



**Haute Ecole de Bruxelles-Brabant
Ecole Supérieure d'Informatique**

Rue Royale, 67 – 1000 Bruxelles
02/219.15.46 – esi@he2b.be

ALGORITHMIQUE

ALGO I

Bachelor en Informatique

Activité d'apprentissage enseignée par :

Laurent Beeckmans
(coordinateur),
Grégory Baltus,
Jonas Beleho,
Marcelo Burda,
Geneviève Cuvelier,
Amine Hallal
et Arnaud Pollaris

ALGORITHMIQUE

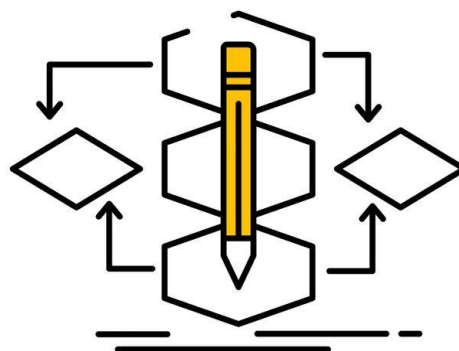
TABLE DES MATIERES

Chapitre 1 – Introduction
Chapitre 2 – Algorithmes linéaires
Chapitre 3 – Alternatives
Chapitre 4 – Modularisation
Chapitre 5 – Variables structurées
Chapitre 6 – Itérations
Chapitre 7 – Tableaux

ALGORITHMIQUE

Chapitre 1

Introduction



Quelques définitions et concepts

La plupart des activités de la vie courante s'exécutent en effectuant une suite logique d'actions basiques et élémentaires : se rendre à l'école, préparer à manger, prendre l'ascenseur, faire les courses, remplacer une ampoule électrique, payer une facture sur un site de self-banking, renouveler un abonnement de train ... Pour vous permettre de réaliser ces actions, que ce soit consciemment ou inconsciemment, votre cerveau suit une série d'instructions acquises par l'apprentissage, par l'habitude ou par la démarche d' « essai-erreur ». Petit à petit, on apprend à exécuter les différentes tâches de la façon la plus efficace.

La rédaction de la manière d'effectuer les différentes opérations pour effectuer une certaine action ou pour résoudre un problème donné porte le nom de *marche à suivre*. Elle se décline sous des formes très diverses : par exemple la *recette* pour préparer un plat culinaire, le *plan de montage* d'un meuble ou d'un jeu de construction, l'*itinéraire* pour se rendre d'un point A à un point B...

Ces différentes marches à suivre se présentent le plus souvent comme une suite d'*instructions* qui permettent, à partir des *données* du problème, d'arriver au *résultat* final souhaité.



Les instructions correspondent à des actions élémentaires qui agissent sur les données et qui sont exprimées par des phrases impératives (par exemple,

« épluchez un oignon », « tournez à gauche », « entrez votre mot de passe »...) ou encore sous forme graphique.

Ces instructions doivent être claires et précises, et rédigées dans un langage compréhensible par l'*exécutant*, c'est-à-dire la personne qui exécute les étapes dictées par la marche à suivre.

Elles doivent de plus être *cohérentes*, et désigner des actions possibles sur les données utilisées. Par exemple « épluchez un oignon » ou « élevez le nombre x au carré » sont des instructions ayant chacune un sens clair dans le contexte où elles apparaissent, tandis que « épluchez le nombre x » ou « élevez un oignon au carré » sont dénués de sens.

Les *données* vont donc de pair avec les *actions* susceptibles d'agir sur ces données. Dans le cadre de ce cours, les données seront les entités de bases utilisées en informatique : nombres, booléens, chaînes de caractères. Progressivement, ces données seront regroupées en structures plus complexes appelées tableaux, listes, piles, files, objets...

Quelques exemples de marche à suivre

- La recette de cuisine



Tarte à la rhubarbe

Ingrédients :

- 150 g de sucre
- 1 cuillère à soupe de crème fraîche
- 1 pâte sablée
- 500 g de rhubarbe
- 2 œufs
- 150 g d'amandes en poudre

Préparation :

1. Préchauffer le four à 180°C.
2. Battre les oeufs entiers avec le sucre pour les faire blanchir, puis ajouter la crème fraîche et la poudre d'amande. Mélanger pour obtenir une préparation homogène.
3. Laver et éplucher la rhubarbe en retirant les gros fils si nécessaire. La couper en petits morceaux.
4. Étaler la pâte, la déposer dans le moule (mettre du papier cuisson facilitera le démoulage), piquer le fond à l'aide d'une fourchette.
5. Ajouter la rhubarbe sur la pâte, puis recouvrir le plus uniformément possible avec la préparation de départ.
6. Cuire à four chaud pendant 30 min.

- Le mode d'emploi

Prendre une capture d'écran avec son iPhone

1. Effectuez l'une des opérations suivantes :

- Sur un iPhone avec Face ID : Appuyez simultanément sur le bouton latéral et le bouton pour monter le volume, puis relâchez-les.
- Sur un iPhone avec un bouton principal : Appuyez simultanément sur le bouton principal et le bouton latéral, puis relâchez-les.



2. Touchez la capture d'écran dans le coin inférieur gauche, puis touchez OK.

3. Choisissez « Enregistrer dans Photos », « Enregistrer dans Fichiers » ou « Supprimer la capture d'écran ».

- L'itinéraire : *aller de l'ESI à la Gare Centrale*

Rue Royale

↑ Prendre la direction ouest sur Pl. du Congrès vers Rue Vandermeulen

53 m

↶ Tourner à gauche pour rester sur Pl. du Congrès

77 m

↑ Continuer sur Rue de Ligne

240 m

↑ Continuer tout droit sur Rue du Bois Sauvage

210 m

↶ Prendre à gauche sur Bd de l'Impératrice

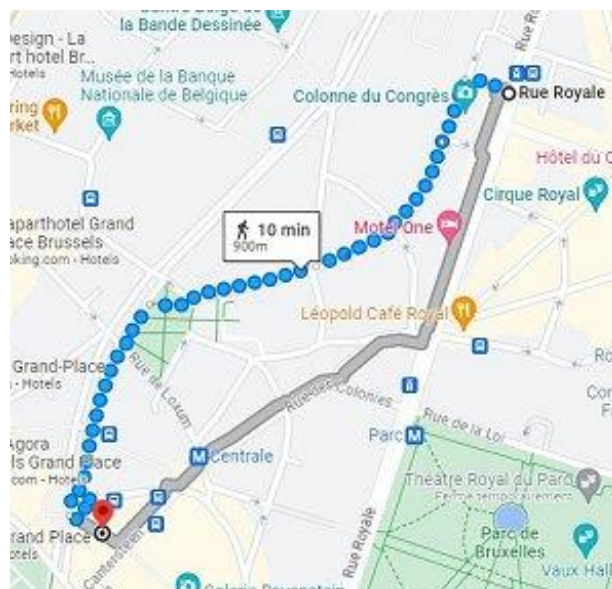
280 m

↶ Tourner à gauche

19 m

Bruxelles-Central

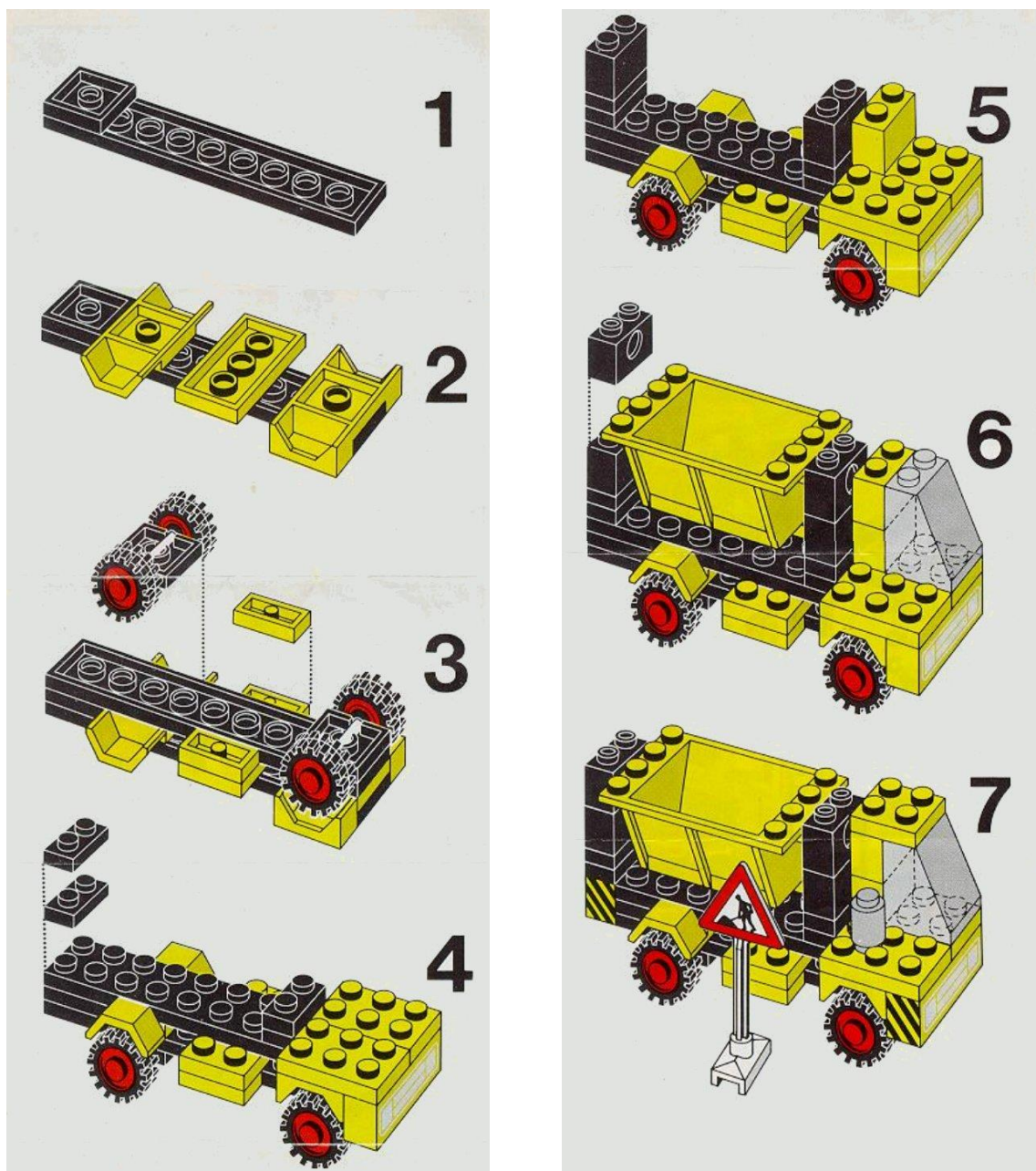
Carr de l'Europe, 1000 Bruxelles



- Les instructions graphiques : *comment faire un nœud de cravate*



- Le plan de montage d'un célèbre jeu de construction



Algorithmes et programmes

L'ordinateur est devenu un outil indispensable pour résoudre des problèmes et effectuer des tâches diverses de façon rapide et efficace. Une marche à suivre exécutée par un ordinateur s'appelle un *programme*. Il est rédigé dans un langage strict appelé *langage de programmation*, et il en existe plusieurs milliers !

A titre d'exemple, voici une série de lignes de code qui ont chacun pour effet d'afficher « Hello, World ! » sur l'écran d'un ordinateur, rédigé dans quelques langages informatiques :



// en Assembleur sous Linux

```
global _start
section .rodata
    msg db 'Hello, World!', 0x0a
    lg dq $-msg
section .text
_start:
    mov rax,1
    mov rdi,1
    mov rsi, msg
    mov rdx,[lg]
    syscall
_end:
    mov rax,60
    mov rdi,0
    syscall
```

// en Fortran

```
program hello
    print *, 'Hello, World!'
end program
```

// en C

```
#include <stdio.h>
int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

// en Java

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

// en PHP

