;
Clrscr ;

La suite de ces deux instructions donnera un fond de couleur rouge.

2.6. TextColor (x)

Cette fonction qui demande un paramètre x (valeur comprise entre 0 et 15), permet de choisir une couleur pour le texte.
Si on y ajoute au paramètre x « *+blink* » alors le texte clignotera ou sera surligné.

2.7. Window (x_1, x_2, y_1, y_2)

Cette fonction permet de créer une fenêtre à écran. x_1 et y_1 sont les coordonnées du caractère en haut à gauche et x_2 et y_2 sont les coordonnées du caractère en bas à droite.
Il est à rappeler qu'un écran en mode texte comporte 80 colonnes et 25 lignes.

2.8. **GotoXY (x, y)**

Cette fonction permet de positionner le curseur à la position dans l'écran ou dans un fenêtre si celle-ci a été définie auparavant. x et y sont respectivement le numéro de la colonne et de la ligne.

2.9. **WhereX – WhereY**

Ces fonctions permettent se connaître la position en colonne pour « WhereX » et en ligne pour « WhereY » du curseur. Il faut donc associer ces fonctions à une variable de type « integer ».

3. L'unité « DOS »

Cette unité est liée à l'environnement DOS. L'ensemble des fonctions et procédures pour cette unité se trouve dans l'annexe.
Voici quelques unités intéressantes :

3.1. **Exec(chemin,commande)**

Cette procédure associée à la procédure SwapVectors permet d'exécuter un programme externe, spécifié par les paramètres « chemin » qui est le chemin complet et commande qui sont les paramètres associés à ce programme.

3.2. **Fsplit(nom : string, d : dirStr, n : nameStr, e : extstr)**

Cette procédure divise la chaîne de caractère « nom »en 3 parties, le chemin dans la variable « d », le nom dans la variable « n » et l'extension dans la variable « e ».

3.3. **GetDate(var an,mois,jour,journee : word)**

Cette procédure permet de placer dans les variables :
- ✓ « an » l'année système actuelle ;
- ✓ « mois », le mois système actuel ;
- ✓ « jour » le jour système actuel ;
- ✓ « journee » la valeur numérique du jour correspondant :
  0 = dimanche, 1 = lundi, 2 = mardi, 3 = mercredi, 4 = jeudi, 5 = vendredi et 6 = samedi.

3.4. **GetTime(h,m,s,s100 : word)**

Cette procédure permet de placer dans les variables :
- ✓ « h » les heures système actuelles ;
- ✓ « m » les minutes système actuels ;
- ✓ « s » les secondes système actuelles ;
- ✓ « s100 » les centièmes de seconde système actuelles.

3.5. **SetTime(h,m,s,s100 : word)**

Cette procédure permet de changer l'heure système.

4. L'unité « GRAPH »

Cette unité a la particularité de créer des éléments graphiques tel que des lignes, cercles, rectangles, arc, … .
Il est à noter que cette unité utilise l'environnement DOS et donc la résolution maximale est 640 X 480 X 16.

4.1. Initialisation

```
Uses graph ;
Var pilote, mode : smallint ;
Begin
Pilote := detect ;
Initgraph(pilote, mode, '') ;
…
Closegraph ;
End ;
```

L'origine de l'écran graphique se trouve en haut à gauche (0,0) et l'extrémité est le point (629,479) qui se trouve en bas à droite.

4.2. Les fonctions et les procédures

- ✓ setColor(couleur) : procédure qui permet de choisir la couleur d'avant-plan de l'objet dessiné.
  Les couleurs possibles sont :

Code ou Nom Couleur	Description	Code ou Nom Couleur	Description
0 ou Black	noir	8 ou DarkGray	gris foncé
1 ou Blue	bleu	9 ou LightBlue	bleu flou
2 ou Green	vert foncé	10 ou LightGreen	vert clair
3 ou Cyan	cyan foncé	11 ou LightCyan	cyan clair
4 ou Red	rouge	12 ou LightRed	rouge clair
5 ou Magenta	mauve foncé	13 ou LightMagenta	mauve clair
6 ou Brown	marron	14 ou Yellow	Jaune
7 ou LightGray	gris clair	15 ou White	Blanc

- ✓ setBkColor(couleur) : procédure qui permet de choir la couleur d'arrière-plan après avoir effacé l'écran.
- ✓ getBkColor() : fonction qui renvoie la valeur du code couleur de l'arrière-plan.
- ✓ clearDevice : procédure qui permet d'effacer l'écran.
- ✓ closeGraph : procédure qui permet de fermer le mode graphique.
- ✓ getMaxX() : fonction qui renvoie la valeur maximale de l'abscisse.
- ✓ getMaxY() : fonction qui renvoie la valeur maximale de l'ordonnée.
- ✓ getX() : fonction qui renvoie la valeur de l'abscisse du curseur.

