Haute École de Bruxelles
École Supérieure d'Informatique

Rue Royale, 67 – 1000 Bruxelles
02/219.15.46 – esi@heb.be

Bachelor en Informatique

Algorithmique I

DEV1 – 2014

Activité d'apprentissage enseignée par :

L. Beeckmans *M. Codutti* *G. Cuvelier* *A. Hallal*
C. Leruste *E. Levy* *N. Pettiaux* *F. Servais*

Ce syllabus a été écrit à l'origine par M. Monbaliu à une époque où le cours s'appelait « Logique et techniques de programmation ». Il a ensuite été adapté par Mme Leruste, M. Beeckmans et M. Codutti. Qu'ils en soient tous remerciés. Nous remercions également tous ceux qui ont contribué à son amélioration grâce à leur lecture attentive et leurs remarques.

Document produit avec LaTeX.
Version du 7 septembre 2014.

Table des matières

Chapitre 1

Qu'est-ce qu'un algorithme ?

« L'algorithmique est le permis de conduire de l'informatique. Sans elle, il n'est pas concevable d'exploiter sans risque un ordinateur. » [1]

Ce chapitre a pour but de vous faire comprendre ce qu'est un **algorithme** et à quel moment de l'**activité de programmation** il intervient. Nous tenterons de préciser la différence entre un algorithme et un **programme**.

Il situe, enfin, le cours d'algorithmique dans l'ensemble des cours donnés dans le baccalauréat et en trace les lignes principales.

1.1 La notion de problème

1.1.1 Préliminaires : utilité de l'ordinateur

L'ordinateur est une machine. Mais une machine intéressante dans la mesure où elle est destinée d'une part, à nous décharger d'une multitude de tâches peu valorisantes, rébarbatives telles que le travail administratif répétitif, mais surtout parce qu'elle est capable de nous aider, voire nous remplacer, dans des tâches plus ardues qu'il nous serait impossible de résoudre sans son existence (conquête spatiale, prévision météorologique, jeux vidéo, ...).

En première approche, nous pourrions dire que l'ordinateur est destiné à nous remplacer, à faire à notre place (plus rapidement et probablement avec moins d'erreurs) un travail nécessaire à la résolution de **problèmes** auxquels nous devons faire face. Attention ! Il s'agit bien de résoudre des *problèmes* et non des mystères (celui de l'existence, par exemple). Il faut que la question à laquelle on souhaite répondre soit **accessible à la raison**.

1.1.2 Poser le problème

Un préalable à l'activité de résolution d'un problème est bien de **définir** d'abord quel est le problème posé, en quoi il consiste exactement ; par exemple, faire un baba au rhum, réussir une année d'études, résoudre une équation mathématique...

1. [CORMEN e.a., Algorithmique, Paris, Edit. Dunod, 2010, (Cours, exercices et problèmes), p. V]

Un problème bien posé doit mentionner l'**objectif à atteindre**, c'est-à-dire la situation d'arrivée, le but escompté, le résultat attendu. Généralement, tout problème se définit d'abord explicitement par ce que l'on souhaite obtenir.

La formulation d'un problème ne serait pas complète sans la connaissance des **données du problème** et **du cadre dans lequel se pose le problème :** de quoi dispose-t-on, quelles sont les hypothèses de base, quelle est la situation de départ ? Faire un baba au rhum est un problème tout à fait différent s'il faut le faire en plein désert ou dans une cuisine super équipée ! D'ailleurs, dans certains cas, la première phase de la résolution d'un problème consiste à mettre à sa disposition les éléments nécessaires à sa résolution : dans notre exemple, ce serait se procurer les ingrédients et les ustensiles de cuisine.

Un problème ne sera véritablement bien spécifié que s'il s'inscrit dans le schéma suivant :

étant donné [les données] **on demande** [l'objectif]

Parfois, la première étape dans la résolution d'un problème est de préciser ce problème à partir d'un énoncé flou : il ne s'agit pas nécessairement d'un travail facile !

Exercice : Un problème flou

Soit le problème suivant : « Calculer la moyenne de nombres entiers. ».
Qu'est-ce qui vous parait flou dans cet énoncé ?

Une fois le problème correctement posé, on passe à la recherche et la description d'une **méthode de résolution**, afin de savoir comment faire pour atteindre l'objectif demandé à partir de ce qui est donné. Le **nom** donné à une méthode de résolution varie en fonction du cadre dans lequel se pose le problème : *façon de procéder, mode d'emploi, marche à suivre, guide, patron, modèle, recette de cuisine, méthode ou plan de travail, algorithme mathématique, programme, directives d'utilisation, ...*

1.2 Procédure de résolution

Une **procédure de résolution** est une description en termes compréhensibles par l'exécutant de la **marche à suivre** pour résoudre un problème donné.

On trouve beaucoup d'exemples dans la vie courante : recette de cuisine, mode d'emploi d'un GSM, description d'un itinéraire, plan de montage d'un jeu de construction, etc. Il est clair qu'il y a une infinité de rédactions possibles de ces différentes marches à suivre. Certaines pourraient être plus précises que d'autres, d'autres par contre pourraient s'avérer exagérément explicatives.

Des différents exemples de procédures de résolution se dégagent les caractéristiques suivantes :

- toutes ont un **nom**
- elles s'expriment dans un **langage** (français, anglais, dessins...)
- l'ensemble de la procédure consiste en une **série chronologique** d'instructions ou de phrases (parfois numérotées)
- une instruction se caractérise par un ordre, une action à accomplir, une **opération** à exécuter sur les **données** du problème
- certaines phrases justifient ou expliquent ce qui se passe : ce sont des **commentaires**.

On pourra donc définir, en première approche, une procédure de résolution comme un texte, écrit dans un certain langage, qui décrit une suite d'actions à exécuter dans un ordre précis, ces actions opérant sur des objets issus des données du problème.

1.2.1 Chronologie des opérations

Pour ce qui concerne l'ordinateur, le travail d'exécution d'une marche à suivre est impérativement **séquentiel**. C'est-à-dire que les instructions d'une procédure de résolution sont exécutées **une et une seule fois** dans l'ordre où elles apparaissent dans le code. Cependant certains artifices d'écriture permettent de **répéter** l'exécution d'opérations ou de la **conditionner** (c'est-à-dire de choisir si l'exécution aura lieu oui ou non en fonction de la réalisation d'une condition).

1.2.2 Les opérations élémentaires

Dans la description d'une marche à suivre, la plupart des opérations sont introduites par un **verbe** (*remplir*, *verser, prendre, peler*, etc.). L'exécutant ne pourra exécuter une action que s'il la comprend : cette action doit, pour lui, être une action élémentaire, une action qu'il peut réaliser sans qu'on ne doive lui donner des explications complémentaires. Ce genre d'opération élémentaire est appelée **primitive**.

Ce concept est évidement relatif à ce qu'un exécutant est capable de réaliser. Cette capacité, il la possède d'abord parce qu'il est **construit** d'une certaine façon (capacité innée). Ensuite parce que, par construction aussi, il est doté d'une faculté d'**apprentissage** lui permettant d'assimiler, petit à petit, des procédures non élémentaires qu'il exécute souvent. Une opération non élémentaire pourra devenir une primitive un peu plus tard.

1.2.3 Les opérations bien définies

Il arrive de trouver dans certaines marches à suivre des opérations qui peuvent dépendre d'une certaine manière de l'appréciation de l'exécutant. Par exemple, dans une recette de cuisine on pourrait lire : *ajouter **un peu** de vinaigre, saler et poivrer **à volonté**, laisser cuire une **bonne** heure dans un four **bien** chaud, etc.*

Des instructions floues de ce genre sont dangereuses à faire figurer dans une bonne marche à suivre car elles font appel à une appréciation arbitraire de l'exécutant. Le résultat obtenu risque d'être imprévisible d'une exécution à l'autre. De plus, les termes du type *environ, beaucoup, pas trop* et *à peu près* sont intraduisibles et proscrites au niveau d'un langage informatique ![2]

Une **opération bien définie** est donc une opération débarrassée de tout vocabulaire flou et dont le résultat est **entièrement prévisible**. Des versions « bien définies » des exemples ci-dessus pourraient être : *ajouter 2 cl de vinaigre, ajouter 5 g de sel et 1 g de poivre, laisser cuire 65 minutes dans un four chauffé à 220 ˚, etc.*

Afin de mettre en évidence la difficulté d'écrire une marche à suivre claire et non ambigüe, on vous propose l'expérience suivant.

2. Le lecteur intéressé découvrira dans la littérature spécialisée que même les procédures de génération de nombres aléatoires sont elles aussi issues d'algorithmes mathématiques tout à fait déterminés.

Expérience : Le dessin

Cette expérience s'effectue en groupe. Le but est de faire un dessin et de permettre à une autre personne, qui ne l'a pas vu, de le reproduire fidèlement, au travers d'une « marche à suivre ».

1. Chaque personne prend une feuille de papier et y dessine quelque chose en quelques traits précis. Le dessin ne doit pas être trop compliqué ; on ne teste pas ici vos talents de dessinateur ! (ça peut-être une maison, une voiture, ...)
2. Sur une **autre** feuille de papier, chacun rédige des instructions permettant de reproduire fidèlement son propre dessin. Attention ! Il est important de ne **jamais faire référence à la signification du dessin**. Ainsi, on peut écrire : « dessine un rond » mais certainement pas : « dessine une roue ».
3. Chacun cache à présent son propre dessin et échange sa feuille d'instructions avec celle de quelqu'un d'autre.
4. Chacun s'efforce ensuite de reproduire le dessin d'un autre en suivant **scrupuleusement** les instructions indiquées sur la feuille reçue en échange, **sans tenter d'initiative** (par exemple en croyant avoir compris ce qu'il faut dessiner).
5. Nous examinerons enfin les différences entre l'original et la reproduction et nous tenterons de comprendre pourquoi elles se sont produites (par imprécision des instructions ou par mauvaise interprétation de celles-ci par le dessinateur...)

Quelles réflexions cette expérience vous inspire-t-elle ? Quelle analogie voyez-vous avec une marche à suivre donnée à un ordinateur ?

Dans cette expérience, nous imposons que la « marche à suivre » ne mentionne aucun mot expliquant le sens du dessin (mettre « rond » et pas « roue » par exemple). Pourquoi, à votre avis, avons-nous imposé cette contrainte ?

1.2.4 Opérations soumises à une condition

En français, l'utilisation de conjonctions ou locutions conjonctives du type *si*, *selon que*, *au cas où*, ... présuppose la possibilité de ne pas exécuter certaines opérations en fonction de certains événements. D'une fois à l'autre, certaines de ses parties seront ou non exécutées.

Exemple : Si la viande est surgelée, la décongeler à l'aide du four à micro-ondes.

1.2.5 Opérations à répéter

De la même manière, il est possible d'exprimer en français une exécution répétitive d'opérations en utilisant les mots *tous*, *chaque*, *tant que*, *jusqu'à ce que*, *chaque fois que*, *aussi longtemps que*, *faire x fois*, ...

Dans certains cas, le nombre de répétitions est connu à l'avance (*répéter 10 fois*) ou déterminé par une durée (*faire cuire pendant 30 minutes*) et dans d'autres cas il est inconnu. Dans ce cas, la fin de la période de répétition d'un bloc d'opérations dépend alors de la réalisation d'une condition ((*lancer le dé jusqu'à ce qu'il tombe sur 6*), ..., *faire cuire jusqu'à évaporation complète*...). C'est ici que réside le danger de boucle infinie, due à une mauvaise formulation de la condition d'arrêt. Par exemple : *lancer le dé jusqu'à ce que le point obtenu soit 7*...Bien sûr, un humain doté d'intelligence comprend que la condition est impossible à réaliser, mais un robot appliquant cette directive à la lettre lancera le dé perpétuellement...

1.2.6 À propos des données

Les types d'objets figurant dans les diverses procédures de résolution sont fonction du cadre dans lequel s'inscrivent ces procédures, du domaine d'application de ces marches à suivre. Par exemple, pour une recette de cuisine, ce sont les ingrédients. Pour un jeu de construction ce sont les briques.

L'ordinateur, quant à lui, manipule principalement des données numériques et textuelles. Nous verrons plus tard comment on peut combiner ces données élémentaires pour obtenir des données plus complexes.

1.3 Algorithmes informatiques

Notre but étant de faire de l'informatique, il convient de restreindre notre étude à des notions plus précises, plus spécialisées, gravitant autour de la notion de *traitement automatique de l'information*.

1.3.1 Algorithme

Un algorithme appartient au vaste ensemble des *marches à suivre*.

Algorithme : Procédure de résolution d'un problème ou d'un ensemble de problèmes de même type contenant des opérations bien définies portant sur des informations, s'exprimant dans une séquence définie sans ambigüité, destinée à être traduite dans un langage de programmation.

Comme toute marche à suivre, un algorithme doit s'exprimer dans un certain langage : à priori le langage naturel, mais il y a d'autres possibilités : ordinogramme, arbre programmatique, pseudo-code ou LDA (langage de description d'algorithmes) que nous allons utiliser dans le cadre de ce cours.

1.3.2 Programme

Un **programme** n'est rien d'autre que la représentation d'un algorithme dans un langage plus technique compris par un ordinateur (par exemple : Assembleur, Cobol, Java, C++, ...). Ce type de langage est appelé **langage de programmation**.

Écrire un programme correct suppose donc la parfaite connaissance du langage de programmation et de sa **syntaxe**, qui est en quelque sorte la grammaire du langage. Mais ce n'est pas suffisant ! Puisque le programme est la représentation d'un algorithme, il faut que celui-ci soit correct pour que le programme le soit. Un programme correct résulte donc d'une démarche logique correcte (algorithme correct) et de la connaissance de la syntaxe d'un langage de programmation.

Il est donc indispensable d'élaborer des algorithmes corrects avant d'espérer concevoir des programmes corrects.

1.3.3 Les constituants principaux de l'ordinateur

Les constituants d'un ordinateur se divisent en **hardware** (matériel) et **software d'exploitation** (logiciel).

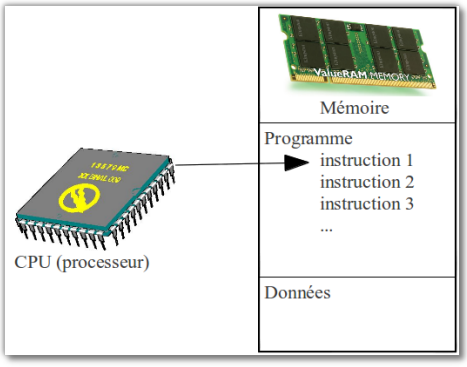
Le **hardware** est constitué de l'ordinateur proprement dit et regroupe les entités suivantes :

- ▷ **l'organe de contrôle :** c'est le cerveau de l'ordinateur. Il est l'organisateur, le contrôleur suprême de l'ensemble. Il assume l'enchainement des opérations élémentaires. Il s'occupe également d'organiser l'exécution effective de ces opérations élémentaires reprises dans les programmes.
- ▷ **l'organe de calcul :** c'est le calculateur où ont lieu les opérations arithmétiques ou logiques. Avec l'organe de contrôle, il constitue le **processeur** ou **unité centrale**.
- ▷ **la mémoire centrale :** dispositif permettant de mémoriser, pendant le temps nécessaire à l'exécution, les programmes et certaines données pour ces programmes.
- ▷ **les unités d'échange avec l'extérieur :** dispositifs permettant à l'ordinateur de recevoir des informations de l'extérieur (unités de lecture telles que clavier, souris, écran tactile, ...) ou de communiquer des informations vers l'extérieur (unités d'écriture telles que écran, imprimantes, signaux sonores, ...).
- ▷ **les unités de conservation à long terme :** ce sont les mémoires auxiliaires (disques durs, CD ou DVD de données, clés USB, ...) sur lesquelles sont conservées les procédures (programmes) ou les informations résidentes dont le volume ou la fréquence d'utilisation ne justifient pas la conservation permanente en mémoire centrale.

Le **software d'exploitation** est l'ensemble des procédures (programmes) s'occupant de la gestion du fonctionnement d'un système informatique et de la gestion de l'ensemble des ressources de ce système (le matériel – les programmes – les données). Il contient notamment des logiciels de traduction permettant d'obtenir un programme écrit en langage machine (langage technique qui est le seul que l'ordinateur peut comprendre directement, c'est-à-dire exécuter) à partir d'un programme écrit en langage de programmation plus ou moins « évolué » (c'est-à-dire plus ou moins proche du langage naturel).

1.3.4 Exécution d'un programme

Isolons (en les simplifiant) deux constituants essentiels de l'ordinateur afin de comprendre ce qui se passe quand un ordinateur exécute un programme. D'une part, la mémoire contient le programme et les données manipulées par ce programme. D'autre part, le processeur va « exécuter » ce programme.



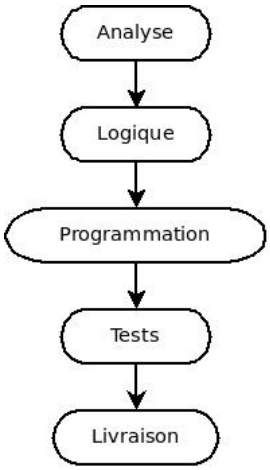
Comment fonctionne le processeur ?
De façon très simplifiée, on passe par les étapes suivantes :

1. Le processeur lit l'instruction courante.
2. Il exécute cette instruction. Cela peut amener à manipuler les données.
3. L'instruction suivante devient l'instruction courante.
4. On revient au point 1.

On voit qu'il s'agit d'un travail automatique ne laissant aucune place à l'initiative !

1.4 Les phases d'élaboration d'un programme

Voyons pour résumer un schéma **simplifié** des phases par lesquelles il faut passer quand on développe un programme.



- Lors de **l'analyse**, le problème doit être compris et clairement précisé. Vous aborderez cette phase dans le cours d'analyse.
- Une fois le problème analysé, et avant de passer à la phase de programmation, il faut réfléchir à l'**algorithme** qui va permettre de résoudre le problème. C'est à cette phase précise que s'attache ce cours.
- On peut alors **programmer** cet algorithme dans le langage de programmation choisi. Vos cours de langage (Java, Cobol, Assembleur, ...) sont dédiés à cette phase.
- Vient ensuite la phase de **tests** qui ne manquera pas de montrer qu'il subsiste des problèmes qu'il faut encore corriger. (Vous aurez maintes fois l'occasion de vous en rendre compte lors des séances de laboratoire)
- Le produit sans bug (connu) peut être **mis en application** ou **livré** à la personne qui vous en a passé la commande.

Notons que ce processus n'est pas linéaire. À chaque phase, on pourra détecter des erreurs, imprécisions ou oublis des phases précédentes et revenir en arrière.

Pourquoi passer par la phase « algorithmique » et ne pas directement passer à la programmation ?

Voilà une question que vous ne manquerez pas de vous poser pendant votre apprentissage cette année. Apportons quelques éléments de réflexion.

- Passer par une phase « algorithmique » permet de séparer deux difficultés : quelle est la marche à suivre ? Et comment l'exprimer dans le langage de programmation choisi ? Le langage que nous allons utiliser en algorithmique est plus souple et plus général que le langage Java par exemple (où il faut être précis au « ; » près).
- De plus, un algorithme écrit facilite le dialogue dans une équipe de développement. « J'ai écrit un algorithme pour résoudre le problème qui nous occupe. Qu'en pensez-vous ? Pensez-vous qu'il est correct ? Avez-vous une meilleure idée ? ». L'algorithme est plus adapté à la communication car plus lisible.
- Enfin, si l'algorithme est écrite, elle pourra facilement être traduite dans n'importe quel langage de programmation. La traduction d'un langage de programmation à un autre est un peu moins facile à cause des particularités propres à chaque langage.

Bien sûr, cela n'a de sens que si le problème présente une réelle difficulté algorithmique. Certains problèmes (en pratique, certaines parties de problèmes) sont suffisamment simples que pour être directement programmés. Mais qu'est-ce qu'un problème simple ? Cela va évidemment changer tout au long de votre apprentissage. Un problème qui vous paraitra difficile en début d'année vous paraitra (enfin, il faut l'espérer !) une évidence en fin d'année.

1.5 Une approche ludique de l'algorithmique

Il existe de nombreux programmes qui permettent de s'initier à la création d'algorithmes. Nous voudrions mettre en avant le projet `learn.code.org`. Soutenu par des grands noms de l'informatique comme Google, Microsoft, Facebook et Twitter il permet de s'initier aux concepts de base (les boucles, les alternatives et la découpe en modules) au travers d'exercices ludiques faisant intervenir des personnages issus de jeux que les jeunes connaissent bien (Angry birds, Plantes et zombies).

Sur le site learn.code.org vous trouverez :

- « The Hour of Code » : un survol des notions fondamentales en une heure au travers de vidéos explicatives et d'exercices interactifs.
- « Intro aux cours d'informatique pour ado » : un cours de 20 heures, reprenant et approfondissant les éléments de « The Hour of Code ».

1.6 Conclusion

L'informatisation de problèmes est un processus essentiellement dynamique, contenant des allées-venues constantes entre les différentes étapes. Codifier un algorithme dans un langage de programmation quelconque n'est certainement pas la phase la plus difficile de ce processus. Par contre, élaborer une démarche logique de résolution d'un problème est probablement plus complexe.

Le but du cours d'**algorithmique** est double :

- essayer de définir une bonne démarche d'élaboration d'algorithmes (apprentissage de la **logique** de programmation) ;
- faire comprendre l'intérêt d'utiliser certaines méthodes ou **techniques** classiques qui ont fait leurs preuves.

Le tout devrait avoir pour résultat l'élaboration de *bons programmes*, c'est-à-dire *des programmes dont il est facile de se persuader qu'ils sont corrects* et des programmes dont la maintenance est la plus aisée possible. Dans ce sens, ce cours se situe idéalement en aval d'un cours d'**analyse**, et en amont des cours de **langage de programmation**. Ceux-ci sont idéalement complétés par les notions de **système d'exploitation** et de **persistance des données**.

Afin d'envisager la résolution d'une multiplicité de problèmes prenant leur source dans des domaines différents, le contenu minimum de ce cours envisage l'étude des points suivants (dans le désordre) :

- la représentation des algorithmes
- la programmation structurée
- la programmation procédurale : les modules et le passage de paramètres
- les bases de la programmation orientée objet
- les algorithmes de traitement des tableaux
- les algorithmes de traitement des fichiers
- la résolution de problèmes récursifs
- les algorithmes liés au traitement des structures de données particulières telles que listes, files d'attente, piles, arbres, graphes, tables de hachage, etc.

Voilà bien un programme trop vaste pour un premier cours, Un choix devra donc être fait et ce, en fonction de critères tels que la rapidité d'assimilation, l'intérêt des étudiants et les besoins exprimés pour des cours annexes. Les matières non traitées ici, le seront dans les cours d'Algorithmique II et III.

Chapitre 2

Algorithmes séquentiels

Dans ce chapitre, vous apprendrez à écrire un algorithme séquentiel pour résoudre un problème informatique. C'est aussi l'occasion de fixer la syntaxe du pseudo-code que nous allons utiliser. Nous commençons avec les algorithmes séquentiels qui sont de simples séquences de suites d'instructions. Les structures alternatives et répétitives seront vues dans les chapitres suivants.

2.1 Introduction

Supposons que nous voulions rédiger une marche à suivre détaillée qui explique comment additionner deux fractions. Une possibilité de l'écrire en langage « naturel » serait la suivante :

- Rechercher le dénominateur commun des deux fractions
- Mettre chaque fraction au même dénominateur
- Additionner les numérateurs des deux fractions, ce qui donne le numérateur de la somme
- Simplifier la fraction obtenue

Cet algorithme, bien que résultant d'un effort d'explicitation, est encore très imprécis et risque fort de ne pas faciliter l'effort de programmation qui en suivrait. N'oublions pas qu'un ordinateur n'est qu'une machine dépourvue de toute intelligence et incapable de comprendre les sous-entendus qu'un être humain pourrait comprendre !

De plus, les notions de dénominateur commun et de simplification de fraction, qui vous sont familières depuis l'école primaire, ne sont pas immédiates d'un point de vue algorithmique. Un algorithme proche d'un langage de programmation ne devrait mentionner que les opérations élémentaires de calcul telles que $+$, -, $*$, /.

Ceci dit, afin d'être plus proche d'un programme écrit dans un langage compréhensible par l'ordinateur, l'algorithme ci-dessus pourrait s'écrire :

- Prendre connaissance du premier numérateur et du premier dénominateur ;
- Prendre connaissance du second numérateur et du second dénominateur ;
- Multiplier les deux dénominateurs pour obtenir le dénominateur commun ;
- Multiplier le premier numérateur par le second dénominateur
  et le second numérateur par le premier dénominateur ;
- Additionner ces deux produits pour obtenir le numérateur du résultat ;
- Communiquer ce résultat ainsi que le dénominateur commun.

Notons encore que cet algorithme n'est pas le plus efficace pour ce type de problème. Il vaudrait mieux utiliser le PPCM (*Plus Petit Commun Multiple*) pour le calcul du déno-

minateur commun, mais la façon de calculer ce dernier serait très fastidieuse à rédiger en langage « naturel ».

On comprend bien, au travers de cet exemple, que le français n'est pas adapté à la description de problèmes au contenu mathématique ou scientifique. Le langage naturel est, en effet, un mode de représentation trop riche des algorithmes. Il faut comprendre par « riche », la présence de synonymes, de nombreux mots aux significations proches et voisines. L'utiliser amènerait probablement à une perte de temps (donc d'argent), et à des risques d'erreurs dans la conception des programmes. Il convient dès lors d'élaborer des méthodes de représentation plus rigoureuses (donc moins riches) mais nécessitant un effort moindre de programmation.

Ceci dit, le français peut toujours être utilisé dans la formalisation d'une première approche de la résolution d'un problème devant être traduit ensuite en algorithme puis en programme.

2.2 Le pseudo-code

Les inconvénients de l'utilisation du français dans la représentation des algorithmes sont dus à la trop grande richesse de ce langage en comparaison à la pauvreté (mais aussi la rigueur) du langage compris par la machine. De plus, la lourdeur et la longueur des phrases risquent de rendre pénible la relecture et le travail d'élaboration des algorithmes. La solution consiste à édulcorer le langage naturel, d'une part en remplaçant les noms parfois fastidieux des objets sur lesquels portent des opérations par des mots (noms de variables) plus courts, d'autre part, en remplaçant les opérations elles-mêmes par des opérateurs symboliques suffisamment explicites. De plus, des structures types remplaceront les multiples possibilités linguistiques signifiant la même chose.