```
<? php
    echo '<p>Hello, World!</p>';
?>
```

// en Python

```
print("Hello, World!")
```

Remarquez la présence d'une série de caractères « techniques », tels que les parenthèses, les accolades, les points-virgules... ainsi que la symétrie des instructions (à une balise ou parenthèse *ouvrante* correspond toujours l'équivalent *fermant*). Ces caractères indispensables pour la rédaction correcte d'un programme forment néanmoins une contrainte inutile lors de l'ébauche d'un programme sous sa forme algorithmique, qui pourrait se formuler simplement ainsi :

écrire "Hello, World ! "

Un *algorithme* représente donc, sous une forme textuelle, une forme simplifiée d'un programme informatique. Il établit la suite des opérations nécessaires à la résolution d'un problème dans un langage plus proche du langage naturel, appelé *Langage de Description d'Algorithme* (LDA) et qui pourra être ensuite traduit facilement par un programmeur dans le langage informatique de son choix.

Ce n'est donc pas un langage de programmation strict, au sens où il n'est pas immédiatement compréhensible par un ordinateur, mais sa forme simplifiée est telle qu'un programmeur peut le comprendre instantanément pour pouvoir le traduire dans un langage informatique précis dont il possède la maîtrise.

L'algorithme est rédigé à partir des *données* du problème (ce que nous appellerons les *variables*) et contient une série d'instructions qui agissent sur ces données, ainsi que des structures de contrôle simples telles que les alternatives et les itérations, et son vocabulaire est limité à quelques poignées de mots-clés seulement, que nous allons introduire au fil des chapitres qui suivent.

Ces mots-clés forment le vocabulaire de base pour la rédaction des algorithmes, on parle aussi de *mots réservés*, ils ne pourront jamais être utilisés pour nommer des variables. Il s'agit de mot qui apparaîtront en gras dans les algorithmes, tels que : **algorithme, si, sinon, fin si, selon que, pour, tant que...**

Quelques références utiles

Pour prolonger votre réflexion sur le concept d'algorithme nous vous proposons quelques ressources en ligne :

- Les Sépas 18 - Les algorithmes
<https://www.youtube.com/watch?v=hG9Jty7P6Es>
- Les Sépas 11 - Un bug
<https://www.youtube.com/watch?v=deI0GV5sWTY>
- Le crêpier psycho-rigide comme algorithme : <https://pixees.fr/?p=446>
- Le baseball multicolore comme algorithme : <https://pixees.fr/?p=450>
- Le jeu de Nim comme algorithme : <https://pixees.fr/?p=443>

Chapitre 2

Algorithmes séquentiels

Dans ce chapitre, nous introduisons quelques concepts clés qui vont nous permettre d'écrire nos premiers algorithmes : les données, les variables, les types et les instructions basiques pour prendre connaissance des données et communiquer le résultat attendu.



Définition

Un algorithme *séquentiel* (ou *linéaire*) contient une suite d'instructions qui s'effectuent entièrement dans leur ordre d'écriture, c'est un algorithme ne contenant pas d'alternatives ou d'itérations : toutes les instructions sont exécutées dans l'ordre, et ce une et une seule fois. C'est le cas des différents exemples de marche à suivre présentés dans le chapitre précédent.

Données

Les exemples de marche à suivre vus au chapitre précédent utilisent des données très diverses : les recettes de cuisine agissent sur des *ingrédients*, les jeux de constructions utilisent des *briques*, l'itinéraire indique les *rues* et les *routes* à emprunter, ... et nous avons vu qu'à chaque type de données sont associées des opérations (ou des actions) dont l'exécution est possible sur ces données.

En informatique, les données de base classiques sont les nombres, les booléens, les chaînes de caractères, et dans le cadre de ce cours introductif, les algorithmes présentés agiront principalement sur ce type de données.

Variables

Les variables sont des entités qui permettent de stocker la valeur des données dans la mémoire de l'ordinateur. Comme son nom l'indique, son contenu est « variable » et peut donc évoluer au cours du déroulement d'un algorithme.

Types

Dans la plupart des langages, les variables ont un *type* prédéfini. Le type indique le genre de données qui peut être stocké par la variable. Les types que nous utiliserons dans ce cours sont les suivants :

- type **entier** : une variable de type *entier* contient – comme son nom l'indique – des valeurs entières, telles que 5, 0, 42, 2023 ou -50, c'est-à-dire les éléments de l'ensemble \mathbb{Z} en mathématiques.

- type **réel** : une variable de type *réel* permet de stocker des nombres qui possèdent une partie décimale, comme 2,5 ou 6,802 ou $1/3 = 0,3333\dots$. En particulier, son contenu peut aussi être un entier, puisque tout nombre entier est aussi un nombre réel (le contraire n'est bien entendu pas vrai).
- type **booléen** : une variable *booléenne* ne peut contenir que les valeurs *vrai* ou *faux*, à l'exception de toute autre possibilité. Elles seront présentées de manière plus approfondie dans le chapitre 3.
- type **chaîne** : une variable de type *chaîne*, ou plus précisément *chaîne de caractères* permet de stocker une suite de caractères alpha-numériques : lettres, chiffres ou tout autre caractère spécial que l'on peut obtenir à partir du clavier d'un ordinateur. Son contenu est habituellement noté entre guillemets, par exemple "ESI", "Rue Royale 67" ou "info@machin.be". Il peut même être limité à un seul caractère, par exemple "x", ou encore n'en contenir aucun, c'est le cas particulier de la *chaîne vide* "".

Types et langages

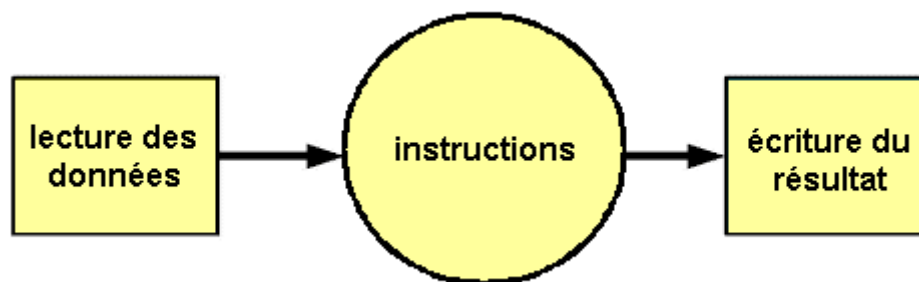
Le choix des types présentés ci-dessus est forcément arbitraire et simplifié dans un but pédagogique. Chaque langage informatique apporte en effet des nuances spécifiques. Ainsi, le type entier peut se décliner en fonction de la taille des entiers à stocker (par exemple *byte*, *short*, *int*, *long* en Java). En algorithmique, nous ne nous soucierons pas de cette limitation, et ne considérons que des entiers au sens mathématique du terme, c'est-à-dire tous les éléments de l'ensemble \mathbb{Z} .

Il en va de même pour les valeurs réelles. Outre la grandeur des nombres, un autre problème se pose ici : la capacité de stockage des nombres étant limitée au niveau d'un ordinateur, les nombres réels illimités (tels que $1/3 = 0,33333\dots$, π , e ou $\sqrt{2}$) sont forcément tronqués et seul une certaine quantité de leurs décimales peut être conservée en mémoire. Le niveau de précision va dépendre de la taille du type de variable réelle utilisée. Encore une fois, cet aspect sera ignoré en algorithmique.

En ce qui concerne les chaînes, certains langages définissent un type particulier pour les chaînes d'un seul caractère pour les différencier des autres chaînes. La connaissance de ces différentes nuances fait partie intégrale de la maîtrise d'un langage de programmation spécifique, et c'est pourquoi l'algorithmique aborde cette notion de type de la façon la plus générale. Ce sera lors de la traduction dans un langage informatique que le programmeur adaptera ces types simples avec le choix des types le plus adéquat.

De plus, certains langages (comme Python) ne nécessitent pas de spécifier le type des variables avant leur utilisation, c'est le contenu placé dans une variable qui va définir son type. Nous gardons ici néanmoins l'approche « type prédéfini » par choix pédagogique.

Forme générale d'un algorithme



Les premiers algorithmes que nous allons écrire suivront le modèle suivant :

```
algorithme [nom]
    // déclaration des données
    // lecture des données
    // instructions agissant sur les données
    // écriture du résultat
fin
```

Dans ce modèle, chaque ligne représente une section de l'algorithme. Chacune de ces sections consistera en une ou plusieurs instructions qui suivront un format bien précis que nous allons détailler dans les paragraphes suivants, en introduisant le formalisme qui sera utilisé dans ce cours.

L'ensemble de ces sections est encadré d'une part par la ligne introductive « algorithme [nom] », aussi appelée *en-tête* et où apparaît le nom choisi pour l'algorithme ; d'autre part par la ligne finale « fin » signalant l'arrêt de la série d'instructions.

Déclaration des données

La déclaration des données est la section qui donne la liste des variables qui seront utilisées dans l'algorithme. Elle découle de la bonne compréhension du problème posé et de l'identification des données indispensables à la résolution du problème, ainsi que du résultat attendu. Elle consiste en une série de lignes qui ont la forme suivante :

```
nom de variable : type
```

Le *nom de variable* peut être choisi librement, mais il est conseillé de choisir un nom ayant un rapport direct avec la variable qu'il représente pour une bonne compréhension de l'algorithme.

Voici quelques exemples :

```
quantité : entier  
nombre : entier  
prix : réel  
hauteur : réel  
adresse : chaîne  
exact : booléen
```

Il est fréquent qu'un nom de variable soit composé de plusieurs mots ; il faut cependant que le nom soit d'un seul tenant (il ne peut pas contenir le caractère d'espacement). On peut dans ce cas assembler les différents mots directement, ou à l'aide du « tiret bas » comme dans les exemples suivants :

```
nombreSpectateurs : entier  
nombre1, nombre2, nombre3 : entier  
prix_beurre, tarif_entrée : réel  
plaqueImmatriculation : chaîne
```

Comme le suggèrent les exemples ci-dessus, on peut déclarer plusieurs variables sur une seule ligne, et les noms peuvent contenir des caractères numériques (bien que ceux-ci ne sont jamais placés au début du nom de variable).

La plupart des langages sont sensibles à la casse des lettres (minuscule/majuscule). Comme un algorithme n'est pas destiné à l'encodage et à la compilation, nous ne nous préoccupons pas de cette distinction, mais il est conseillé pour éviter toute confusion de ne pas utiliser dans un même algorithme (ou programme) des variables de même nom qui se différencieraient seulement par la casse (comme par exemple nombrePersonnes et NombrePersonnes).

Notez que la déclaration des données est une étape facultative dans certains langages informatiques (comme en Python : c'est l'ordinateur qui « devine » et adapte le type des variables en fonction des valeurs qu'on y introduit). Nous conservons néanmoins cette étape dans un but pédagogique.

Lecture des données

La lecture des données est la séquence d'instructions par laquelle l'ordinateur va prendre connaissances des données de l'algorithme.

Les variables présentées dans la déclaration de variable ont au départ un contenu vide ou non défini. Pour pouvoir les utiliser, il faut leur donner une valeur. Dans un premier temps, ce sera à l'utilisateur de l'algorithme de fournir ces valeurs via l'*instruction de lecture* suivante :

```
lire variable
```

Cette instruction demande à l'utilisateur d'entrer une valeur (par exemple au clavier ou par un clic de souris) et sera ensuite affectée à la variable. Cette instruction porte aussi le nom d'*affectation externe*.

On peut également faire une lecture multiple de la façon suivante :

lire variable1, variable2, variable3, ...

Cette instruction est alors équivalente à plusieurs lectures : l'ordinateur va demander le contenu de variable1, ensuite celui de variable2, etc.

Notez que cette forme très simple du point de vue algorithmique deviendra bien plus complexe lors de sa programmation. Lors de l'implémentation, le programmeur veillera à l'inclure dans une interface graphique conviviale, incluant par exemple un texte du style « veuillez introduire votre login et votre mot de passe », ou permettant le choix de certaines valeurs dans une liste déroulante... La tâche du programmeur ne s'arrêtera pas là, il faudra de plus vérifier si la valeur encodée est correcte (lecture de donnée « robuste »)... Mais il est encore trop tôt pour se préoccuper de tout cela...

Instructions agissant sur les données

La section suivante dans le modèle d'algorithme contient des instructions qui ont pour but de transformer les données, de calculer des nouvelles valeurs et de les stocker éventuellement dans de nouvelles variables.

On y trouvera typiquement des instructions appelées *affectations internes* et qui ont la forme suivante :

variable ← expression

Dans le cadre de variables numériques, une *expression* est le plus souvent une formule de nature mathématique composée d'opérateurs et de variables, qui calcule une nouvelle valeur qui sera affectée à la variable à gauche de la flèche. Une expression peut aussi être réduite à un seul nom de variable ou à une valeur numérique. Il existe d'autre type d'expressions que nous verrons ultérieurement.

Les principaux opérateurs arithmétiques sont les suivants :

+	addition
−	soustraction
*	multiplication
/	division réelle
DIV	division entière
MOD	reste de la division entière

Exemples

```
// on peut supposer que toutes les variables apparaissant
// à droite du signe d'affectation ont été initialisées !
a ← b + 2*c
x ← y
delta ← b*b – 4*a*c
r ← x + y*z
r ← (x + y)*z
prixAvecTVA ← prixHorsTVA*1,21
unité ← nombre MOD 10
siècle ← année DIV 100 + 1
quotient ← numérateur / dénominateur
```

On appliquera toujours les règles de priorités usuelles, avec l'usage éventuel de parenthèses pour lever les ambiguïtés.

Pour être valide et compréhensible, il faut que l'expression à droite de la flèche soit correctement écrite, et que le type calculé par l'expression corresponde au type de la variable à gauche de la flèche.

Si leur usage s'avère nécessaire, les fonctions mathématiques peuvent être écrites sous la forme habituelle : $\sin(x)$, $\cos(x)$, $\log(x)$, $\exp(x)$ pour l'exponentielle de x , et \sqrt{x} pour la racine carrée. Le calcul d'exposant – assez rare dans les programmes informatiques – peut se noter sous la forme $x^{**}y$ (x élevé à la puissance y).

Écriture du résultat

Une fois que toutes les variables ont reçu le contenu désiré, il faut communiquer le(s) résultat(s) attendu(s) par l'utilisateur. Pour afficher le contenu d'une variable, on utilisera l'instruction suivante :

écrire variable

Cette instruction a pour effet de faire apparaître le contenu de la variable à l'écran. Ici encore, l'algorithmique simplifie les choses de façon extrême, car on ne se soucie absolument pas du format de l'affichage.

En implémentant cette ligne dans un programme bien écrit, le programmeur devra ajouter toutes les instructions nécessaires pour obtenir un affichage soigné et précis, par exemple l'endroit de l'écran où doit apparaître l'affichage, sa mise en forme, la fonte et la couleur désirée, un cadre avec un texte annonçant de quoi il s'agit... Une fois de plus, ces préoccupations sont du ressort du programmeur, et n'interviennent pas au niveau de l'écriture de l'algorithme, qui se concentre uniquement sur l'aspect logique de la séquence des instructions.

Commentaires

Un algorithme peut aussi contenir des *commentaires*, ce sont des lignes contenant des remarques ou des notes qui peuvent être utiles au lecteur de l'algorithme ou au programmeur chargé d'implémenter l'algorithme. Ils se présentent sous la forme de texte librement rédigé et précédés de la double barre oblique // :

```
// ceci est un commentaire
```

Ces commentaires ne sont donc pas des instructions et ils seront ignorés lors de l'exécution de l'algorithme.

On admet aussi les notations alternatives suivantes :

```
# ceci est un commentaire précédé d'un dièse  
/* ceci est un commentaire encadré par 2 séries de symboles */
```

Exemples

1. L'algorithme suivant calcule la somme de deux entiers x et y, et affiche ensuite le résultat.