- ✓ getY() : fonction qui renvoie la valeur de l'ordonnée du curseur.
- ✓ getPixel(x, y) : fonction qui renvoie la valeur du code couleur du point (x, y).
- ✓ arc(x, y, angledeb, anglefin, r) : procédure qui dessine un arc de cercle de centre (x, y), de rayon r, commençant à l'angle angledeb et se terminant à l'angle anglefin.
- ✓ Bar`(x1, y1, x2, y2) : procedure qui d`essine un rectangle plein ayant pour coin supérieur gauche `(x1, y1)` et pour coin inférieur droit `(x2, y2)`.
- ✓ Bar3D`(x1, y1, x2, y2, prof, top) : procedure qui d`essine un rectangle plein en 3D ayant pour coin supérieur gauche `(x1, y1)`, pour coin inférieur droit `(x2,y2)`, pour profondeur `prof`, et dont le dessus est dessiné ou non selon la valeur de `top`.
- ✓ Circle`(x, y, r) : procedure qui d`essine un cercle de centre `(x, y)` et de rayon `r`.
- ✓ Ellipse `(x, y, a1, a2, r1, r2) : procédure qui d`essine un arc d'ellipse de centre `(x, y)`, allant de l'angle `a1` à `a2`, et de rayon horizontal `r1` et vertical `r2`.
- ✓ FillEllipse `(x, y, r1, r2) : procedure qui d`essine une ellipse pleine de centre `(x, y)`, de rayon horizontal `r1` et vertical `r2`.
- ✓ `FillPoly (nbpoints, poly) : procedure qui d`essine un polygone plein à `nbpoints` points de coordonnées données par le tableau.
- ✓ FloodFill(x, y, Bord) : procédure qui remplit la surface fermée contenant le point (x, y) par la couleur de sa bordure.
- ✓ Line (x1, y1, x2, y2) : procédure qui dessine une ligne du point de coordonnées (x_1, y_1) au point (x_2, y_2).
- ✓ LineRel (x, y) : procédure qui dessine une ligne du point courant jusqu'au point de coordonnées $(x_{courant} + x, y_{courant} + y)$.
- ✓ LineTo (x, y) : procédure qui dessine une ligne du point courant au point de coordonnées (x, y).
- ✓ MoveTo (x, y) : procédure qui déplace le point courant au point de coordonnées (x, y).
- ✓ OutText (st) : procedure qui affiche à l'écran, à la position actuelle du point, le texte st.
- ✓ OutTextXY (x, y, st) : procédure qui affiche à l'écran, à la position (x, y), le texte st. La position du point est inchangée.
- ✓ PieSlice (x_C, y_C, angleDebut, angleFin, rayon) : procedure qui dessine un secteur circulaire de centre C de coordonnées x_c et y_c, l'angle est en degré.
- ✓ PutPixel (x, y, c) : procédure qui affiche un point de couleur c aux coordonnées (x, y).
- ✓ Rectangle (x1, y1, x2, y2) : procédure qui dessine un rectangle ayant pour coin supérieur gauche `(`x_1`, `y_1`)` et pour coin inférieur droit `(`x_2`, `y_2`)`.

D'autres fonctions et procédures se trouvent dans l'annexe qui suit.