Le **pseudo-code** ou *Langage de Description des Algorithmes* (LDA en abrégé) est un langage formel et symbolique utilisant :

- ▷ des **noms symboliques** destinés à représenter les objets sur lesquels s'effectuent des actions ;
- ▷ des **opérateurs symboliques** ou des mots-clés traduisant les opérations primitives exécutables par un exécutant donné ;
- ▷ des **structures de contrôle** types.

Ce langage repose sur des conventions d'écriture. Il est destiné à servir de vecteur de la compréhension permettant par exemple une relecture faite par l'élaborateur de l'algorithme lui-même ou faite par autrui et facilitant le travail de programmation.

C'est de cette dernière exigence que nait la polémique consistant à savoir jusqu'à quel niveau de détail il convient d'aller dans la représentation d'un algorithme ! Nous dirons, pour notre part, qu'il ne doit jamais être aussi détaillé que le programme lui-même, celui-ci étant, par essence, la description ultime. Point n'est besoin de donner deux descriptions quasi identiques d'un algorithme, l'une dans un langage *pseudo-codé*, l'autre exprimée dans un langage de programmation.

Nous dirons qu'un algorithme *idéal*, appelé **algorithme général**, exprimé en pseudo-code, devrait se situer à mi-chemin entre la démarche globale exprimée dans un langage naturel (langue française) ou structuré (ordinogramme) et l'algorithme ultime, c'est-à-dire le **programme**, exprimé en langage de programmation.

On pourrait d'ailleurs concevoir la possibilité d'établir d'autres **algorithmes plus détaillés** se situant entre l'algorithme général et le programme et qui tiendraient compte de certaines particularités du langage de programmation dans lequel ils sont destinés à être traduits.

Enfin, et ce n'est pas la chose la moins importante, la définition d'un pseudo-code doit être telle qu'il puisse **faciliter l'apprentissage de la logique de programmation** par des

personnes désireuses de faire de l'informatique un métier. Dans cette optique, il est inutile de les embarrasser de contraintes syntaxiques ou autres qui risquent de les éloigner du but poursuivi.

En conclusion, à partir du moment où des conventions sont prises dans un contexte bien déterminé (un service informatique d'une entreprise, un groupe scolaire...), il convient que **tous** respectent ces conventions. Ce sera à chacun de juger jusqu'où il ne faut pas aller trop loin dans la liberté d'écriture.

2.3 Variables et types

Nous savons que les opérations que l'ordinateur devra exécuter portent sur des éléments qui sont les **données** du problème. Lorsqu'on attribue un **nom** et un **type** à ces données, on parle alors de **variables**. Dans un algorithme, une variable conserve toujours son nom et son type, mais peut changer de **valeur**.

Le **nom** d'une variable permet de la caractériser et de la reconnaitre. Ainsi, dans l'exemple donné ci-dessus, nous pourrions donner le nom *num1* au *premier numérateur* et *num2* au *second numérateur*, ce qui permet déjà de nommer les objets de l'algorithme de façon tout aussi reconnaissable mais plus courte et donc plus commode. Quant au **type** d'une variable, il décrit la nature de son contenu.

2.3.1 Les types autorisés

Dans un premier temps, les seuls **types** utilisés sont les suivants :

entier	pour les nombres entiers
réel	pour les nombres réels
caractère	pour les différentes lettres et caractères (par exemple ceux qui apparaissent sur un clavier : 'a', '1', '#', etc.)
chaine	pour les variables contenant plusieurs caractères ou aucun (la chaine vide) (par exemple : "Bonjour", "Bonjour le monde", "a", "", etc.)
booléen	les variables de ce type ne peuvent valoir que vrai ou faux

On veillera au cours de logique à ne pas utiliser les valeurs 0 et 1 pour les variables booléennes, même si leur emploi est correct dans beaucoup de langages de programmation.

Exercices

Quel(s) type(s) de données utiliseriez-vous pour représenter

- une date du calendrier ?
- un moment dans la journée ?
- le prix d'un produit en grande surface ?
- votre nom ?
- vos initiales ?
- votre adresse ?

2.3.2 Déclaration de variables

La déclaration d'une variable est l'instruction qui définit son nom et son type. On pourrait écrire :

```
num1 et num2 seront les noms de deux objets destinés à recevoir
les numérateurs des fractions, c'est-à-dire des nombres à valeurs entières.
```

Mais, bien entendu, cette formulation, trop proche du langage parlé, serait trop floue et trop longue. Dès lors, nous abrégerons par :

```
num1, num2 : entiers
```

L'ensemble des instructions de la forme

```
variable1, variable2,... : type
```

forme la partie d'un algorithme nommée **déclaration des variables**. La déclaration des informations apparaitra toujours en début d'algorithme, ou dans un bloc annexe appelé **dictionnaire des variables** ou encore **dictionnaire des données**.

Par exemple, pour l'algorithme des fractions, la déclaration des informations pourrait être la suivante :

```
num1, den1, num2, den2, numRes, denRes : entiers
```

avec la signification suivante :

- ▷ num1 (num2) : le numérateur de la première (seconde) fraction ;
- ▷ den1 (den2) : le dénominateur de la première (seconde) fraction ;
- ▷ numRes (denRes) : le numérateur (dénominateur) du résultat.

Attention, lors de la déclaration d'une variable, celle-ci n'a pas de valeur ! Nous verrons plus loin que c'est l'instruction d'**affectation** qui va servir à donner un contenu aux variables déclarées. En logique, nous n'envisageons pas d'*affectation par défaut* consistant à donner une valeur initiale de façon automatique aux variables déclarées (par exemple 0 pour les variables numériques, comme c'est le cas dans certains langages informatiques).

2.3.3 Comment nommer correctement une variable ?

Le but est de trouver un nom qui soit suffisamment court, tout en restant explicite et ne prêtant pas à confusion.

Ainsi num1 est plus approprié pour désigner le *premier numérateur* que zozo1, tintin, bidule ou premierNumérateur. De même, ne pas appeler appeler den la variable représentant le numérateur.

Il faut aussi tenir compte que les langages de programmation imposent certaines limitations (parfois différentes d'un langage à l'autre) ce qui peut nécessiter une modification du nom lors de la traduction.

Voici quelques règles et limitations traditionnelles dans les langages de programmation :

- ▷ Un nom de variable est généralement une suite de caractères alphanumériques d'un seul tenant (pas de caractères blancs) et ne commençant jamais par un chiffre. Ainsi x1 est correct mais pas 1x.
- ▷ Pour donner un nom composé à une variable, on peut utiliser le « tiret bas » ou *underscore* (par ex. premier_numérateur) mais on déconseille d'utiliser le signe « – » qui est

plutôt réservé à la soustraction. Ainsi, dans la plupart des langages, premier-numérateur serait interprété comme la soustraction des variables premier et numérateur [1].

- ▷ Une alternative à l'utilisation du tiret bas pour l'écriture de noms de variables composés est la notation « chameau » (*camelCase* en anglais), qui consiste à mettre une majuscule au début des mots (généralement à partir du deuxième), par exemple premierNombre ou dateNaissance.
- ▷ Les indices et exposants sont proscrits (par ex. x_1, z_6 ou m^2)
- ▷ Les mots-clés du langage sont interdits (par exemple **for**, **if**, **while** pour Java et Cobol) et on déconseille d'utiliser les mots-clés du pseudo-code (tels que **Lire**, **Afficher**, **pour**...)
- ▷ Certains langages n'autorisent pas les caractères accentués (tels que *à, ç, ê, ø,* etc.) ou les lettres des alphabets non latins (tel que Δ) mais d'autres oui ; certains font la distinction entre les minuscules et majuscules, d'autres non. En logique, nous admettons dans noms de variables les caractères accentués du français, par ex. : durée, intérêts, etc.

Il est impossible de décrire ici toutes les particularités des différents langages, le programmeur se reportera aux règles spécifiques du langage qu'il sera amené à utiliser.

Exercice : Déclarer une date

Déclarer le(s) variable(s) permettant de représenter la date d'anniversaire de quelqu'un.

Exercice : Déclarer un rendez-vous

Déclarer le(s) variable(s) permettant de représenter l'heure de début, l'heure de fin et l'objet d'un rendez-vous.

2.4 Opérateurs et expressions

Les opérateurs agissent sur les variables et les constantes pour former des **expressions**. Une expression est donc une combinaison **cohérente** de variables, de constantes et d'opérateurs, éventuellement accompagnés de parenthèses.

2.4.1 Opérateurs arithmétiques élémentaires

Ce sont les opérateurs binaires bien connus :

+	addition
-	soustraction
*	multiplication
/	division réelle

Ils agissent sur des variables ou expressions à valeurs entières ou réelles. Plusieurs opérateurs peuvent être utilisés pour former des expressions plus ou moins complexes, en tenant compte des règles de calcul habituelles, notamment la priorité de la multiplication et de la division

1. Signalons que le tiret « – » est autorisé en Cobol, où il récupère son rôle arithmétique s'il est précédé et suivi d'au moins un blanc.

sur l'addition et la soustraction. Il est aussi permis d'utiliser des parenthèses, par exemple a − (b + c * d)/x. Tout emploi de la division devra être accompagné d'une réflexion sur la valeur du dénominateur, une division par 0 entrainant toujours l'arrêt d'un algorithme.

2.4.2 DIV et MOD

Ce sont deux opérateurs très importants qui ne peuvent s'utiliser qu'avec des variables entières :

DIV	division entière
MOD	reste de la division entière

Par exemple, 47 DIV 7 donne 6 tandis que 47 MOD 7 donne 5.

2.4.3 Fonctions mathématiques complexes

L'élévation à la puissance sera notée ** ou ^ . Pour la racine carrée d'une variable x nous écrirons $\sqrt{x}$. Attention, pour ce dernier, de veiller à ne l'utiliser qu'avec un radicant positif !

Exemple : $(-b + \sqrt{(b**2 - 4 * a * c)})/(2 * a)$

Mais on peut aussi accepter la notation mathématique usuelle $\frac{-b+\sqrt{b^2-4*a*c}}{2*a}$

Pourquoi ne pas avoir écrit « 4ac » et « 2a » ?

Si nécessaire, on se permettra d'utiliser les autres fonctions mathématiques sous leur forme la plus courante dans la plupart des langages de programmation (exemples : $sin(x)$, $tan(x)$, $log(x)$, $exp(x)\ldots$)

2.4.4 Opérateurs de comparaison

Ces opérateurs agissent généralement sur des variables numériques ou des chaines et donnent un résultat booléen.

=	égal
<> ou ≠	différent de
<	(strictement) plus petit que
>	(strictement) plus grand que
≤	plus petit ou égal
≥	plus grand ou égal

Pour les chaines, c'est l'ordre alphabétique qui détermine le résultat (par exemple "milou" < "tintin" est **vrai** de même que "assembleur" ≤ "java")

2.4.5 Opérateurs logiques

Ils agissent sur des expressions booléennes (variables ou expressions à valeurs booléennes) pour donner un résultat du même type.

NON	négation
ET	conjonction logique
OU	disjonction logique

Pour rappel, cond1 ET cond2 n'est vrai que lorsque les deux conditions sont vraies. cond1 OU cond2 est toujours vrai, sauf quand les deux conditions sont fausses.

Veillez à mettre des parenthèses dans le cas de combinaisons de ET et de OU : (cond1 ET cond2) OU cond3 étant différent de cond1 ET (cond2 OU cond3). En cas d'oubli de parenthèses, il faudra se rappeler que ET est prioritaire sur le OU.

Pour rappel aussi, pour un booléen ok : ok = faux est équivalent à NON ok, ok = vrai est équivalent à ok et NON NON ok est équivalent à ok. Dans les trois cas, nous préconiserons la seconde écriture.

2.4.6 Évaluation complète et court-circuitée

On définit deux modes d'évaluation des opérateurs ET et OU :

l'évaluation *complète*
pour connaitre la valeur de cond1 ET cond2 (respectivement cond1 OU cond2), les deux conditions sont chacune évaluées, après quoi on évalue la valeur de vérité de l'ensemble de l'expression.

l'évaluation *court-circuitée*
dans un premier temps, seule la première condition est testée. Dans le cas du ET, si cond1 s'avère faux, il est inutile d'évaluer cond2 puisque le résultat sera faux de toute façon ; l'évaluation de cond2 et de l'ensemble de la conjonction ne se fera que si cond1 est vrai. De même, dans le cas du OU, si cond1 s'avère vrai, il est inutile d'évaluer cond2 puisque le résultat sera vrai de toute façon ; l'évaluation de cond2 et de l'ensemble de la disjonction ne se fera que si cond1 est faux.

Dans le cadre de ce cours, nous opterons pour la deuxième interprétation. Montrons son avantage sur un exemple. Considérons l'expression $n \neq 0$ ET $m/n > 10$. Si on teste sa valeur de vérité avec une valeur de n non nulle, la première condition est vraie et le résultat de la conjonction dépendra de la valeur de la deuxième condition. Supposons à présent que n soit nul. L'évaluation court-circuitée donne le résultat **faux** immédiatement après test de la première condition sans évaluer la seconde, tandis que l'évaluation complète entrainerait un arrêt de l'algorithme pour cause de division par 0 !

Notez que l'évaluation court-circuitée a pour conséquence la non-commutativité du ET et du OU : cond1 ET cond2 n'est donc pas équivalent à cond2 ET cond1, puisque l'ordre des évaluations des deux conditions entre en jeu. Nous conseillons cependant de limiter les cas d'utilisation de l'évaluation court-circuitée et d'opter pour des expressions dont l'évaluation serait similaire dans les deux cas. La justification d'utiliser l'évaluation court-circuitée apparaitra dans plusieurs exemples tout au long du cours.

2.4.7 Manipuler les chaines

Pour les chaines, nous allons introduire quelques notations [2] qui vont nous permettre de les manipuler plus facilement.

- ▷ long(maChaine) donne la taille (le nombre de caractères) de la chaine maChaine.
- ▷ car(maChaine,pos) donne le caractère en position pos (à partir de 1) dans la chaine maChaine.
- ▷ souschaine(maChaine,pos,long) donne la sous-chaine de maChaine commençant en position pos et de longueur long. Exemple : souschaine("Bonjour",2,3) donne "onj".
- ▷ concat(maChaine1,...,maChaineN) concatène les chaines maChaine1 à maChaineN. Exemple : concat("Bon","j", "our") donne "Bonjour".

2. Le lecteur averti reconnaitra la notation modulaire

2.4.8 Manipuler les caractères

Introduisons également quelques notations pour les caractères.

- ▷ chaine(car) transforme le caractère car en une chaine de taille 1.
- ▷ estLettre(car) est vrai si le caractère car est une lettre (idem pour estChiffre, estMajuscule, estMinuscule).
- ▷ majuscule(car) donne la majuscule de la lettre car (idem pour minuscule).
- ▷ numLettre(car) donne la position de car dans l'alphabet (ex : numLettre('E') donne 5, idem pour numLettre('e')).
- ▷ lettre(num) l'inverse de la précédente (ex : lettre(4) donne le caractère 'D').

2.5 L'affectation d'une valeur à une variable

Cette opération est probablement l'opération la plus importante. En effet, une variable ne prend son sens réel que si elle reçoit à un moment donné une valeur. Il y a deux moyens de donner une valeur à une variable.

2.5.1 Affectation externe

On parle d'**affectation externe** lorsque la valeur à affecter à une variable est donnée par l'utilisateur qui la communique à l'exécutant quand celui-ci le lui demande : cette valeur est donc *externe* à la procédure (l'ordinateur ne peut la deviner lui-même !)

L'affectation externe est donc la primitive qui permet de recevoir de l'utilisateur, au moment où l'algorithme se déroule, une ou plusieurs valeur(s) et de les affecter à des variables en mémoire. Nous noterons :

```
lire liste_de_variables_à_lire
```

Exemples

```
lire nombre1, nombre2
lire num1, den1, num2, den2
```

L'exécution de cette instruction provoque une pause dans le déroulement de l'algorithme ; l'exécutant demande alors à l'utilisateur les valeurs des variables à lire. Ces valeurs viennent donc de l'*extérieur* ; une fois introduites dans le système, elles sont affectées aux variables concernées et l'algorithme peut reprendre son cours. Les possibilités d'introduction de données sont nombreuses : citons par exemple l'encodage de données au clavier, un clic de souris, le toucher d'un écran tactile, des données provenant d'un fichier, etc.

2.5.2 Affectation interne

On parle d'affectation interne lorsque la valeur d'une variable est « calculée » par l'exécutant de l'algorithme lui-même à partir de données qu'il connait déjà :

```
nomVariable ← expression
```

Rappelons qu'une **expression** est une combinaison de variables et d'opérateurs. L'expression a une valeur.

Exemples d'affectation corrects

```
somme ← nombre1 + nombre2
denRes ← den1 * den2
cpt ← cpt + 1
delta ← b**2 – 4*a*c
test ← a < b                                          // pour une variable logique
maChaine ← "Bon"
uneChaine ← concat(maChaine, "jour")
```

Exemples d'affectation incorrects

```
somme + 1 ← 3                                  // somme + 1 n'est pas une variable
somme ← 3n          // 3n n'est ni un nom de variable correct ni une expression correcte
```

Remarques

- ▷ Il est de règle que le résultat de l'expression à droite du signe d'affectation (←) soit de même type que la variable à sa gauche. On tolère certaines exceptions :
  - ▷ varEntière ← varRéelle : dans ce cas le contenu de la variable sera la valeur **tronquée** de l'expression réelle. Par exemple si « nb » est une variable de type entier, son contenu après l'instruction « nb ← 15/4 » sera 3
  - ▷ varRéelle ← varEntière : ici, il n'y a pas de perte de valeur.
  - ▷ varChaine ← varCaractère : équivalent à varChaine ← chaine(varCaractère). Le contraire n'est évidemment pas accepté.
- ▷ Seules les variables déclarées peuvent être affectées, que ce soit par l'affectation externe ou interne !
- ▷ Nous ne mélangerons pas la déclaration d'une variable et son affectation interne dans une même ligne de code, donc pas d'instructions hybrides du genre x ← 2 : entier ou encore x : entier(0).
- ▷ Pour l'affectation interne, toutes les variables apparaissant dans l'expression doivent avoir été affectées préalablement. Le contraire provoquerait un arrêt de l'algorithme.

2.6 Communication des résultats

L'instruction de communication des résultats consiste à donner à l'extérieur (donc à l'utilisateur) la valeur d'un résultat calculé au cours de l'exécution de l'algorithme. Nous écrirons :

afficher *expression* ou *liste de variables séparées par des virgules*

qui signifie que la valeur d'une expression (ou celles des différentes variables mentionnées) sera fournie à l'utilisateur (par exemple par un affichage à l'écran ou par impression sur listing via l'imprimante, etc.).

Remarques

- ▷ Ce ne serait pas une erreur fondamentale de remplacer lire par recevoir ou afficher par écrire. Il n'y a évidemment pas de confusion possible à partir du moment où l'on sait qu'il s'agit de primitives d'échange entre l'extérieur et l'ordinateur exécutant la procédure, mais par principe, il est conseillé d'utiliser une syntaxe commune et limitée à un petit nombre de mots-clés.

▷ Comme pour l'affectation interne, on ne peut afficher que des expressions dont les variables qui la composent ont été affectées préalablement.

2.7 Structure générale d'un algorithme

La traduction d'un algorithme en pseudo-code constituera le contenu d'un **module**. Un module contient donc la solution algorithmique d'un problème donné (ou d'une de ses parties). Sa structure générale sera la suivante :

```
module nomDuModule()
    déclaration des variables et constantes utilisées dans le module
    lecture des données
    instructions utilisant les données lues
    communication des résultats
fin module
```

Remarques

▷ Le code d'un algorithme sera toujours compris entre la première ligne, appelée « entête » qui commence par le mot « module » suivi du nom choisi pour l'algorithme, et la ligne finale « fin module ». Le code compris entre l'entête et la ligne finale sera toujours légèrement décalé vers la droite, c'est un premier exemple d'**indentation** indispensable pour la lisibilité d'un programme, nous y reviendrons lors de l'étude des structures alternatives et répétitives.

▷ Comme pour les variables, le nomDuModule devra être approprié au contenu ! Par exemple, sommerNombres), additionnerFractions() plutôt que goPartez() ! Le rôle des parenthèses qui suivent le nom du module sera expliqué plus tard.

▷ Il va de soi que toutes les parties de cette structure générale ne seront pas toujours nécessaires : certains algorithmes ne nécessiteront pas de lecture de données, d'autres ne devront pas communiquer des résultats...

▷ Pour la lisibilité, on veillera toujours à ce qu'un module tienne sur une vingtaine de lignes (donc, en pratique, sur un écran de 40 x 80 caractères ou une page). Ceci implique que si le module devait être plus long, il faudrait le découper, comme nous le verrons plus loin.

Analysons les algorithmes suivants :

```
module exempleValide()
    a, b, c : entiers
    a ← 12
    b ← 5
    c ← a - b
    a ← a + c
    b ← a
fin module
```

	a	b	c
a, b, c : entiers	indéfini	indéfini	indéfini
a ← 12	12	indéfini	indéfini
b ← 5	12	5	indéfini
c ← a - b	12	5	7
a ← a + c	19	5	7
b ← a	19	19	7

```
module exempleInvalide()
    a, b, c : entiers
    a ← 12
    c ← a - b
    d ← c - 2
fin module
```

	a	b	c
a, b, c : entiers	indéfini	indéfini	indéfini
a ← 12	12	indéfini	indéfini
c ← a - b	12	indéfini	???
d ← c - 2	12	indéfini	???

c ne peut pas être calculé car b n'a pas été initialisé ; quant à d, il n'est même pas déclaré !

À titre d'exemple, récrivons l'algorithme d'addition de fractions décrit en début de chapitre :

```
module additionnerFractions()
    num1, den1, num2, den2, numRes, denRes : entiers
    lire num1, den1, num2, den2
    denRes ← den1 * den2
    numRes ← num1*den2 + num2*den1
    afficher numRes, "/", denRes
fin module
```

Remarque : rappelons que la fraction affichée n'est sans doute pas simplifiée. Nous n'avons pas encore tous les atouts suffisants pour réaliser cela à ce niveau. Patience !

2.8 Commenter un algorithme

On n'insistera jamais assez sur la nécessité de **documenter** un algorithme en y insérant des **commentaires** judicieux, clairs et suffisants ! Un commentaire est un texte placé dans l'algorithme et destiné à faciliter au maximum la compréhension d'un algorithme par le lecteur (parfois une autre personne, mais aussi souvent l'auteur qui se perd dans son propre texte lorsqu'il s'y replonge après une interruption). Ces commentaires (introduits par « // ») seront bien entendu ignorés par l'exécutant de l'algorithme.

```
// Lit les contenus de 2 fractions et affiche leur somme
module additionnerFractions()
    num1, den1, num2, den2, numRes, denRes : entiers
    lire num1, den1, num2, den2
    denRes ← den1 * den2                                  // calcul du dénominateur
    numRes ← num1*den2 + num2*den1                        // calcul du numérateur
                                          // la fraction n'est sans doute pas simplifiée
    afficher numRes, "/", denRes
fin module
```

Notez qu'un excès de commentaires peut être aussi nuisible qu'un trop-peu pour la compréhension d'un algorithme. Par exemple, un choix judicieux de noms de variables peut s'avérer bien plus efficace que des commentaires superflus. Ainsi, l'instruction

```
nouveauCapital ← ancienCapital * (1 + taux / 100)
```

dépourvue de commentaires est bien préférable aux lignes suivantes :

```
c1 ← c0 * (1 + t / 100)                                   // calcul du nouveau capital
                          // c1 est le nouveau capital, c0 est l'ancien capital, t est le taux
```

Nous prendrons l'habitude de commenter chaque module en précisant ce qu'il fait.

2.9 Compléments de pseudo-code

Présentons quelques notions algorithmiques peu utilisées en première mais qui vous seront peut-être utiles dans l'un ou l'autre des exercices.

2.9.1 Constantes

Une **constante** est une information pour laquelle nom, type et valeur sont figés. La liste des constantes utilisées dans un algorithme apparaitra dans la section déclaration des variables sous la forme suivante :

```
constante PI = 3,14
constante ESI = "École Supérieure d'Informatique"
```

Il est inutile de spécifier leur type, celui-ci étant défini implicitement par la valeur de la constante.

2.9.2 Énumération

Parfois, une variable ne peut prendre qu'un ensemble fixe et fini de valeurs. Par exemple une variable représentant une saison ne peut prendre que quatre valeurs (HIVER, PRINTEMPS, ÉTÉ, AUTOMNE). On va l'indiquer grâce à l'énumération qui introduit un **nouveau type** de donnée.

```
énumération Saison { HIVER, PRINTEMPS, ÉTÉ, AUTOMNE }
```

Il y a deux avantages à cela : une indication claire des possibilités de la variable lors de la déclaration et une lisibilité du code grâce à l'utilisation des valeurs explicites.

Par exemple,

```
// Lit une saison et affiche sa particularité
module particularitéSaisonnière()
    uneSaison : Saison
    lire uneSaison        // on lira la valeur HIVER ou PRINTEMPS ou ÉTÉ ou AUTOMNE
    si uneSaison = HIVER alors
        afficher "il neige"
    sinon
        si uneSaison = PRINTEMPS alors
            afficher "les fleurs poussent"
        sinon
            si uneSaison = ÉTÉ alors
                afficher "le soleil brille"
            sinon
                afficher "les feuilles tombent"
            fin si
        fin si
    fin si
fin module
```

Exercice : Autres situations

Pouvez-vous identifier d'autres données qui pourraient avantageusement s'exprimer avec une énumération ?

Quid des langages de programmation ?

Certains langages (comme Java) proposent un type énuméré complet. D'autres (comme C et C++) proposent un type énuméré incomplet mais qui permet néanmoins une écriture comme celle ci-dessus. Cobol propose des « noms conditions » qui représentent l'ensemble des valeurs possibles d'une variable. D'autres langages, enfin, ne proposent rien. Pour ces langages, le truc est de définir des constantes entières qui vont permettre une écriture proche de celle ci-dessus (mais sans une déclaration explicite).