```
1  algorithme somme  
2      x, y, somme : entier  
3      lire x, y  
4      somme  $\leftarrow$  x + y  
5      écrire somme  
6  fin
```

2. L'algorithme suivant calcule le prix total à payer pour un produit, à partir du prix à l'unité et la quantité achetée :

```
1  algorithme prixTotal  
2      prixUnité, prixTotal : réel  
3      quantité : entier  
4      lire prixUnité  
5      lire quantité  
6      prixTotal  $\leftarrow$  prixUnité*quantité  
7      écrire "Le prix total à payer est : ", prixTotal  
8  fin
```

Traçage d'algorithme

« Tracer un algorithme » signifie suivre « à la trace » l'évolution de toutes les variables d'un algorithme. L'utilité du traçage est de pouvoir valider un code, et aussi de comprendre et repérer une éventuelle erreur de logique lorsque l'algorithme ne livre pas le résultat attendu.

Pour ce faire, on utilise un tableau dont les colonnes représentent chacune une variable de l'algorithme, et dont chaque ligne est associée à une modification d'une variable donnée, dans l'ordre chronologique de l'exécution de l'algorithme. Le tableau présente aussi une colonne indiquant le numéro des lignes concernées par une modification d'une variable.

Exemple pour l'algorithme somme ci-dessus, en supposant que les données entrées sont 18 et 24 :

ligne	x	y	somme
2	// pas de contenu	// pas de contenu	// pas de contenu
3	18	24	// pas de contenu
4	18	24	42

Mots réservés

Les mots apparaissant en gras dans les exemples d'algorithmes (**algorithme**, **lire**, **écrire**, **fin**) sont appelés mots réservés. D'autres vont apparaître par la suite (**si**, **selon**, **tant que**, **pour**...). Ce sont des mots-clés qui font partie des structures de contrôle, et il est fortement déconseillé de les utiliser à d'autres fins, comme par exemple un nom de variable ou d'algorithme !

Qualité d'un algorithme

Un bon programmeur doit acquérir ce qu'on appelle les « bonnes pratiques » pour pouvoir écrire un « bon algorithme ». Les critères qui font la qualité d'un algorithme sont forcément un peu flous à définir et subjectifs, l'appréciation pouvant varier d'un programmeur à l'autre. Nous allons néanmoins nous mettre d'accord sur certains points :

- La **validité** : un algorithme doit forcément être *valide*, c'est-à-dire qu'il doit fournir le résultat attendu ; en pratique, on vérifie la validité de l'algorithme par une série de *tests* qui portent sur des données couvrant les différents cas particuliers du problème.
- La **lisibilité** : un algorithme doit être compréhensible à sa simple lecture, on doit pouvoir immédiatement voir de quoi il s'agit, sur quelles données il fonctionne, et quel résultat il fournit. Un algorithme clair et lisible s'impose pour pouvoir juger de sa validité, pour le corriger en cas d'erreur de programmation et pour pouvoir aisément le modifier. Un algorithme peu lisible pourrait même paraître incompréhensible à son auteur qui relirait son code quelque temps plus tard ! Un algorithme clair et lisible s'obtient par un choix

pertinent des noms de variables, une **indentation** correcte (nous y reviendrons plus loin), la présence de commentaires utiles et judicieux...

- La **performance** : ce critère juge si un algorithme est efficient ou rapide pour résoudre un problème donné. Il n'est pas très pertinent au début de votre apprentissage, mais le devient lorsqu'on aborde les boucles et les parcours de tableaux. Pour juger de la performance d'un algorithme, il faut se poser les questions : combien d'étapes (en moyenne) sont nécessaires pour fournir le résultat ? Y a-t-il une façon plus rapide de faire, en économisant des variables au rôle restreint ou en limitant le nombre d'instructions ? Ce critère est aussi lié à la notion de **complexité** qui sera étudié plus tard.

Exemples

Voici une série d'algorithmes qui calculent chacun l'âge du capitaine. La plupart ont des qualités et des défauts, pouvez-vous les identifier ? Sont-ils tous valides ? Quelles pratiques vous semblent « bonnes » ou « mauvaises » ? Tous les commentaires sont-ils utiles ? Quel est selon vous le *meilleur* algorithme des 4 ?

```

1  algorithme ageCapitaine1
2      age : entier
3      année : entier
4      lire année // année de naissance du capitaine
5      age ← 2023 – année // cette instruction calcule son âge
6      écrire "L'âge du capitaine est : ", age // affichage de l'âge
7  fin
```

```

1  algorithme ageCapitaine2
2      x : entier
3      lire x
4      écrire "L'âge du capitaine est : ", 2023 – x
5  fin
```

```

1  algorithme ageCapitaine3
2      // N.B. l'algorithme ne tient pas compte des jours et des mois,
3      // à améliorer quand on aura vu les alternatives...
4      ageCapitaine : entier
5      annéeNaissance : entier
6      lire annéeNaissance
7      ageCapitaine ← 2023 – annéeNaissance
8      écrire "L'âge du capitaine est : ", ageCapitaine
9  fin
```

```
1  algorithme ageCapitaine4
2      haddock : entier
3      tintin : entier
4      lire tintin
5      haddock  $\leftarrow$  2023 – tintin
6      écrire "L'âge du capitaine est : ", haddock
7  fin
```

Exercices

1. Choix de type

Quel type de variable, parmi les 4 types présentés dans ce chapitre, choisiriez-vous pour stocker les types de données suivants ?

- | | |
|---|--|
| <input type="radio"/> l'âge d'une personne | <input type="radio"/> un code postal belge |
| <input type="radio"/> le poids d'une personne | <input type="radio"/> un numéro de maison |
| <input type="radio"/> le nom d'une personne | <input type="radio"/> une adresse email |

2. On trace !

Tracer l'algorithme suivant, dans le cas où les nombres lus sont 4 et 10.

```
1  algorithme calculs
2      a, b, c : entier
3      lire a, b
4      c  $\leftarrow$  a + 2*b
5      a  $\leftarrow$  c DIV b
6      b  $\leftarrow$  b MOD a
7      écrire a, b, c
8  fin
```

3. On retrace !

Tracer l'algorithme suivant, en supposant que les nombres lus sont 2 et 3.

```
1  algorithme bizarre
2      x, y : entier
3      lire x
4      x  $\leftarrow$  2*x
5      y  $\leftarrow$  3*x
6      lire y
7      y  $\leftarrow$  3*y
8      écrire y MOD x
9  fin
```

4. Echange de variables

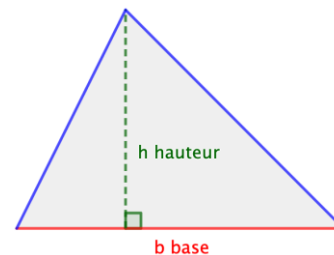
On voudrait dans l'algorithme suivant échanger le contenu des variables x et y de sorte qu'à la ligne 7, les nombres s'affichent dans l'ordre inverse de leur lecture (par exemple, si on lit 30 et 40, les nombres affichés seront – dans l'ordre – 40 et 30). Compléter le code pour obtenir le résultat voulu. Aide : il faudra peut-être introduire une variable supplémentaire pour réaliser cet exercice.

```
1  algorithme échange
2      x, y : entier
3      lire x, y
4      ...
5      ...
6      ...
7      écrire x, y
8      fin
```

5. Surface du triangle

Ecrire un algorithme qui calcule la surface d'un triangle à partir des mesures de sa base et de sa hauteur.

Par exemple, si la base mesure 6 cm et la hauteur 4 cm, l'algorithme affichera 12 cm².



6. En voiture !

Ecrire un algorithme qui calcule la vitesse d'un véhicule en km/h, connaissant la distance qu'il a parcourue en kilomètres et le temps du trajet en minutes.

Attention aux unités, cet algorithme nécessite une petite conversion de données...



7. Histoire de chiffres

Ecrire un algorithme qui lit un nombre entier de 3 chiffres, et affiche ensuite séparément les chiffres des centaines, dizaines et unités.

Exemple : si le nombre lu est 723, l'algorithme affichera par exemple :

- *le chiffre des centaines est 7*
- *le chiffre des dizaines est 2*
- *le chiffre des unités est 3*

8. Le nombre « miroir »

Le *miroir* d'un nombre entier est ce nombre écrit en inversant l'ordre de ses chiffres. Par exemple, le miroir de 189 est 981, et celui de 752 est 257.

Ecrire un algorithme qui lit un nombre entier de 3 chiffres, et affiche ensuite le miroir de ce nombre. Vérifiez si votre algorithme fonctionne aussi lorsque le nombre se termine par 1 ou 2 zéros.



9. Conversion HMS → S

Ecrire un algorithme qui convertit une durée exprimée sous la forme de trois nombres (heure, minute, seconde) en nombre total de secondes.

Par exemple, pour 12h 3' 40'', l'algorithme affichera 43420 secondes.



10. Conversion S → HMS

Plus difficile : conversion inverse de la précédente : convertir un nombre total de secondes au format heure, minute, seconde.

Par exemple, pour 43420 secondes, l'algorithme affichera 12h 3' 40''.

Chapitre 3

Alternatives



Le plus souvent, le déroulement d'un algorithme dépend des données initiales, de conditions, d'interactions non prévisibles avec l'utilisateur. Suivant la valeur d'une ou de plusieurs variables, il peut se passer telle ou telle autre chose, et ce choix ne peut être déterminé au départ de l'algorithme.

Plusieurs scénarios sont donc possibles suivant l'évolution des données d'un problème. C'est au programmeur d'imaginer ces différents scénarios, qui vont être insérés dans le programme par des *structures alternatives*.

La structure alternative de base : si – sinon – fin si

```
si condition alors
    // bloc d'instructions qui sont exécutées lorsque la condition est vraie
sinon
    // bloc d'instructions qui sont exécutées lorsque la condition est fausse
fin si
```

La *condition* est une expression à valeur booléenne ; ce peut être une simple variable booléenne ou une expression calculée à partir des variables de l'algorithme, et qui aura une valeur finale vraie ou fausse.

Si la valeur booléenne de l'expression est vraie, seul le premier bloc d'instructions sera exécuté, et le second sera ignoré. Dans le cas contraire, à savoir si la valeur booléenne de l'expression est fausse, c'est le 2^{ème} bloc d'instructions qui sera exécuté, et le premier qui sera ignoré.

Exemple

Voici à titre d'exemple l'algorithme « classique » qui détermine la valeur du maximum de 2 nombres :

```
1  algorithme maximum
2      x, y, max : entier
3      lire x
4      lire y
5      si x > y alors
6          max ← x
7      sinon
8          max ← y
9      fin si
10     écrire "Le maximum vaut", max
11 fin
```

Imaginons que les nombres lus sont 4 et 1 dans l'ordre de lecture. Dans ce cas, la condition de la ligne 5 est vraie, et c'est la valeur de x (c'est-à-dire 4) qui est prise par la variable max à la ligne 6. La ligne 8 n'est pas exécutée. Si les nombres lus sont 4 et 6, alors la condition de la ligne 5 est fausse, et c'est la ligne 8 qui est exécutée ensuite, où la variable max prendra la valeur de y, soit 6.

Réflexion : que se passe-t-il si x et y reçoivent la même valeur ? L'algorithme est-il encore valide si on remplace le signe « plus grand » à la ligne 5 par « plus grand ou égal » ?

Alternative... sans alternative

Parfois, l'exécution d'instructions est soumise à une condition, mais rien n'est prévu ou précisé dans le cas où la condition n'est pas satisfaite. Par exemple, en disant « si il fait beau ce week-end, on ira à la mer », on ne précise pas ce qui se passera si il ne fait pas beau ce week-end... Il en est de même dans les algorithmes, une structure **si... alors...** ne doit pas forcément contenir un bloc **sinon**. La forme est alors :

```
si condition alors
    // bloc d'instructions qui sont exécutées lorsque la condition est vraie
fin si
```

Exemple

```
1  algorithme casino
2      age : entier
3      lire age
4      si age < 18 alors
5          écrire "trop jeune pour entrer au casino !"
6      fin si
7  fin
```

Dans cet exemple, le message de la ligne 5 ne s'affiche que lorsque le nombre lu est inférieur à 18. Dans le cas contraire, rien ne se passe.

Alternatives multiples

Inversement, une situation peut donner lieu à de multiples alternatives, menant à plusieurs blocs d'instructions, dont un seul sera toutefois exécuté. La forme générale est la suivante :

```

si condition1 alors
    // bloc d'instructions qui sont exécutées lorsque condition1 est vraie
sinon si condition2 alors
    // bloc d'instructions qui sont exécutées lorsque condition1 est
    // fausse et condition2 est vraie
sinon si condition3 alors
    // bloc d'instructions qui sont exécutées lorsque condition1 et
    // condition2 sont fausses et la condition3 est vraie
    ...
sinon
    // bloc d'instructions qui ne sont exécutées que lorsque toutes
    // les conditions précédentes sont fausses
fin si
  
```

Dans cette structure, les conditions sont évaluées en cascade dans leur ordre de présentation, et c'est le bloc d'instructions de la première condition vraie qui est exécuté. Si aucune des conditions sont vraies, alors c'est le bloc d'instructions final du **sinon** qui va être exécuté. Comme précédemment, ce bloc **sinon** est facultatif.

Exemple

L'algorithme suivant indique à partir de l'âge d'une personne si elle est mineure (de 0 à 17 ans), majeure (de 18 à 59) ou senior (à partir de 60).