Annexe : Index des fonctions et procédures en Pascal

1. L'unité « System »

Abs	function Abs(X): résultat du type que le paramètre	Fournit la valeur absolue de X.
Addr	function Addr(X): pointeur;	Fournit l'adresse de X.
Append	procedure Append(var f: Text);	Ouvre le fichier F existant pour ajouter des éléments en fin de fichier.
ArcTan	function ArcTan(X: Real): Real;	Fournit l'arc dont la tangente vaut X.
Assign	procedure Assign(var f; String);	Assigne le nom d'un fichier sur disque à la variable f de type fichier.
Assigned	function Assigned(var P): Boolean;	Teste si un pointeur ou une variable procédurale est le pointeur NIL.
Blockread	procedure BlockRead(var F: File; var Buf; Count: Word [; var Result:Word]);	Lit, sur F, Count RECORDs et les stocke dans la variable Buf. Result contient le nombre de RECORD réellement lus.
Blockwrite	procedure BlockWrite(var f: File; var Buf; Count: Word [; var Result:Word]);	Ecrit, sur F, Count RECORDs stockées dans la variable Buf. Result contient le nombre de RECORD réellement écrits.
Break	procedure Break	Termine une boucle FOR, WHILE ou REPEAT.
ChDir	procedure ChDir(S: String);	Change le répertoire actuel pour le répertoire S.
Chr	function Chr(X: Byte): Char;	Fournit le caractère dont le code ASCII est X.
Close	procedure Close(var F);	Ferme le fichier F ouvert.
Concat	function Concat(s1 [, s2,..., sn]: String): String;	Concatène les chaînes stockées dans les variables s1, s2, ... Est avantageusement remplacé par le symbole +
Continue	procedure Continue	Continue une boucle FOR, WHILE, REPEAT interrompue par Break.
Copy	function Copy(S: String; Index: Integer; Count: Integer): String;	Fournit la sous chaîne de S commençant en position Index et comportant les Count caractères consécutifs (vers la droite).
Cos	function Cos(X: Real): Real;	Fournit le cosinus de l'angle X exprimé en radians.
Cseg	function CSeg: Word;	Fournit la valeur actuelle du registre CS (Code Segment).
Dec	procedure Dec(var X[; N: Longint]);	Décrémente la variable X de N.
Delete	procedure Delete(var S: String; Index: Integer; Count:Integer);	Supprime, dans S, Count caractères à partir du caractère en position Index.
Dispose	procedure Dispose(var P: Pointer [, Destructor]);	Libère une variable dynamique (e.a. pointeur).
Dseg	function DSeg: Word;	Fournit la valeur actuelle du registre DS (Data Segment).
EOF(text files)	function Eof [(var F: Text)]: Boolean;	Indique si le pointeur de fichier est en fin de fichier F ou non.
EOF(untyped, typed files)	function Eof(var F): Boolean;	Indique si le pointeur de fichier est en fin de fichier F ou non.
EoLn	function Eoln [(var F: Text)]: Boolean;	Indique si le pointeur de fichier est en fin de lign, dans un fichier texte F.
Erase	procedure Erase(var F);	Efface un fichier F.
Exit	procedure Exit;	Quitte immédiatement le bloc en cours. Si le bloc en cours est le programme principal, le programme s'arrête.
Exp	function Exp(X: Real): Real;	Fournit l'exponentielle de X.
FilePos	function FilePos(var F): Longint;	Fournit la position actuelle (numéro du RECORD) du pointeur de fichier dans le fichier F. Le premier RECORD porte le numéro 0.
FileSize	function FileSize(var F): Longint;	Fournit la taille du fichier F en nombre de RECORD.
FillChar	procedure FillChar(var X; Count: Word; value);	Remplit la variable X avec Count valeurs Value (byte ou char)
Flush	procedure Flush(var F: Text);	Vide le tampon d'un fichier Text ouvert en écriture.