Lien avec les entiers

Dans l'exemple ci-dessus, on lit une Saison mais souvent, si on travaille avec les Mois par exemple, on disposera plutôt d'un entier. Il faut pouvoir convertir les valeurs. Chaque langage de programmation propose sa propre technique ; nous allons adopter la syntaxe suivante :

```
Saison(3)                  // donne l'énumération de la saison numéro 3 (on commence à 1) ;
                                                    // donne ÉTÉ dans notre exemple.
position(uneSaison)                          // donne l'entier associé à une saison ;
                        // si on a lu HIVER comme valeur pour uneSaison, donne la valeur 1.
```

2.10 Exercices

2.10.1 Pour s'échauffer

1 Compréhension d'algorithme

Pour ces exercices, nous vous demandons de comprendre des algorithmes donnés. Plus précisément, que vont-ils afficher si à chaque fois les deux nombres lus au départ sont successivement 2 et 3 ?

```
module exerciceA()
    a,b : entiers
    lire a,b
    a ← a + 2 * b
    afficher a
fin module
```

```
module exerciceB()
    a,b : entiers
    quotient : réel
    lire b,a
    quotient ← a / b
    afficher quotient
fin module
```

```
module exerciceC()
    a,b,c,d : entiers
    lire c,d
    a ← 2 * c + 5 * d
    b ← 2 + c * 3 + d
    c ← a MOD b
    afficher a DIV c
fin module
```

```
module exerciceD()
    x,y : réels
    lire x,y
    x ← x * x
    x ← x * x + y * y
    x ← √x
    afficher x
fin module
```

```
module exerciceE()
    x,y : réels
    lire x,x
    x ← x MOD x + (x + 1) DIV 2
    afficher x + 3
fin module
```

2 Le jeu des n erreurs

Identifier les erreurs et/ou les problèmes dans les modules suivants :

```
module exA()    // Division
    nb1, nb2 : réels
    afficher nb1 / nb2
fin module
```

```
module exB()    // Somme
    nb1, nb2 : réels
    lire nb1, nb2
    m ← (nb1 + nb2) / 2
    afficher m
fin module
```

3 Simplification d'algorithme

Voici quelques extraits d'algorithmes corrects du point de vue de la syntaxe mais contenant des lignes inutiles ou des lourdeurs d'écriture. Remplacer chacune de ces portions d'algorithme par un minimum d'instructions qui auront un effet équivalent.

```
hauteur ← largeur * 50    // les deux variables sont des entiers
hauteur ← hauteur MOD 5 + 7
```

```
lire var
var ← 0
```

```
var ← 0
lire var
```

```
ok2 ← NON (ok1 = vrai) ET NON (ok1 = faux)
```

2.10.2 Exercices d'apprentissage

Il est temps de se lancer et d'écrire vos premiers modules. Voici quelques précieux conseils pour vous guider dans la résolution de tels problèmes :

- ▷ il convient d'abord de bien comprendre le problème posé ; assurez-vous qu'il est parfaitement spécifié ;

- ▷ déclarez ensuite les variables (et leur type) qui interviennent dans l'algorithme ; les noms des variables risquant de ne pas être suffisamment explicites ;
- ▷ mettez en évidence les variables « données », les variables « résultats » et les variables de travail ;
- ▷ n'hésitez pas à faire une ébauche de résolution en français avant d'élaborer l'algorithme définitif pseudo-codé.

4 Surface d'un triangle

Écrire un algorithme qui calcule et affiche la surface d'un triangle connaissant sa base et sa hauteur.

5 Placement

Écrire un algorithme qui, étant donné le montant d'un capital placé (en €) et le taux d'intérêt annuel (en %), calcule et affiche la valeur nouvelle de ce capital après un an de ce placement.

6 Prix TTC

Écrire un algorithme qui, étant donné le prix unitaire d'un produit (hors TVA), le taux de TVA (en %) et la quantité de produit vendue à un client, calcule et affiche le prix total à payer par ce client.

7 Durée de trajet

Écrire un algorithme qui, étant donné la vitesse moyenne en **m/s** d'un véhicule et la distance parcourue en **km** par ce véhicule, calcule et affiche la durée en secondes du trajet de ce véhicule.

8 Permutation

Écrire une séquence d'instructions qui réalise la permutation du contenu de deux variables.

9 Cote moyenne

Écrire un algorithme qui, étant donné les résultats (cote entière sur 20) de trois examens passés par un étudiant (exprimés par six nombres, à savoir, la cote et la pondération de chaque examen), calcule et affiche la moyenne globale exprimée en pourcentage. Vérifiez bien votre algorithme avec des valeurs concrètes ; il est facile de se tromper dans la formule !

10 Somme des chiffres

Écrire un algorithme qui calcule la somme des chiffres d'un nombre entier de 3 chiffres. Réflexion : l'algorithme est-il aussi valide pour les entiers inférieurs à 100 ?

11 Conversion HMS en secondes

Écrire un algorithme qui, étant donné un moment dans la journée donné par trois nombres lus, à savoir, heure, minute et seconde, calcule le nombre de secondes écoulées depuis minuit.

12 Conversion secondes en HMS

Écrire un algorithme qui, étant donné un temps écoulé dans la journée exprimé en secondes, calcule et affiche ce temps sous la forme de trois nombres (heure, minute et seconde).

Ex : 10000 secondes donnera 2h 46'40"

Chapitre 3

Les alternatives

Dans ce chapitre, nous abordons les structures alternatives qui permettent de conditionner des parties d'algorithmes. Elles ne seront exécutées que si une condition est satisfaite.

3.1 « si – alors – sinon »

Cette structure permet d'exécuter une partie de code ou une autre en fonction de la valeur de vérité d'une condition.

si – alors

```
si condition alors
    // instructions à réaliser si la condition est VRAIE
fin si
```

La **condition** est une expression délivrant un résultat booléen (**vrai** ou **faux**) ; elle associe des variables, constantes, expressions arithmétiques, au moyen des opérateurs logiques ou de comparaison. En particulier, cette condition peut être réduite à une seule variable booléenne.

Dans cette structure, lorsque la condition est vraie, il y a exécution de la séquence d'instructions contenue entre les mots **alors** et **fin si** ; ensuite, l'algorithme continue de façon séquentielle.

Lorsque la condition est fausse, les instructions se trouvant entre **alors** et **fin si** sont tout simplement ignorées.

si – alors – sinon

```
si condition alors
    // instructions à réaliser si la condition est VRAIE
sinon
    // instructions à réaliser si la condition est FAUSSE
fin si
```

Dans cette structure, une et une seule des deux séquences est exécutée.

Exemple : Signe d'un nombre

Écrire un algorithme qui affiche si un nombre lu est positif (zéro inclus) ou strictement négatif.

Solution

```
// Lit un nombre et affiche si ce nombre est positif (zéro inclus) ou strictement négatif
module signeNombre()
    nb : entier
    lire nb
    si nb < 0 alors
        afficher "le nombre", nb, " est négatif"
    sinon
        afficher "le nombre", nb, " est positif ou nul"
    fin si
fin module
```

Exercice : Signe d'un nombre (amélioré)

Écrire un algorithme qui dit si un nombre lu est positif, négatif ou nul.

3.2 Indentation

Dans l'écriture de tout algorithme, on veillera à **indenter** correctement les lignes de codes afin de faciliter sa lecture ; cela veut dire que :

- ▷ Les **balises** encadrant toute structure de contrôle devront être parfaitement à la verticale l'une de l'autre : **module** et **fin module** ; **si** [, **sinon**] et **fin si**. Ce sera aussi vrai pour celles que nous allons voir plus tard : *selon que*, *tant que*, *faire jusqu'à ce*, *pour*...
- ▷ Les lignes situées entre toute paire de balises devront être décalées d'une tabulation vers la droite.
- ▷ On pensera aussi à tracer une **ligne verticale** entre le début et la fin d'une structure de contrôle afin de mieux la délimiter encore (surtout lorsqu'on travaille sur papier).

3.3 « selon que »

Avec ces structures, plusieurs branches d'exécution sont disponibles. L'ordinateur choisit la branche à exécuter en fonction de la valeur d'une variable (ou parfois d'une expression) ou de la condition qui est vraie.

selon que (version avec listes de valeurs)

```
selon que variable vaut
 liste_1 de valeurs séparées par des virgules :
    // instructions lorsque la valeur de la variable est dans liste_1
 liste_2 de valeurs séparées par des virgules :
    // instructions lorsque la valeur de la variable est dans liste_2
    ...
 liste_n de valeurs séparées par des virgules :
    // instructions lorsque la valeur de la variable est dans liste_n
 autres :
    // instructions lorsque la valeur de la variable
    // ne se trouve dans aucune des listes précédentes
fin selon que
```

Dans ce type de structure, comme pour la structure **si-alors-sinon**, une seule des séquences d'instructions sera exécutée. On veillera à ne pas faire apparaitre une même valeur dans plusieurs listes. Cette structure est une simplification d'écriture de plusieurs alternatives imbriquées. Elle est équivalente à :

```
si variable = une des valeurs de la liste_1 alors
    // instructions lorsque la valeur est dans liste_1
sinon
    si variable = une des valeurs de la liste_2 alors
        // instructions lorsque la valeur est dans liste_2
    sinon
        ...
        si variable = une des valeurs de la liste_n alors
            // instructions lorsque la valeur est dans liste_n
        sinon
            // instructions lorsque la valeur de la variable
            // ne se trouve dans aucune des listes précédentes
        fin si
    fin si
fin si
```

Notez que le cas **autres** est facultatif.

selon que (version avec conditions)

```
selon que
 condition_1 :
    // instructions lorsque la condition_1 est vraie
 condition_2 :
    // instructions lorsque la condition_2 est vraie
    ...
 condition_n :
    // instructions lorsque la condition_n est vraie
 autres :
    // instructions à exécuter quand aucune
    // des conditions précédentes n'est vérifiée
fin selon que
```

Comme précédemment, une et une seule des séquences d'instructions est exécutée. On veillera à ce que les conditions ne se « recouvrent » pas, c'est-à-dire que deux d'entre elles

ne soient jamais vraies simultanément. C'est équivalent à :

```
si condition_1 alors
    // instructions lorsque la condition_1 est vraie
sinon
    si condition_2 alors
        // instructions lorsque la condition_2 est vraie
    sinon
        ...
        si condition_n alors
            // instructions lorsque la condition_n est vraie
        sinon
            // instructions à exécuter quand aucune
            // des conditions précédentes n'est vérifiée
        fin si
    fin si
fin si
```

Exemple : Jour de la semaine en clair

Écrire un algorithme qui lit un jour de la semaine sous forme d'un nombre entier (1 pour lundi, ..., 7 pour dimanche) et qui affiche en clair ce jour de la semaine.

Solution

```
// Lit un nombre entre 1 et 7 et affiche en clair le jour de la semaine correspondant.
module jourSemaine()
    jour : entier
    lire jour
    selon que jour vaut
        1 : afficher "lundi"
        2 : afficher "mardi"
        3 : afficher "mercredi"
        4 : afficher "jeudi"
        5 : afficher "vendredi"
        6 : afficher "samedi"
        7 : afficher "dimanche"
    fin selon que
fin module
```

Exemple : Nombre de jours (avec énumération)

Reprendre l'algorithme qui affiche le nombre de jours dans un mois en utilisant une énumération.

Solution

```
énumération Mois {JANVIER, FÉVRIER, MARS, AVRIL, MAI, JUIN, JUILLET, AOÛT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DÉCEMBRE}
```

```
// Lit un Mois et affiche le nombre de jours correspondant
// (en ne tenant pas compte des années bissextiles).
module nbJours()
    unMois : Mois
    lire unMois            // on lira la valeur JANVIER ou FÉVRIER ou ... ou DÉCEMBRE
    selon que unMois vaut
     JANVIER, MARS, MAI, JUILLET, AOÛT, OCTOBRE, DÉCEMBRE:
        afficher 31
     AVRIL, JUIN, SEPTEMBRE, NOVEMBRE:
        afficher 30
     FÉVRIER:                          // on ne tient pas compte ici des années bissextiles
        afficher 28
    fin selon que
    // Pas de clause « autre » car toutes les valeurs possibles ont été envisagées
fin module
```

3.4 Exercices

1 Compréhension

Qu'affichent les algorithmes suivants si à chaque fois les deux nombres lus au départ sont, dans l'ordre, 2 et 3 ? Même question avec 4 et 1.

```
module exerciceA()
    a,b : entier
    lire a,b
    si a > b alors
        a ← a + 2 * b
    fin si
    afficher a
fin module
```

```
module exerciceB()
    a,b,c : entier
    lire b,a
    si a > b alors
        c ← a DIV b
    sinon
        c ← b MOD a
    fin si
    afficher c
fin module
```

```
module exerciceC()
    x1,x2 : entier
    ok : booléen
    lire x1,x2
    ok ← x1 > x2
    si ok alors
        ok ← ok ET x1 = 4
    sinon
        ok ← ok OU x2 = 3
    fin si
    si ok alors
        x1 ← x1 * 1000
    fin si
    afficher x1 + x2
fin module
```

2 Simplification d'algorithmes

Voici quelques extraits d'algorithmes corrects du point de vue de la syntaxe mais contenant des lignes inutiles ou des lourdeurs d'écriture. Remplacer chacune de ces portions d'algorithme par un minimum d'instructions qui auront un effet équivalent.

```
si ok = vrai alors
    afficher nombre
fin si
```

```
si ok = faux alors
    afficher nombre
fin si
```

```
si condition alors
    ok ← vrai
sinon
    ok ← faux
fin si
```

```
si a > b alors
    ok ← faux
sinon
    si a ≤ b alors
        ok ← vrai
    fin si
fin si
```

```
si ok1 = vrai ET ok2 = vrai alors
    afficher x
fin si
```

3 Maximum de 2 nombres

Écrire un algorithme qui, étant donné deux nombres quelconques, recherche et affiche le plus grand des deux. Attention ! On ne veut pas savoir si c'est le premier ou le deuxième qui est le plus grand mais bien quelle est cette plus grande valeur. Le problème est donc bien défini même si les deux nombres sont identiques.

4 Maximum de 3 nombres

Écrire un algorithme qui, étant donné trois nombres quelconques, recherche et affiche le plus grand des trois.

5 Le signe

Écrire un algorithme qui affiche un message indiquant si un entier est strictement négatif, nul ou strictement positif.

6 La fourchette

Écrire un algorithme qui, étant donné trois nombres, recherche et affiche si le premier des trois appartient à l'intervalle donné par le plus petit et le plus grand des deux autres (bornes exclues). Qu'est-ce qui change si on inclut les bornes ?

7 Équation du deuxième degré

Écrire un algorithme qui, étant donné une équation du deuxième degré, déterminée par le coefficient de x^2, le coefficient de x et le terme indépendant, recherche et affiche la (ou les) racine(s) de l'équation (ou un message adéquat s'il n'existe pas de racine réelle).

8 Une petite minute

Écrire un algorithme qui, à partir d'un moment exprimé par 2 entiers, heure et minute, affiche le moment qu'il sera une minute plus tard.

9 Calcul de salaire

Dans une entreprise, une retenue spéciale de 15% est pratiquée sur la partie du salaire mensuel qui dépasse 1200 €. Écrire un algorithme qui calcule le salaire net à partir du salaire brut. En quoi l'utilisation de constantes convient-elle pour améliorer cet algorithme ?

10 Nombre de jours dans un mois

Écrire un algorithme qui donne le nombre de jours dans un mois. Le mois est lu sous forme d'un entier (1 pour janvier...). On considère dans cet exercice que le mois de février comprend toujours 28 jours.

11 Année bissextile

Écrire un algorithme qui vérifie si une année est bissextile. Pour rappel, les années bissextiles sont les années multiples de 4. Font exception, les multiples de 100 (sauf les multiples de 400 qui sont bien bissextiles). Ainsi 2012 et 2400 sont bissextile mais pas 2010 ni 2100.

12 Valider une date

Écrire un algorithme qui valide une date donnée par trois entiers : l'année, le mois et le jour.

13 Le jour de la semaine

Écrire un algorithme qui lit un jour du mois de novembre de cette année (sous forme d'un entier entre 1 et 30) et qui affiche le nom du jour (« lundi », « mardi »...)

14 Quel jour serons-nous ?

Écrire un algorithme qui indique le nom du jour qui sera n jours plus tard qu'un jour donné. Par exemple si le jour donné est « mercredi » et $n = 10$, l'algorithme indiquera « samedi ».

15 Un peu de trigono

Écrire un algorithme qui pour un entier n donné, affiche la valeur de $cos(n * \pi/2)$.

16 Le stationnement alternatif

Dans une rue où se pratique le stationnement alternatif, du 1 au 15 du mois, on se gare du côté des maisons ayant un numéro impair, et le reste du mois, on se gare de l'autre côté. Écrire un algorithme qui, sur base de la date du jour et du numéro de maison devant laquelle vous vous êtes arrêté, indique si vous êtes bien stationné ou non.

Chapitre 4

Les modules

Dans ce chapitre, nous voyons pourquoi et comment découper un algorithme en modules (morceaux d'algorithmes).

4.1 Introduction

Outre le choix de noms de variables explicites et une indentation correcte, tout bon programmeur se doit de veiller à la clarté et la lisibilité de ses algorithmes. Si ces qualités sont relativement aisées à assurer dans les premiers exemples et exercices abordés, il n'en est pas de même lorsque croissent la complexité et la longueur des algorithmes. Il est courant que les lignes de codes de programmes se comptent en centaines ou en milliers. Dès lors, le découpage en modules est indispensable. En outre, lors de l'écriture d'un algorithme, on conseille, dans un souci de lisibilité, qu'aucun des modules ne dépasse la longueur d'une vingtaine de lignes.

Illustrons l'approche modulaire sur l'exercice du chapitre précédent demandant de calculer le maximum de 3 nombres. Commençons par écrire la solution du problème plus simple : le maximum de 2 nombres.

```
module max2()                    // Lit deux nombres et affiche le maximum des deux.
    a,b,max : réels
    lire a,b
    si a > b alors
        max ← a
    sinon
        max ← b
    fin si
    afficher max
fin module
```

Pour le maximum de 3 nombres, il existe plusieurs approches. Voyons celle-ci :

```
Calculer le maximum des deux premiers nombres, soit temp
Calculer le maximum de temp et du troisième nombre, ce qui donne le résultat.
```

Sur base de cette idée, comment faire à présent pour introduire le calcul du maximum de 2 nombres dans l'algorithme calculant le maximum de 3 nombres ? Une solution consiste à « copier-coller » les lignes de code du module max2 dans le module max3, toutefois en adaptant son contenu au contexte de ce module : les nombres a et b ne doivent plus être systématiquement lus et la valeur du maximum ne doit plus être systématiquement affichée. Ainsi temp est calculé et ré-utilisé dans un calcul ultérieur. Ceci donnerait :

```
module max3()                    // Lit trois nombres et affiche le maximum des trois.
    a, b, c, temp, max : réels
    lire a,b,c
    si a > b alors
        temp ← a
    sinon
        temp ← b
    fin si
    si temp > c alors
        max ← temp
    sinon
        max ← c
    fin si
    afficher max
fin module
```

Bien que correcte, cette démarche est cependant déconseillée et peut s'avérer fastidieuse, spécialement dans le cas de « grands » algorithmes. Imaginons qu'il aurait fallu de cette façon « mixer » deux algorithmes d'une cinquantaine de lignes chacun. Le résultat serait un algorithme d'une centaine de lignes qui perdrait en lisibilité et clarté. De plus, l'opération effectuée n'est pas sans risques : que se passerait-il si les deux modules contiennent chacun une variable nommée de manière identique ? Cette « transplantation » demande donc la vérification de toutes les variables utilisées, la réécriture des lignes de déclarations de variables de façon à y inclure celles du module « greffé », le « raccord » correct des données lues et écrites par ce module aux variables du module principal, etc.

Imaginons, par exemple, que l'on doive calculer le maximum de 4 ou même 5 nombres. Le résultat serait un code long et à l'allure répétitive. Une erreur serait vite arrivée et serait difficile à détecter.

L'idéal serait de pouvoir garder deux modules séparés, conservant chacun leur spécificité (l'un calculant le maximum de deux nombres et l'autre le maximum de trois nombres) mais de leur permettre de communiquer entre eux pour s'échanger des données ou des résultats de calculs. Nous allons voir deux solutions.

4.2 Passage de paramètres

Pour pouvoir faire communiquer les modules entre eux, il faut les équiper d'une « interface » de transmission des variables appelée l'**en-tête** du module et qui contient une déclaration de variables qu'on appellera ici **paramètres** du module.

Les variables accompagnées d'une flèche vers le bas (↓) sont des **paramètres d'entrée** qui reçoivent des données au **début** de l'exécution du module (données qui ne doivent donc plus être lues) tandis que celles accompagnées d'une flèche vers le haut (↑) sont des **paramètres de sortie** qui permettent de renvoyer des résultats à l'**issue** de l'exécution du module (résultats qui ne doivent donc plus être affichés).

Par exemple, ceci donnerait pour le module max2 :

```
module max2(a↓, b↓ : réels, max↑ : réel)      // Affiche deux nombres et sort le maximum
des deux.
    si a > b alors
        max ← a
    sinon
        max ← b
    fin si
fin module
```

Notons que, sous cette forme, ce module est devenu « inactif », il ne lit rien ni n'affiche rien,

mais il est cependant prêt à être utilisé. Il suffit pour cela de lui fournir les valeurs de a et b (paramètres d'entrée) pour qu'il entre en action. Une fois le maximum calculé, celui-ci est affecté à la variable max qui – en tant que paramètre de sortie – contiendra et renverra le résultat voulu.

Pour faire appel aux services de ce module, il suffit à présent d'écrire son nom suivi d'un nombre de variables (ou, en entrée, d'expressions) en accord avec son en-tête. Montrons par un exemple comment le module max3 peut faire appel au module max2.

```
module max3()                          // Lit trois nombres et affiche le maximum des trois.
    a, b, c, temp, max : réels
    lire a, b, c
    max2( a, b, temp )
    max2( temp, c, max )
    afficher max
fin module
```

L'instruction max2(a, b, temp) se déroule comme suit :

- le contenu des variables a et b est affecté aux paramètres d'entrée a et b du module max2, puis ce module peut entrer en action ;
- à l'issue du module max2, la valeur du paramètre de sortie max est communiquée à la variable temp, qui contient à présent le maximum de a et b

L'instruction max2(temp, c, max) se déroule comme suit :

- le contenu des variables temp et c est affecté aux paramètres d'entrée a et b du module max2, puis ce module peut entrer en action ;
- à l'issue du module max2, la valeur du paramètre de sortie max est communiquée à la variable max qui contient à présent le maximum des 3 nombres de départ.

Il est évident que la présence de ces deux instructions dans le module max3 implique la présence du module max2 « aux alentours » du premier, c'est-à-dire que ces deux modules doivent se trouver sur le même support d'écriture de l'algorithme complet.

Implicitement, on a introduit ici un nouveau type de primitive, l'**appel à un module** qui consiste en la simple écriture de son nom dans le code. Cette instruction est reconnue comme telle car elle ne s'apparente pas aux autres types d'instruction vus précédemment (primitives **lire**, **afficher** affectation (← , structures de contrôle **si**, **selon que**, **tant que**, **pour**, etc.) Lorsque l'ordinateur rencontre cette instruction, il va rechercher si un module portant ce nom existe. Une fois ce module trouvé, les paramètres *ad hoc* lui sont communiqués et toutes les instructions le composant sont exécutées séquentiellement. Arrivé à l'issue du module, les paramètres de sortie sont communiqués au module appelant et le déroulement de celui-ci se poursuit là où il avait été interrompu.

Remarques

- un paramètre peut être à la fois paramètre d'entrée et de sortie ; on le fait suivre alors de la double flèche ↓↑ ;
- si tous les paramètres sont en entrée, on peut omettre les flèches ;
- la présence de ces paramètres n'est pas obligatoire, on peut envisager un module sans paramètre de sortie (par ex. un module qui reçoit en entrée un nombre et dont la seule fonction est de l'afficher à l'écran), ou inversement sans paramètre d'entrée (un module qui simule un lancer de dé, et renvoie en paramètre une valeur aléatoire entre 1 et 6) ;
- pour appeler correctement un module, il faut fournir des noms de variables, des expressions ou des constantes **en même nombre** et en même ordre que les paramètres du module ;

- ▷ en outre, les types des variables doivent correspondre entre l'appel et l'en-tête du module ;
- ▷ ne peut être affectée à un paramètre d'entrée du module qu'une constante, une expression ou une variable préalablement affectée ;
- ▷ à un paramètre de sortie d'un module doit toujours correspondre une autre variable, jamais une constante ou une expression ;
- ▷ il faut s'assurer qu'à l'issue du module, tous les paramètres de sortie aient été affectés lors de l'exécution de ce module.

4.3 Variables locales

Dans le cadre de ce cours d'algorithmique, nous n'envisagerons que l'utilisation de **variables locales**. Toute variable est **locale** au module dans lequel elle apparait, ce qui veut dire que son existence est ignorée en dehors de ce module. Nous n'envisageons donc pas ici l'utilisation déconseillée de **variables globales** (variables dont le contenu est reconnu par tous les modules d'un programme).

Précisons qu'une variable locale est aussi **dynamique**, c'est-à-dire qu'un emplacement en mémoire lui est réservé durant l'exécution du module où elle est déclarée. Une fois l'exécution terminée, cet emplacement est récupéré en mémoire.

Plutôt qu'une restriction, cette propriété est une aide confortable au programmeur : si, de façon fortuite, des variables appartenant à des modules différents possédaient le même nom, il n'y aurait pas d'interférence entre ces variables, ni confusion possible de leur contenu. Par exemple, dans le module max2, les paramètres d'entrée a, b ainsi que le paramètre de sortie max sont locaux à ce module. Ce serait aussi le cas de toute autre variable temporaire déclarée dans le module.

Notons au passage que les variables déclarées dans l'en-tête du module ne doivent plus l'être dans la partie « déclaration de variables » de ce module. Mais toutes les variables utilisées dans un module doivent être déclarées, soit dans son en-tête, soit dans sa déclaration. Le non-respect de cette règle provoquerait une erreur d'exécution.

Pour illustrer le fonctionnement des variables locales, voici encore une nouvelle version de l'algorithme incluant une fonctionnalité qui va empêcher l'utilisateur d'entrer des nombres strictement négatifs. Le module lirePositif ci-dessous vérifie si un nombre lu est positif et affiche un message d'erreur sinon. Le texte complet de l'algorithme est le suivant :

```
// Lit trois nombres positifs et affiche le maximum des trois.
module max3()
    a, b, c, temp, max : réels
    lirePositif(a)
    lirePositif(b)
    lirePositif(c)
    max2(a, b, temp)
    max2(temp, c, max)
    afficher max
fin module
```

```
// Lit un nombre, le sort s'il est positif et sinon lance une erreur.
module lirePositif(a↑ : réel)
    lire a
    si a < 0 alors
        erreur Le nombre n'est pas positif !
    fin si
fin module
```

```
// Affiche deux nombres et sort le maximum des deux.
module max2(a↓, b↓ : réels, max↑ : réel)
    si a > b alors
        max ← a
    sinon
        max ← b
    fin si
fin module
```

Dans cet exemple, les trois modules en jeu utilisent une variable nommée a ; il n'y a cependant aucune confusion possible. Ainsi, à l'issue du deuxième appel du module lirePositif, le contenu de son paramètre a est affecté à la variable b du module max2.