```

1    algorithme catégorie
2        age : entier
3        lire age
4        si age ≥ 60 alors
5            écrire "La personne est un senior"
6        sinon si age ≥ 18
7            écrire "La personne est majeure"
8        sinon
9            écrire "La personne est mineure"
10       fin si
11    fin
  
```

Réflexion : l'algorithme est-il encore valide si on inverse l'ordre des conditions des lignes 4 et 6 ?

Structure « selon que »

Cette structure est utile dans le cas où le nombre d'alternatives est relativement élevé, et elle permet d'éviter une longue et incommode série de blocs « sinon ». Sa forme est la suivante :

```
selon que variable vaut  
    valeur(s)1 : instruction(s)1  
    valeur(s)2 : instruction(s)2  
    valeur(s)3 : instruction(s)3  
    ...  
    autre : instruction(s)  
fin selon
```

Les zones « valeur(s) » représentent ici une ou plusieurs valeurs séparées par des virgules. De même, le nombre d'instructions peut être unique ou multiple. Le bloc d'instructions exécuté est celui correspondant à la première liste de valeurs contenant la valeur de la variable. Si celle-ci n'apparaît dans aucune liste de valeur, c'est le bloc d'instructions suivant le mot réservé **autre** qui sera exécuté. Ce dernier bloc est aussi facultatif.

Pour un bon usage de cette structure, on veillera à ce qu'une même valeur n'apparaisse pas dans plusieurs listes de valeurs.

Exemple

```
1      algorithme localExamen  
2          groupe : entier  
3          local : chaîne  
4          lire groupe  
5          selon que groupe vaut  
6              111, 112, 121, 122 : local ← "004"  
7              131, 132 : local ← "003"  
8              211, 212 : local ← "101"  
9              221, 222 : local ← "102"  
10             231, 232 : local ← "103"  
11             autre : local ← "105"  
12          fin selon  
13          écrire "L'examen a lieu au local", local  
14      fin
```

Si le groupe entré est 212, cet algorithme affichera « L'examen a lieu au local 101 », et si le groupe entré est 321, cet algorithme affichera « L'examen a lieu au local 105 »

Variables booléennes

En dépit de leur simplicité, les variables booléennes jouent un rôle primordial en programmation ; souvent, elles permettent d'écrire le code de façon élégante et compréhensible. Elles doivent leur nom à George Boole (1815 – 1864), un mathématicien anglais considéré comme le père de la logique moderne. Ces variables n'ont que 2 possibilités de contenu : *vrai* ou *faux*.

Il n'est pas d'usage de leur donner un contenu par une affectation externe (**lire**), mais plutôt par une affectation interne, via une expression qui a elle-même une valeur booléenne (par ex. en utilisant les opérateurs de comparaison =, <, >, ...).

Exemple

```

1  algorithme parité
2      nombre : entier
3      estPair : booléen
4      lire nombre
5      estPair ← nombre MOD 2 = 0
6      si estPair alors
7          écrire "Le nombre est pair "
8      sinon
9          écrire "Le nombre est impair "
10     fin si
11 fin
```

Opérateurs booléens

Un opérateur booléen agit sur une ou plusieurs expressions booléennes pour construire une expression plus complexe ayant également une valeur booléenne. Les 3 principaux opérateurs booléens sont ET (conjonction), OU (disjonction inclusive) et NON (négation). Ces opérateurs sont définis par les tables de vérité suivantes :

p	q	p ET q
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

p	q	p OU q
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

p	NON p
vrai	faux
faux	vrai

Dans le cas où plusieurs opérateurs apparaissent dans une formule booléenne, il convient d'utiliser des parenthèses pour lever toute ambiguïté quant à l'ordre des opérations à effectuer, par exemple :

- p ET (q OU r)
- (p ET q) OU r

En effet, ces 2 expressions ne donnent pas le même résultat booléen !

Evaluation court-circuitée

En mathématique, l'expression booléenne $p \text{ ET } q$ est équivalente à $q \text{ ET } p$ (on parle de la *commutativité* de l'opération). Il n'en est pas exactement de même en informatique, et le choix d'une des 2 possibilités peut même avoir un impact sur la rapidité d'exécution d'un algorithme !

Supposons qu'au cours d'un algorithme contenant cette expression, p ait la valeur « faux ». Dans ce cas, la valeur booléenne de q ne sera pas prise en considération car la valeur finale de l'expression « $p \text{ ET } q$ » est déjà connue, ce sera forcément « faux ».

Il en est de même pour $p \text{ OU } q$: si p est vrai, le résultat de l'opération est déjà connu, ce sera « vrai », quelle que soit la valeur booléenne de q .

Exercices

1. Traçage d'algorithme

Qu'affichent les algorithmes ci-dessous si les nombres lus au départ sont successivement 2 et 3 ? Même question avec 4 et 1.

```

1  algorithme exercice1a
2    a, b : entier
3    lire a, b
4    si a > b alors
5      a ← a + 2 * b
6    fin si
7    écrire a
8  fin
```

```

1  algorithme exercice1b
2    a, b, c : entier
3    lire b, a // attention, piège !
4    si a > b alors
5      c ← a DIV b
6    sinon
7      c ← b MOD a
8    fin si
9    écrire c
10 fin
```

```

1  algorithme exercice1c
2    x1, x2 : entier
3    ok : booléen
4    lire x1, x2
5    ok ← x1 > x2
6    si ok alors
7      ok ← ok ET x1 = 4
8    sinon
9      ok ← ok OU x2 = 3
10   finsi
11   si ok alors
12     x1 ← x1 * 100
13   fin si
14   écrire x1 + x2
15 fin
```

```

1  algorithme exercice1d
2    x, y, z : entier
3    ok : booléen
4    lire x, y
5    z ← 10*x + y + 1
6    selon que z vaut
7      14, 24 : ok ← z MOD 6 = 0
8      30, 41 : ok ← z MOD 8 = 0
9      autre : ok ← z MOD 7 = 0
10   fin selon
11   si ok alors
12     z ← 2*z
13   fin si
14   écrire z
15 fin
```


2. Simplification de code

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 code plus court qui aura un effet équivalent.

Toutes les variables apparaissant dans ces portions de code sont booléennes, à l'exception de a et b, et toutes contiennent une valeur.

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

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

```
si nerveux alors
    cool ← faux
sinon
    cool ← vrai
fin si
```

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

```
si ok1 = faux alors
    ok2 ← faux
fin si
```

```
si ok1 = vrai alors
    si ok2 = faux alors
        ok3 ← ok1 ET ok2
    fin si
fin si
```

3. Majeur ou mineur ?

Écrire un algorithme qui indique si une personne est mineure ou majeure en fonction de son âge.

4. Affaire de signe

Écrire un algorithme qui lit un nombre réel et affiche un message indiquant s'il est strictement négatif, strictement positif ou nul.

5. Equation du second degré

Écrire un algorithme qui résout une équation du second degré, déterminée par le coefficient de x^2 , le coefficient de x et le terme indépendant. L'algorithme affichera la ou les racine(s) de l'équation, ou un message adéquat si elle ne possède pas de solution réelle.

6. Maximum de 3 nombres

Ecrire un algorithme qui lit 3 nombres et affiche la valeur du plus grand des trois.

7. Les poissons

Une personne est du signe astrologique des Poissons si elle est née entre le 19 février et le 20 mars inclus. Ecrire un algorithme qui indique si une personne possède ce signe ou non à partir de sa date d'anniversaire (communiquée sous la forme de 2 entiers *jour* et *mois*).



8. Jour de la semaine

Ecrire un algorithme qui affiche le nom du jour d'un jour du mois de novembre de cette année, donné par son numéro.

Par exemple, si on entre 6, l'ordinateur affichera « lundi ».

9. Nombre de jour du mois

Ecrire un algorithme qui indique le nombre de jours qu'il y a dans un mois de l'année 2024. Le mois est donné sous forme d'un entier entre 1 et 12

Exemples :

- si mois = 1, l'algorithme affichera 31
- si mois = 2, l'algorithme affichera 29

10. Année bissextile

Ecrire un algorithme qui indique si une année est bissextile ou non.

Pour rappel, les années bissextiles on un numéro divisible par 4. Font exception à la règle, les années multiples de 100, sauf les multiples de 400 qui eux sont bien bissextiles ! Ainsi :

- 2022 n'est pas bissextile
- 2024 est bissextile
- 2100 n'est pas bissextile
- 2400 est bissextile

11. La cote finale d'ALG1

Pour le cours d'algorithme, il y a 2 évaluations : le bilan et l'examen. La note finale est calculée comme suit : le bilan compte pour 25% de la cote finale, dans le cas où sa note est supérieure à celle de l'examen. Dans le cas contraire, seul l'examen compte alors pour 100% des points. Ecrire un algorithme qui affiche la note finale à partir des cotes du bilan et de l'examen (tout étant coté sur 20).

12. Le stationnement alterné

Dans une rue où se pratique le stationnement alterné, 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.



13. La boule de cristal

Cet algorithme est destiné à prédire l'avenir, et il doit être infallible ! Il lira au clavier les heures et les minutes, et il affichera l'heure qu'il sera une minute plus tard !

Par exemple, si l'utilisateur entre 21 puis 32, l'algorithme doit répondre « Dans une minute, il sera 21h 33 ».



14. What time is it?

Écrire un algorithme qui convertit un moment de la journée donnée au format HM (heure-minute) dans le format anglais.

Exemples :

- 3h10 devient 3:10 AM
- 12h25 devient 12:25 PM
- 15h21 devient 3:21 PM



15. Prix des photocopies

Chez Happy Copy, le tarif affiché est le suivant :

- moins de 10 copies : 0.10 € la copie
- à partir de 10 copies : 0.07 € par copie
- à partir de 100 copies : 0.05 € par copie
- à partir de 1000 copies : 0.04 € par copie

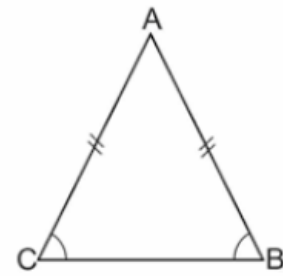
Écrire un algorithme qui affiche le prix à payer connaissant le nombre de copies effectuées



16. Le triangle isocèle

Un triangle est isocèle s'il possède 2 côtés de même longueur, ou de façon équivalente, s'il possède 2 angles égaux. Ecrire un algorithme qui reçoit en paramètres 2 des angles de ce triangle, et affiche un message indiquant si le triangle est isocèle ou non.

N.B. : les données se limitent ici à 2 angles du triangle... si cela vous perturbe, rappelez vous que la somme des angles d'un triangle vaut toujours 180° ...



17. Les saisons

Les quatre saisons sont fixées comme suit :

- le printemps : du 21 mars au 20 juin
- l'été : du 21 juin au 20 septembre
- l'automne : du 21 septembre au 20 décembre
- l'hiver : du 21 décembre au 20 mars

Ecrire, sur base de ces données, un algorithme qui lit une date sous forme du jour (entier entre 1 et 31) et du mois (entier entre 1 et 12), et qui affiche la saison correspondant à cette date.



18. Les promos

Un magasin a décidé d'une campagne promotionnelle pour ses clients : plus grande est la quantité achetée pour un produit, plus grande est la remise (ristourne) sur le prix normal du produit.

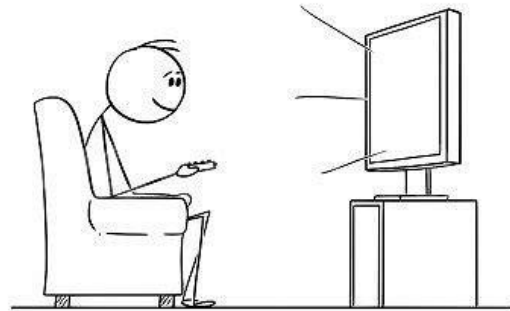
- il n'y a pas de remise pour moins de 5 fois le même produit
- il y a une remise de 5% pour une quantité minimale de 5 et inférieure à 10 fois le même produit
- il y a une remise de 10% pour une quantité minimale de 10 et inférieure à 50 fois le même produit
- il y a une remise de 15% pour une quantité minimale de 50 fois le même produit

Ecrivez un algorithme qui calcule et affiche le prix à payer par le client à partir du prix unitaire et de la quantité achetée du même produit.

19. Moins fort !

Le volume d'un téléviseur est mesuré sur une échelle de nombres entiers entre 0 et 20. Il est modifié par le bouton '+' de la télécommande, qui augmente le volume d'une unité (sauf s'il est au maximum) ou par le bouton '-' qui descend le volume d'une unité (sauf s'il est au minimum).

Ecrire un algorithme qui simule le fonctionnement du réglage du volume. Cet algorithme lira la valeur du volume actuel ainsi qu'un caractère qui peut valoir 'p' (pour le bouton '+') ou 'm' (pour le bouton '-') et affichera la nouvelle valeur du volume.



Chapitre 4

Modularisation



Bien souvent, un problème devient assez vite complexe. Il convient alors de le découper en une série de sous-problèmes plus simples, chacun réalisant une certaine étape du problème de départ. Pour pouvoir agir de la sorte, il nous faut d'abord trouver un moyen de découper un algorithme en plusieurs parties, et de pouvoir les faire communiquer entre-elles ; nous allons pour cela introduire les notions de paramètre et de valeur de retour d'un algorithme, qui vont par la suite remplacer les instructions de lecture et d'écriture vues précédemment.

Cette approche a plusieurs utilités : d'une part, elle permet par la suite d'utiliser des algorithmes déjà écrits précédemment et de les intégrer dans des problèmes plus complexes sans devoir réécrire à chaque fois du code déjà produit ; d'autre part, elle permet aussi une meilleure lisibilité du code, et de remplacer un code long par une série de petits modules n'exécutant chacun qu'une seule chose plus simple.

Algorithme avec paramètres

La première étape consiste à remplacer les instructions de lecture des données par des *paramètres* en entrée. Ces paramètres sont des variables dont le contenu est communiqué lors de l'*appel* de l'algorithme. Une fois ces variables affectées, l'algorithme peut s'enclencher sans nécessiter de lecture de données, ceci ayant été fait « quelque part » en amont de l'appel de l'algorithme.

Exemple

Nous allons adapter l'algorithme *maximum* écrit précédemment en remplaçant la lecture des nombres par des paramètres en entrée, ce qui donne ceci :