Frac	function Frac(X: Real): Real;	Fournit la partie décimale du nombre X.
FreeMem	procedure FreeMem(var P: Pointer; Size: Word);	Libère de Size octets la place mémoire utilisée par une variable dynamique.
GetDir	procedure GetDir(D: Byte; var S: String);	Fournit le répertoire actuel du disque D.
GetMem	procedure GetMem(var P: Pointer; Size: Word);	Crée une variable dynamique de la taille Size et place l'adresse du bloc dans P.
Halt	procedure Halt [(Exitcode: Word)];	Stoppe l'exécution du programme et renvoie la valeur Exitcode au système d'exploitation.
Hi	function Hi(X): Byte;	Fournit l'octet de poids fort de X.
High	function High(X)	Fournit la plus grande valeur possible que peut prendre X.
Inc	procedure Inc(var X [; N: Longint]);	Incrémente X de N.
Include	procedure Include(var S: set of T; I:T);	Inclut l'élément I dans l'ensemble S.
Insert	procedure Insert(Source: String; var S: String; Index: Integer);	Insère la sous chaîne S dans la chaîne Source à la position Index.
Int	function Int(X: Real): Real;	Fournit la partie entière de X.
IOResult	function IOResult: Integer;	Fournit l'état de la dernière entrée/sortie réalisée.
Length	function Length(S: String): Integer;	Fournit la longueur de la chaîne S.
Ln	function Ln(X: Real): Real;	Fournit le logarithme naturel (base e) de X.
Lo	function Lo(X): Byte;	Fournit l'octet de poids faible de X.
Low	function Low(X);	Fournit la plus petite valeur que peut prendre la variable X.
MaxAvail	function MaxAvail: Longint;	Fournit la taille du plus grand bloc contigu libre dans le tas (heap).
MemAvail	function MemAvail: Longint;	Fournit la taille de toute la mémoire libre dans le tas (heap).
MkDir	procedure MkDir(S: String);	Crée le sous répertoire S.
Move	procedure Move(var Source, Dest; Count: Word);	Copie Count octets de l'adresse Source à l'adresse Dest.
New	procedure New(var P: Pointer [, Init: Constructor]);	Crée une nouvelle variable dynamique et indique à la variable P de type pointeur de pointer vers elle.
Odd	function Odd(X: Longint): Boolean;	Teste si X est pair.
Ofs	function Ofs(X): Word;	Fournit l'offset de X.
Ord	function Ord(X): Longint;	Fournit la position de X dans l'ensemble ordonné dont il fait partie. Ord est surtout utilisé pour déterminer le code ASCII d'un caractère.
ParamCount	function ParamCount: Word;	Fournit le nombre de paramètres spécifiés sur la ligne de commande.
ParamStr	function ParamStr(Index): String;	Fournit le paramètre en position Index sur la ligne de commande.
Pi	function Pi: Real;	Fournit la valeur de Pi = 3.1415926535897932385.
Pos	function Pos(Substr: String; S: String): Byte;	Fournit la position de la sous chaîne Substr dans S et renvoie 0 si elle est absente.
Pred	function Pred(X): < Same type as parameter >;	Fournit le prédécesseur de X dans l'ensemble ordonné auquel il appartient.
Ptr	function Ptr(Seg, Ofs: Word): Pointer;	Convertit le couple segment/offset en une adresse de type Pointer.
Random	function Random [(Range: Word)]: < Same type as parameter >;	Fournit un nombre pseudo aléatoire.
Randomize	procedure Randomize;	Initialise le générateur de nombres pseudo aléatoires.
Read (text files)	procedure Read([var F: Text;] V1 [, V2,...,Vn]);	Lit, éventuellement sur le fichier Text F, les variables V1, ...
Read (typed files)	procedure Read(F , V1 [, V2,...,Vn]);	Lit sur le fichier F les variables V1, ...
Readln	procedure Readln([var F: Text;] V1 [, V2, ...,Vn]);	Exécute la lecture et passe à la ligne suivante du fichier F.

Annexe : Index des fonctions et procédures en Pascal

Rename	procedure Rename(var F; Newname);	Renomme un fichier externe.
Reset	procedure Reset(var F [: File; Recsize: Word]);	Ouvre un fichier existant.
Rewrite	procedure Rewrite(var F: File [; Recsize: Word]);	Crée et ouvre un fichier. Si le fichier existait déjà, il est effacé.
RmDir	procedure RmDir(S: String);	Supprime un sous répertoire vide.
Round	function Round(X: Real): Longint;	Arrondit un nombre réel en un nombre entier.
RunError	procedure RunError [(Errorcode: Byte)];	A le même effet que HALT sauf que Errorcode est affiché à l'écran .
Seek	procedure Seek(var F; N: Longint);	Déplace le pointeur de position dans le fichier F à l'enregistrement N.
SeekEof	function SeekEof [(var F: Text)]: Boolean;	Fournit l'état du end-of-file du fichier.
SeekEoln	function SeekEoln [(var F: Text)]: Boolean;	Fournit l'état du end-of-line du fichier.
Seg	function Seg(X): Word;	Fournit la partie segment du premier octet de l'adresse de X.
SetTextBuf	procedure SetTextBuf(var F: Text; var Buf [; Size: Word]);	Assigne de la mémoire tampon pour les I/O sur les fichiers Text.
Sin	function Sin(X: Real): Real;	Fournit le sinus de X.
SizeOf	function SizeOf(Var): Integer;	Fournit le nombre d'octets occupés par la variable Var.
Sptr	function SPtr: Word;	Fournit la valeur du registre SP (Stack Pointer).
Sqr	function Sqr(X): (Same type as parameter);	Fournit le carré de X.
Sqrt	function Sqrt(X: Real): Real;	Fournit la racine carrée de X.
Sseg	function SSeg: Word;	Fournit la valeur du registre SS (Stack Segment).
Str	procedure Str(X [: Width [: Decimals]]; var S:string);	Convertit le nombre X en la chaîne S avec le format Width:Decimals.
Succ	function Succ(X): (même type que X);	Fournit le successeur de X.
Swap	function Swap(X): (même type que X);	Echange les parties de poids fort et faible dans X.
Trunc	function Trunc(X: Real): Longint;	Tronque un nombre de type REAL en un nombre entier en supprimant la partie décimale.
Truncate	procedure Truncate(var F);	Tronque le fichier à la position actuelle du pointeur de fichiers.
UpCase	function UpCase(Ch: Char): Char;	Convertit un caractère en majuscule.
Val	procedure Val(S; var V; var Code: Integer);	Convertit une chaîne de caractère S en sa valeur numérique V. Code contient le code de l'erreur qui serait apparue lors de la conversion.
Write (text files)	procedure Write([var F: Text;] P1 [,P2,...,Pn]);	Affiche les variables V1, V2, ... sur l'écran.
Write (typed files)	procedure Write(F, V1 [, V2,...,Vn]);	Ecrit les variables V1, V2, ... sur le fichier F.
Writeln	procedure Writeln([var F: Text;] P1 [, P2, ...,Pn]);	Exécute l'écriture suivi d'un marqueur de fin de ligne.