4.4 Module renvoyant une valeur

Un deuxième type de module est le **module renvoyant une valeur**. (On désigne aussi ce type de modules par le terme **fonction**). Son en-tête est du type suivant :

```
module exemple (var1↓ : type1, var2↓ : type2, ..., varN↓ : typeN) → typeRes
```

Il se distingue du module précédemment étudié par la flèche de renvoi à droite, et possède les particularités suivantes :

- ce module renvoie **une valeur qui n'est pas affectée à une variable de sortie** ;
- à droite de la flèche de renvoi ne se trouve donc pas le nom d'un paramètre de sortie, mais le **type** de la valeur renvoyée ;
- typeRes est le type de la valeur renvoyée ; en théorie, ce peut être un type simple (entier, réel, booléen, chaine, caractère), un type structuré ou même un tableau (ces types seront vus dans les chapitres suivants). En pratique il conviendra de s'en tenir aux limitations du langage utilisé ;
- var1, ..., varN sont les paramètres du module (aussi appelés **arguments**) ; ce sont, le plus souvent, des paramètres d'entrée, les paramètres de sortie étant plus rares dans ce type de module ;
- ces arguments deviennent automatiquement des variables locales du module ; déclarées dans son en-tête, elles ne doivent plus l'être dans la partie déclarative ;
- la valeur renvoyée est définie à la fin du module via la primitive **retourner** résultat, où résultat est une variable (ou plus généralement une expression) de type typeRes ;
- pour appeler ce type de module, on utilise son nom **comme celui d'une variable** ou dans une expression apparaissant dans le module appelant, mais jamais à gauche du signe d'affectation.

Comme précédemment, il faut veiller, lors de l'appel de ce type de module, à ce que le nombre d'arguments ainsi que leur type correspondent à ceux spécifiés dans l'en-tête.

Pour exemple, transformons une fois de plus la dernière version de notre algorithme. Plutôt que de renvoyer un résultat via un paramètre de sortie, les modules lirePositif et max2 renvoient à présent chacun uniquement une valeur de type *réel*. Notez que le premier de ces modules n'a pas d'argument. Notez également la façon d'appeler ces deux modules.

```
// Lit trois nombres positifs et affiche le maximum des trois.
module max3()
    a, b, c, temp, max : réels
    a ← lirePositif()
    b ← lirePositif()
    c ← lirePositif()
    temp ← max2(a, b)
    max ← max2(temp, c)
    afficher max
fin module
```

```
// Lit un nombre, le retourne s'il est positif et sinon lance une erreur.
module lirePositif() → réel
    a : réel
    lire a
    si a < 0 alors
        erreur Le nombre n'est pas positif !
    fin si
    retourner a
fin module
```

```
// Affiche deux nombres et retourne le maximum des deux.
module max2(a↓, b↓ : réels) → réel
    max : réel
    si a > b alors
        max ← a
    sinon
        max ← b
    fin si
    retourner max
fin module
```

L'écriture avec une valeur de retour permet une écriture plus compacte :

```
// Lit trois nombres positifs et affiche le maximum des trois.
module max3()
    a, b, c, max : réels
    a ← lirePositif()
    b ← lirePositif()
    c ← lirePositif()
    max ← max2(max2(a, b), c)
    afficher max
fin module
```

ou encore (mais on perd un peu en clarté) :

```
// Lit trois nombres positifs et affiche le maximum des trois.
module max3()
    a, b, c : réels
    a ← lirePositif()
    b ← lirePositif()
    c ← lirePositif()
    afficher max2(max2(a, b), c)
fin module
```

ou même (et là, ce n'est plus du tout lisible !) :

```
// Lit trois nombres positifs et affiche le maximum des trois.
module max3()
    afficher max2(max2(lirePositif(), lirePositif()), lirePositif())
fin module
```

4.5 Un exemple complet

Synthétisons les notions vues dans ce chapitre au travers d'un exemple complet.

Exercice résolu : Écart entre 2 moments

Étant donné deux moments dans une journée donnés chacun par trois nombres (heure, minute, seconde), écrire un algorithme qui calcule et affiche le délai écoulé entre ces deux moments en heure(s), minute(s), seconde(s) sachant que le deuxième moment donné est chronologiquement postérieur au premier.

Analyse du problème

Le problème n'est pas simple. On pourrait se dire qu'il suffit de soustraire les heures, les minutes et les secondes.

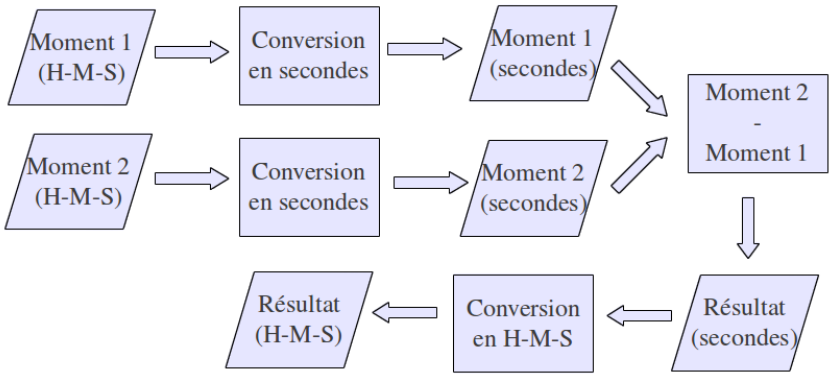
Ainsi, pour l'écart entre 1h22'23" et 3h34'56", on calcule 3h-1h=2h ; 34'-22'=12' et 56"-23"=33" ce qui donne un écart de 2h12'33".

Mais il s'agit là d'un cas favorable sans résultat négatif. Si une des soustractions est négative, comme par exemple pour l'écart entre 1h22'23" et 3h11'12", de simples soustractions ne suffisent pas.

On pourrait bien sûr compléter notre code pour tenir compte de tous les cas mais il existe aussi une autre approche. Rappelons-nous que nous avons déjà écrit des algorithmes proches de ce problème dans les exercices du chapitre sur les algorithmes séquentiels. Plus précisément :

```
module heuresEnSecondes()
module secondesEnHeures()
```

Cela peut nous mettre sur la voie d'une approche détournée pour arriver à nos fins. Le chemin sera plus long mais plus simple à écrire. L'idée est de tout convertir en secondes car alors la soustraction est simple. Le résultat est alors reconverti en heures-minutes-secondes.



Conversion en secondes

Une des sous-tâches est donc la conversion en secondes d'un moment exprimé en heures-minutes-secondes (h-m-s). Il nous faut adapter la solution trouvée pour l'exercice du chapitre sur les algorithmes séquentiels car il n'est pas question ici que les données soient lues ni que le

résultat soit écrit ; l'interaction ne se fait pas avec l'utilisateur mais avec le module principal qui va l'utiliser [1].

La première question à se poser est donc celle des paramètres :

- ▷ Quelles sont les données dont a besoin l'algorithme pour travailler ?
- ▷ Quels résultats fournit-il ?

Dans notre cas, la réponse est simple :

- ▷ Les données sont le moment à convertir en secondes. Ce moment est représenté par trois entiers : les heures, les minutes et les secondes ;
- ▷ Le résultat est le moment converti en secondes. Il est représenté par un entier.

Lorsque le résultat est représenté par une seule donnée, on a le choix entre un paramètre en sortie :

```
// Reçoit trois entiers : des heures, des minutes et des secondes
// et sort le nombre de secondes correspondant.
module hmsVersSec(h↓, m↓, s↓, secondes↑ : entiers)
```

ou une valeur de retour :

```
// Reçoit trois entiers : des heures, des minutes et des secondes
// et retourne le nombre de secondes correspondant.
module hmsVersSec(h↓, m↓, s↓ : entiers) → entier
```

Dans de pareils cas, on privilégie souvent la valeur de retour car cela facilite l'écriture lors de l'appel.

L'algorithme s'écrit alors :

```
// Reçoit trois entiers : des heures, des minutes et des secondes
// et retourne le nombre de secondes correspondant.
module hmsVersSec(h↓, m↓, s↓ : entiers) → entier
    secondes : entier                    // À déclarer puisque ce n'est pas un paramètre
    secondes ← h * 3600 + m * 60 + s
    retourner secondes
fin module
```

ou de manière équivalente mais plus concise :

```
// Reçoit trois entiers : des heures, des minutes et des secondes
// et retourne le nombre de secondes correspondant.
module hmsVersSec(h↓, m↓, s↓ : entiers) → entier
    retourner h * 3600 + m * 60 + s
fin module
```

4.5.1 Conversion en heures-minutes-secondes

À la fin de notre algorithme, il nous faudra reconvertir un résultat exprimé en secondes sous la forme heures-minutes-secondes. À nouveau, on a déjà résolu ce problème dans le chapitre sur les algorithmes séquentiels. Mais il faut l'adapter à l'usage de paramètres.

- ▷ Quelles sont les données ? Une seule, le moment exprimé en secondes

1. On pourrait, dans notre cas précis, imaginer que le module de conversion demande les données à l'utilisateur mais un tel module serait moins souvent réutilisable.

▷ Quels sont les résultats calculés par le module ? Ce même moment exprimé en heures-minutes-secondes. Trois entiers sont requis ce qui fait que le choix entre un paramètre en sortie et une valeur de retour ne se pose pas ici ; impossible d'utiliser une valeur de retour (qui doit être unique) ; on doit utiliser des paramètres en sortie.

Ce qui donne :

```
// Reçoit un nombre de secondes et sort les heures, les minutes et les secondes correspondant.
module secVersHMS(secondes↓, h↑, m↑, s↑ : entiers)
    h ← secondes DIV 3600
    m ← (secondes MOD 3600) DIV 60
    s ← secondes MOD 60
fin module
```

4.5.2 La solution

À présent, on a tout pour écrire la solution à notre problème

```
// Lit deux moments (h-m-s) et affiche le moment de la différence entre les deux.
module différenceEntreHeures()                                  // Pas de paramètre !
    h1, m1, s1, h2, m2, s2 : entiers                          // Les 2 moments à soustraire
    secondes1, secondes2 : entiers                            // Ces 2 moments en secondes
    diffSecondes : entier                                       // La différence en secondes
    diffH, diffM, diffS : entiers                               // La différence en H-M-S
    lire h1, m1, s1, h2, m2, s2
    secondes1 ← hmsVersSec( h1, m1, s1 )
    secondes2 ← hmsVersSec( h2, m2, s2 )
    diffSecondes ← secondes2 – secondes1
    secVersHMS( diffSecondes, diffH, diffM, diffS )
    afficher diffH, diffM, diffS
fin module
```

Réflexion

Dans la solution ci-dessus, quelle est ou quelles sont la/les variable(s) locale(s) dont on pourrait se passer moyennant une réécriture de l'algorithme ?

4.6 Blocs

Un bloc est l'écriture d'une portion de module à l'extérieur de celui-ci. C'est un simple *déplacement* de lignes de code vers un autre endroit du texte de l'algorithme. La raison de découper un module en blocs peut être le souci de clarifier un algorithme en le découpant en étapes bien distinctes, ou tout simplement un manque de place. Les variables d'un bloc ne sont donc pas des variables locales du bloc dans lequel elles apparaissent, mais bien des variables locales du module auquel appartient ce bloc. L'appel de l'exécution des instructions se trouvant dans un bloc est similaire à celui d'un module avec paramètres, on écrit simplement le nom du bloc comme s'il s'agissait d'une instruction.

Pour exemple, le module additionnerFractions (première version dans le chapitre sur les algorithmes séquentiels) pourrait se découper ainsi :

```
// Lit les contenus de 2 fractions et affiche leur somme
module additionnerFractions()
    déclaration
    lectureDonnées
    calculs
    écritureRésultat
fin module
```

```
bloc déclaration
    num1, den1, num2, den2, numRes, denRes : entiers
fin bloc
```

```
bloc lectureDonnées
    lire num1, den1, num2, den2
fin bloc
```

```
bloc calculs
    numRes ← num1 * den2 + num2 * den1
    denRes ← den1 * den2
fin bloc
```

```
bloc écritureRésultat
    afficher numRes, "/", denRes                    // la fraction n'est pas simplifiée
fin bloc
```

Le bloc contenant la déclaration des variables est aussi appelé **dictionnaire des variables** du module.

4.7 Qu'est-ce qu'un algorithme de qualité ?

Nous voulons tous produire du code de qualité mais qu'est-ce que cette notion recouvre vraiment ? Qu'est-ce qui permet de juger de la qualité d'un code ? De nombreux critères existent. Citons ici, parmi les plus pertinents, ceux qui sont le plus liés à ce cours [2].

4.7.1 La validité

C'est l'évidence, le code doit réaliser les tâches pour lesquelles il a été écrit, même dans tous les cas particuliers imaginés. C'est là le critère principal et les autres ne sont là que pour aider à atteindre celui-ci plus facilement. Ne nous leurrons pas, ce critère est loin d'être facile à rencontrer. Il suffit pour s'en convaincre de se rappeler toutes les fois où des logiciels pourtant courants que nous utilisons se sont déjà arrêtés sur des erreurs.

4.7.2 L'extensibilité (ou évolutivité)

Un code valide résout un problème particulier. Mais dans les situations réelles, le problème change régulièrement. Nous voudrons adapter le logiciel à de nouveaux besoins, à une modification de la législation, parce qu'un besoin était mal compris... Ces modifications peuvent parfois être prévues mais pas toujours.

Le code doit être écrit de telle sorte qu'un changement mineur dans le problème n'implique qu'un changement mineur dans le code. Ce changement doit être évident, facile à opérer et rapide. Il ne doit pas mettre à mal la validité du code.

2. Pour approfondir cette notion, l'étudiant pourra lire par exemple : Bertrand Meyer, « *Conception et programmation par objets : pour du logiciel de qualité* », aux éditions « *interéditions* »

4.7.3 La réutilisabilité

On désigne par ce terme la capacité de réutiliser un bout de code d'un projet dans un autre projet de façon aisée et sûre. L'avantage est de pouvoir employer à nouveau un code qui a déjà fait ses preuves dans le but de gagner du temps que l'on pourra investir dans les aspects algorithmiques propres au nouveau projet. Le découpage en modules joue ici un rôle essentiel.

4.7.4 La lisibilité

La lisibilité d'un code recouvre une notion globale et locale. Au niveau global, on doit facilement comprendre la structure générale du code afin de pouvoir aisément comprendre où se trouve la portion de code qui nous intéresse. Au niveau local, on doit pouvoir comprendre aisément chaque bout de code.

Au niveau du procédural (on nomme ainsi la programmation qui se structure à l'aide de modules), cela revient à dire qu'on doit rapidement voir quelles sont les fonctions et leur rôle (niveau global) et que chaque fonction sera aisément compréhensible (niveau local).

Est-ce que la lisibilité d'un code passe par l'abondance de commentaire ? Non, pas forcément. Le commentaire sert à expliquer ce qui n'est pas immédiatement clair dans le code. La première source de lisibilité provient donc du code lui-même. C'est pourquoi on est attentif à la bonne découpe en modules, au choix judicieux des noms de modules et de variables... Le respect de ces bonnes pratiques diminue le besoin de commentaires.

4.7.5 L'efficience

Ce critère s'intéresse à la bonne utilisation des ressources informatiques. Est-ce que le programme va tourner suffisamment vite, utiliser peu de mémoire ? Faut-il chercher systématiquement à optimiser son code ? Non ! On a souvent tendance à surévaluer ce critère ce qui est dommageable pour trois raisons.

- ▷ Souvent, l'approche directe est suffisante. Par exemple, la plupart des applications sont interactives. C'est alors l'utilisateur qui est le plus lent, pas la machine qui a tout le temps d'effectuer sa tâche pendant que l'utilisateur réfléchit à ce qu'il doit taper.
- ▷ En général, un programme passe 80% de son temps dans 20% de son code. Chercher à optimiser chaque ligne de code est une perte de temps.
- ▷ Accroitre la rapidité d'un code est presque toujours en contradiction avec sa lisibilité. On risque d'utiliser un algorithme moins connu ou des raccourcis difficiles à suivre.

C'est pourquoi on recommande en général, face à un problème, d'utiliser la solution la plus courante, la plus claire. Une fois le programme terminé, et seulement si le besoin s'en fait sentir, on identifiera les portions de code où le programme s'attarde le plus et on cherchera à les optimiser.

4.8 Exercices

1 Compréhension

Indiquer quels nombres sont successivement affichés lors de l'exécution des modules ex1, ex2, ex3 et ex4.

```
module ex1()
    x, y : entiers
    addition(3, 4, x)
    afficher x
    x ← 3
    y ← 5
    addition(x, y, y)
    afficher y
fin module

module addition(a↓, b↓, c↑ : entiers)
    somme : entier
    somme ← a + b
    c ← somme
fin module
```

```
module ex2()
    a, b : entiers
    addition(3, 4, a)    // voir ci-dessus
    afficher a
    a ← 3
    b ← 5
    addition(b, a, b)
    afficher b
fin module
```

```
module ex3()
    a, b, c : entiers
    calcul(3, 4, c)
    afficher c
    a ← 3
    b ← 4
    c ← 5
    calcul(b, c, a)
    afficher a, b, c
fin module

module calcul(a↓, b↓, c↑ : entiers)
    a ← 2 * a
    b ← 3 * b
    c ← a + b
fin module
```

```
module ex4()
    a, b, c : entiers
    a ← 3
    b ← 4
    c ← f(b)
    afficher c
    calcul2(a, b, c)
    afficher a, b, c
fin module

module calcul2(a↓, b↓, c↑ : entiers)
    a ← f(a)
    c ← 3 * b
    c ← a + c
fin module

module f(a↓ : entier) → entier
    b : entier
    b ← 2 * a + 1
    retourner b
fin module
```

2 Appels de module

Parmi les instructions suivantes (où les variables a, b et c sont des entiers), lesquelles font correctement appel au module d'en-tête suivant ?

```
module PGCD(a↓,b↓ : entiers) → entier
```

```
[1] a ← PGCD(24, 32)
[2] a ← PGCD(a, 24)
[3] b ← 3 * PGCD(a + b, 2*c) + 120
[4] PGCD(20, 30)
[5] a ← PGCD(a, b, c)
[6] a ← PGCD(a, b) + PGCD(a, c)
[7] a ← PGCD(a, PGCD(a, b))
[8] lire PGCD(a, b)
[9] afficher PGCD(a, b)
[10] PGCD(a, b) ← c
```

3 Comparaison d'algorithmes

Soit un module testant si un nombre entier est pair. Comparez les différentes solutions proposées.

```
// Affiche un entier et retourne vrai s'il est pair et faux sinon.
module estPair(nb↓ : entier) → booléen
    pair : booléen
    pair ← faux
    si nb MOD 2 = 0 alors
        pair ← vrai
    fin si
    retourner pair
fin module
```

```
// Affiche un entier et retourne vrai s'il est pair et faux sinon.
module estPair(nb↓ : entier) → booléen
    pair : booléen
    si nb MOD 2 = 0 alors
        pair ← vrai
    sinon
        pair ← faux
    fin si
    retourner pair
fin module
```

```
// Affiche un entier et retourne vrai s'il est pair et faux sinon.
module estPair(nb↓ : entier) → booléen
    pair : booléen
    pair ← nb MOD 2 = 0
    retourner pair
fin module
```

```
// Affiche un entier et retourne vrai s'il est pair et faux sinon.
module estPair(nb↓ : entier) → booléen
    retourner nb MOD 2 = 0
fin module
```

```
// Affiche un entier et retourne vrai s'il est pair et faux sinon.
module estPair(nb↓ : entier) → booléen
    si nb MOD 2 = 0 alors
        retourner vrai
    sinon
        retourner faux
    fin si
fin module
```

```
// Affiche un entier et retourne vrai s'il est pair et faux sinon.
module estPair(nb↓ : entier) → booléen
    si nb MOD 2 = 0 alors
        retourner vrai
    fin si
    retourner faux
fin module
```

4 Échange de variables

Écrire un module *swap* qui échange le contenu de deux variables entières passées en paramètres.

5 Valeur absolue

Écrire un module qui retourne la valeur absolue d'un réel reçu en paramètre. Quel est l'avantage pour un module de retourner la réponse plutôt que de l'afficher ?

6 Maximum de 4 nombres

Écrire un module qui retourne le maximum de 4 nombres donnés en paramètre en utilisant les modules max2 et/ou max3 déjà développés dans ce chapitre.

7 Validité d'une date

Reprendre l'algorithme de validation d'une date développé au chapitre précédent et le rendre modulaire.

Chapitre 5

Les variables structurées

5.1 Le type structuré

Dans les chapitres précédents, nous avons utilisé des variables de types dits « simples » (entier, réel, booléen, caractère, chaine) ne pouvant contenir qu'une seule valeur à la fois. Cependant, certains types d'information consistent en un regroupement de plusieurs données élémentaires. Quelques exemples :

- Une **date** est composée de trois éléments (le jour, le mois, l'année). Le mois peut s'exprimer par un entier (15/10/2014) ou par une chaine (15 octobre 2014).
- Un **moment** de la journée est un triple d'entiers (heures, minutes, secondes).
- La localisation d'un **point** dans un plan nécessite la connaissance de deux coordonnées cartésiennes (l'abscisse x et l'ordonnée y) ou polaires (le rayon r et l'angle θ).
- Une **adresse** est composée de plusieurs données : un nom de rue, un numéro de maison, parfois un numéro de boite postale, un code postal, le nom de la localité, un nom ou code de pays pour une adresse à l'étranger...

Pour stocker et manipuler de tels ensembles de données, nous utiliserons des **types structurés** ou **structures** [1]. Une **structure** est donc un ensemble fini d'éléments pouvant être de types distincts. Chaque élément de cet ensemble, appelé **champ** de la structure, possède un nom unique.

Notez qu'un champ d'une structure peut lui-même être une structure. Par exemple, une **carte d'identité** inclut parmi ses informations une **date** de naissance, l'**adresse** de son propriétaire (cachée dans la puce électronique !)...

5.2 Définition d'une structure

La définition d'un type structuré adoptera le modèle suivant :

```
structure NomDeLaStructure
    nomChamp1 : type1
    nomChamp2 : type2
    ...
    nomChampN : typeN
fin structure
```

1. Nous verrons dans un chapitre ultérieur que l'approche « orientée objet » offre également une solution élégante pour manipuler des données complexes.

nomChamp1, ..., nomChampN sont les noms des différents champs de la structure, et type1, ..., typeN les types de ces champs. Ces types sont soit les types « simples » étudiés précédemment (entier, réel, booléen, caractère, chaine) soit d'autres types structurés dont la structure aura été préalablement définie.

Pour exemple, nous définissons ci-dessous trois types structurés que nous utiliserons souvent par la suite :

```
structure Date
    jour : entier
    mois : entier
    année : entier
fin structure
```

```
structure Moment
    heure : entier
    minute : entier
    seconde : entier
fin structure
```

```
structure Point
    x : réel
    y : réel
fin structure
```

5.3 Déclaration d'une variable de type structuré

Une fois un type structuré défini, la déclaration de variables de ce type est identique à celle des variables simples. Par exemple :

```
anniversaire, jourJ : Date
départ, arrivée, unMoment : Moment
a, b, centreGravité : Point
```

5.4 Utilisation des variables de type structuré

En général, pour manipuler une variable structurée ou en modifier le contenu, il faut agir au niveau de ses champs en utilisant les opérations permises selon leur type. Pour accéder à l'un des champs d'une variable structurée, il faut mentionner le nom de ce champ ainsi que celui de la variable dont il fait partie. Nous utiliserons la notation « pointée » :

```
nomVariable.nomChamp
```

Exemple d'instructions utilisant les variables déclarées au paragraphe précédent :

```
anniversaire.jour ← 15
anniversaire.mois ← 10
anniversaire.année ← 2014
arrivée.heure ← départ.heure + 2
centreGravité.x ← (a.x + b.x) / 2
```

On peut cependant, dans certains cas, utiliser une variable structurée de manière globale (c'est-à-dire d'une façon qui agit simultanément sur chacun de ses champs). Le cas le plus courant est l'affectation interne entre deux variables structurées de même type, par exemple :

```
arrivée ← départ
```

qui résume les trois instructions suivantes :

```
arrivée.heure ← départ.heure
arrivée.minute ← départ.minute
arrivée.seconde ← départ.seconde
```

Une variable structurée peut aussi être le paramètre d'un module, et un module peut également renvoyer une « valeur » de type structuré. Par exemple, l'entête d'un module renvoyant le nombre de secondes écoulées entre une heure de départ et d'arrivée serait :

```
module nbSecondesEcoulées( départ↓, arrivée↓ : Moment) → entier
```

On pourra aussi lire ou afficher une variable structurée [2].

```
lire unMoment
afficher unMoment
```

Par contre, il n'est pas autorisé d'utiliser les opérateurs de comparaison ($<$, $>$) pour comparer des variables structurées (même de même type), car une relation d'ordre n'accompagne pas toujours les structures utilisées. En effet, s'il est naturel de vouloir comparer des dates ou des moments, comment définir une relation d'ordre avec les points du plan ou avec des cartes d'identités ?

Si le besoin de comparer des variables structurées se fait sentir, il faudra dans ce cas écrire des modules de comparaison adaptés aux structures utilisées.

Par facilité d'écriture, on peut assigner tous les champs en une fois via des « {} ». Exemple :

```
anniversaire ← {1, 9, 1989}
```

5.5 Exemple d'algorithme

Le module ci-dessous reçoit en paramètre deux dates ; la valeur renvoyée est –1 si la première date est antérieure à la deuxième, 0 si les deux dates sont identiques ou 1 si la première date est postérieure à la deuxième.

```
// Affiche 2 dates en paramètres et retourne la valeur
// –1 si la première date est antérieure à la deuxième
// 0 si les deux dates sont identiques
// 1 si la première date est postérieure à la deuxième.
module comparerDates(date1↓, date2↓ : Date) → entier
    résultat : entier
    résultat ← -1                                              // valeur choisie par défaut
    si date1.année ≥ date2.année alors
        si date1.année > date2.année alors
            résultat ← 1
        sinon                                                  // les années sont identiques
            si date1.mois ≥ date2.mois alors
                si date1.mois > date2.mois alors
                    résultat ← 1
                sinon                                  // les années et les mois sont identiques
                    si date1.jour ≥ date2.jour alors
                        si date1.jour > date2.jour alors
                            résultat ← 1
                        sinon                  // les années, les mois et les jours sont identiques
                            résultat ← 0
                        fin si
                    fin si
                fin si
            fin si
        fin si
    fin si
    retourner résultat
fin module
```

2. Bien que, dans certains langages, ces opérations devront être décomposées en une lecture ou écriture de chaque champ de la structure.