```
1  algorithme maximum(x, y : entier)
2      max : entier
3      si x > y alors
4          max ← x
5      sinon
6          max ← y
7      fin si
8      écrire "Le maximum vaut", max
9  fin
```


On voit dans l'écriture de cet algorithme que la déclaration des variables x et y se fait à présent dans l'en-tête de l'algorithme (et il serait inutile – et erroné – de refaire une déclaration de x et y plus bas). Noter aussi que cet algorithme ne fonctionnera que si les 2 paramètres x et y ont reçu une valeur. Cela se fait lors de l'*appel* de l'algorithme.

Appel d'un algorithme par un autre algorithme

Pour appeler un algorithme avec paramètres, il suffit d'écrire son nom accompagné de valeurs ou de variables en même nombre et type que ceux figurant dans son en-tête. Voici à titre d'exemple un algorithme qui appelle l'algorithme `maximum` à plusieurs reprises :

```
1  algorithme exemple
2      a, b : entier
3      lire a, b
4      maximum(a, b)
5      maximum(10, b)
6      maximum(10, 20)
7  fin
```

Supposons que les valeurs lues sont respectivement 3 et 4. La ligne 4 va appeler l'algorithme `maximum` avec ces valeurs qui seront communiquées respectivement à x et y , et la ligne 8 de l'algorithme `maximum` aura pour effet d'afficher le message « le maximum vaut 4 ». Arrivé en fin d'algorithme `maximum`, l'algorithme `exemple` se poursuit à la ligne 5, qui va faire appel à `maximum` avec les valeurs 10 et 4, ce qui a pour effet l'affichage du message « le maximum vaut 10 ». Ensuite, l'algorithme `maximum` est appelé une dernière fois avec les valeurs 10 et 20 et le message « le maximum vaut 20 » est affiché.

L'exemple ci-dessus montre plusieurs appels de l'algorithme `maximum` : d'abord avec des variables (ligne 4), des constantes (ligne 6) et une combinaison des 2 (ligne 5). Il serait aussi possible de mettre des expressions comme par exemple `maximum(a + b, a - b)` ; l'appel est valide si les 2 expressions entre parenthèses ont bien reçu une valeur ayant le type attendu qui sont communiquées ensuite à x et y au début de l'algorithme `maximum`.

En fait, on peut considérer que l'appel de `maximum(expression1, expression2)` provoque de façon implicite les 2 affectations internes :

$$\begin{aligned} x &\leftarrow \text{expression1} \\ y &\leftarrow \text{expression2} \end{aligned}$$

Valeur de retour

La 2^{ème} étape de notre transformation consiste à éliminer à présent l'écriture au niveau de l'algorithme appelé. En effet, dans notre exemple, nous ne voulons peut-être pas forcément que le maximum s'affiche à l'écran, nous pourrions avoir besoin de connaître le maximum de 2 nombres pour en faire un autre usage. Pour récupérer le résultat d'un algorithme sans qu'il s'affiche, nous allons utiliser une *valeur de retour*.

Voici ce que devient l'algorithme maximum avec ce retour de valeur :

```
1  algorithme maximum(x, y : entier) → entier
2      si x > y alors
3          max ← x
4      sinon
5          max ← y
6      fin si
7      retourner max
8  fin
```

Il y a 2 changements majeurs :

- à la ligne 1, l'en-tête est suivi d'une flèche vers la droite et d'un type : ceci signifie qu'à l'issue de l'algorithme, une valeur du type indiqué va être renvoyée
- à la ligne 7, on trouve un nouveau mot réservé : retourner. C'est ici que la valeur annoncée dans l'en-tête est renvoyée (ou retournée) là où on l'attend, c'est-à-dire au niveau de l'appel de cet algorithme.

L'appel d'un algorithme avec valeur de retour diffère légèrement : il faut maintenant utiliser le nom de l'algorithme non comme une instruction indépendante, mais comme une expression. L'exemple précédent pourrait à présent s'écrire ainsi :

```
1  algorithme exemple
2      a, b : entier
3      lire a, b
4      écrire maximum(a, b)
5      écrire maximum(10, b)
6      écrire maximum(10, 20)
7  fin
```

Remarques

- l'instruction de retour se met obligatoirement en fin d'algorithme ; dans le cas contraire, on peut considérer que toute instruction qui serait écrite après la ligne **retourner** serait ignorée. Le code qui serait écrit dans les lignes suivantes ne serait jamais exécuté, et s'appelle « code mort »...

- il ne peut y avoir qu'une seule instruction **retourner** (s'il y en aurait plusieurs, seule la première serait exécutée, et les autres ignorées...)
- retourner est le plus souvent suivi d'une variable contenant la valeur de retour, mais il est aussi possible d'y mettre une expression, comme par exemple **retourner** $x + y$
- les variables utilisées au niveau d'un algorithme sont toujours locales, c'est-à-dire qu'elles sont inconnues en dehors du corps de l'algorithme où elles apparaissent. Ainsi, a et b n'auraient pas de sens dans l'algorithme maximum, et x et y seraient inconnus de l'algorithme exemple.
- le caractère local des variables permet aussi d'utiliser des mêmes noms de variables dans des algorithmes différents, sans crainte de confusion.
- un des gros avantages du découpage d'algorithme en sous-algorithmes est de pouvoir déléguer les tâches et de les séparer de façon claire en fonction de leur spécificité. Dans notre exemple, l'algorithme appelant s'occupe de la lecture des valeurs et de l'affichage, tandis que l'algorithme maximum contient la partie purement mathématique du processus.

Exemple avec max de 3 nombres

Voici encore un exemple d'utilisation de l'algorithme maximum pour calculer le maximum de 3 nombres :

```
1  algorithme maximum3nombres
2      a, b, c : entier
3      lire a, b, c
4      max ← maximum(a, b)
5      max ← maximum(max, c)
6      écrire "Le maximum des 3 nombres vaut", max
7  fin
```

Supposons que les nombres lus sont dans l'ordre 5, 12 et 7. La ligne 4 fait appel à maximum avec les valeurs 5 et 12, et la valeur retournée sera 12, qui est stockée temporairement dans max. Ensuite, la ligne 5 refait un appel à maximum avec les valeurs 12 et 7, la valeur retournée est 12 qui est à nouveau affectée à max. Enfin, à la ligne 6, s'affiche le message « Le maximum des 3 nombres vaut 12 ».

On pourrait même se passer de la variable intermédiaire et écrire ceci :

```
1  algorithme maximum3nombres
2      a, b, c : entier
3      lire a, b, c
4      écrire "Le maximum vaut", maximum(maximum(a, b), c)
5  fin
```

Bien que valide et pertinente, cette écriture n'est pas forcément la plus compréhensible... mais nous vous laissons le choix de l'utiliser ou non...

Exercices

1. A l'appel !

Dans le code ci-dessous, l'algorithme principal fait appel à l'algorithme somme, mais parfois de façon erronée. Quels sont les appels corrects ? Quels sont les lignes incorrectes, et pourquoi ? Après avoir éliminé les lignes incorrectes, tracer l'algorithme en prenant 5 et 10 comme valeurs lues.

```
1  algorithme principal
2      a, b, c : entier
3      lire a, b
4      somme(a, b)
5      a ← somme(a, b)
6      écrire somme(a, 3)
7      écrire somme(a, a)
8      écrire somme(a, c)
9      c ← somme(a, b)
10     écrire somme(a, c)
11     écrire somme(x, y)
12     écrire somme(a, b, c)
13     écrire somme(a, a + b)
14     a ← a + somme(a, b)
15     b ← somme(a, somme(a, b))
16 fin
17
18 algorithme somme(x, y : entier) → entier
19     z : entier
20     z ← x + y
21     retourner z
22 fin
```

2. On trace !

Tracer le déroulement de l'algorithme principal ci-dessous en prenant 1 et 4 comme valeurs de départ.

```
1  algorithme principal
2      a, b : entiers
3      lire a, b
4      a ← calculs(a, 2*b)
5      b ← calculs(b, a)
6      écrire a
7  fin
8
// suite du code page suivante...
```

```
9  algorithme calculs(a, b : entier) → entier
10  c : entier
11  c ← a + b
12  a ← fonction(c)
13  retourner a
14  fin
15
16  algorithme fonction(x : entier) → entier
17  y : entier
18  si x > 10 alors
19  y ← x MOD 10
20  sinon
21  y ← 2*x
22  fin si
23  retourner x + y
24  fin
```

3. Maximum de 3 nombres

Voici encore une variante de l'algorithme pour le calcul du maximum de 3 nombres, qui fait appel à l'algorithme maximum (qui calcule lui le maximum de 2 nombres). Cet algorithme n'utilise cependant que 2 variables. Vérifiez la validité de l'algorithme, et corrigez-le si nécessaire.

```
1  algorithme maximum3nombres
2  a, b : entier
3  lire a, b
4  a ← maximum(a, b)
5  lire b
6  a ← maximum(a, b)
7  écrire "Le maximum des 3 nombres vaut", a
8  fin
```

4. Réécriture

Réécrire au choix quelques algorithmes des exercices du chapitre précédent en remplaçant les données lues par des variables en paramètres, et les écritures par des valeurs de sortie.

5. Valeur absolue

Ecrire un algorithme qui retourne la valeur absolue d'un nombre réel reçu en paramètre

6. Maximum de 4 nombres

Ecrire un algorithme qui retourne le maximum de 4 nombres entrés en paramètres. Cet algorithme fera appel de façon imaginative au module qui calcule le maximum de 2 nombres.

7. Validation de date

Ecrire un algorithme qui vérifie si une date est valide ou non. Pour ce faire, suivre les étapes suivantes :

- écrire un algorithme qui vérifie si une année est bissextile : cet algorithme recevra une année en paramètre et retournera un booléen indiquant si l'année est bissextile ou non
- écrire un algorithme qui reçoit un mois et une année en paramètre, et retourne le nombre de jours qu'il y a dans ce mois (cet algorithme fera appel au précédent pour traiter le cas particulier du mois de février)
- écrire un algorithme qui reçoit deux entiers en paramètre (x et max), et vérifie si x est compris dans l'intervalle [1, max]
- enfin, écrire l'algorithme principal qui reçoit une date en paramètre sous la forme de 3 entiers J, M, A et retourne un booléen indiquant si la date est valide ou non. Il faut vérifier pour cela que le mois M est compris entre 1 et 12, et que le jour J est compris entre 1 et le nombre de jours de ce mois.

Réflexion : que proposeriez-vous comme contrainte sur l'année ?

Chapitre 5

Variables structurées

CARTE D'IDENTITÉ SCOLAIRE

PHOTO

NOM

PRÉNOM

ANNÉE SCOLAIRE

CLASSE

AUTORISÉ(E) À SORTIR

OUI ☐ NON ☐

SIGNATURE

Jusqu'ici, les variables utilisées (entier, réel, booléen, chaîne), sont de type « simple », c'est-à-dire qu'elles ne peuvent contenir qu'une seule valeur à la fois. Hors, dans certains problèmes, les données sont parfois constituées de plusieurs parties, c'est le cas d'une date qui est déterminée par 3 entiers (jour, mois, année) ou d'un moment de la journée, également déterminé par 3 entiers (heure, minute, seconde).

Une adresse en Belgique est déterminée par un nom de rue (chaîne), un numéro de maison (chaîne), un code postal (entier), une ville (chaîne), et éventuellement un pays (chaîne).

L'introduction de la *variable structurée* va permettre de pouvoir manier l'ensemble de données éparées en les regroupant dans une seule donnée unifiée.

Type structuré

Pour pouvoir utiliser une variable structurée, il faudra d'abord définir son type, qui consiste en une liste énonçant les différents *champs* de la variable et les types de ces différents champs. Pour les exemples décrits ci-dessus, cela donnerait :

<p>type date</p> <p>jour : entier</p> <p>mois : entier</p> <p>année : entier</p> <p>fin type</p> <p>type moment</p> <p>heure : entier</p> <p>minute : entier</p> <p>seconde : entier</p> <p>fin type</p>	<p>type adresse</p> <p>rue : chaîne</p> <p>numéroMaison : chaîne</p> <p>codePostal : entier</p> <p>ville : chaîne</p> <p>pays : chaîne</p> <p>fin type</p>
--	--

Lorsque ces types ont été définis, on peut utiliser les mots date, moment, adresse avec les mêmes usages que les types simples.

Noter qu'un champ d'une variable structurée peut lui-même être structuré ; ainsi on pourrait définir un type `carteIdentité` qui contiendrait un champs de type `date` et un autre de type `adresse`, comme ceci :

```
type carteIdentité
    nom : chaîne
    prénom : chaîne
    dateNaissance : date
    domicile : adresse
    numRegistre : chaîne
fin type
```

Déclaration d'une variable structurée

La déclaration d'une variable structurée est identique à celle d'une variable simple :

```
nom de variable : type
```

le type pouvant être ici un des nouveaux types structurés préalablement définis. Par exemple :

```
dateAnniversaire, dateDuJour : date
heureRéveil : moment
adresseEcole, monAdresse, maNouvelleAdresse : adresse
maCarte : carteIdentité
```

Accès aux champs d'une variable structurée

Quelle que soit la complexité de la structure définie, une variable structurée est dans tous les cas une collection de variables simples. Comme la plupart des instructions vues précédemment agissent sur des variables simples, il faut pouvoir manier ou accéder à un champ particulier d'une variable structurée. Pour cela, on utilise la notation pointée. Voici quelques exemples possibles d'instruction portant sur des champs des structures :

```
dateAnniversaire.jour ← 5
heureRéveil.heure ← 7
adresseEcole.rue ← "Rue Royale"
écrire carteIdentité.nom
retourner carteIdentité.domicile.codePostal
```

Dans certains cas, on peut manier l'ensemble des données d'une variable structurée, pour autant que l'action désirée est compréhensible :