2. L'unité « Crt »

AssignCrt	procedure AssignCrt(var f: Text);	Associe une variable f de type fichier Text à l'écran.
ClrEol	procedure ClrEol;	Efface tous les caractères, à partir de la position actuelle du curseur, et ce jusqu'à la fin de la ligne. La position du curseur n'est pas modifiée.
ClrScr	procedure ClrScr;	Efface la fenêtre active et place le curseur dans le coin supérieur gauche de l'écran.
Delay	procedure Delay(X: Word);	Réalise une attente de X millisecondes.
DelLine	procedure DelLine;	Efface la ligne où le curseur se trouve.
GotoXY	procedure GotoXY(X, Y: Byte);	Place le curseur en position X (colonne), Y (ligne) sur un écran virtuel défini par Window.
HighVideo	procedure HighVideo;	Sélectionne le mode haute intensité des caractères (RED --> LIGHTRED).
InsLine	procedure InsLine;	Insère une ligne vide à la position actuelle du curseur.
KeyPressed	function KeyPressed: Boolean;	Détermine si une touche du clavier a été pressée.
LowVideo	procedure LowVideo;	Sélectionne l'attribut intensité faible pour l'affichage des caractères.
NormVideo	procedure NormVideo;	Sélectionne l'attribut du texte qui était en dessous du curseur au lancement du programme.
NoSound	procedure NoSound;	Coupe le haut-parleur interne de l'ordinateur.
ReadKey	function ReadKey: Char;	Lit un caractère au clavier, sans affichage à l'écran.
Sound	procedure Sound(Hz: Word);	Emet de manière continue un son à la fréquence Hz par le haut-parleur interne.
TextBackground	procedure TextBackground(Color: Byte);	Sélectionne la couleur du fond pour l'affichage en mode texte.
TextColor	procedure TextColor(Color: Byte);	Sélectionne la couleur pour l'affichage du texte.
TextMode	procedure TextMode(Mode: Integer);	Sélectionne un mode texte spécifique.
WhereX	function WhereX: Byte;	Fournit le numéro de la ligne où se trouve le curseur.
WhereY	function WhereY: Byte;	Fournit le numéro de la colonne où se trouve le curseur.
Window	procedure Window(X1, Y1, X2, Y2: Byte);	Définit une fenêtre sur l'écran, en mode texte.