5.6 Exercices sur les structures

1 Conversion moment-secondes

Écrire un module qui reçoit en paramètre un moment d'une journée et qui retourne le nombre de secondes écoulées depuis minuit jusqu'à ce moment.

2 Conversion secondes-moment

Écrire un module qui reçoit en paramètre un nombre de secondes écoulées dans une journée à partir de minuit et qui retourne le moment correspondant de la journée.

3 Temps écoulé entre 2 moments

Écrire un module qui reçoit en paramètres deux moments d'une journée et qui retourne le nombre de secondes séparant ces deux moments.

4 Milieu de 2 points

Écrire un module recevant les coordonnées de deux points distincts du plan et qui retourne les coordonnées du point situé au milieu des deux.

5 Distance entre 2 points

Écrire un module recevant les coordonnées de deux points distincts du plan et qui retourne la distance entre ces deux points.

6 Un cercle

Définir un type **Cercle** pouvant décrire de façon commode un cercle quelconque dans un espace à deux dimensions. Écrire ensuite

a) un module calculant la surface du cercle reçu en paramètre ;
b) un module recevant 2 points en paramètre et retournant le cercle dont le diamètre est le segment reliant ces 2 points ;
c) un module qui détermine si un point donné est dans un cercle ;
d) un module qui indique si 2 cercles ont une intersection.

7 Un rectangle

Définir un type **Rectangle** pouvant décrire de façon commode un rectangle dans un espace à deux dimensions et dont les côtés sont parallèles aux axes des coordonnées. Écrire ensuite

a) un module calculant le périmètre d'un rectangle reçu en paramètre ;
b) un module calculant la surface d'un rectangle reçu en paramètre ;
c) un module recevant en paramètre un rectangle R et les coordonnées d'un point P, et renvoyant **vrai** si et seulement si le point P est à l'intérieur du rectangle R ;
d) un module recevant en paramètre un rectangle R et les coordonnées d'un point P, et renvoyant **vrai** si et seulement si le point P est sur le bord du rectangle R ;
e) un module recevant en paramètre deux rectangles et renvoyant la valeur booléenne **vrai** si et seulement si ces deux rectangles ont une intersection.

8 Validation de date

Écrire un algorithme qui valide une date reçue en paramètre sous forme d'une structure.

Chapitre 6

Les boucles

Les ordinateurs révèlent tout leur potentiel dans leur capacité à répéter inlassablement les mêmes tâches. Nous voyons ici comment incorporer des boucles dans nos codes et comment les utiliser à bon escient.

Attention ! D'expérience, nous savons que ce chapitre est difficile à appréhender. Beaucoup d'entre vous perdent pied ici. Accrochez-vous et faites bien tous les exercices proposés !

6.1 La notion de travail répétitif

Si on veut faire effectuer un travail répétitif, il faut indiquer deux choses :

1. Le travail à répéter
2. La manière de continuer la répétition ou de l'arrêter.

Examinons quelques exemples pour fixer notre propos.

Exemple 1 : Pour traiter des dossiers, on dira quelque chose comme « tant qu'il reste un dossier à traiter, le traiter » ou encore « traiter un dossier puis passer au suivant jusqu'à ce qu'il n'en reste plus à traiter ».

- ▷ La tâche à répéter est : « traiter un dossier »
- ▷ On indique qu'on doit continuer s'il reste encore un dossier à traiter.

Remarque : On aurait aussi pu le formuler ainsi : « traiter un dossier et passer au suivant jusqu'à ce qu'il n'en reste plus ».

Exemple 2 : Pour calculer la cote finale de tous les étudiants, on aura quelque chose du genre « Pour tout étudiant, calculer sa cote ».

- ▷ La tâche à répéter est : « calculer la cote d'un étudiant »
- ▷ On indique qu'on doit le faire pour tous les étudiants. On doit donc commencer par le premier, passer à chaque fois au suivant et s'arrêter quand on a fini le dernier.

Exemple 3 : Pour afficher tous les nombres de 1 à 100, on aura « Pour tous les nombres de 1 à 100, afficher ce nombre ».

- ▷ La tâche à répéter est : « afficher un nombre »

▷ On indique qu'on doit le faire pour tous les nombres de 1 à 100. On doit donc commencer avec 1, passer à chaque fois au nombre suivant et s'arrêter après avoir affiché le nombre 100.

Attention ! Comprenez bien que c'est toujours la même tâche qui est exécutée mais pas avec le même effet à chaque fois. Ainsi, on traite un dossier mais à chaque fois un différent ; on affiche un nombre mais à chaque fois un différent. Nous verrons comment y arriver en logique.

6.2 Structures itératives

Chacune des structures suivantes traduit une volonté de faire exécuter de façon répétée une séquence d'instructions.

6.2.1 « tant que »

Le « tant que » est une structure qui demande à l'exécutant de répéter une tâche (une ou plusieurs instructions) tant qu'une condition donnée est vraie.

```
tant que condition faire
    séquence d'instructions à exécuter
fin tant que
```

Comme pour la structure si, la condition est une expression à valeur booléenne. Dans ce type de structure, il faut qu'il y ait dans la séquence d'instructions comprise entre tant que et fin tant que au moins une instruction qui modifie une des composantes de la condition de telle manière qu'elle puisse devenir **fausse** à un moment donné. Dans le cas contraire, la condition reste indéfiniment vraie et la boucle va tourner sans fin, c'est ce qu'on appelle une **boucle infinie**. La figure 6.1 est un ordinogramme décrivant le déroulement de cette structure. On remarquera que si la condition est fausse dès le début, la tâche n'est jamais exécutée.

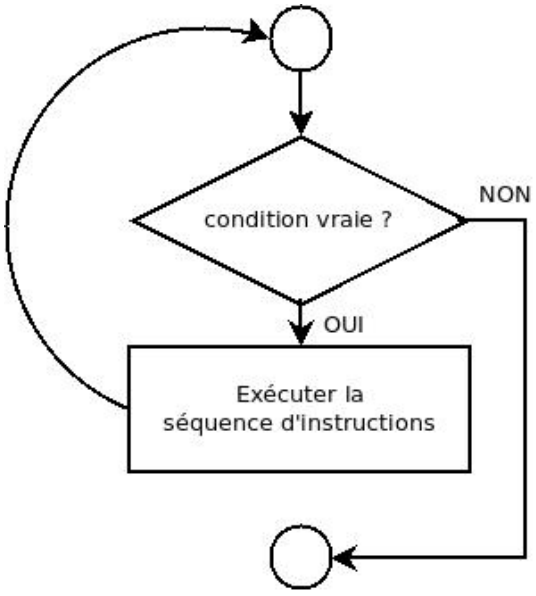


FIGURE 6.1 – Ordinogramme de la boucle "tant-que"

6.2.2 « faire – jusqu'à ce que »

Cette structure est très proche du « tant que » à deux différences près :

1. Le test est fait à la fin et pas au début. La tâche est donc toujours exécutée au moins une fois.
2. On donne la condition pour arrêter et pas pour continuer ; il s'agit d'une différence mineure.

```
faire
    séquence d'instructions à exécuter
jusqu'à ce que condition
```

Comme ci-dessus, il faut que la séquence d'instructions comprise entre faire et jusqu'à ce que contienne au moins une instruction qui modifie la condition de telle manière qu'elle puisse devenir **vraie** à un moment donné pour arrêter l'itération. La figure 6.2 décrit le déroulement de cette boucle.

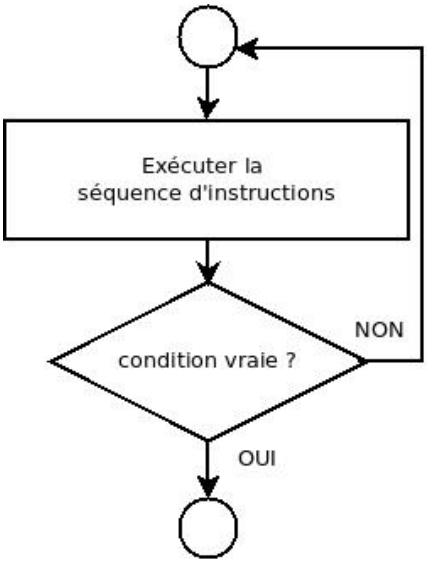


FIGURE 6.2 – Ordinogramme de la boucle "faire – jusqu'à ce que"

6.2.3 « pour »

Ici, on va plutôt indiquer **combien de fois** la tâche doit être répétée. Cela se fait au travers d'une **variable de contrôle** dont la valeur va évoluer à partir d'une valeur de départ jusqu'à une valeur finale.

```
pour variable de début à fin [par pas] faire
    séquence d'instructions à exécuter
fin pour
```

Dans ce type de structure, début, fin et pas peuvent être des constantes, des variables ou des expressions (le plus souvent à valeurs entières mais on admettra parfois des réels). Le pas est facultatif, et généralement omis (dans ce cas, sa valeur par défaut est 1). Ce pas est parfois négatif, dans le cas d'un compte à rebours, par exemple pour n de 10 à 1 par -1.

Quand le pas est positif, la boucle s'arrête lorsque la variable dépasse la valeur de fin. Par contre, avec un pas négatif, la boucle s'arrête lorsque la variable prend une valeur plus petite que la valeur de fin (cf. le test dans l'organigramme de la figure 6.3 page suivante).

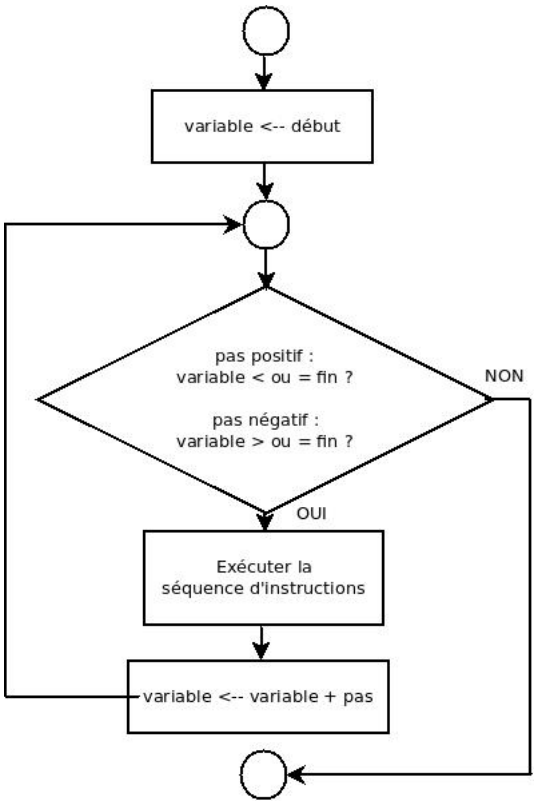


FIGURE 6.3 – Ordinogramme de la boucle "pour"

Veiller à la cohérence de l'écriture de cette structure. On considérera qu'au cas (à éviter) où début est strictement supérieur à fin et le pas est positif, la séquence d'instructions n'est jamais exécutée (mais ce n'est pas le cas dans tous les langages de programmation !). Idem si début est strictement inférieur à fin mais avec un pas négatif.

Exemples :

```
pour i de 0 à 2 faire                    // La boucle est exécutée 3 fois.
pour i de 2 à 0 faire                    // La boucle n'est pas exécutée.
pour i de 1 à 10 par -1 faire            // La boucle n'est pas exécutée.
pour i de 1 à 1 par 5 faire              // La boucle est exécutée 1 fois.
```

Veiller aussi à ne pas modifier dans la séquence d'instructions une des variables de contrôle début, fin ou pas ! Il est aussi fortement déconseillé de modifier « manuellement » la variable de contrôle au sein de la boucle pour. Il ne faut pas l'initialiser en début de boucle, et ne pas s'occuper de sa modification, l'instruction variable ← variable + pas étant automatique et implicite à chaque étape de la boucle. Il est aussi déconseillé d'utiliser variable à la sortie de la structure pour sans lui affecter une nouvelle valeur (son contenu pouvant varier selon le langage de programmation).

6.2.4 Quel type de boucle choisir ?

En pratique, il est possible d'utiliser systématiquement la boucle tant que qui peut s'adapter à toutes les situations. Cependant, il est plus clair d'utiliser la boucle pour dans les cas où le nombre d'itérations est fixé et connu à l'avance (par là, on veut dire que le nombre d'itérations est déterminé au moment où on arrive à la boucle). La boucle faire convient quant à elle dans les cas où le contenu de la boucle doit être parcouru au moins une fois,

alors que dans tant que, le nombre de parcours peut être nul si la condition initiale est fausse. La figure 6.4 propose un petit schéma récapitulatif.

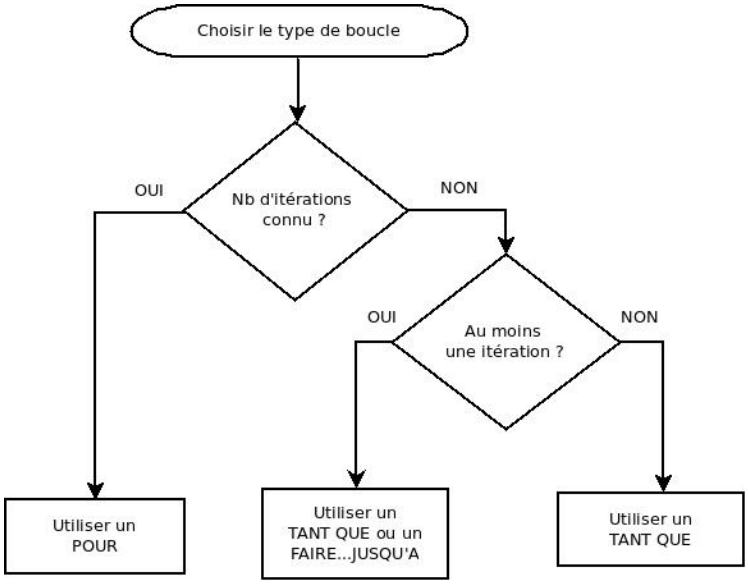


FIGURE 6.4 – Quel type de boucle choisir ?

6.3 Exemples

Nous présentons quelques exemples de difficulté croissante pour montrer comment bien utiliser les boucles.

6.3.1 Exemple 1 – Compter de 1 à 10

Imaginons qu'on veuille afficher tous les nombres de 1 à 10. On va évidemment rejeter cette première solution :

```
// Affiche les nombres de 1 à 10.
module compterJusque10()                    // une mauvaise solution !
    afficher 1
    afficher 2
    afficher 3
    afficher 4
    afficher 5
    afficher 6
    afficher 7
    afficher 8
    afficher 9
    afficher 10
fin module
```

Non seulement c'est long à écrire (imaginer l'algorithme pour afficher les nombres de 1 à 10000 !) mais c'est très peu souple. Cela ne fonctionne que pour 10 : il faut modifier l'algorithme pour un autre comptage. La boucle va nous permettre d'obtenir un algorithme qui s'adapte à la limite du décompte.

Posons-nous les bonnes questions pour déterminer la boucle :

- ▷ Quelle est la tâche à répéter ? Afficher un nombre.

- ▷ Comment savoir si on continue ? On arrête quand « 10 » est affiché.
- ▷ Comment afficher à chaque fois un nombre différent ? Au travers d'une variable qui prendra toutes les valeurs de 1 à 10. Il faut donc ajouter dans le corps de la boucle une incrémentation de la variable

Ce qui donne :

```
// Affiche les nombres de 1 à 10.
module compterJusque10()                                    // version avec tant que
    nb : entier
    nb ← 1                                                  // c'est le premier nombre à afficher
    tant que nb ≤ 10 faire                                  // tant que le nb à afficher est toujours bon
        afficher nb                                         // on affiche la valeur de la variable nb
        nb ← nb + 1                                         // on passe au nombre suivant
    fin tant que
fin module
```

Mais ici, on pourrait aussi l'écrire avec un « pour » vu qu'on connait exactement le nombre d'exécutions de la boucle (10). La variable de contrôle va évoluer de 1 à 10 ce qui tombe bien car c'est justement le nombre à afficher à chaque fois.

```
// Affiche les nombres de 1 à 10.
module compterJusque10()                                    // version avec pour
    nb : entier
    pour nb de 1 à 10 faire                                 // par défaut le pas est de 1
        afficher nb
    fin pour
fin module
```

On obtient une solution plus courte et plus lisible car l'en-tête du « pour » indique clairement comment va évoluer la boucle (valeur de départ, pas et valeur finale)

6.3.2 Exemple 2 – Compter de 1 à beaucoup

Dans l'exercice précédent, on a affirmé que la boucle pouvait s'adapter à la limite du décompte. Montrons-le ! Supposons qu'on veuille afficher les nombres de 1 à n où n est une valeur donnée par l'utilisateur.

Rien de plus simple, il suffit de lire cette valeur au début et de l'utiliser comme limite de boucle

```
// Lit un nombre et affiche les nombres de 1 à ce nombre.
module afficherN()
    nb, n : entier
    lire n
    pour nb de 1 à n faire
        afficher nb
    fin pour
fin module
```

Réflexion : Que se passe-t-il si l'utilisateur entre une valeur négative ? Comment améliorer le code pour que le programme le signale à l'utilisateur ?

6.3.3 Exemple 3 – Afficher les nombres pairs

Cela se complique un peu. Cette fois-ci on affiche uniquement les nombres **pairs** jusqu'à la limite n.

Exemple : les nombres pairs de 1 à 10 sont : 2, 4, 6, 8, 10.

Notez que n peut être impair. Si n vaut 11, l'affichage est le même que pour 10.

Est-ce qu'on peut utiliser un « pour » ? Oui. De 1 à n, il y a exactement « n DIV 2 » nombres à afficher. La difficulté vient du lien à faire entre la variable de contrôle et le nombre à afficher.

Solution 1 : on garde le lien entre la variable de contrôle et le nombre à afficher. Dans ce cas, on commence à 2 et le pas doit être de 2.

```
// Lit un nombre et affiche les nombres pairs jusqu'à ce nombre.
module afficherPair()
    nb, n : entier
    lire n                                          // limite des nombres à afficher
    pour nb de 2 à n par 2 faire
        afficher nb
    fin pour
fin module
```

Solution 2 : la variable de contrôle compte simplement le nombre d'itérations. Alors il faut calculer le nombre à afficher en fonction de la variable de contrôle (ici le double de celle-ci convient)

```
// Lit un nombre et affiche les nombres pairs jusqu'à ce nombre.
module afficherPair()
    i, n : entier
    lire n                                          // limite des nombres à afficher
    pour i de 1 à n DIV 2 faire
        afficher 2 * i
    fin pour
fin module
```

Par une vieille habitude des programmeurs [1], une variable de contrôle qui se contente de compter les passages dans la boucle est souvent nommée i. On l'appelle aussi « itérateur ».

6.3.4 Exemple 4 – Afficher les premiers nombres pairs

Voici un problème proche du précédent : on affiche cette fois les n premiers nombres pairs.

Exemple : les 10 premiers nombres pairs sont : 2, 4, 6, 8, 10, 12, 14, 16, 18, 20.

Il est plus simple de partir de la solution 2 de l'exemple précédent en changeant simplement la valeur finale de la boucle.

```
// Lit un nombre et affiche ce nombre de nombres pairs.
module afficherPair()
    i, n : entier
    lire n                                          // le nombre de nombres à afficher
    pour i de 1 à n faire
        afficher 2 * i
    fin pour
fin module
```

6.3.5 Exemple 5 – Somme de nombres

Changeons de problème. On veut pouvoir calculer (et retourner) la somme d'une série de nombres donnés par l'utilisateur. Il faut d'abord se demander comment l'utilisateur va pouvoir indiquer combien de nombres il faut additionner ou quand est-ce que le dernier nombre à additionner a été entré. Voyons quelques possibilités.

1. Née avec le langage FORTRAN où la variable i était par défaut une variable entière.

Variante 1 :

L'utilisateur indique le nombre de termes au départ. Ce problème est proche de ce qui a déjà été fait.

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
module sommeNombres() → entier                                    // Variante 1
    nbValeurs : entier                                  // nb de valeurs à additionner
    valeur : entier                                     // un des termes de l'addition
    somme : entier                                                  // la somme
    i : entier                                                      // itérateur
    somme ← 0                          // la somme se construit petit à petit. 0 au départ
    lire nbValeurs
    pour i de 1 à nbValeurs faire
        lire valeur
        somme ← somme + valeur
    fin pour
    retourner somme
fin module
```

Variante 2 :

Après chaque nombre, on demande à l'utilisateur s'il y a encore un nombre à additionner.

Ici, il faut chercher une solution différente car on ne connait pas au départ le nombre de valeurs à additionner et donc le nombre d'exécution de la boucle. On va devoir passer à un « tant que » ou un «faire - jusqu'à ce que». On peut envisager de demander en fin de boucle s'il reste encore un nombre à additionner. Ce qui donne :

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
module sommeNombres() → entier                                   // Variante 2a
    encore : booléen                 // est-ce qu'il reste encore une valeur à additionner ?
    valeur : entier                                     // un des termes de l'addition
    somme : entier                                                  // la somme
    somme ← 0
    faire
        lire valeur
        somme ← somme + valeur
        lire encore
    jusqu'à ce que NON encore
    retourner somme
fin module
```

Avec cette solution, on additionne au moins une valeur. Si on veut pouvoir tenir compte du cas très particulier où l'utilisateur ne veut additionner aucune valeur, il faut utiliser un « tant que » et donc poser la question avant d'entrer dans la boucle.

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
module sommeNombres() → entier                                   // Variante 2b
    encore : booléen                 // est-ce qu'il reste encore une valeur à additionner ?
    valeur : entier                                     // un des termes de l'addition
    somme : entier                                                  // la somme
    somme ← 0
    lire encore
    tant que encore faire
        lire valeur
        somme ← somme + valeur
        lire encore
    fin tant que
    retourner somme
fin module
```

Variante 3 :

L'utilisateur entre une valeur spéciale pour indiquer la fin. On parle de valeur **sentinelle**. Ceci n'est possible que si cette valeur **sentinelle** ne peut pas être un terme valide de l'addition. Par exemple, si on veut additionner des nombres positifs uniquement, la valeur -1 peut servir de valeur sentinelle. Mais sans limite sur les nombres à additionner (positifs, négatifs ou nuls) il n'est pas possible de choisir une sentinelle.

Ici, on se base sur la valeur entrée pour décider si on continue ou pas. Il faut donc **toujours** effectuer un test après une lecture de valeur. C'est pour cela qu'il faut effectuer une lecture avant et une autre à la fin de la boucle.

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
module sommeNombres() → entier                                    // Variante 3
    valeur : entier                                    // un des termes de l'addition
    somme : entier                                                    // la somme
    somme ← 0
    lire valeur
    tant que valeur ≥ 0 faire
        somme ← somme + valeur
        lire valeur                          // remarquer l'endroit où on lit une valeur.
    fin tant que
    retourner somme
fin module
```

Réflexions :

- ▷ Quelle valeur sentinelle prendrait-on pour additionner une série de cotes d'interrogations ? Une série de températures ?
- ▷ Dans les solutions 2 et 3 on lit une variable booléenne. Comment un programmeur pourrait-il réaliser cette instruction de façon pratique ?

6.3.6 Exemple 6 : Suite des nombres pairs

Ce chapitre propose en exercice des suites. Toutes ces suites peuvent se résoudre en suivant un même squelette d'algorithme que nous allons expliquer ici.

Un exemple simple pourrait être celui-ci : « Écrire l'algorithme qui affiche les n premiers termes de la suite : 2, 4, 6... »

Puisqu'on doit écrire plusieurs nombres et qu'on sait exactement combien, on se tournera tout naturellement vers une boucle **pour**.

Le cas le plus simple est lorsque le nombre à afficher à l'étape i peut être calculé en fonction de i seulement. L'algorithme est alors

```
pour i de 1 à n faire
    afficher f(i)
fin pour
```

Dans ce cas, la fonction est $f(i) = 2 * i$ (Si vous n'êtes pas convaincu, vérifiez qu'à l'étape 1 on affiche 2, à l'étape 2 on affiche 4...) ce qui donne

```
module nombrePair(n↓ : entier)
    i : entier
    pour i de 1 à n faire
        afficher 2 * i
    fin pour
fin module
```

Parfois, il est difficile (voire impossible) de trouver $f(i)$. On suivra alors une autre approche qui revient à calculer un nombre à afficher à partir du nombre précédemment affiché (ou, plus

exactement, de calculer le suivant à partir du nombre qu'on vient d'afficher). La structure générale est alors

```
nb ← {1ère valeur à afficher}
pour i de 1 à n faire
    afficher nb
    nb ← {calculer ici le nb suivant}
fin pour
```

Dans l'exemple de la suite paire, le 1er nombre à afficher est 2 et le nombre suivant se calcule en ajoutant 2 au nombre courant.

```
module suite1(n↓ : entier)
    nb, i : entiers
    nb ← 2
    pour i de 1 à n faire
        afficher nb
        nb ← nb + 2
    fin pour
fin module
```

On remarque que, lors du dernier passage dans la boucle, on calcule une valeur qui ne sera pas affichée. Cette petite perte de temps est dommage mais négligeable et permet de garder une structure claire et générale à la solution.

Dans certains cas, il n'est pas possible de déduire directement le nombre suivant en connaissant juste le nombre précédent. Prenons un exemple un peu plus compliqué pour l'illustrer.

6.3.7 Exemple 7 : 3 pas en avant, 2 pas en arrière

Compliquons un peu : « Écrire l'algorithme qui affiche les n premiers termes de la suite : 1, 2, 3, 4, 3, 2, 3, 4, 5, 4, 3... »

Si on vient d'écrire, disons un 3, impossible sans information supplémentaire, de connaitre le nombre suivant. Il faudrait savoir si on est en phase d'avancement ou de recul et combien de pas il reste à faire dans cette direction.

Ajoutons des variables pour retenir l'*état* où on est.

```
module suite3Avant2Arrière(n↓ : entier)
    nb, nbPasRestants, direction, i : entiers
    nb ← 1
    nbPasRestants ← 3                                          // 3 pas
    direction ← 1                                              // en avant
    pour i de 1 à n faire
        afficher nb
        nb ← nb + direction                  // faire un pas dans la bonne direction
        nbPasRestants ← nbPasRestants - 1
        si nbPasRestants = 0 alors           // il est temps de changer de direction
            direction ← -direction
            si direction = 1 alors
                nbPasRestants ← 3
            sinon
                nbPasRestants ← 2
            fin si
        fin si
    fin pour
fin module
```

On obtient un algorithme plus long mais qui respecte toujours le schéma vu.