```
dateDuJour ← dateAnniversaire  
retourner heureRéveil  
maNouvelleAdresse ← monAdresse  
carteIdentité.domicile ← maNouvelleAdresse
```

Le retour d'une variable structurée est particulièrement utile, car il permet dès à présent de pourvoir retourner un ensemble de valeurs en une seule fois...

Exercices

Les exercices utilisent les types structurés **moment** et **date** définis dans ce chapitre. A partir de maintenant, nous privilégions l'utilisation des paramètres et des valeurs de retour aux lectures et écritures, sauf si ces dernières sont explicitement demandées.

1. Une seconde plus tard

Ecrire un algorithme qui reçoit un moment en paramètre et retourne ce moment augmenté d'une seconde.

Exemples :

- *si le moment reçu est 13h 21' 23'', le retour sera 13h 21' 24''*
- *si le moment reçu est 15h 17' 59'', le retour sera 15h 18' 00''*

2. Comparaison de dates

Ecrire un algorithme qui compare 2 dates reçues en paramètres. L'algorithme retournera un entier valant :

- -1 si la première date est antérieure à la 2^{ème}
- 1 si la première date est postérieure à la 2^{ème}
- 0 si les 2 dates sont identiques

3. Reconversions

- a) Réécrire l'exercice 9 du chapitre 2 en utilisant une variable structurée (conversion d'une durée en nombre total de secondes)
- b) Réécrire l'exercice 10 du chapitre 2 en utilisant une variable structurée (conversion d'un nombre total de secondes au format HMS)

4. Combien de temps ?

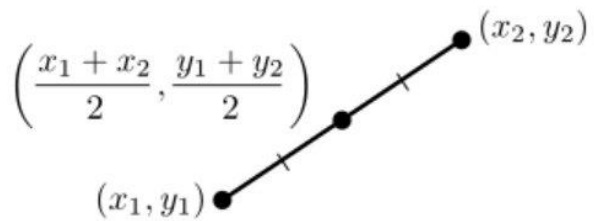
Ecrire un algorithme qui retourne le temps écoulé entre 2 moments de la journée. Utiliser pour cela les 2 algorithmes précédents de façon judicieuse.

Exemple : si les moments donnés sont 12h 59' 50'' et 14h 0' 10'', le résultat attendu sera 1h 0' 20''

5. Les points

Définir un type structuré permettant de manier des points dans le plan cartésien Oxy.

a) Ecrire ensuite un algorithme qui donne les coordonnées du point situé au milieu de 2 points donnés.



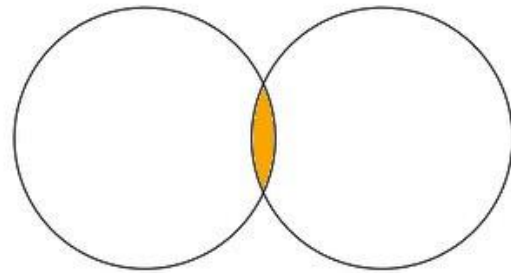
b) Ecrire ensuite un algorithme qui calcule la distance qui sépare 2 points donnés.

$$d = \sqrt{|y_2 - y_1|^2 + |x_2 - x_1|^2}$$

6. Les cercles

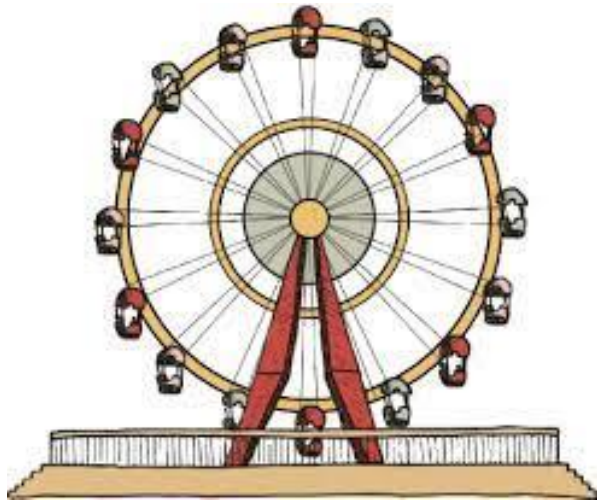
Définir un type structuré permettant de manier des cercles dans le plan cartésien Oxy.

Ecrire ensuite un algorithme qui détermine si 2 cercles donnés possèdent une intersection.



Chapitre 6

Itérations



L'ordinateur est capable de pouvoir exécuter à grande vitesse et de façon infatigable un nombre énorme de tâches répétitives. L'écriture des boucles (ou itérations) est un point crucial dans l'apprentissage de la programmation, elles permettent de faire répéter par l'ordinateur l'exécution d'une série d'instructions aussi longtemps que nécessaire. Nous étudierons dans ce chapitre les 2 types de boucle principaux : la boucle « pour » et la boucle « tant que ».

Boucle « pour »

Cette structure de boucle s'utilise lorsque le nombre précis de répétitions d'une série d'instructions est connu à l'avance. Par exemple, on veut faire 10 fois quelque chose, ou 100 fois, ou encore n fois (avec bien sûr une valeur connue de n)...

La forme générale de la boucle est la suivante :

```
pour var de a à b faire  
    [ instructions ]  
fin pour
```

Ce code signifie que pour toutes les valeurs entières de la variable *var* comprises entre *a* et *b*, les instructions à l'intérieur de la boucle vont être exécutées. Les valeurs de départ *a* et d'arrivée *b* doivent être connues avant d'entamer la boucle, mais pas celle de *var*. Cette variable est appelée *variable de contrôle* de la boucle, elle est implicitement affectée à *a* au début de la boucle et est ensuite incrémentée d'une unité automatiquement à chaque étape.

Cette variable peut aussi apparaître dans les instructions, ce qui est d'ailleurs presque toujours le cas : la boucle ne produit donc pas à chaque tour exactement le même résultat, qui dépend alors de la valeur de *var*.

Théoriquement, la valeur finale de *var* en fin de boucle est *b*, mais cela peut être différent selon le langage de programmation (par ex. *var* pourrait éventuellement

valoir $b + 1$), et il faut donc considérer que sa valeur est indéterminée lorsque la boucle est terminée.

Quelques exemples

```
// exemple 1
pour  $i$  de 1 à 10 faire
    écrire "Hello, World ! "
fin pour
```

Cette boucle a pour effet d'afficher 10 fois de suite "Hello, World ! " à l'écran de l'ordinateur. L'instruction d'écriture ne dépendant pas de la variable de contrôle i , les 10 exécutions de la boucle sont rigoureusement pareils.

```
// exemple 2
pour  $i$  de 1 à 10 faire
    écrire  $i$ 
fin pour
```

Cette boucle a pour effet d'afficher à l'écran tous les entiers de 1 à 10 : à la 1^{ère} itération, i a comme valeur 1, et donc l'instruction dans la boucle est équivalente à « écrire 1 » ; ensuite i est automatiquement incrémenté d'une unité et vaut 2 à la 2^{ème} itération, de sorte que l'instruction dans la boucle est à présent équivalente à « écrire 2 », et ainsi de suite jusqu'à ce que i atteigne la valeur 10.

```
// exemple 3
pour  $i$  de 1 à 10 faire
    écrire  $11 - i$ 
fin pour
```

Cette boucle a pour effet d'afficher à l'écran tous les entiers de 1 à 10 mais en ordre inverse. En effet, comme i vaut 1 au début, l'instruction réalisée est « écrire $11 - 1$ », c'est-à-dire 10 ; à la 2^{ème} itération, l'instruction devient « écrire $11 - 2$ », c'est-à-dire 9 et ainsi de suite. La 1^{ère} valeur de i ne doit pas être obligatoirement 1, par exemple le code suivant produirait exactement le même résultat :

```
// exemple 3bis
pour  $i$  de 0 à 9 faire
    écrire  $10 - i$ 
fin pour
```

Il est important de bien comprendre et de comparer ces 2 derniers exemples (3 et 3bis). On passe de l'un à l'autre par un décalage des valeurs de la variable de contrôle, ce qui modifie aussi le code dans la boucle, mais l'effet est invisible de

l'extérieur : les 2 boucles produisent le même affichage. Ceci montre aussi qu'il y a plusieurs façon différentes d'écrire les boucles... c'est l'entraînement et la pratique qui vous guideront dans le meilleur choix d'écriture, c'est-à-dire celui qui rend le code le plus clair et compréhensible.

Boucle « pour » avec pas

Cette variante est plus rarement utilisée, mais peut s'avérer utile dans quelque cas de figure. Sans mention d'un « pas », la variable de contrôle est automatiquement incrémentée d'une unité. Si on désire une autre incrémentation, on peut la spécifier de cette façon :

```
pour var de a à b par pas faire  
    [ instructions ]  
fin pour
```

Ce code signifie que la variable de contrôle *var* va être à chaque itération incrémentée de la valeur contenue dans la variable *pas*. Ce pas peut aussi être négatif, ce qui est commode pour parcourir une plage de valeur « à contre-sens »

L'exemple 4 ci-dessous a ainsi rigoureusement le même effet que l'exemple 3 qui précède.

```
// exemple 4  
pour i de 10 à 1 par -1 faire  
    écrire i  
fin pour
```

Dans l'exemple ci-dessous, ce sont les nombres impairs de 1 à 19 qui vont s'afficher :

```
// exemple 5  
pour i de 1 à 20 par 2 faire  
    écrire i  
fin pour
```

Le fait d'avoir écrit 20 comme valeur finale n'est pas une erreur de programmation. La valeur 19 aurait été plus claire pour la compréhension du code, mais l'effet est identique : toutes les valeurs impaires de *i* entre 1 et 20 ont bien été parcourues.

Recommendations

Dans la rubrique des bonnes pratiques, il est fortement recommandé :

- de ne jamais modifier la variable de contrôle par des instructions à l'intérieur de la boucle !
- de faire un choix cohérent des variables a , b et pas ... : si $a < b$, alors pas doit être positif, et si $a > b$, pas doit être négatif. Hors de cette cohérence, on peut considérer que la boucle n'est jamais exécutée.

Boucle « tant que »

Cette structure de boucle s'utilise de façon légèrement plus subtile que la boucle « pour » : on l'utilise dans une situation où le nombre d'itérations n'est pas forcément connu à l'avance : l'arrêt de la boucle va dépendre d'une condition qui doit changer de valeur booléenne.

Prenons quelques exemples de la vie courante

- si vous dites « je reste assis sur ce banc tant que le bus n'arrive pas », vous ne savez pas exactement combien de temps vous resterez assis... il se peut que le bus arrive assez vite, ou dans très longtemps en cas d'incident technique ou de grève sauvage...
- si dans une recette de cuisine il est écrit « tournez dans la soupe jusqu'à ébullition », vous ne savez pas à l'avance combien de temps cela va prendre, ni combien de fois vous allez tourner avec la louche dans la casserole...

La forme générale de la boucle est la suivante :

```
tant que condition faire  
    [ instructions ]  
fin tant
```

Elle signifie que tant que la condition énoncée dans l'en-tête est vraie, on va répéter les instructions écrites à l'intérieur de la boucle. La boucle ne s'arrête donc que lorsque la condition devient fausse. La condition est bien entendu une expression booléenne qui doit contenir une variable présente dans les instructions, car sinon, elle ne changerait jamais de valeur et on a alors ce qu'on appelle une *boucle infinie*.

En général, la condition est vraie au départ, car sinon, on ne rentrerait pas dans la boucle et elle ne s'exécuterait pas du tout.

Quelques exemples

```
// exemple 6
i ← 1
tant que i ≤ 10 faire
    écrire i
    i ← i + 1
fin tant
```

Cet exemple réalise exactement la même chose que l'exemple 2 ci-dessus, c'est en fait une boucle « pour » déguisée en boucle « tant que » (ce qui montre au passage qu'on peut toujours remplacer une boucle « pour » par une boucle « tant que », le contraire étant moins évident...).

Notez les différences fondamentales : la variable *i* doit ici être initialisée avant la boucle, et son incrémentation doit s'écrire explicitement dans le code.

```
// exemple 7
onContinue ← vrai
tant que onContinue faire
    écrire « veuillez entrer un nombre »
    lire x
    onContinue ← x ≠ 0
fin tant
```

L'exemple 7 est un exemple d'algorithme « imprévisible » : on ne peut absolument pas connaître à l'avance le nombre d'itérations, car ici le déroulement de l'algorithme dépend aussi de l'interaction avec l'utilisateur. La boucle ne se termine que lorsque celui-ci a entré le nombre 0.

À ce moment, le booléen onContinue devient faux, et on quitte la boucle. Notez que si l'utilisateur n'est pas au courant qu'il doit encoder 0 pour quitter la boucle, lui-même ne sait pas quand s'arrêtera le processus !

Voyons à présent quelques algorithmes modèles utilisant la boucle « tant que » :

Lecture d'une série de valeurs avec valeur sentinelle

L'algorithme suivant est un grand « classique » algorithmique : il permet de lire une série de données, dont la fin est signalée par une **valeur sentinelle** : c'est une valeur spéciale qui ne fait pas logiquement partie de l'ensemble des autres valeurs.

On peut la comparer au bâtonnet « client suivant » que vous placez sur le tapis de la caisse au supermarché pour signaler la fin de vos achats (pour ne pas les mélanger avec ceux du client suivant...). La caissière reconnaît ce bâtonnet comme un objet différent des autres produits et termine alors l'encodage de vos achats et vous présente le montant total à payer.



Prenons pour exemple une série de cotes d'interrogation, dont les valeurs sont des entiers compris entre 0 et 20. On peut choisir comme valeur sentinelle -1 , qui n'est pas une possibilité de cote. Voici un exemple d'une telle série :

12 15 10 8 7 9 15 10 15 17 12 1 14 10 6 5 18 20 15 10 -1

Pour exemple, voici un algorithme qui compte le nombre de cotes encodées :

```

1  algorithme nombreCotes
2      cote, cpt : entier
3      cpt  $\leftarrow$  0
4      lire cote
5      tant que cote  $\neq -1$  faire
6          cpt  $\leftarrow$  cpt + 1
7          lire cote
8      fin tant
9      écrire « Le nombre de cotes est », cpt
10 fin
```

Quelques points importants :

- ligne 4 : la première lecture se fait hors boucle ; c'est nécessaire pour couvrir le cas particulier de la série « vide » qui ne contiendrait alors que la valeur sentinelle
- ligne 7 : la lecture de la valeur suivante se fait juste avant la fin de la boucle ; lorsque la valeur sentinelle -1 est atteinte, la condition de la ligne 5 devient fausse et la boucle se termine

Noter qu'il n'existe pas de valeur sentinelle « universelle », le choix se fait au cas par cas selon la nature des valeurs présentes dans la série. Par exemple, pour une série de températures, -1 ne serait pas un bon choix de valeur sentinelle...

Parcours des chiffres d'un nombre

Certains problèmes nécessitent de parcourir en détail les différents chiffres d'un nombre, par exemple pour voir s'il contient un chiffre donné...

Voici un exemple type d'algorithme parcourant un nombre donné, les chiffres étant parcourus de droite à gauche ; cet algorithme compte le nombre de 5 contenus dans le nombre lu.