3. L'unité « DOS »

DiskFree	function DiskFree(Drive: Byte): Longint;	Fournit le nombre d'octets libres sur le disque Drive.
DiskSize	function DiskSize(Drive: Byte): Longint;	Fournit le nombre total d'octets du disque Drive.
DosExitCode	function DosExitCode: Word;	Fournit le code de sortie d'un sous processus.
DosVersion	function DosVersion: Word;	Fournit le numéro de version du DOS.
Envcount	function EnvCount: Integer;	Fournit le nombre de chaînes contenues dans l'environnement DOS.
EnvStr	function EnvStr(Index: Integer): string;	Fournit la chaîne d'environnement spécifiée par Index.
Exec	procedure Exec(Path, CmdLine: string);	Exécute le programme spécifié par Path avec les paramètres CmdLine. Les erreurs rencontrées sont renvoyées dans DosError.
Fexpand	function FExpand (Path: PathStr): PathStr;	Etend le nom de fichier Path en son nom avec chemin.
FindFirst	procedure FindFirst(Path: PChar; Attr: Word; var F: TSearchRec);	Cherche, dans le répertoire Path, le premier fichier correspondant à Attr.
FindNext	procedure FindNext(var F: TSearchRec);	Cherche le fichier suivant dont les nom et attributs correspondent à ce qui a été spécifié par le précédent FindFirst.
Fsearch	function FSearch(Path: PathStr; DirList: string): PathStr;	Cherche dans Path le fichier DirListSearches for a file.
Fsplit	procedure FSplit(Path: PathStr; var Dir: DirStr; var Name: NameStr;var exit : ExitStr);	Sépare un nom de fichier en ses 3 composants.
GetCBreak	procedure GetCBreak(var Break: Boolean);	Fournit l'état du Ctrl-Break comme indiqué dans le Config.Sys.
GetDate	procedure GetDate(var Year, Month, Day, DayOfWeek: Word);	Fournit la date actuelle telle que fournie par l'OS.
GetEnv	function GetEnv(EnvVar: string): string;	Fournit la valeur de la variable d'environnement EnVar.
GetFAttr	procedure GetFAttr(var F; var Attr: Word);	Fournit les attributs du fichier F.
GetFTime	procedure GetFTime(var F; var Time: Longint);	Fournit les date et heure de dernière écriture du fichier F.
GetIntVec	procedure GetIntVec(IntNo: Byte; var Vector: Pointer);	Fournit l'adresse stockée dans le vecteur d'interruption IntNo.
GetTime	procedure GetTime(var Hour, Minute, Second, Sec100: Word);	Fournit l'heure actuelle telle que fournie par l'OS.
GetVerify	procedure GetVerify(var Verify: Boolean);	Fournit l'état du drapeau verify sous DOS.
Intr	procedure Intr(IntNo: Byte; var Regs: TRegisters);	Exécute l'interruption IntNo avec passage et mise et à jour des registres Regs.
Keep	procedure Keep(ExitCode: Word);	Conserve le programme en mémoire (Terminate Stay Resident=TSR).
MsDos	procedure MsDos(var Regs: TRegisters);	Exécute un appel à une fonction DOS.
PackTime	procedure PackTime(var T: DateTime; var Time: Longint);	Convertit une heure et une date au format DateTime en un entier.
SetCBreak	procedure SetCBreak(Break: Boolean);	Permet de configurer la variable BREAK de MS DOS.
SetFTime	procedure SetFTime(var F; Time: Longint);	Change la date/heure de dernière écriture du fichier F.
SetIntVec	procedure SetIntVec(IntNo: Byte; Vector: Pointer);	Change l'adresse pointe le vecteur de l'interruption IntNo.
SetTime	procedure SetTime(Hour, Minute, Second, Sec100: Word);	Change l'heure dans l'OS.
SetVerify	procedure SetVerify(Verify: Boolean);	Change l'état de la variable Verify flag sous DOS.
SwapVectors	procedure SwapVectors;	Echange les pointeurs SaveIntXX dans l'unit System avec les vecteurs actuels.
UnPackTime	procedure UnpackTime(Time: Longint; var DT: TDateTime);	Convertit le LONGINT fourni par GetFTime, FindFirst ou FindNext en un RECORD au format date/heure.