Un conseil : essayez de respecter ce schéma et vous obtiendrez plus facilement un algorithme correct et lisible, également dans les cas particuliers.

6.4 Exercices

Voilà ! Vous devriez à présent être en mesure de résoudre les exercices qui suivent. Courage ! Et n'en passez aucun !

1 Compréhension d'algorithmes

Quels sont les affichages réalisés lors de l'exécution des algorithmes suivants ?

```
module boucle1()
    x : entier
    x ← 0
    tant que x < 12 faire
        x ← x + 2
    fin tant que
    afficher x
fin module
```

```
module boucle2()
    ok : booléen
    x : entier
    ok ← vrai
    x ← 5
    tant que ok faire
        x ← x + 7
        ok ← x MOD 11 ≠ 0
    fin tant que
    afficher x
fin module
```

```
module boucle3()
    ok : booléen
    cpt, x : entiers
    x ← 10
    cpt ← 0
    ok ← vrai
    tant que ok ET cpt < 3 faire
        si x MOD 2 = 0 alors
            x ← x + 1
            ok ← x < 20
        sinon
            x ← x + 3
            cpt ← cpt + 1
        fin si
    fin tant que
    afficher x
fin module
```

```
module boucle4()
    pair, grand : booléens
    p, x : entiers
    x ← 1
    p ← 1
    faire
        p ← 2 * p
        x ← x + p
        pair ← x MOD 2 = 0
        grand ← x > 15
    jusqu'à ce que pair OU grand
    afficher x
fin module
```

```
module boucle5()
    i, x : entiers
    ok : booléen
    x ← 3
    ok ← vrai
    pour i de 1 à 5 faire
        x ← x + i
        ok ← ok ET (x MOD 2 = 0)
    fin pour
    si ok alors
        afficher x
    sinon
        afficher 2 * x
    fin si
fin module
```

```
module boucle6()
    i, j, fin : entiers
    pour i de 1 à 3 faire
        fin ← 6 * i - 11
        pour j de 1 à fin par 3 faire
            afficher 10 * i + j
        fin pour
    fin pour
fin module
```

2 Simplification

Voici quelques extraits d'algorithmes corrects du point de vue de la syntaxe mais contenant des lignes inutiles ou des lourdeurs d'écriture. Remplacez ces portions d'algorithme par un minimum d'instructions qui auront un effet équivalent.

```
a ← 1
b ← 1
tant que a < 10 faire
    a ← a + 1
    b ← b + 1
    afficher b
fin tant que
```

```
a ← 1
b ← 1
tant que a < 10 faire
    a ← a + 1
    b ← b + 1
fin tant que
afficher b
```

```
i ← 1
pour i de 1 à 10 faire
    b ← i
fin pour
```

3 Afficher les n premiers

Écrire un algorithme qui lit un naturel n et affiche

- ▷ les n premiers entiers strictement positifs ;
- ▷ les n premiers entiers strictement positifs en ordre décroissant ;
- ▷ les n premiers carrés parfaits ;
- ▷ les n premiers naturels impairs ;
- ▷ les naturels impairs qui sont inférieurs ou égaux à n.

4 Maximum de nombres

Écrire un algorithme qui lit une série de cotes d'interrogation (entiers entre 0 et 20) et affiche ensuite la plus grande. Pour signaler la fin de l'encodage des cotes, l'utilisateur introduit un nombre négatif (**valeur sentinelle**).

5 Afficher les multiples de 3

Écrire un algorithme qui lit une série de nombres entiers positifs, jusqu'à ce que l'utilisateur encode la valeur 0. Les nombres multiples de 3 seront affichés au fur et à mesure et le nombre de ces multiples sera affiché en fin de traitement.

6 Placement d'un capital

On placera le 1er janvier de l'année à venir un capital pendant un certain nombre d'années à un certain taux. Le capital évolue suivant le système des intérêts composés. Écrire un algorithme qui, à partir du capital de départ, du nombre d'années et du taux de placement (en %), affiche pour chaque année les informations suivantes : la date, le capital et les intérêts obtenus au 1er janvier de cette année.

7 Produit de 2 nombres

Écrire un algorithme qui retourne le produit de deux entiers quelconques sans utiliser l'opérateur de multiplication, mais en minimisant le nombre d'opérations.

8 Génération de suites

Écrire un algorithme qui affiche les n **premiers termes** des suites suivantes :

a) Le pas croissant
1, 2, 4, 7, 11, 16, ...

b) La boiteuse
1, 2, 4, 5, 7, 8, 10, 11, ...

c) La suite de Fibonacci
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

d) La procession d'Echternach
1, 2, 3, 4, 3, 2, 3, 4, 5, 4, 3, 4, 5, 6, 5, 4, 5, 6, ...

e) La combinaison de deux suites
1, 2, 3, 3, 5, 4, 7, 5, 9, 6, 11, 7, 13, 8, ...

f) La capricieuse
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 40, 39, 38, ..., 31, 41, 42, ...

9 Factorielle

Écrire un algorithme qui retourne la factorielle de n (entier positif ou nul). Rappel : la factorielle de n, notée $n\,!$, est le produit des n premiers entiers strictement positifs.

Par convention, $0\,! = 1$.

10 Somme de chiffres

Écrire un algorithme qui retourne la somme des chiffres qui forment un nombre naturel n. Attention, on donne au départ **le** nombre et pas ses chiffres. Exemple : 133045 doit donner comme résultat 16, car $1 + 3 + 3 + 0 + 4 + 5 = 16$.

11 Conversion binaire-décimale

Écrire un algorithme qui lit un nombre entier positif censé représenter une configuration binaire (et donc ne contenant que des chiffres 0 et 1) et qui retourne la conversion de ce nombre en base 10. Attention, on lit **un** nombre et on affiche **un** nombre. Améliorer l'algorithme de sorte qu'il signale une éventuelle erreur d'encodage.

12 Conversion décimale-binaire

Écrire un algorithme qui lit un entier positif et retourne sa configuration binaire.

13 PGCD

Écrire un algorithme qui calcule le PGCD (plus grand commun diviseur) de deux entiers positifs selon l'algorithme d'Euclide.

14 PPCM

Écrire un algorithme qui calcule le PPCM (plus petit commun multiple) de deux entiers positifs.

15 Nombre premier

Écrire un algorithme qui vérifie si un entier positif est un **nombre premier**.

Rappel : un nombre est premier s'il n'est divisible que par 1 et par lui-même. Le premier nombre premier est 2.

16 Nombres premiers

Écrire un algorithme qui affiche les nombres premiers inférieurs ou égaux à un entier positif donné. Le module de cet algorithme fera appel de manière répétée mais économique à celui de l'exercice précédent.

17 Nombre parfait

Écrire un algorithme qui vérifie si un entier positif est un **nombre parfait**, c'est-à-dire un nombre égal à la somme de ses diviseurs (sauf lui-même).

Par exemple, 6 est parfait car $6 = 1 + 2 + 3$.

De même, 28 est parfait car $28 = 1 + 2 + 4 + 7 + 14$.

18 Décomposition en facteurs premiers

Écrire un algorithme qui affiche la décomposition d'un entier en facteurs premiers. Par exemple, 1001880 donnerait comme décomposition $2^3 * 3^2 * 5 * 11^2 * 23$.

19 Palindrome

Écrire un algorithme qui vérifie si un entier donné forme un palindrome ou non. Un nombre palindrome est un nombre qui lu dans un sens (de gauche à droite) est identique au nombre lu dans l'autre sens (de droite à gauche). Par exemple, 1047401 est un nombre palindrome.

20 Jeu de la fourchette

Écrire un algorithme qui simule le jeu de la fourchette. Ce jeu consiste à essayer de découvrir un nombre quelconque compris entre 1 et 100 inclus, tiré au sort par l'ordinateur (la primitive hasard(n : entier) retourne un entier entre 1 et n). L'utilisateur a droit à huit essais maximum. À chaque essai, l'algorithme devra afficher un message indicatif « nombre donné trop petit » ou « nombre donné trop grand ». En conclusion, soit « bravo, vous avez trouvé en [nombre] essai(s) » soit « désolé, le nombre était [valeur] ».

21 IMC

L'IMC (*Indice de Masse Corporelle*) est une mesure permettant à chacun d'entre nous de se situer sur une échelle de corpulence. On obtient l'IMC en divisant le poids en kilo par le carré de la taille en mètre.

La catégorie de corpulence en fonction de l'IMC est donnée par le tableau suivant :

Corpulence	IMC homme	IMC femme
maigre	strictement moins de 20	strictement moins de 19
normal	entre 20 et 25 inclus	entre 19 et 24 inclus
excès pondéral	entre 25 exclu et 30 inclus	entre 24 exclu et 30 inclus
obèse	plus de 30	plus de 30

Écrire un algorithme qui lit les données d'une série de personnes (sexe ('H' ou 'F'), poids et taille) et affiche à la fin le pourcentage de personnes obèses.

Remarque : Choisissez une valeur sentinelle pour indiquer la fin des données.

22 Cotes

Écrire un algorithme qui permet d'introduire, pour chaque étudiant d'un groupe de logique, les trois cotes d'interrogations (sur 20) qu'il a passées durant l'année, cotes données dans l'ordre chronologique ainsi que la cote d'examen (sur 100). La valeur –1 est introduite en cas d'absence justifiée à une interrogation. Pour une absence non justifiée, la cote de l'interrogation est de 0. En ce qui concerne l'examen, toute absence (justifiée ou non) est introduite comme un « -1 » et implique un « 0 » comme cote finale.

Pour rappel, l'examen est coté sur 100 et la cote finale se calcule comme suit : on additionne les points des trois interrogations passées (y compris les zéros des absences non justifiées) à la cote de l'examen. Dans le cas d'absences justifiées (1, 2 ou 3), la cote d'examen est ramenée respectivement sur 120, 140 ou 160 et additionnée aux interrogations passées. Dans tous les cas, le total sur 160 est divisé par 8 pour obtenir la cote finale sur 20.

L'algorithme demande après chaque étudiant s'il reste encore des notes à encoder et affiche à la fin le nombre de cotes, la meilleure cote et le pourcentage de cotes supérieures à 12.

Chapitre 7

Les chaines

Dans ce chapitre, nous allons apprendre à manipuler du texte en étudiant les différentes fonctions associées aux variables de type chaine et caractère. Ce sera aussi l'occasion de revoir quelques techniques algorithmiques déjà acquises pour les nombres (alternatives, boucles) afin de les consolider dans un contexte plus « littéraire ».

7.1 Introduction

Nous avons défini au chapitre 2 deux types de variables qui permettent de stocker du texte : le caractère (pour les différentes lettres et symboles qui apparaissent sur le clavier de votre ordinateur, par exemple 'a', 'B', ' ?', '3', '@', etc.) et la chaine (qui est un assemblage de plusieurs caractères) comme par exemple "Nous voici déjà arrivés au chapitre 7 !".

Observez la petite nuance d'écriture : un caractère sera entouré de simples guillemets (' ') et une chaine de doubles guillemets (" ").

La taille (ou longueur) d'une chaine est le nombre de caractères qu'elle contient. Remarquez qu'une chaine peut être vide, mais pas un caractère ! Ne confondez pas la chaine vide ("", de taille nulle) avec le caractère blanc (' ') qui contient l'espace séparant 2 mots d'un texte et que vous obtenez en enfonçant la touche d'espacement au bas du clavier.

La chaine (string en anglais) est un type à part entière dans la plupart des langages informatiques, bien que certains langages utilisent d'autres artifices pour simuler les chaines (par exemple le C où la chaine s'apparente plus à un tableau de caractères). Au début de l'histoire de l'informatique, les chaines étaient limitées à 255 caractères ; actuellement, la majorité des langages admettent des chaines de longueur 65535 ou plus (soit suffisament pour contenir tout un chapitre de ce syllabus). La plupart des langages orienté objet offrent une classe dédiée à la représentation et la manipulation des chaines de caractères.

La manipulation de chaines peut s'avérer être une source de complications pour le programmeur, en particulier pour le développeur de sites internet en raison des problèmes de compatibilité des différents systèmes informatiques, des différentes normes de représentations et des différents alphabets utilisés à travers le monde. Dans tout ce qui suit, nous simplifierons sensiblement cette problématique (même au niveau du français) en ne considérant pas les problèmes liés aux différentes accentuations des lettres (accents, trémas, cédilles...).

7.2 Manipuler les caractères

Introduisons quelques fonctions (ou primitives) agissant sur les caractères. Leur écriture s'apparente à celle de modules que vous pourrez utiliser tels quels dans les exercices. Nous ne détaillons pas leur code ici, car il fait intervenir des aspects techniques extérieurs au cadre du cours de logique (codage des caractères, code ASCII, normes ISO...)

estLettre(car : caractère) → booléen

Cette fonction indique si un caractère est une lettre. Par exemple elle retourne vrai pour 'a', 'e', 'G', 'K', mais faux pour '4', '$', '@'...

Si on veut savoir si une lettre est une minuscule ou majuscule, on utilisera les fonctions analogues

estMinuscule(car : caractère) → booléen

et

estMajuscule(car : caractère) → booléen

Il va de soi que si *car* n'est pas une lettre, ces deux fonctions retournent faux.

estChiffre(car : caractère) → booléen

Cette fonction permet de savoir si un caractère est un chiffre. Elle retourne vrai uniquement pour les dix caractères '0', '1', '2', '3', '4', '5', '6', '7', '8' et '9' et faux dans tous les autres cas.

On peut aussi convertir une majuscule en minuscule, grâce à la fonction :

majuscule(car : caractère) → caractère

Par exemple, si *car* vaut 'h', cette fonction retourne 'H'. L'opération inverse se fait avec :

minuscule(car : caractère) → caractère

Il peut aussi être pratique de connaitre la position d'une lettre dans l'alphabet. Ceci se fera à l'aide de la fonction :

numLettre(car : caractère) → entier

qui retourne toujours un entier entre 1 et 26. Par exemple numLettre('E') donnera 5, ainsi que numLettre('e'), cette fonction traite donc de la même manière les majuscules et les minuscules. En vertu de ce qui a été écrit plus haut, numLettre retournera aussi 5 pour les caractères 'é', 'è', 'ê', 'ë'...). N.B. : attention, il est interdit d'utiliser cette fonction si le caractère n'est pas une lettre !

Il peut être utile d'avoir un outil qui fait l'opération inverse, à savoir associer la lettre de l'alphabet correspondant à une position donnée. Pour cela, nous aurons :

lettreMaj(n : entier) → caractère

et

lettreMin(n : entier) → caractère

qui retournent respectivement la forme majuscule ou minuscule de la n-ème lettre de l'alphabet (où n sera obligatoirement compris entre 1 et 26). Par exemple, lettreMaj(13) retourne 'M' tandis que lettreMin(26) retourne 'z'.

7.3 Convertir en chaine

Un caractère peut être vu comme une chaine de taille 1, mais techniquement, il est important de faire la distinction entre ces deux types. Si on veut transformer un caractère en chaine,

on utilisera la fonction :

chaine(car : caractère) → chaine

Par exemple, si *car* vaut 'a', cette fonction retournera "a". Il est cependant admissible d'utiliser le simple signe d'affectation qui réalise la conversion de façon automatique :

varChaine ← varCaractère

De façon similaire, on pourra transformer une variable numérique en chaine avec :

chaine(n : entier) → chaine

ou

chaine(x : réel) → chaine

Ainsi, chaine(23) donnera "23" et chaine (3,14) donnera "3,14". Notez qu'on utilise le même nom de fonction dans ces derniers cas (concept de « surcharge » qui sera développé dans le chapitre orienté objet du cours « Algorithmique II »). La différence entre ces différentes fonctions se fait par le type de la variable en entrée.

On peut aussi transformer une chaine contenant des caractères numériques en nombre par la fonction inverse :

nombre(ch : chaine) → réel

Ainsi, nombre("3,14") retournera 3,14. Notez que si n est un entier, l'instruction n ← nombre("3,14") affectera n à 3, conformément à la remarque du paragraphe 2.5.2 page 20. On supposera encore que si la chaine *ch* ne représente pas un nombre de façon correcte, la fonction retournera 0.

7.4 Manipuler les chaines

Venons-en maintenant aux différentes fonctions qui permettent de manipuler les chaines ; ces fonctions ont des formes équivalentes dans la plupart des langages de programmation. Nous adapterons pour le pseudo-code les fonctions les plus courantes et les plus utiles.

7.4.1 Longueur d'une chaine

La longueur d'une chaine (ou encore sa taille) est le nombre de caractères qu'elle contient. Nous pourrons l'évaluer avec la fonction

longueur(ch : chaine) → entier

Par exemple : longueur("algorithmique") retourne 13 et longueur("") retourne 0.

7.4.2 Accès aux différents caractères d'une chaine

La fonction

car(ch : chaine, n : entier) → caractère

permet de connaitre le n-ème caractère contenu dans une chaine. Pour une utilisation correcte de cette fonction, il faut que la valeur de n soit au moins égale à 1 et qu'elle ne dépasse pas la longueur de la chaine. Par exemple car("Spirou", 4) retourne 'r', mais car("Spirou", 8) générera un message d'erreur.

7.4.3 Extraction de sous-chaine

Cette fonction importante permet d'extraire une portion d'une certaine longueur d'une chaine donnée, et ceci à partir d'une position donnée. L'en-tête de cette fonction (qui reçoit donc 3 paramètres entrants) est la suivante :

sousChaine(ch : chaine, pos : entier, long : entier) → chaine

Il faudra aussi être très vigilant pour une utilisation correcte : pos doit être compris entre 1 et la taille de la chaine, et la valeur de long doit être telle qu'on ne déborde pas de la chaine !

Par exemple, sousChaine("algorithmique", 5, 3) donne "rit".

Cette fonction est très utile pour sélectionner des portions d'une chaine contenant des informations codées sous un certain format. Prenons par exemple une date stockée dans une chaine *ch* de format "JJ/MM/AAAA" (de longueur 10). Pour extraire le jour de cette date, on écrira sousChaine(ch, 1, 2) ; le mois s'obtiendra avec sousChaine(ch, 4, 2) et l'année avec sousChaine(ch, 7, 4). Attention, les 3 résultats obtenus sont des chaines de chiffres et non des nombres !

7.4.4 Recherche de sous-chaine

Cette fonction permet de savoir si une sous-chaine donnée (ou un caractère) est présent dans une chaine donnée. Elle permet d'éviter d'écrire le code correspondant à une recherche :

estDansChaine(ch : chaine, sous-chaine : chaine [ou caractère]) → entier

La valeur de l'entier renvoyé est la position où commence la sous-chaine recherchée. Par exemple, estDansChaine("algorithmique", "mi") retournera 9. Si la sous-chaine ne s'y trouve pas, la fonction retourne 0. On peut admettre ici d'écrire un caractère à la place de la sous-chaine. Par exemple, estDansChaine("algorithmique", 'm') retournera également 9.

7.4.5 Concaténation de chaines

Il est fréquent de devoir rassembler plusieurs chaines pour former une seule chaine plus grande, il s'agit de l'opération de concaténation. La syntaxe est la suivante :

concat(ch1, ch2, ..., chN : chaine) → chaine

On remarquera ici le cas exceptionnel d'une fonction admettant un nombre indéfini de paramètres. Par exemple, concat("al", "go", "rithmique") donnera "algorithmique". On admettra aussi comme notation alternative l'utilisation du signe « + » qui est ici sans équivoque entre des variables non numériques :

ch ← ch1 + ch2 + ...+ chN

7.5 Exercices

1 Calcul de fraction

Écrire un algorithme qui reçoit une fraction sous forme de chaine, et retourne la valeur numérique de celle-ci. Par exemple, si la fraction donnée est "5/8", l'algorithme renverra 0,625. On peut considérer que la fraction donnée est correcte, elle est composée de 2 entiers séparés par le caractère de division '/'.

2 Conversion de nom

Écrire un algorithme qui reçoit le nom complet d'une personne dans une chaine sous la forme "nom, prénom" et la renvoie au format "prénom nom" (sans virgule séparatrice). Exemple : "De Groote, Jan" deviendra "Jan De Groote".

3 Gauche et droite

Écrire un module gauche et un droite qui retourne la chaine formée respectivement des n premiers et des n derniers caractères d'une chaine donnée.

4 Grammaire

Écrire un algorithme qui met un mot en « ou » au pluriel. Pour rappel, un mot en « ou » prend un « s » à l'exception des 7 mots bijou, caillou, chou, genou, hibou, joujou et pou qui prennent un « x » au pluriel. Exemple : un clou, des clous, un hibou, des hiboux. Si le mot soumis à l'algorithme n'est pas un mot en « ou », un message adéquat sera affiché.

5 Le code correct

Le code IBAN de votre compte en banque est composé de 16 caractères alphanumériques (2 lettres suivies de 14 chiffres). Pour la lisibilité, on l'écrit souvent en 4 paquets de 4 caractères séparés par des blancs, par ex. : "BE89 6352 4390 0285" (soit une chaine de 19 caractères au total). On peut vérifier si un compte est valide par la manipulation suivante : le reste de la division par 97 du nombre formé par les 10 chiffres entre le 5ème et 14ème caractères du code doit donner le nombre formé par les 2 derniers chiffres. On peut voir directement que, pour l'exemple donné, 896352439002 MOD 97 est bien égal à 85.

Écrire l'algorithme qui reçoit un code IBAN sous la forme d'une chaine de 19 caractères et retourne un booléen indiquant si ce code est correct.

6 Email valide

Écrire un algorithme qui vérifie si une chaine représentant une adresse email est valide. Pour simplifier, nous dirons qu'une adresse email valide est une chaine de la forme `ch1@ch2.ch3`, où les sous-chaines ch1 et ch2 ne peuvent contenir que des caractères alpha-numériques, et ch3 ne contient que 2 ou 3 lettres. (Dans la réalité, c'est un peu plus complexe, ch1 pouvant contenir d'autres caractères tels que points, tirets, tirets bas (underscore)...).

7 Les palindromes

Cet exercice a déjà été réalisé pour des entiers, en voici 2 versions « chaine » !

a) Écrire un algorithme qui vérifie si un mot donné sous forme de chaine constitue un palindrome (comme par exemple "kayak", "radar" ou "saippuakivikauppias" (marchand de savon en finnois)

b) Écrire un algorithme qui vérifie si une phrase donnée sous forme de chaine constitue un palindrome (comme par exemple "Esope reste ici et se repose" ou "Tu l'as trop écrasé, César, ce Port-Salut !"). Dans cette seconde version, on fait abstraction des majuscules/minuscules et on néglige les espaces et tout signe de ponctuation.

8 Remplacement de sous-chaines

Écrire un algorithme qui remplace dans une chaine donnée toutes les sous-chaines ch1 par la sous-chaine ch2. Attention, cet exercice est plus coriace qu'il n'y parait à première vue... Assurez-vous que votre code n'engendre pas de boucle infinie.

9 Conversion nombres binaires et décimaux

Réécrire les exercices de conversion de nombres binaires et décimaux du chapitre sur les boucles en considérant que les nombres entrés et affichés sont des chaines. D'après-vous, en quoi cette transformation est-elle une amélioration de la version purement numérique ?

10 Normaliser une chaine

Écrire un module qui reçoit une chaine et retourne une autre chaine, version normalisée de la première. Par normalisée, on entend : enlever tout ce qui n'est pas une lettre et tout mettre en majuscule.
Exemple : "Le <COBOL>, c'est la santé !" devient "LECOBOLCESTLASANTE".

11 Le chiffre de César

Depuis l'antiquité, les hommes politiques, les militaires, les hommes d'affaires cherchent à garder secret les messages importants qu'ils doivent envoyer. L'empereur César utilisait une technique (on dit un *chiffrement*) qui porte à présent son nom : remplacer chaque lettre du message par la lettre qui se situe k position plus loin dans l'alphabet (cycliquement).

Exemple : si k vaut 2, alors le texte clair "CESAR" devient "EGUCT" lorsqu'il est chiffré et le texte "ZUT" devient "BWV".

Bien sûr, il faut que l'expéditeur du message et le récepteur se soient mis d'accord sur la valeur de k.

On vous demande d'écrire un module qui reçoit une chaine ne contenant que des lettres majuscules ainsi que la valeur de k et qui retourne la version chiffrée du message.

On vous demande également d'écrire le module de déchiffrement. Ce module reçoit un message chiffré et la valeur de k qui a été utilisée pour le chiffrer et retourne le message en clair. Attention ! Ce second module est **très simple**.

Chapitre 8

Les tableaux

Dans ce chapitre nous étudions les tableaux, une structure qui peut contenir plusieurs exemplaires de données similaires.

8.1 Utilité des tableaux

Nous allons introduire la notion de tableau à partir d'un exemple dans lequel l'utilité de cette structure de données apparaitra de façon naturelle.

Exemple : Statistiques de vente

Un gérant d'une entreprise commerciale souhaite connaitre l'impact d'une journée de promotion publicitaire sur la vente de dix de ses produits. Pour ce faire, les numéros de ces produits (numérotés de 1 à 10 pour simplifier) ainsi que les quantités vendues pendant cette journée de promotion sont encodés au fur et à mesure de leurs ventes. En fin de journée, le vendeur entrera la valeur 0 pour signaler la fin de l'introduction des données. Ensuite, les statistiques des ventes seront affichées.

La démarche générale se décompose en trois parties :

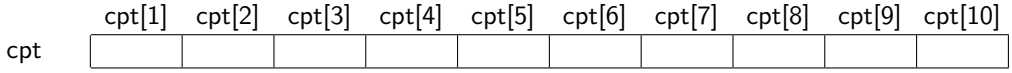
- le traitement de début de journée, qui consiste essentiellement à mettre les compteurs des quantités vendues pour chaque produit à 0
- le traitement itératif durant toute la journée : au fur et à mesure des ventes, il convient de les enregistrer, c'est-à-dire d'ajouter au compteur des ventes d'un produit la quantité vendue de ce produit ; ce traitement itératif s'interrompra lorsque la valeur 0 sera introduite
- le traitement final, consistant à communiquer les valeurs des compteurs pour chaque produit.