```

1  algorithme nombreDeCinq
2      nombre, cpt, chiffre : entier
3      cpt ← 0
4      lire nombre
5      tant que nombre ≠ 0 faire
6          chiffre ← nombre MOD 10
7          si chiffre = 5 alors
8              cpt ← cpt + 1
9          fin si
10         nombre ← nombre DIV 10
11     fin tant
12     écrire « Le nombre de 5 est », cpt
13 fin

```

Voici le traçage de cet algorithme si le nombre lu est 57556 :

ligne	nombre	chiffre	cpt
2	// pas de contenu	// pas de contenu	// pas de contenu
3	// pas de contenu	// pas de contenu	0
4	57556	// pas de contenu	0
6	57556	6	0
10	5755	6	0
6	5755	5	0
8	5755	5	1
10	575	5	1
6	575	5	1
8	575	5	2
10	57	5	2
6	57	7	2
10	5	7	2
6	5	5	2
8	5	5	3
10	0	5	3

L'algorithme se termine par le message « Le nombre de 5 est 3 ».

Remarque : il faut se rappeler que les nombres sont stockés en binaire dans la mémoire de l'ordinateur. Le parcours des chiffres d'un nombre qui semble évident

avec son écriture en base 10 devient beaucoup moins banal lorsqu'il est écrit en binaire, ce n'est qu'une suite de chiffres 0 et 1 dans laquelle les chiffres de la base 10 n'apparaissent d'aucune façon.

Exercices

1. C'est reparti, on trace !

Tracer les 4 algorithmes suivants :

```
1  algorithme 1a
2    i, n : entier
3    n ← 1
3  pour i de 1 à 5 faire
4      n ← 2*n + i
5  fin pour
7  fin
```

```
1  algorithme 1b
2    i, s, p : entier
3    p ← 1
4    s ← 0
5    pour i de 5 à 0 par -1 faire
6        s ← s + p
7        p ← 2*p
8    fin pour
9  fin
```

```
1  algorithme 1c
2    n, c, s : entier
3    n ← 20
4    s ← 0
5    tant que n ≠ 0 faire
6        c ← n MOD 3
7        s ← s + c
8        n ← n DIV 3
9    fin tant
10 fin
```

```
1  algorithme 1d
2    n, somme : entier
3    ok : booléen
4    ok ← vrai
5    n ← 100
6    somme ← 0
7    tant que ok faire
8        somme ← somme + n
9        n ← n DIV 2
10       ok ← n > 5
11  fin tant
12 fin
```

2. Affichage

Ecrire un algorithme qui affiche :

- a) les nombres entiers de 1 à 10
- b) les nombres entiers par ordre décroissant de 10 à 1
- c) les nombres pairs de 2 à 100
- d) les entiers positifs se terminant par 1 et inférieurs à 100
- e) les puissances de 2 de 1 à 1024

3. Affichage (suite)

Ecrire un algorithme qui reçoit un entier n en paramètre et affiche :

- a) les nombres entiers de 1 à n
- b) les nombres entiers de n à 1 (par ordre décroissant)
- c) les n premiers nombres naturels impairs
- d) les multiples de 5 inférieurs à n
- e) les n premiers multiples de 5 (non nuls)
- f) les n premiers carrés parfaits

4. Affichage (suite et fin)

Ecrire un algorithme qui reçoit un entier n en paramètre et affiche les n premiers termes des séries suivantes :

- a) 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
- b) 1, -1, 1, -1, 1, -1, 1, -1, ...
- c) 1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, ...
- d) 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... (suite de Fibonacci)

5. Statistiques sur les interros

Ecrire un algorithme qui lit une série de cotes d'interrogations (entiers positifs entre 0 et 20) se terminant par la valeur sentinelle -1 et affiche ensuite :

- a) le nombre de cotes encodées
- b) le nombre de cotes nulles
- c) la moyenne des cotes
- d) la meilleure cote encodée
- e) la meilleure cote encodée et le nombre de fois qu'elle apparaît dans la série (ce dernier point est plus compliqué à gérer... vous pouvez éventuellement y revenir plus tard...)

6. Evolution d'un capital

A partir des données suivantes :

- un capital (en €)
- un taux d'intérêts (en %)
- un nombre d'années n

écrire un algorithme qui affiche l'évolution de ce capital au cours des n prochaines années.

Exemple d'affichage pour un capital de 1000 € placé à un taux de 2% pendant 3 ans :

année	capital
0	1000 €
1	1020 €
2	1040,40 €
3	1061,21 €

7. Le prix d'entrée

Un parc d'attraction vient de publier ses tarifs d'entrée pour la prochaine saison touristique :

Age	Prix d'entrée
moins de 6 ans	gratuit
de 6 à 12 ans	8 €
de 13 à 18 ans	15 €
de 19 à 60 ans	25 €
plus de 60 ans	15 €

Ecrire un algorithme qui lit une série d'âges d'un groupe de personnes se rendant dans le parc et affiche le montant total à payer. La fin de la série encodée est signalée par la valeur sentinelle 0. Attention, tenir compte de la promotion suivante : les groupes d'au moins 10 personnes bénéficient d'une réduction de 25% sur le prix total !

8. Y a-t-il un zéro ?

- Ecrire un algorithme qui reçoit un entier positif en paramètre et retourne un booléen indiquant si cet entier contient le chiffre 0
- Ecrire un algorithme qui lit une série de nombres positifs (se terminant par la valeur sentinelle -1) et affiche combien parmi ces nombres contiennent le chiffre 0. L'algorithme fera bien sûr appel à l'algorithme précédent !

9. Le prof clément

Pour calculer la cote finale qu'un professeur donne à un étudiant à la fin de l'année, il fait la moyenne de toutes les cotes d'interrogations qui ont été faites au cours de l'année, en oubliant toutefois la cote la moins bonne. Toutes les cotes sont sur 20.



Par exemple, si un étudiant a obtenu la série de cotes suivantes :

15 8 12 6 17 13

sa cote finale sera $(15 + 8 + 12 + 17 + 13)/5$ c'est-à-dire 13. La cote la moins bonne – dans ce cas le 6 – a donc été retirée du calcul.

Ecrire l'algorithme qui permet à ce professeur d'encoder les points d'un étudiant et qui affiche ensuite la cote finale. La fin de l'encodage sera signalée par une valeur négative (par exemple -1).

10. Le tyrannosaure

Un chercheur en génétique a publié une thèse dans laquelle il explique comment, sur base d'une chaîne d'ADN, identifier un tyrannosaure.

L'ADN est une longue chaîne qui consiste en une suite de bases A, C, G et T. D'après ce chercheur, l'ADN est celui du tyrannosaure si et seulement si on y trouve exactement 177 fois la séquence AGCC.

Ecrire un algorithme qui permet de lire une chaîne d'ADN (en entrant les bases une par une...) et qui détermine s'il s'agit de l'ADN du tyrannosaure ou non. Remarque: l'utilisateur de l'algorithme entrera la valeur sentinelle X pour signaler la fin de l'encodage.



Ex. d'encodage: ACGAGCCCCAGGGCTTAGCCAGCCCTTX, et ce n'est visiblement pas l'ADN du tyrannosaure !

11. Factorielle d'un nombre

Vous avez vu en mathématiques que la factorielle de n (notée $n!$) est le produit de tous les entiers compris entre 1 et n . Par exemple $4! = 1*2*3*4 = 24$. On définit aussi le cas particulier $0! = 1$.

Ecrire un algorithme qui reçoit un entier n en paramètre, et retourne sa factorielle.

12. Le premier chiffre

Ecrire un algorithme qui reçoit un entier en paramètre et retourne le premier chiffre de ce nombre.

Par exemple, pour 23540, l'algorithme renverra 2.

13. Le nombre miroir

Nous avons précédemment déjà appris à écrire le nombre miroir pour des entiers de 3 chiffres maximum. A l'aide d'une boucle parcourant les chiffres d'un nombre donné, écrire un algorithme qui renvoie le miroir d'un entier reçu en paramètre.

14. La somme des chiffres

Ecrire un algorithme qui retourne la somme des chiffres d'un nombre entier reçu en paramètre.

15. Nombre réduit

La **réduction** d'un nombre se calcule en additionnant la somme de ses chiffres, et en répétant cette opération jusqu'à ce qu'on obtienne un nombre d'un seul chiffre. Par exemple, la réduction de 468605 vaut 2, puisque les sommes successives des chiffres valent respectivement 29, 11 et 2.

Ecrire un algorithme qui **lit** un nombre et **affiche** sa réduction, en faisant appel de manière répétée au module de l'exercice précédent.

16. Le Plus Grand Commun Diviseur

Il existe plusieurs méthodes pour déterminer le PGCD de 2 nombres entiers, en particulier cet algorithme très simple qui est dû à Euclide (mathématicien grec qui a vécu vers l'an -300).

Voici son fonctionnement : on travaille dans un tableau à 2 colonnes, et on indique dans la 1^{ère} ligne les 2 valeurs dont on recherche le PGCD, soit a et b . Pour remplir les lignes suivantes, on procède à chaque fois comme suit : on remplace a par b , et b par $a \text{ MOD } b$. Lorsque b devient nul, alors la dernière valeur de a obtenue est le PGCD.

a	b
112	63
63	49
49	14
14	7
7	0

Dans l'exemple ci-dessus, on recherche le PGCD de 112 et 63. Après 4 étapes, b s'annule. Le PGCD est la dernière valeur de a obtenue, c'est-à-dire 7.



Euclide

Ecrire un algorithme qui reçoit 2 entiers en paramètres et retourne leur PGCD, en se basant sur la méthode d'Euclide.

Réflexion : dans l'exemple, à chaque étape, on remarque que $a > b$. Que se passerait-il si au départ on a $b > a$? Faut-il adapter l'algorithme pour traiter ce cas ?

17. Le Plus Petit Commun Multiple

On vous propose 2 techniques différentes pour écrire un algorithme calculant le PPCM de 2 entiers a et b :

- la 1^{ère} consiste à déterminer dans la suite des multiples de a le premier entier divisible par b (ou alors dans la suite des multiples de b le premier entier divisible par a ...). Comment s'y prendre pour minimiser le nombre d'itérations ?
- la 2^{ème} consiste à utiliser judicieusement l'algorithme qui calcule le PGCD de a et b en considérant la formule suivante :

$$\text{PGCD}(a, b) * \text{PPCM}(a, b) = a * b$$

18. Les nombres premiers

Les nombres premiers jouent un rôle important dans le domaine de la cryptographie. Un nombre est premier si il possède exactement 2 diviseurs : 1 et lui-même. La suite des nombres premiers commence ainsi : 2, 3, 5, 7, 11, 13, 17, 19, 23... (N.B. : 1 n'est pas considéré comme un nombre premier, car il ne possède qu'un seul diviseur).

Pour vérifier si un nombre n est premier, on vérifie qu'il ne possède pas de diviseur compris entre 2 et la racine carré de n .

Ecrire un algorithme qui reçoit un nombre en paramètre, et renvoie un booléen qui indique si ce nombre est premier ou non. Vérifiez bien si votre algorithme donne le résultat attendu pour les petites valeurs de n . Optimisez également votre code en ne testant que la division par 2 et par les entiers impairs.

19. Encore des séries

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

- a) 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 7, 1, 2, ...
- b) 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, ...
- c) 1, 2, 1, 2, 3, 2, 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 5, 4, 3, 2, 1, ...

n est un entier strictement positif lu au début de l'algorithme.

20. Un peu de dessin

Ecrire un algorithme qui affiche à l'écran les dessins suivants, consistants uniquement en une succession de caractères O de X. Les exemples sont donnés pour des carrés de côté 6, mais votre algorithme doit reproduire les modèles donnés dans un carré de côté n quelconque, où n est un entier lu au départ de l'algorithme.

Pour passer à la ligne, on utilisera dans cet exercice l'instruction « sautDeLigne »

a) le monochrome	b) les lignes	c) les colonnes	d) le damier
000000	000000	OXOXOX	OXOXOX
000000	XXXXXX	OXOXOX	XOXOXO
000000	000000	OXOXOX	OXOXOX
000000	XXXXXX	OXOXOX	XOXOXO
000000	000000	OXOXOX	OXOXOX
000000	XXXXXX	OXOXOX	XOXOXO

e) la diagonale	f) les 2 diagonales	g) le contour	h) les triangles
X00000	X0000X	XXXXXX	000000
OX0000	OX00XO	X0000X	00000X
00X000	00XX00	X0000X	0000XX
000X00	00XX00	X0000X	000XXX
0000X0	OX00XO	X0000X	00XXXX
00000X	X0000X	XXXXXX	0XXXXX

21. La frise

Une frise est constituée d'un motif de dessin qui peut se répéter à l'infini, le plus souvent dans le sens horizontal. Considérons la frise suivante constituée de 3 lignes :

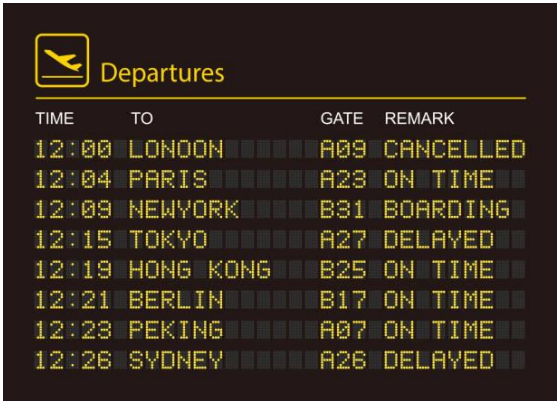
```
0000XX0000XX0000XX0000XX0000XX0000XX0000XX...
0XX0XX0XX0XX0XX0XX0XX0XX0XX0XX0XX0XX0XX0XX...
0XX0000XX0000XX0000XX0000XX0000XX0000XX000...
```

Comme dans l'exercice précédent, c'est une succession de caractères O et X. Si vous observez bien le dessin, on voit que le motif qui constitue la frise se répète tous les 6 caractères, et la longueur de la portion représentée est 42.

Ecrire un algorithme qui lit une longueur et dessine à l'écran une portion de cette frise de la longueur désirée. Comme dans l'exercice précédent, on utilisera l'instruction « sautDeLigne » pour indiquer le saut de ligne.

Chapitre 7

Tableaux



TIME	TO	GATE	REMARK
12:00	LONDON	A09	CANCELLED
12:04	PARIS	A23	ON TIME
12:09	NEWYORK	B31	BOARDING
12:15	TOKYO	A27	DELAYED
12:19	HONG KONG	B25	ON TIME
12:21	BERLIN	B17	ON TIME
12:23	PEKING	A07	ON TIME
12:26	SYDNEY	A26	DELAYED

En informatique, un tableau est une structure permettant de stocker une série de variables de même type sous une seule appellation, l'emplacement des variables étant différencié et déterminé par un indice. On peut comparer la situation aux appartements d'un immeuble : tous les appartements partagent la même adresse (rue et numéro) mais se distinguent dans l'immeuble par un numéro de boîte postale.

La **taille** (ou **longueur**) d'un tableau est le nombre de variables qu'il peut contenir. Dans la plupart des langages informatiques, l'indice du premier élément est 0 (c'est le cas en Java et aussi en Python, où la notion de tableau se confond avec celle de liste), ce qui fait qu'il y a un décalage entre l'indice d'un élément et son numéro d'ordre : le 1^{er} élément occupe la case d'indice 0, le 2^{ème} la case d'indice 1, le 3^{ème} la case d'indice 2, et ainsi de suite jusqu'au dernier élément qui occupe la case d'indice $\text{taille} - 1$.

Pour accéder ou utiliser un élément d'un tableau, on fait suivre après le nom du tableau l'indice voulu entre crochets : ainsi, `tab[i]` désigne l'élément d'indice `i` du tableau `tab`. Le schéma suivant représente un tableau nommé `tab` et de taille 10 :

<code>tab[0]</code>	<code>tab[1]</code>	<code>tab[2]</code>	<code>tab[3]</code>	<code>tab[4]</code>	<code>tab[5]</code>	<code>tab[6]</code>	<code>tab[7]</code>	<code>tab[8]</code>	<code>tab[9]</code>
---------------------	---------------------	---------------------	---------------------	---------------------	---------------------	---------------------	---------------------	---------------------	---------------------

Déclaration d'un tableau

Pour déclarer un tableau, on mentionnera son nombre d'éléments et le type de ces éléments comme dans les exemples suivants :

- | | |
|---|--|
| 1 | <code>tab</code> : tableau de 10 entiers |
| 2 | <code>prix</code> : tableau de 100 réels |
| 3 | <code>calendrier</code> : tableau de 365 date(s) |

La ligne 1 va créer un tableau de 10 entiers indicé de 0 à 9, la ligne 2 va créer un tableau de 100 réels indicé de 0 à 99 et la ligne 3 va créer un tableau de 365 éléments de type date (voir chapitre 5).

Attention, comme pour la déclaration des variables « simples », le contenu d'un tableau est indéfini lors de sa déclaration. Même si dans certains langages, le tableau est initialisé avec des valeurs par défaut (par exemple 0 pour un tableau de type numérique), nous adopterons ici la convention qu'un tableau doit d'abord être explicitement initialisé avant de pouvoir être utilisé.

Dans la plupart des cas, nous écrirons des algorithmes pour lesquels un tableau est reçu en paramètre et dont le contenu a déjà été précisé. Nous donnons néanmoins à titre d'exemple un algorithme qui montre comment on peut créer un tableau et l'initialiser avec des valeurs entrées par l'utilisateur :