4. L'unité « Graph »

Arc	`procedure Arc(x, y: INTEGER, angledeb, anglefin, rayon: WORD) ; …`	Dessine un arc de cercle de centre `(x,y)`, de rayon `rayon`, commençant à l'angle `angledeb` et allant jusqu'à l'angle `anglefin`.
Bar	procedure `Bar (x1, y1, x2, y2:INTEGER) ; …`	Dessine un rectangle plein ayant pour coin supérieur gauche `(x1, y1)` et pour coin inférieur droit `(x2,y2)`.
Bar3D	procedure `Bar3D(x1, y1, x2, y2 : INTEGER, prof : WORD; top : BOOLEAN) ; …`	Dessine un rectangle plein en 3D ayant pour coin supérieur gauche `(x1, y1)`, pour coin inférieur droit `(x2,y2)`, pour profondeur `prof`, et dont le dessus est dessiné ou non selon la valeur de `top` (ce qui permet par exemple d'emboîter des rectangles).
Circle	procedure `Circle (x, y : Integer, rayon : Word) ; …`	Dessine un cercle de centre `(x, y)` et de rayon `rayon`.
CloseGraph	procedure CloseGraph ; …	Désactive le mode graphique.
ClearDevice	procedure ClearDevice ; …	Efface le périphérique de sortie en mode `GRAPH` (généralement l'écran).
DetectGraph	procedure `DetectGraph(Var GraphPilote, GraphMode : Integer) ; …`	Vérifie la configuration matérielle et détermine le pilote et le mode graphique courant.
`DrawPoly`	procedure `DrawPoly (nbpoints :Word, Var poly : Array[...] of PointType) ; …`	Dessine un polygone à `nbpoints` points de coordonnées données par le tableau.
Ellipse	procedure `Ellipse (x, y : Integer; a1, a2, r1, r2: Word) ; …`	Dessine un arc d'ellipse de centre `(x, y)`, allant de l'angle `a1` à `a2`, et de rayon horizontal `r1` et vertical `r2`.
FillEllipse	procedure `FillEllipse (x, y : Integer ; r1, r2 : Word) ; …`	Dessine une ellipse pleine de centre `(x, y)`, de rayon horizontal `r1` et vertical `r2`.
FillPoly	procedure `FillPoly (nbpoints : Word, Var poly : Array[...] of PointType)`	Dessine un polygone plein à `nbpoints` points de coordonnées données par le tableau.
FloodFill	procedure FloodFill(x, y : Integer, Bord : Word) ; …	Remplit la surface fermée contenant le point (x, y) par la couleur de sa bordure.
GetBkColor	function GetBkColor() : Integer ; …	Donne la valeur de la couleur de l'arrière-plan.
GetImage	procedure GetImage(x, y, x2, y2 : Integer, Var bm) ; …	Copie la partie d'écran définie par le rectangle (x_1, y_1) et (x_2, y_2) dans une zone mémoire désignée par le pointeur Image.
GetMaxX	function GetMaxX() : Integer ; …	Donne la valeur de l'abscisse maximale de l'écran.
GetMaxY	function GetMaxY() : Integer ; …	Donne la valeur de l'ordonnée maximale de l'écran.
GetPixel	function GetPixel (x, y : Integer) : Word ; …	Donne le numéro de la couleur actuelle du point de coordonnées (x, y).
GetX	function GetX () : Integer ;	Donne la valeur de l'abscisse du curseur.
GetY	function GetY () : Integer ;	Donne la valeur de l'ordonnée du curseur.
ImageSize	function ImageSize (x1, y1, x2, y2 : Integer) : Word ; …	Renvoie la taille en octets nécessaire à la sauvegarde de la partie d'écran limitée par le rectangle défini par (x_1, y_1) et (x_2, y_2).
InitGraph	procedure InitGraph (i, j : integer, Const path : String) ; …	Initialise le mode graphique avec les valeurs i et j pour le pilote et le mode et path pour le chemin du fichier qui gère le mode graphique.
InstallUserFont	function InstallUserFont (name : String) : Integer ; …	Donne la valeur qui correspond à l'index de la police donnée par name.
Line	procedure Line (x1, y1, x2, y2 : Integer) ; …	Dessine une ligne du point de coordonnées (x_1, y_1) au point (x_2, y_2).
LineRel	procedure LineRel (x, y : Integer) ; …	Dessine une ligne du point courant jusqu'au point de