Vous trouverez sur la page suivante une version possible de cet algorithme.

```
// Calcule et affiche la quantité vendue de 10 produits.
module statistiquesVentesSansTableau()

    cpt1, cpt2, cpt3, cpt4, cpt5, cpt6, cpt7, cpt8, cpt9, cpt10 : entiers
    numéroProduit, quantité : entiers

    cpt1 ← 0
    cpt2 ← 0
    cpt3 ← 0
    cpt4 ← 0
    cpt5 ← 0
    cpt6 ← 0
    cpt7 ← 0
    cpt8 ← 0
    cpt9 ← 0
    cpt10 ← 0

    afficher "Introduisez le numéro du produit :"
    lire numéroProduit

    tant que numéroProduit > 0 faire

        afficher "Introduisez la quantité vendue :"
        lire quantité

        selon que numéroProduit vaut
            1 : cpt1 ← cpt1 + quantité
            2 : cpt2 ← cpt2 + quantité
            3 : cpt3 ← cpt3 + quantité
            4 : cpt4 ← cpt4 + quantité
            5 : cpt5 ← cpt5 + quantité
            6 : cpt6 ← cpt6 + quantité
            7 : cpt7 ← cpt7 + quantité
            8 : cpt8 ← cpt8 + quantité
            9 : cpt9 ← cpt9 + quantité
            10 :cpt10 ← cpt10 + quantité
        fin selon que

        afficher "Introduisez le numéro du produit :"
        lire numéroProduit

    fin tant que

    afficher "quantité vendue de produit 1 :", cpt1
    afficher "quantité vendue de produit 2 :", cpt2
    afficher "quantité vendue de produit 3 :", cpt3
    afficher "quantité vendue de produit 4 :", cpt4
    afficher "quantité vendue de produit 5 :", cpt5
    afficher "quantité vendue de produit 6 :", cpt6
    afficher "quantité vendue de produit 7 :", cpt7
    afficher "quantité vendue de produit 8 :", cpt8
    afficher "quantité vendue de produit 9 :", cpt9
    afficher "quantité vendue de produit 10 :", cpt10

fin module
```

Il est clair, à la lecture de cet algorithme, qu'une simplification d'écriture s'impose ! Et que ce passerait-il si le nombre de produits à traiter était de 20 ou 100 ? Le but de l'informatique étant de dispenser l'humain des tâches répétitives, le programmeur peut en espérer autant de la part d'un langage de programmation ! La solution est apportée par un nouveau type de variables : les **variables indicées** ou **tableaux**.

Au lieu d'avoir à manier dix compteurs distincts (cpt1, cpt2, etc.), nous allons envisager une seule « grande » variable cpt compartimentée en dix « sous-variables » qui se distingueront les unes des autres par un indice : cpt1 deviendrait ainsi cpt[1], cpt2 deviendrait cpt[2], et ainsi de suite jusqu'à cpt10 qui deviendrait cpt[10].



Un des intérêts de cette notation est la possibilité de faire apparaitre une variable entre les crochets, par exemple cpt[i], ce qui permet une grande économie de lignes de code.

Voici la version avec tableau.

```
// Calcule et affiche la quantité vendue de 10 produits.
module statistiquesVentesAvecTableau()

    cpt : tableau [1 à 10] d'entiers
    i, numéroProduit, quantité : entiers

    pour i de 1 à 10 faire
        cpt[i] ← 0
    fin pour

    afficher "Introduisez le numéro du produit :"
    lire numéroProduit

    tant que numéroProduit > 0 faire

        afficher "Introduisez la quantité vendue :"
        lire quantité

        cpt[numéroProduit] ← cpt[numéroProduit] + quantité

        afficher "Introduisez le numéro du produit :"
        lire numéroProduit

    fin tant que

    pour i de 1 à 10 faire
        afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour

fin module
```

8.2 Définitions

Un **tableau** est une suite d'éléments de même type portant tous le même nom mais se distinguant les uns des autres par un indice.

L'**indice** est un entier donnant la position d'un élément dans la suite. Cet indice varie entre la position du premier élément et la position du dernier élément, ces positions correspondant aux bornes de l'indice. Notons qu'il n'y a pas de « trou » : tous les éléments existent entre le premier et le dernier indice.

La **taille** d'un tableau est le nombre (strictement positif) de ses éléments. Attention ! la taille d'un tableau ne peut pas être modifiée pendant son utilisation.

Souvent on utilise un tableau plus grand que le nombre utile de ses éléments. Seule une partie du tableau est utilisée. On parle alors de taille **physique** (la taille maximale du tableau) et de taille **logique** (le nombre d'éléments effectivement utilisés)

8.3 Notations

Pour déclarer un tableau, on écrit :

```
nomTableau : tableau [borneMin à borneMax] de TypeElément
```

où TypeElément est le type des éléments que l'on trouvera dans le tableau. Les éléments sont d'un des types élémentaires vus précédemment (entier, réel, booléen, chaine, caractère) ou encore des variables structurées. À ce propos, remarquons aussi qu'un tableau peut être un champ d'une structure. D'autres possibilités apparaitront lors de l'étude de l'orienté objet.

Les bornes apparaissant dans la déclaration sont des constantes ou des paramètres ayant une valeur connue lors de la déclaration. Une fois un tableau déclaré, seuls les éléments d'indice compris entre borneMin et borneMax peuvent être utilisés. Par exemple, si on déclare :

```
tabEntiers : tableau [1 à 100] d'entiers
```

Il est interdit d'utiliser tabEntiers[0] ou tabEntiers[101]. De plus, chaque élément tabEntiers[i] (avec $1 \leq i \leq 100$) doit être manié avec la même précaution qu'une variable simple, c'est-à-dire qu'on ne peut utiliser un élément du tableau qui n'aurait pas été préalablement affecté ou initialisé.

N.B. : Il n'est pas interdit de prendre 0 pour la borne inférieure ou même d'utiliser des bornes négatives (par exemple : tabTempératures : **tableau** [-20 à 50] de réels).

En Java, un tableau est défini par sa taille n et les bornes sont automatiquement 0 et $n-1$. Ce n'est pas le cas en Logique où on a plus de liberté dans le choix des bornes.

8.4 Tableau statique vs tableau dynamique

Les tableaux étudiés jusqu'ici sont dit **statiques**. Un tableau est dit **statique** si sa taille est connue lors de l'écriture du programme

Un tableau est dit **dynamique** si sa taille n'est connue qu'à l'exécution du programme (elle est calculée, donnée par l'utilisateur, lue dans un fichier de configuration, fournie par une autre partie du programme, ...)

Exemples

- ▷ Reprenons l'exemple ci-dessus. S'il y a toujours exactement 10 produits, alors on peut utiliser un tableau statique.
- ▷ Dans la pratique, il est probable que ce nombre puisse évoluer au cours de l'histoire de l'entreprise. On peut au moins demander au gérant le nombre d'articles dans son stock. On utilisera alors un tableau dynamique.
- ▷ Si on désire stocker le chiffre d'affaire d'une entreprise pour une année donnée, mois par mois, un tableau de taille 12 est suffisant (tableau statique)

Concrètement, comment choisir ? Le choix dépend du langage.

Certains langages (c'est le cas de Cobol) ne permettent que des tableaux statiques. Dans ce cas, il faudra souvent imposer une **taille maximale** au tableau. Il sera également nécessaire d'utiliser une variable pour retenir la taille utile (ou logique) du tableau, à savoir la partie du tableau réellement utilisée. Bien sûr cette solution entraine souvent une perte de place mémoire due à une réservation inutilement grande (ou, pire, une saturation du tableau qui n'aura pas été prévu assez grand).

D'autres, comme Java, n'ont que des tableaux dynamiques. La taille ne doit pas forcément être connue à la compilation, mais doit être connue à l'exécution. Vous verrez les détails dans votre cours de Java.

Exemple statique

Reprenons encore l'exemple de la vente de produits. Si on ne dispose pas de tableau dynamique, on peut réserver un tableau de taille 1000 par exemple (si on est sûr qu'on ne vendra pas plus de 1000 produits différents) et ajouter une variable entière (nbProduits) pour retenir le nombre exact de produits différents actuellement en vente.

```
// Calcule et affiche la quantité vendue de x produits.
module statistiquesVentes()

    cpt : tableau [1 à 1000] d'entiers
    i, numéroProduit, quantité : entiers
    nbArticles : entier
    lire nbArticles

    pour i de 1 à nbArticles faire
        cpt[i] ← 0
    fin pour

    afficher "Introduisez le numéro du produit :"
    lire numéroProduit

    tant que numéroProduit > 0 ET numéroProduit <= nbArticles faire

        afficher "Introduisez la quantité vendue :"
        lire quantité

        cpt[numéroProduit] ← cpt[numéroProduit] + quantité

        afficher "Introduisez le numéro du produit :"
        lire numéroProduit

    fin tant que

    pour i de 1 à nbArticles faire
        afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour

fin module
```

Exemple dynamique

Pour déclarer un tableau dynamique, on sépare la déclaration du tableau (où la taille n'est pas donnée) de sa création effective (où on donne la taille)

```
nomTableau : tableau de TypeElément
// Le code peut déterminer ici les bornes
nomTableau ← nouveau tableau [ borneMin à borneMax ] de TypeElément
```

où borneMin et borneMax sont des expressions entières quelconques.

```
// Calcule et affiche la quantité vendue de x produits.
module statistiquesVentesAvecTableau()

    cpt : tableau d'entiers
    i, numéroProduit, quantité : entiers
    nbArticles : entier
    lire nbArticles
    cpt ← nouveau tableau [1 à nbArticles] d'entiers

    pour i de 1 à nbArticles faire
        cpt[i] ← 0
    fin pour

    afficher "Introduisez le numéro du produit :"
    lire numéroProduit

    tant que numéroProduit > 0 ET numéroProduit <= nbArticles faire

        afficher "Introduisez la quantité vendue :"
        lire quantité

        cpt[numéroProduit] ← cpt[numéroProduit] + quantité

        afficher "Introduisez le numéro du produit :"
        lire numéroProduit

    fin tant que

    pour i de 1 à nbArticles faire
        afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour

fin module
```

8.5 Tableau et paramètres

Un tableau peut être passé en paramètre à un module mais qu'en est-il de sa taille ? Il serait utile de pouvoir appeler le même module avec des tableaux de tailles différentes. Pour permettre cela, la taille du tableau reçu en paramètre est déclarée avec une variable (qui peut être considéré comme un paramètre entrant).

Exemples :

```
module brol(tab↓ : tableau [1 à n] d'entiers)
    afficher "J'ai reçu un tableau de ", n, " éléments".
fin module
```

Ce n va prendre la taille précise du tableau (statique ou dynamique) utilisé à chaque appel et peut être utilisé dans le corps du module. Bien sûr il s'agit là de la taille physique du tableau. Si une partie seulement du tableau doit être traitée, il convient de passer également la taille logique en paramètre.

```
module brol(tab↓ : tableau [1 à n] d'entiers, tailleLogique : entier)
    afficher "J'ai reçu un tableau rempli de ", tailleLogique, " éléments "
    afficher "sur ", n, " éléments au total."
fin module
```

Notons qu'on admet également qu'un module retourne un tableau.

```
// Crée un tableau statique d'entiers de taille 10, l'initialise à 0 et le retourne.
module créerTableau() → tableau [1 à 10] d'entiers
    tab : tableau [1 à 10] d'entiers
    i : entier
    pour i de 1 à 10 faire
        tab[i] ← 0
    fin pour
    retourner tab
fin module

module principalAppelTableau()
    entiers : tableau [1 à 10] d'entiers
    i : entier

    entiers ← créerTableau()

    pour i de 1 à 10 faire
        afficher entiers[i]
    fin pour
fin module
```

Par contre, on ne peut pas lire ou afficher un tableau en une seule instruction ; il faut des instructions de lecture ou d'affichage individuelles pour chacun de ses éléments.

8.6 Parcours d'un tableau à une dimension

Soit le tableau statique *tab* déclaré ainsi

```
tab : tableau [1 à n] de T                    // où T est un type quelconque
```

ou soit le tableau dynamique *tab* déclaré ainsi

```
tab : tableau de T                            // où T est un type quelconque
tab ← nouveau tableau [1 à n] de T
```

Envisageons d'abord le parcours complet et voyons ensuite les parcours avec arrêt prématuré.

8.6.1 Parcours complet

Les tableaux interviennent dans de nombreux problèmes et il est primordial de savoir les parcourir en utilisant des algorithmes corrects, efficaces et lisibles.

Examinons les situations courantes et voyons quelles solutions conviennent. On pourrait aussi envisager d'autres parcours (à l'envers, une case sur deux, ...) mais ils ne représentent aucune difficulté nouvelle.

Pour parcourir complètement un tableau, on peut utiliser la boucle **pour** comme dans l'algorithme suivant où « traiter » va dépendre du problème concret posé : afficher, modifier, sommer, ...

```
// Parcours complet d'un tableau via une boucle pour
// Les déclarations sont omises pour ne pas alourdir les algorithmes.
pour i de 1 à n faire
    traiter tab[i]
fin pour
```

8.6.2 Parcours avec sortie prématurée

Parfois, on ne doit pas forcément parcourir le tableau jusqu'au bout mais on pourra s'arrêter prématurément si une certaine condition est remplie. Par exemple :

- ▷ on cherche la présence d'un élément et on vient de le trouver ;
- ▷ on vérifie qu'il n'y a pas de 0 et on vient d'en trouver un.

La première étape est de transformer le **pour** en **tant que** ce qui donne l'algorithme

```
// Parcours complet d'un tableau via une boucle tant-que
i ← 1
tant que i ≤ n faire
    traiter tab[i]
    i ← i + 1
fin tant que
```

On peut à présent introduire le test d'arrêt. Une contrainte est qu'on voudra, à la fin de la boucle, savoir si oui ou non on s'est arrêté prématurément et, si c'est le cas, à quel indice.

Il existe essentiellement deux solutions, avec ou sans variable booléenne. En général, la solution [A] sera plus claire si le test est court.

[A] Sans variable booléenne

```
// Parcours partiel d'un tableau sans variable booléenne
i ← 1
tant que i ≤ n ET test sur tab[i] dit que on continue faire
    i ← i + 1
fin tant que
si i > n alors
    // on est arrivé au bout
sinon
    // arrêt prématuré à l'indice i.
fin si
```

On pourrait inverser les deux branches du **si-sinon** en inversant le test mais attention à ne pas tester tab[i] car i n'est peut-être pas valide.

[B] Avec variable booléenne

```
// Parcours partiel d'un tableau avec variable booléenne
i ← 1
trouvé ← faux
tant que i ≤ n ET NON trouvé faire
    si test sur tab[i] dit que on a trouvé alors
        trouvé ← vrai
    sinon
        i ← i + 1
    fin si
fin tant que
// tester le booléen pour savoir si arrêt prématuré.
```

Attention à bien choisir un nom de booléen adapté au problème et à l'initialiser à la bonne valeur. Par exemple, si la variable s'appelle « continue »

- ▷ initialiser la variable à vrai ;
- ▷ le test de la boucle est « . . .ET continue » ;
- ▷ mettre la variable à faux pour sortir de la boucle.

8.7 Exercices

8.7.1 Exercices de base

1 Somme

Écrire un module qui reçoit en paramètre le tableau tabEnt de n entiers et qui retourne la somme de ses éléments.

2 Maximum/minimum

Écrire un module qui reçoit en paramètre le tableau tabEnt de n entiers et qui retourne la plus grande valeur de ce tableau. Idem pour le minimum.

3 Indice du maximum/minimum

Écrire un module qui reçoit en paramètre le tableau tabEnt de n entiers et qui retourne l'indice de l'élément contenant la plus grande valeur de ce tableau. En cas d'ex-æquo, c'est l'indice le plus petit qui sera renvoyé.

Que faut-il changer pour renvoyer l'indice le plus grand ? Et pour retourner l'indice du minimum ? Réécrire l'algorithme de l'exercice précédent en utilisant celui-ci.

4 Nombre d'éléments d'un tableau

Écrire un module qui reçoit en paramètre le tableau tabRéels de n réels et qui retourne le nombre d'éléments du tableau.

5 Y a-t-il un pilote dans l'avion

Écrire un module qui reçoit en paramètre le tableau avion de n chaines et qui retourne un booléen indiquant s'il contient au moins un élément de valeur «pilote».

6 Plus grand écart absolu

Écrire un module qui reçoit en paramètre le tableau températures de n réels et qui retourne le plus grand écart absolu entre deux températures consécutives de ce tableau. Et si on veut le plus petit écart ?

7 Remplacement de valeurs

Écrire un module qui reçoit en paramètre le tableau prénoms de n chaines et qui contrôle si tous les prénoms commencent par une majuscule. Dans la négative, il remplacera la première lettre du prénom par une majuscule.

Rappel : pour manipuler les chaines et les caractères, vous avez à votre disposition les modules fournis en annexe.

8 Tableau ordonné ?

Écrire un module qui reçoit en paramètre le tableau valeurs de n entiers et qui vérifie si ce tableau est ordonné (strictement) croissant sur les valeurs. Le module retournera **vrai** si le tableau est ordonné, **faux** sinon.

9 Positions du minimum

Écrire un module qui reçoit en paramètre le tableau cotes de n entiers et qui affiche le ou les indice(s) des éléments contenant la valeur minimale du tableau.

a) Écrire une première version « classique » avec deux parcours de tableau

b) Écrire une deuxième version qui ne parcourt qu'une seule fois cotes en stockant dans un deuxième tableau (de quelle taille ?) les indices du plus petit élément rencontrés (ce tableau étant à chaque fois réinitialisé lorsqu'un nouveau minimum est rencontré)

c) Écrire une troisième version qui **retourne** le tableau contenant les indices. Écrire également un module qui appelle cette version puis affiche les indices reçus.

10 Renverser un tableau

Écrire un module qui reçoit en paramètre le tableau tabCar de n caractères, et qui « renverse » ce tableau, c'est-à-dire qui permute le premier élément avec le dernier, le deuxième élément avec l'avant-dernier et ainsi de suite.

11 Tableau symétrique ?

Écrire un module qui reçoit en paramètre le tableau tabChaines de n chaines et qui vérifie si ce tableau est symétrique, c'est-à-dire si le premier élément est identique au dernier, le deuxième à l'avant-dernier et ainsi de suite.

12 Cumul des ventes

Soit le tableau ventes : **tableau** [1 à 12] d'entiers où le premier élément contient le montant total des ventes pour janvier, le second pour février, et ainsi de suite jusqu'à décembre. Écrire l'algorithme qui reçoit ce tableau en paramètre, et renvoie le tableau cumul : **tableau**[1 à 12] ; chaque élément de ce tableau devra contenir le cumul de toutes les ventes depuis le début de l'année jusqu'au mois correspondant à l'indice de l'élément en question. Le dernier élément de cumul devra donc contenir le total des ventes de l'année.

13 Occurrence des chiffres

Écrire un module qui reçoit un nombre entier positif ou nul en paramètre et qui affiche pour chacun de ses chiffres le nombre de fois qu'il apparait dans ce nombre. Ainsi, pour le nombre 10502851125, l'affichage mentionnera que le chiffre 0 apparait 2 fois, 1 apparait 3 fois, 2 apparait 2 fois, 5 apparait 3 fois et 8 apparait une fois (l'affichage ne mentionnera donc pas les chiffres qui n'apparaissent pas).

14 Palindrome

Soit le tableau phrase : **tableau**[1 à n] de caractères (caractère alphabétique, le caractère d'espacement ou un caractère de ponctuation). Dans ce tableau, des lettres qui se suivent constituent un mot. Ces mots sont séparés les uns des autres par un ou plusieurs caractères d'espacement ou de ponctuation. Écrire un module qui reçoit en paramètre ce tableau et qui vérifie si la phrase formée par les mots de ce tableau est un palindrome. Le résultat sera communiqué par le renvoi d'une valeur booléenne. Pour rappel, une phrase palindrome est une phrase qui peut se lire dans les deux sens sans tenir compte des espacements et de la ponctuation.

Exemple du contenu du tableau phrase :

A		L	'	E	T	A	P	E	,			E	P	A	T	E	-	L	A	!

15 Moyenne d'éléments

Soit le tableau tabEnt contenant n entiers **différents**. Écrire l'algorithme qui calcule et affiche la moyenne des éléments situés entre les valeurs minimale et maximale du tableau (ces deux valeurs participant au calcul de cette moyenne). Dans l'exemple ci-dessous, le maximum est 85, le minimum 5 et la moyenne des 8 éléments concernés est 21. Exploitez l'exercice 3 page 83 pour réaliser cet algorithme.

12	85	21	17	8	6	10	16	5	74	64	29	41	11	73	72	28	66	55	44

16 OXO

Soit le tableau oxo contenant n caractères. Chaque élément est le caractère 'O' ou 'X'. Écrire l'algorithme qui permet de compter et d'afficher le nombre de séquences distinctes des valeurs consécutives 'O', 'X', 'O'. Lorsqu'un 'O' fait partie de deux séquences, on ne comptabilise qu'une seule séquence (ainsi le 'O' en position 6 ci-dessous est le dernier d'une séquence et le premier d'une autre : on ne comptabilisera qu'une seule séquence pour ces deux séquences qui se chevauchent). Dans l'exemple ci-dessous, il y a donc 3 séquences à comptabiliser :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
O	X	X	O	X	O	X	O	X	O	O	O	O	X	X	O	X	O	O	X

17 Les doublons

Vérifiez si un tableau contient au moins 2 éléments égaux.

18 Mastermind

Dans le jeu du Mastermind, un joueur A doit trouver une combinaison de k pions de couleur, choisie et tenue secrète par un autre joueur B. Cette combinaison peut contenir éventuellement des pions de même couleur. À chaque proposition du joueur A, le joueur B indique le nombre de pions de la proposition qui sont corrects et bien placés et le nombre de pions corrects mais mal placés.

Pour implémenter une simulation de ce jeu, on utilise le type Couleur, dont les valeurs possibles sont les couleurs des pions utilisés. (Attention, le nombre exact de couleurs n'est pas précisé.) Les seules manipulations permises avec ce type sont la comparaison (tester si deux couleurs sont identiques ou non) et l'affectation (affecter le contenu d'une variable de type Couleur à une autre variable de ce type). Les propositions du joueur A, ainsi que la combinaison secrète du joueur B sont contenues dans des tableaux de k composantes de type Couleur.

Écrire le module suivant qui renvoie dans les variables bienPlacés et malPlacés respectivement le nombre de pions bien placés et mal placés dans la « proposition » du joueur A en la comparant à la « solution » cachée du joueur B.

```
module testerProposition( proposition↓, solution↓ : tableau [1 à k ] de Couleur,
                                                    bienPlacés↑, malPlacés↑ : entiers)
```

19 Casser le chiffre de César

Dans le chapitre sur les chaines, l'exercice 11 page 74 vous a montré la technique de César pour chiffrer ses messages.

Ce chiffre est en fait assez facile à casser grâce à l'analyse statistique. En effet, dans un texte général, les lettres de l'alphabet ne sont pas présentes avec la même fréquence. Par exemple,

en français, la lettre 'E' est la plus présente [1]. Ainsi, si on trouve un message chiffré (et si on sait que c'est du français et que c'est chiffré par le chiffre de César) on peut tenter de le comprendre via une analyse statistique. On analyse la fréquence de chaque lettre du message et la plus fréquente remplace probablement le 'E'. Par exemple, si la lettre la plus fréquente est le 'B', il est probable que le message soit chiffré avec un décalage de 3.

Écrivez un module qui reçoit une chaine - le message chiffré par le chiffre de César - et retourne une valeur probable pour le déplacement utilisé pour le chiffrer.

20 Le caractère le plus présent

Écrire un algorithme qui, à partir d'un texte donné sous forme de chaine, retourne le caractère qui s'y trouve le plus de fois. N.B. : on peut simplifier le problème ici en supposant que le texte ne contient que des caractères minuscules non accentués, ainsi que l'absence d'ex-æquo. Exemple : pour "roberto roule en moto à oslo", l'algorithme renverra 'o'.

1. Il y a évidemment des exceptions, cf. "La disparition" de Georges Perec.

Chapitre 9

Le tri

Dans ce chapitre nous voyons quelques algorithmes simples pour trier un ensemble d'informations : recherche des maxima, tri par insertion et tri bulle dans un tableau. Des algorithmes plus efficaces seront vus en deuxième année.

9.1 Motivation

Une application importante de l'informatique est le tri d'informations. La conception d'algorithmes efficaces pour ce type de traitement s'est imposée avec l'apparition de bases de données de taille importante. On citera par exemple la recherche d'un numéro de téléphone ou une adresse dans la version informatisée des pages blanches, ou de façon plus spectaculaire l'efficacité de certains moteurs de recherche qui permettent de retrouver en quelques secondes des mots clés arbitraires parmi plusieurs millions de pages sur le web.

La recherche efficace d'information implique un tri préalable de celle-ci. En effet, si les données ne sont pas classées ou triées, le seul algorithme possible reviendrait à parcourir entièrement l'ensemble des informations. Pour exemple, il suffit d'imaginer un dictionnaire dans lequel les mots seraient mélangés de façon aléatoire au lieu d'être classés par ordre alphabétique. Pour trouver le moindre mot dans ce dictionnaire, il faudrait à chaque fois le parcourir entièrement ! Il est clair que le classement préalable (ordre alphabétique) accélère grandement la recherche.

Ainsi, recherche et tri sont étroitement liés, et la façon dont les informations sont triées conditionne bien entendu la façon de rechercher l'information (cf. algorithme de recherche dichotomique). Pour exemple, prenons cette fois-ci un dictionnaire des mots croisés dans lequel les mots sont d'abord regroupés selon leur longueur et ensuite par ordre alphabétique. La façon de rechercher un mot dans ce dictionnaire est bien sûr différente de la recherche dans un dictionnaire usuel.

Le problème central est donc le tri des informations. Celui-ci a pour but d'organiser un ensemble d'informations qui ne l'est pas *à priori*. On peut distinguer trois grands cas de figure :

1. D'abord les situations impliquant le classement total d'un ensemble de données « brutes », c'est-à-dire complètement désordonnées. Prenons pour exemple les feuilles récoltées en vrac à l'issue d'un examen ; il y a peu de chances que celles-ci soient remises à l'examinateur de manière ordonnée ; celui-ci devra donc procéder au tri de l'ensemble des copies, par exemple par ordre alphabétique des noms des étudiants, ou par numéro de groupe etc.

2. Ensuite les situations où on s'arrange pour ne jamais devoir trier la totalité des éléments d'un ensemble, qui resterait cependant à tout moment ordonné. Imaginons le cas d'une bibliothèque dont les livres sont rangés par ordre alphabétique des auteurs : à l'achat d'un nouveau livre, ou au retour de prêt d'un livre, celui-ci est immédiatement rangé à la bonne place. Ainsi, l'ordre global de la bibliothèque est maintenu par la répétition d'une seule opération élémentaire consistant à insérer à la bonne place un livre parmi la collection. C'est la situation que nous considérerions dans le cas d'une structure où les éléments sont ordonnés.
3. Enfin, les situations qui consistent à devoir re-trier des données préalablement ordonnées sur un autre critère. Prenons l'exemple d'un paquet de copies d'examen déjà triées sur l'ordre alphabétique des noms des étudiants, et qu'on veut re-trier cette fois-ci sur les numéros de groupe. Il est clair qu'une méthode efficace veillera à conserver l'ordre alphabétique déjà présent dans la première situation afin que les copies apparaissent dans cet ordre dans chacun des groupes.