		coordonnées ($x_{courant}$ + x, $y_{courant}$ + y).
LineTo	procedure LineTo (x, y : Integer) ; …	Dessine une ligne du point courant au point de coordonnées (x, y).
MoveTo	procedure MoveTo (x, y : Integer) ; …	Déplace le point courant au point de coordonnées (x, y).
OutText	procedure OutText (Const st : String) ;	Affiche à l'écran, à la position actuelle du point, le texte st.
OutTextXY	procedure OutTextXY (x, y : Integer, Const st : String) ; …	Affiche à l'écran, à la position (x, y), le texte st. La position du point est inchangée.
PieSlice	procedure PieSlice (xC, yC, angleDebut, angleFin, rayon) ; …	Dessine un secteur circulaire de centre C de coordonnées x_c et y_c, l'angle est en degré.
PutImage	procedure putImage(x, y : Integer, Image : Pointer, Oper : Word); …	Affiche à la position (x, y) de l'écran une image préalablement sauvée en mémoire dans la zone indiquée par le pointeur Image. Le paramètre Oper permet de préciser comment l'affichage de l'image tiendra compte du fond.
PutPixel	procedure PutPixel (x, y : Integer, c : Word) ; …	Affiche un point de couleur c aux coordonnées (x,y).
Rectangle	procedure Rectangle (x1, y1, x2, y2 : Integer) ; …	Dessine un rectangle ayant pour coin supérieur gauche (x_1, y_1) et pour coin inférieur droit (x_2, y_2).
RestoreCRTMode	procedure RestoreCRTMode ; …	Ferme la fenêtre graphique pour revenir en fenêtre DOS.
SetBkColor	procedure SetBkColor(y : byte) ; …	Attribue après avoir effacer l'écran, la couleur y à l'arrière-plan.
SetColor	procedure SetColor (y : Byte) ; …	Attribue après avoir effacer l'écran, la couleur y à l'avant-plan (ligne et texte).
SetFillStyle	procedure SetFillStyle (Pattern : Word, Color : Word) ; …	Attribue aux contenus des objets pleins le style et la couleur donnés par Pattern et Color.
SetGraphMode	procedure SetGraphMode (i : Integer) ; …	Attribue le mode graphique i à la fenêtre active.
SetLineStyle	procedure SetLineStyle (LineStyle, Pattern, Thickness : Word) ; …	Attribue aux lignes le style, la forme et l'épaisseur donnés par LineStyle, Pattern et Thickness.
SetTextJustify	procedure SetTextJustify (h, v : Word) ; …	Définit la justification horizontale et verticale au moyen des variables h et v.
SetTextStyle	procedure SetTextStyle (Font, Direction : Integer ; CharSize : Integer) ; …	Définit la police par la variable Font, la direction (gauche à droite ou haut en bas) par la variable Direction et la taille de celle-ci par la variable CharSize.
SetWriteMode	procedure SetWriteMode (WriteMode : Integer) ; …	Choisit le mode de tracés.
TextHeight	function TextHeight (Const s : String) : Integer ; …	Donne la hauteur du texte s.
TextWidth	function TextWidth (Const s : String) : Integer ; …	Donne la longueur du texte s.

5. L'unité « Strings »

StrCat	function StrCat(Dest, Source: PChar): PChar;	Concatène la copie de Source à la fin de la chaîne Dest.
StrCopy	function StrCopy(Dest, Source: PChar): PChar;	Concatène les chaînes Dest et Source.
StrDispose	function StrDispose(Str: PChar);	Récupère la place occupée sur le tas (HEAP).
StrECopy	function StrECopy(Dest, Source: Pchar): PChar;	Copie une chaîne dans une autre et fournit un pointeur vers la fin de la chaîne résultante.
StrEnd	function StrEnd(Str: PChar): Pchar;	Fournit un pointeur vers la fin de la chaîne.
StrLCat	function StrLCat(Dest, Source: PChar; MaxLen: Word): PChar;	Ajoute les caractères d'une chaîne à la fin d'une autre.
StrlComp	function StrLComp(Str1, Str2: PChar; MaxLen: Word): Integer;	Compare deux chaînes jusqu'à la position MaxLen.
StrLCopy	function StrLCopy(Dest, Source: PChar; MaxLen: Word): PChar;	Copie les caractères d'une chaîne dans une autre.
StrLen	function StrLen(Str: PChar): Word;	Fournit le nombre de caractères dans Str.
StrLower	function StrLower(Str: PChar): PChar;	Convertit une chaîne en minuscules.
StrMove	function StrMove(Dest, Source: PChar; Count: Word): PChar;	Copie les caractères d'une chaîne dans une autre.
StrNew	function StrNew(Str: PChar): PChar;	Alloue une chaîne sur le tas (HEAP).
StrPas	function StrPas(Str: PChar): String;	Convertit une chaîne terminée par le caractère #0 en une chaîne du type Pascal.
StrPCopy	function StrPCopy(Dest: PChar; Source: String): PChar;	Copie une chaîne dans une chaîne terminée par le caractère de code 0.
StrPos	function StrPos(Str1, Str2: PChar): PChar;	Fournit un pointeur vers la première occurrence de la chaîne Str1 dans Str2.
StrRScan	function StrRScan(Str: PChar; Chr: Char): PChar;	Fournit un pointeur vers la dernière occurrence du caractère Chr dans Str.
StrScan	function StrScan(Str: PChar; Chr: Char): PChar;	Fournit un pointeur vers la première occurrence du caractère Chr dans Str.
StrUpper	function StrUpper(Str: PChar): PChar;	Convertit une chaîne en majuscules.