Le dernier cas illustre un classement sur une **clé complexe** (ou **composée**) impliquant la comparaison de plusieurs champs d'une même structure : le premier classement se fait sur le numéro de groupe, et à numéro de groupe égal, l'ordre se départage sur le nom de l'étudiant. On dira de cet ensemble qu'il est classé en **majeur** sur le numéro de groupe et en **mineur** sur le nom d'étudiant.

Notons que certains tris sont dits **stables** parce qu'en cas de tri sur une nouvelle clé, l'ordre de la clé précédente est préservé pour des valeurs identiques de la nouvelle clé, ce qui évite de faire des comparaisons sur les deux champs à la fois. Les méthodes nommées **tri par insertion**, **tri bulle** et **tri par recherche de minima successifs** (que nous allons aborder dans ce chapitre) sont stables.

Le tri d'un ensemble d'informations n'offre que des avantages. Outre le fait déjà mentionné de permettre une recherche et une obtention rapide de l'information, il permet aussi l'application de traitements algorithmiques efficaces (comme par exemple celui des ruptures que nous verrons plus loin) qui s'avéreraient trop coûteux (en temps, donc en argent !) s'ils s'effectuaient sur des ensembles non triés.

Certains tris sont évidemment plus performants que d'autres. Le choix d'un tri particulier dépend de la taille de l'ensemble à trier et de la manière dont il se présente, c'est-à-dire déjà plus ou moins ordonné. La performance quant à elle se juge sur deux critères : le nombre de tests effectués (comparaisons de valeurs) et le nombre de transferts de valeurs réalisés.

Les algorithmes classiques de tri présentés dans ce chapitre le sont surtout à titre pédagogique. Ils ont tous une « complexité en n^2 », ce qui veut dire que si n est le nombre d'éléments à trier, le nombre d'opérations élémentaires (tests et transferts de valeurs) est proportionnel à n^2. Ils conviennent donc pour des ensembles de taille « raisonnable », mais peuvent devenir extrêmement lents à l'exécution pour le tri de grands ensembles, comme par exemple les données de l'annuaire téléphonique. Plusieurs solutions existent, comme la méthode de tri **Quicksort**. Cet algorithme très efficace faisant appel à la récursivité et qui sera étudié en deuxième année a une complexité en $n \log(n)$.

Dans ce chapitre, les algorithmes traiteront du tri dans un **tableau d'entiers à une dimension**. Toute autre situation peut bien entendu se ramener à celle-ci moyennant la définition de la relation d'ordre propre au type de données utilisé. Ce sera par exemple, l'ordre alphabétique pour les chaines de caractères, l'ordre chronologique pour des objets Date ou Moment (que nous verrons plus tard), etc. De plus, le seul ordre envisagé sera l'ordre **croissant** des données. Plus loin, on envisagera le tri d'autres structures de données.

Enfin, dans toutes les méthodes de tri abordées, nous supposerons la taille physique du tableau à trier (notée n) égale à sa taille logique, celle-ci n'étant pas modifiée par l'action de tri.

9.2 Tri par insertion

Cette méthode de tri repose sur le principe d'insertion de valeurs dans un tableau ordonné.

Description de l'algorithme

Le tableau à trier sera à chaque étape subdivisé en deux sous-tableaux : le premier cadré à gauche contiendra des éléments déjà ordonnés, et le second, cadré à droite, ceux qu'il reste à insérer dans le sous-tableau trié. Celui-ci verra sa taille s'accroitre au fur et à mesure des insertions, tandis que celle du sous-tableau des éléments non triés diminuera progressivement.

Au départ de l'algorithme, le sous-tableau trié est le premier élément du tableau. Comme il ne possède qu'un seul élément, ce sous-tableau est donc bien ordonné ! Chaque étape consiste ensuite à prendre le premier élément du sous-tableau non trié et à l'insérer à la bonne place dans le sous-tableau trié.

Prenons comme exemple un tableau tab de 20 entiers. Au départ, le sous-tableau trié est formé du premier élément, tab[1] qui vaut 20 :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
20	12	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

L'étape suivante consiste à insérer tab[2], qui vaut 12, dans ce sous-tableau, de taille 2 :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
12	20	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

Ensuite, c'est au tour de tab[3], qui vaut 18, d'être inséré :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
12	18	20	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

Et ainsi de suite jusqu'à insertion du dernier élément dans le sous-tableau trié.

Algorithme

On combine la recherche de la position d'insertion et le décalage concomitant de valeurs.

```
// Trie le tableau reçu en paramètre (via un tri par insertion).
module triInsertion(tab↓↑ : tableau[1 à n] d'entiers)
    i, j, valAInsérer : entiers
    pour i de 2 à n faire
        valAInsérer ← tab[i]
        // recherche de l'endroit où insérer valAInsérer dans le
        // sous-tableau trié et décalage simultané des éléments
        j ← i - 1
        tant que j ≥ 1 ET valAInsérer < tab[j] faire
            tab[j+1] ← tab[j]
            j ← j – 1
        fin tant que
        tab[j+1] ← valAInsérer
    fin pour
fin module
```

9.3 Tri par sélection des minima successifs

Dans ce tri, on recherche à chaque étape la plus petite valeur de l'ensemble non encore trié et on peut la placer immédiatement à sa position définitive.

Description de l'algorithme

Prenons par exemple un tableau de 20 entiers.

La première étape consiste à rechercher la valeur minimale du tableau. Il s'agit de l'élément d'indice 10 et de valeur 17.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
20	52	61	47	82	64	95	66	84	17	32	24	46	48	75	55	19	61	21	30

Celui-ci devrait donc apparaitre en 1ère position du tableau. Hors, cette position est occupée par la valeur 20. Comment procéder dès lors pour ne perdre aucune valeur et sans faire appel à un second tableau ? La solution est simple, il suffit d'échanger le contenu des deux éléments d'indices 1 et 10 :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
17	52	61	47	82	64	95	66	84	20	32	24	46	48	75	55	19	61	21	30

Le tableau se subdivise à présent en deux sous-tableaux, un sous-tableau déjà trié (pour le moment réduit au seul élément $tab[1]$) et le sous-tableau des autres valeurs non encore triées (de l'indice 2 à 20). On recommence ce processus dans ce second sous-tableau : le minimum est à présent l'élément d'indice 17 et de valeur 19. Celui-ci viendra donc en 2ème position, échangeant sa place avec la valeur 52 qui s'y trouvait :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
17	19	61	47	82	64	95	66	84	20	32	24	46	48	75	55	52	61	21	30

Le sous-tableau trié est maintenant formé des deux premiers éléments, et le sous-tableau non trié par les 18 éléments suivants.

Pour un tableau de taille n, à l'étape i de cet algorithme, on sélectionne donc le minimum du sous-tableau non trié (entre les indices i et n). Une fois le minimum localisé, on l'échange avec l'élément d'indice i. À chaque étape, la taille du sous-tableau trié augmente de 1 et celle du sous-tableau non trié diminue de 1. L'algorithme s'arrête à la $n^{\text{ème}}$ étape, lorsque le sous-tableau trié correspond au tableau de départ. En pratique l'arrêt se fait après l'étape $n-1$, car lorsque le sous-tableau non trié n'est plus que de taille 1, il contient nécessairement le maximum de l'ensemble.

Algorithme

```
// Trie le tableau reçu en paramètre (via un tri par sélection des minima successifs).
module triSélectionMinimaSuccessifs(tab↓↑ : tableau[1 à n] d'entiers)
    i, indiceMin : entier
    pour i de 1 à n – 1 faire                    // i correspond à l'étape de l'algorithme
        indiceMin ← positionMin( tab, i, n )
        swap( tab[i], tab[indiceMin] )
    fin pour
fin module
```

```
// Échange le contenu de 2 variables.
module swap(a↓↑, b↓↑ : entiers)
    aux : entiers
    aux ← a
    a ← b
    b ← aux
fin module
```

```
// Retourne l'indice du minimum entre les indices début et fin du tableau reçu.
module positionMin(tab↓ : tableau[1 à n] d'entiers, début, fin : entiers) → entier
    indiceMin, i : entiers
    indiceMin ← début
    pour i de début+1 à fin faire
        si tab[i] < tab[indiceMin] alors
            indiceMin ← i
        fin si
    fin pour
    retourner indiceMin
fin module
```

9.4 Tri bulle

Il s'agit d'un tri par **permutations** ayant pour but d'amener à chaque étape à la « surface » du sous-tableau non trié (on entend par là l'élément d'indice minimum) la valeur la plus petite, appelée la **bulle**. La caractéristique de cette méthode est que les comparaisons ne se font qu'entre éléments consécutifs du tableau.

Description de l'algorithme

Prenons pour exemple un tableau de taille 14. En partant de la fin du tableau, on le parcourt vers la gauche en comparant chaque couple de valeurs consécutives. Quand deux valeurs sont dans le désordre, on les permute. Le premier parcours s'achève lorsqu'on arrive à l'élément d'indice 1 qui contient alors la « bulle », c'est-à-dire la plus petite valeur du tableau, soit 1 :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
10	5	12	15	4	8	1	7	12	11	3	6	5	4
10	5	12	15	4	8	1	7	12	11	3	6	4	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	3	11	4	6	5
10	5	12	15	4	8	1	7	3	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	1	8	3	7	12	11	4	6	5
10	5	12	15	1	4	8	3	7	12	11	4	6	5
10	5	12	1	15	4	8	3	7	12	11	4	6	5
10	5	1	12	15	4	8	3	7	12	11	4	6	5
10	1	5	12	15	4	8	3	7	12	11	4	6	5
1	10	5	12	15	4	8	3	7	12	11	4	6	5

Ceci achève le premier parcours. On recommence à présent le même processus dans le sous-tableau débutant au deuxième élément, ce qui donnera le contenu suivant :

1	3	10	5	12	15	4	8	4	7	12	11	5	6

Comme pour le tri par recherche des minima successifs, à l'issue de l'étape i de l'algorithme, on obtient un sous-tableau trié (entre les indices 1 et i) et un tableau non encore trié (entre les indices $i + 1$ et n). Le tri est donc achevé à l'issue de l'étape $n - 1$. Cependant, on notera que petit à petit, outre le déplacement de la bulle, les éléments plus petits évoluent progressivement vers la gauche, et les plus gros vers la droite, de sorte qu'il est fréquent que le nombre d'étapes effectif est inférieur au nombre d'étapes théorique.

Algorithme

Dans cet algorithme, la variable indiceBulle est à la fois le compteur d'étapes et aussi l'indice de l'élément récepteur de la bulle à la fin d'une étape donnée.

```
// Trie le tableau reçu en paramètre (via un tri bulle).
module triBulle(tab↓↑ : tableau[1 à n] d'entiers)
    indiceBulle, i : entiers
    pour indiceBulle de 1 à n faire
        pour i de n – 1 à indiceBulle par – 1 faire
            si tab[i] > tab[i + 1] alors
                swap( tab[i], tab[i + 1] )                    // voir algorithme précédent
            fin si
        fin pour
    fin pour
fin module
```

9.5 Cas particuliers

Tri partiel

Parfois, on n'est pas intéressé par un tri complet de l'ensemble mais on veut uniquement les « k » plus petites (ou plus grandes) valeurs. Le tri par recherche des minima et le tri bulle sont particulièrement adaptés à cette situation ; il suffit de les arrêter plus tôt. Par contre, le tri par insertion est inefficace car il ne permet pas un arrêt anticipé.

Sélection des meilleurs

Une autre situation courante est la suivante : un ensemble de valeurs sont traitées (lues, calculées...) et il ne faut garder que les k plus petites (ou plus grandes). L'algorithme est plus simple à écrire si on utilise une position supplémentaire en fin de tableau. Notons aussi qu'il faut tenir compte du cas où il y a moins de valeurs que le nombre de valeurs voulues ; c'est pourquoi on ajoute une variable indiquant le nombre exact de valeurs dans le tableau.

Exemple : on lit un ensemble de valeurs strictement positives (un 0 indique la fin de la lecture) et on ne garde que les k plus petites valeurs.

```
module meilleurs(plusPetits↓↑ : tableau[1 à k + 1] d'entiers, nbValeurs↑ : entier)
    val : entier
    nbValeurs ← 0
    lire val
    tant que val ≠ 0 faire
        insérer( val, plusPetits, nbValeurs )
        lire val
    fin tant que
fin module
```

```
module insérer(val↓ : entier, plusPetits↓↑ : tableau[1 à k + 1] d'entiers, nbValeurs↓↑ : entier)
    i : entiers
    i ← nbValeurs
    tant que i > 0 ET val < plusPetits[i] faire
        plusPetits[i + 1] ← plusPetits[i]
        i ← i - 1
    fin tant que
    plusPetits[i + 1] ← val
    si nbValeurs < k alors
        nbValeurs ← nbValeurs + 1
    fin si
fin module
```

9.6 Recherche dichotomique

La recherche dichotomique a pour essence de réduire à chaque étape la taille de l'ensemble de recherche de moitié, jusqu'à ce qu'il ne reste qu'un seul élément dont la valeur devrait être celle recherchée, sauf bien entendu en cas d'inexistence de cette valeur dans l'ensemble de départ. Nous l'expliquons pour un tableau ordonné mais elle peut s'appliquer aussi avec des changements mineurs pour d'autres structures (par exemple la liste ordonnée que nous étudierons plus tard).

Description de l'algorithme

Soit val la valeur recherchée dans une zone délimitée par les indices indiceGauche et indiceDroit. On commence par déterminer l'élément médian, c'est-à-dire celui qui se trouve « au milieu » de la zone de recherche ; son indice sera déterminé par la formule

indiceMédian ← (indiceGauche + indiceDroit) DIV 2

N.B. : cet élément médian n'est pas tout à fait au milieu dans le cas d'une zone contenant un nombre pair d'éléments. On compare alors val avec la valeur de cet élément médian ; il est possible qu'on ait trouvé la valeur cherchée ; sinon, on partage la zone de recherche en deux parties : une qui **ne contient certainement pas** la valeur cherchée et une qui **pourrait la contenir**. C'est cette deuxième partie qui devient la nouvelle zone de recherche. On réitère le processus jusqu'à ce que la valeur cherchée soit trouvée ou que la zone de recherche soit réduite à sa plus simple expression, c'est-à-dire un seul élément.

Algorithme

```
module rechercheDichotomique(tab↓↑ : tableau[1 à n] de T, valeur↓ : T, pos↑ : entier) → booléen
    indiceDroit, indiceGauche, indiceMédian : entiers
    candidat : T
    trouvé : booléen

    indiceGauche ← 1
    indiceDroit ← n
    trouvé ← faux

    tant que NON trouvé ET indiceGauche ≤ indiceDroit faire
        indiceMédian ← (indiceGauche + indiceDroit) DIV 2
        candidat ← tab[indiceMédian]
        selon que
         candidat = valeur: trouvé ← vrai
         candidat < valeur: indiceGauche ← indiceMédian + 1    // on garde la partie droite
         candidat > valeur: indiceDroit ← indiceMédian – 1     // on garde la partie gauche
        fin selon que
    fin tant que

    si trouvé alors
        pos ← indiceMédian
    sinon
        pos ← indiceGauche                    // dans le cas où la valeur n'est pas trouvée,
              // on vérifiera que indiceGauche donne la valeur où elle pourrait être insérée.
    fin si

    retourner trouvé
fin module
```

Exemple de recherche fructueuse

Supposons que l'on recherche la valeur **23** dans un tableau de 20 entiers. La zone ombrée représente à chaque fois la partie de recherche, qui est au départ le tableau dans son entièreté. Au départ, indiceGauche vaut 1 et indiceDroit vaut 20.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Première étape : indiceMédian ← (1 + 20) DIV 2, c'est-à-dire 10. Comme la valeur en position 10 est 15, la valeur cherchée doit se trouver à sa droite, et on prend comme nouvelle zone de recherche celle délimitée par indiceGauche qui vaut 11 et indiceDroit qui vaut 20.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Deuxième étape : indiceMédian ← (11 + 20) DIV 2, c'est-à-dire 15. Comme on y trouve la valeur 25, on garde les éléments situées à la gauche de celui-ci ; la nouvelle zone de recherche est [11, 14].

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Troisième étape : indiceMédian ← (11 + 14) DIV 2, c'est-à-dire 12 où se trouve l'élément 20. La zone de recherche devient indiceGauche vaut 13 et indiceDroit vaut 14.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Quatrième étape : indiceMédian ← (13 + 14) DIV 2, c'est-à-dire 13 où se trouve la valeur cherchée ; la recherche est terminée.

Exemple de recherche infructueuse

Recherchons à présent la valeur **8**. Les étapes de la recherche vont donner successivement

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Arrivé à ce stade, la zone de recherche s'est réduite à un seul élément. Ce n'est pas celui qu'on recherche mais c'est à cet endroit qu'il se serait trouvé ; c'est donc là qu'on pourra éventuellement l'insérer. Le paramètre de sortie prend la valeur 6.

9.7 Introduction à la complexité

L'algorithme de recherche dichotomique est beaucoup plus rapide que l'algorithme de recherche linéaire. Mais qu'est-ce que cela veut dire exactement ? Comment le mesure-t-on ? Quels critères utilise-t-on ?

On pourrait se dire que pour comparer deux algorithmes, il suffit de les traduire tous les deux dans un langage de programmation, de les exécuter et de comparer leur temps d'exécution. Cela pose toutefois quelques problèmes

- ▷ Il faut que les programmes soient exécutés dans des environnements strictement identiques ce qui n'est pas toujours le cas ou facile à vérifier.
- ▷ Le test ne porte que sur un (voir quelques uns) jeu(x) de tests. Comment en tirer un enseignement général ? Que se passerait-il avec des données plus importantes ? Avec des données différentes ?

En fait, un chiffre précis ne nous intéresse pas. Ce qui est intéressant à connaitre, c'est l'évolution de la rapidité de l'algorithme (plus précisément, le nombre d'opérations de base) en fonction de la **taille** du problème à résoudre. Comment va-t-il se comporter sur de « gros » problèmes ?

Dans le cas de la recherche dans un tableau, la taille du problème est la taille du tableau dans lequel on recherche un élément. On peut considérer que l'opération de base est la comparaison avec un élément du tableau. La question est alors la suivante : pour un tableau de taille n, à combien de comparaisons faut-il procéder pour trouver l'élément (ou se rendre compte de son absence) ?

Pour la recherche linéaire

Cela dépend évidemment de la position de la valeur à trouver. Dans le meilleur des cas c'est 1, dans le pire c'est n mais on peut dire, qu'en moyenne, cela entraine « $n/2$ » comparaisons (que ce soit pour la trouver ou se rendre compte de son absence).

Pour la recherche dichotomique

Ici, la zone de recherche est divisée par 2, à chaque étape. Imaginons une liste de 64 éléments : après 6 étapes, on arrive à un élément isolé. Pour une liste de taille n, on peut en déduire que le nombre de comparaisons est au maximum l'exposant qu'il faut donner à 2 pour obtenir n, soit « $\log_2(n)$ ».

Comparaisons

Voici un tableau comparatif du nombre de comparaison en fonction de la taille n

n	**10**	**100**	**1000**	**10.000**	**100.000**	**1 million**
recherche linéaire	5	50	500	5.000	50.000	500.000
recherche dichotomique	4	7	10	14	17	20

On voit que c'est surtout pour des grands problèmes que la recherche dichotomique montre toute son efficacité. On voit aussi que ce qui est important pour mesurer la complexité c'est l'ordre de grandeur du nombre de comparaisons.

On dira que la recherche simple est un algorithme de complexité linéaire (c'est-à-dire que le nombre d'opérations est de l'ordre de n ou proportionnel à n) ce qu'on note en langage plus mathématique $O(n)$ (prononcé « grand O de n »). Pour la recherche dichotomique, la complexité est logarithmique, et on le note $O(\log_2(n))$.

Comparons les complexités les plus courantes.

n	**10**	**100**	**1000**	$\mathbf{10^4}$	$\mathbf{10^5}$	$\mathbf{10^6}$	$\mathbf{10^9}$
$O(1)$	1	1	1	1	1	1	1
$O(\log_2(n))$	4	7	10	14	17	20	30
$O(n)$	10	100	1000	10.000	10^5	10^6	10^9
$O(n^2)$	100	10.000	10^6	10^8	10^{10}	10^{12}	10^{18}
$O(n^3)$	1000	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{27}
$O(2^n)$	1024	10^{30}	10^{301}	10^{3010}	10^{30102}	10^{301029}	$10^{301029995}$

On voit ainsi que si on trouve un algorithme correct pour résoudre un problème mais que cet algorithme est de complexité exponentielle alors cet algorithme ne sert à rien en pratique. Par exemple un algorithme de recherche de complexité exponentielle, exécuté sur une machine pouvant effectuer un milliard de comparaisons par secondes, prendrait plus de dix mille milliards d'années pour trouver une valeur dans un tableau de 100 éléments.

9.8 Exercices

1 Sélection des maxima

Développez un algorithme similaire consistant à trier un tableau par ordre croissant par sélection des **maxima** successifs. Le sous-tableau trié apparaitra donc à droite du tableau, et les maxima sélectionnés seront à chaque étape positionnés à droite du sous-tableau non trié.

2 Maxima et minima

Développez un algorithme combinant les deux recherches. À chaque étape, on sélectionne donc le minimum et le maximum du sous-tableau restant à trier et on les positionnera à l'endroit *ad hoc*. Cette méthode apporte-t-elle une amélioration en temps ou en simplicité aux deux algorithmes de base ?

3 Amélioration 1

Écrire une amélioration du tri bulle consistant à mémoriser à chaque étape l'indice de la dernière permutation, celui-ci délimitant en fait la véritable taille du sous-tableau trié à l'issue d'une étape. En lieu et place de la boucle pour n'incrémentant l'indice bulle que de 1 à la fois, on écrira une boucle tant que à l'issue de laquelle indiceBulle prendra la valeur de l'indice de dernière permutation + 1.

4 Amélioration 2

L'amélioration précédente est issue de l'observation du sous-tableau déjà trié en début du tableau initial. On peut de même étudier la possibilité d'avoir un sous-tableau, trié également, mais cadré à droite dans le tableau à trier. Cette symétrie suggère une amélioration supplémentaire qui consiste à changer de sens à la fin d'un parcours pour entamer le parcours suivant. Lorsqu'on change de sens, on amènera l'élément le plus grand (qu'on peut nommer le « plomb ») au fond du tableau non trié. L'association des deux méthodes donne ce qu'on appelle le **tri shaker**, dont le but est de restreindre le sous-tableau non trié en augmentant sa borne inférieure et en diminuant sa borne supérieure. Écrire l'algorithme qui réalise cette méthode de tri.

5 Tri de dates

Écrire un algorithme qui trie un tableau de dates par ordre chronologique croissant. Les dates sont des structures Date (cfr. chap. 6)

6 Tri de personnes

Écrire un algorithme qui trie un tableau de personnes sur leur date d'anniversaire (sans tenir compte de l'année). Deux personnes nées le même jour seront départagées par leur nom.

Une personne est représentée par la structure

```
structure Personne
    nom : chaine
    dateNaissance : Date
fin structure
```

7 Tableau de Point

Écrire un algorithme qui reçoit un tableau de structures Point et trie celui-ci sur l'ordre d'éloignement de ces points par rapport à l'origine du plan. Les premiers éléments seront donc ceux les plus proches du point (0, 0) et les derniers seront les points les plus éloignés.

8 Calcul de complexités

Quelle est la complexité d'un algorithme qui :

a) recherche le maximum d'un tableau de n éléments ?
b) remplace par 0 toutes les occurrences du maximum d'un tableau de n éléments ?
c) vérifie si un tableau contient deux éléments égaux ?
d) trie par recherche des minima successifs ?
e) vérifie si les éléments d'un tableau forment un palindrome ?

9 Réflexion

L'algorithme de recherche dichotomique est-il toujours à préférer à l'algorithme de recherche linéaire ?

9.9 Références

▷ http://interstices.info/jcms/c_6973/les-algorithmes-de-tri
Ce site permet de visualiser les méthodes de tris en fonctionnement. Il présente tous les tris vus en première plus quelques autres comme le « tri rapide » (« quicksort ») que vous verrez en deuxième année.

Annexe A

Aide-mémoire

Cet aide-mémoire peut vous accompagner lors d'une interrogation ou d'un examen. Il vous est permis d'utiliser ces méthodes sans les développer. Par contre, si vous sentez le besoin d'utiliser une méthode qui n'apparait pas ici, il faudra en écrire explicitement le contenu.

A.1 Pour manipuler les chaines et les caractères

```
// Est-ce ?

estLettre(car : caractère)  → booléen                          // est-ce une lettre ?
estChiffre(car : caractère)  → booléen                         // est-ce un chiffre ?
estMajuscule(car : caractère)  → booléen                       // est-ce une majuscule ?
estMinuscule(car : caractère)  → booléen                       // est-ce une minuscule ?

// Conversions

majuscule(car : caractère)  → caractère      // convertit une minuscule en une majuscule.
minuscule(car : caractère)  → caractère      // convertit une majuscule en une minuscule.
numLettre(car : caractère)  → entier         // donne la position de la lettre dans l'alphabet.
lettreMaj(n : entier)  → caractère           // donne la lettre majuscule de position donnée.
lettreMin(n : entier)  → caractère           // donne la lettre minuscule de position donnée.
chaine(car : caractère)  → chaine                    // convertit le caractère en une chaine.
varChaine ← varCaractère                                                          // idem
chaine(n : entier)  → chaine                          // convertit un entier en une chaine.
chaine(x : réelr)  → chaine                             // convertit un réel en une chaine.
nombre(ch : chaine)  → réel                          // convertit une chaine en un nombre.

// Manipulations

longueur(ch : chaine)  → entier                                  // donne la taille de la chaine.
car(ch : chaine, n : entier)  → caractère        // donne le caractère à une position donnée.
sousChaine(ch : chaine, pos : entier, long : entier)  → chaine      // extrait une sous-chaine
estDansChaine(ch : chaine, sous-chaine : chaine [ou caractère])  → entier
            // dit où commence une sous-chaine dans une chaine donnée (0 si pas trouvé)
concat(ch1, ch2, ..., chN : chaine)  → chaine                    // concatène des chaines
ch ← ch1 + ch2 + ...+ chN                                                         // idem
```