```
1  algorithme initialisationTableau
2      i : entier
3      tab : tableau de 10 entiers
4      pour i de 0 à 9 faire
5          écrire « entrer l'élément d'indice », i
6          lire tab[ i ]
7      fin pour
8  fin
```

Cet algorithme est bien sûr un exemple d'étude qui s'avère impraticable ; en pratique, on n'initialise jamais un tableau manuellement ; il se remplit en général par des procédures automatiques ou par la lecture de données dans des fichiers...

Voici par exemple comment initialiser un tableau de taille 1000 avec des 0 :

```
1  algorithme initialisationTableauAvecZéro
2      i : entier
3      tab : tableau de 1000 entiers
4      pour i de 0 à 999 faire
5          tab[ i ] ← 0
7      fin pour
8  fin
```

Parcours d'un tableau

Lorsqu'un tableau a été déclaré, sa taille est fixée et ne peut être modifiée au cours d'un algorithme. Pour connaître la taille d'un tableau déclaré, on utilisera la notation suivante :

tab.taille

Cette notation pointée est similaire à celle utilisée pour les variables structurées, et qui est utilisée dans la *programmation orientée objet*. On considère ici que la taille est un attribut, ou une propriété d'un tableau que l'on peut récupérer par cette

expression. Voici par exemple un algorithme qui reçoit un tableau en paramètre (qu'on suppose initialisé) et affiche son contenu à l'écran :

```

1  algorithme affichageTableau(tab : tableau d'entiers)
2      i : entier
3      pour i de 0 à tab.taille – 1 faire
4          écrire tab[ i ]
5      fin pour
6  fin

```

Noter que cette façon de faire est bien commode, cet algorithme pouvant fonctionner pour n'importe quel tableau d'entier, indépendamment de sa taille.

Taille physique et taille logique

Un tableau n'est pas forcément utilisé dans sa totalité ; dans certains problèmes, seule une portion de taille variable du tableau est utilisée. Le nombre de cases du tableau effectivement utilisées s'appelle la *taille logique* du tableau, et sa taille totale (celle définie précédemment) s'appelle la *taille physique*.

Voici par exemple un tableau dont seules les cases d'indices 0 à 6 ont été affectées. Sa taille physique est 10, mais sa taille logique est 7.

0	1	2	3	4	5	6	7	8	9
50	45	15	35	20	80	25	?	?	?

Dans les exercices qui suivent, nous ne travaillerons qu'avec des tableaux entièrement remplis, c'est-à-dire dont la taille logique coïncide avec la taille physique. De plus, ces concepts se montreront peu utiles lorsque viendra l'étude des listes qui permettent d'effacer et de camoufler cette problématique de taille logique.

Exercices

1. Traçons !

Suivez l'évolution du tableau `tab` dans l'algorithme suivant :

```

1  algorithme tableau
2      tab : tableau de 6 entiers
3      i : entier
4      pour i de 0 à 5 faire
5          tab[ i ] ← 2*i
6      fin pour
7      tab[1] ← tab[3]
8      tab[2] ← tab[4] + tab[5]
9      pour i de 0 à 5 par 2 faire
10         tab[ i ] ← 2*tab[ i ]
11     fin pour
12 fin

```

2. Initialisation

Ecrire un algorithme qui crée et initialise – à l'aide d'une boucle « pour » – un tableau de taille 10 avec le contenu suivant :

a)

0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	60	70	80	90	100

b)

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	11	13	15	17	19

c)

0	1	2	3	4	5	6	7	8	9
10	9	8	7	6	5	4	3	2	1

3. Somme

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui retourne la somme de ses éléments.

4. Maximum

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui retourne la plus grande valeur de ce tableau.

5. Minimum

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui retourne la plus petite valeur de ce tableau.

6. Indice du maximum

Écrire un algorithme qui reçoit en paramètre un tableau d'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.

Par exemple, pour le tableau suivant, l'algorithme va renvoyer 4 :

0	1	2	3	4	5	6	7	8	9
5	0	30	40	50	25	45	50	25	10

Que faut-il changer à votre algorithme pour renvoyer l'indice le plus grand en cas d'ex-æquo? Et pour retourner l'indice du minimum ?

7. Moyenne d'éléments

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et calcule la moyenne des éléments situés entre les valeurs minimale et maximale du tableau (ces deux valeurs étant incluses dans le calcul de la moyenne).

Dans l'exemple ci-dessous, le maximum est 85, le minimum 5 et la moyenne des 8 éléments concernés est 21.

12	85	21	17	8	6	10	16	5	74	64	29	41	11	73
----	----	----	----	---	---	----	----	---	----	----	----	----	----	----

N.B. : n'hésitez pas à réutiliser certains exercices précédents pour faciliter l'écriture de l'algorithme.

8. Y a-t-il un zéro ?

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui retourne un booléen indiquant si ce tableau contient un élément nul.

9. Plus grand écart absolu

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui retourne le plus grand écart absolu entre **deux éléments consécutifs** de ce tableau.

Par exemple, pour le tableau suivant, l'algorithme va renvoyer 12 :

0	1	2	3	4	5	6	7	8	9
7	9	12	8	4	11	20	8	5	15

10. Pas de négatifs !

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui remplace dans ce tableau toutes les valeurs négatives par leur opposé.

Par exemple, le tableau suivant

0	1	2	3	4	5	6	7	8	9
5	2	-3	9	10	-2	3	-8	-7	6

deviendra après traitement :

0	1	2	3	4	5	6	7	8	9
5	2	3	9	10	2	3	8	7	6

11. Tableau ordonné

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui retourne un booléen indiquant si ce tableau est ordonné de façon croissante.

12. Recherche de valeur

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers, ainsi qu'une valeur **val**, et qui recherche cette valeur dans le tableau. Si cette valeur est trouvée, l'algorithme renvoie l'indice de la position de **val** dans le tableau ; si **val** n'est pas présente dans le tableau, l'algorithme retourne alors -1. Il se pourrait que **val** soit présente plusieurs fois dans le tableau, l'algorithme retourne l'indice de la 1^{ère} valeur trouvée.

13. Recherche dans un tableau ordonné

Idem que l'exercice précédent, mais on suppose cette fois que le tableau reçu en paramètre est ordonné de façon croissante.

14. Le tableau décalé

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui décale tous ses éléments d'un indice. Le dernier élément prendra lui la place du premier.

Par exemple, le tableau suivant

0	1	2	3	4	5	6	7	8	9
4	6	7	18	25	6	8	13	2	9

deviendra après traitement :

0	1	2	3	4	5	6	7	8	9
9	4	6	7	18	25	6	8	13	2

15. Le tableau renversé

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers 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.

Par exemple, le tableau suivant

0	1	2	3	4	5	6	7	8	9
4	6	7	18	25	6	8	13	2	9

deviendra après traitement :

0	1	2	3	4	5	6	7	8	9
9	2	13	8	6	25	18	7	6	4

16. Tableau symétrique

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers 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.

17. Egalité de tableaux

Écrire un algorithme qui reçoit 2 tableaux en paramètres, et retourne un booléen indiquant si ces tableaux sont identiques (en taille et en contenu !)

18. Du nombre au tableau

Ecrire un algorithme qui reçoit un nombre entier en paramètre et retourne un tableau contenant les chiffres de ce nombre (un chiffre par case du tableau !)

Par exemple, si le nombre reçu est 584621, l'algorithme retournera le tableau :

0	1	2	3	4	5
5	8	4	6	2	1

19. Le code postal

On considère un tableau dont les éléments sont de type structuré « commune », défini comme suit :

```

type commune
    code : entier           // code postal de la commune
    nom : chaîne           // nom de la commune
fin type
```


Ce tableau contient (dans un ordre quelconque) les données de toutes les communes de Belgique. Ecrire un algorithme qui reçoit en paramètres ce tableau ainsi que le nom d'une commune de Belgique, et retourne le code postal correspondant.

20. Occurrence des chiffres

Écrire un algorithme qui reçoit un nombre entier positif ou nul en paramètre et qui affiche pour chacun de ses chiffres le nombre de fois qu'il apparaît dans ce nombre.

Par exemple, pour le nombre 10502851125, l'affichage pourrait être le suivant :

- 0 apparaît 2 fois
- 1 apparaît 3 fois
- 2 apparaît 2 fois
- 5 apparaît 3 fois
- 8 apparaît une fois

(l'affichage ne mentionne donc pas les chiffres qui n'apparaissent pas dans le nombre).

21. OXO

Soit le tableau **oxo** dont le contenu des cases est le caractère 'O' ou 'X'. Écrire un 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 qui se chevauchent, on ne comptabilise qu'une seule séquence.

Dans l'exemple ci-dessous, il y a donc 3 séquences à comptabiliser :

O	X	X	O	X	O	X	O	X	O	O	O	O	X	X	O	X	O	O	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

22. Position des minimums

Écrire un algorithme qui reçoit en paramètre un tableau d'entiers et qui affiche le ou les indice(s) des éléments contenant la valeur minimale du tableau.

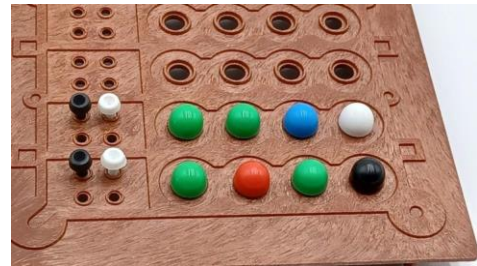
- écrire une première version « classique » avec deux parcours de tableau
- écrire une deuxième version qui ne parcourt qu'une seule fois le tableau en stockant dans un deuxième tableau les indices des plus petits éléments rencontrés (ce tableau étant à chaque fois réinitialisé lorsqu'un nouveau minimum est rencontré)

23. Les doublons

Écrire un algorithme qui retourne un booléen qui indique si le tableau contient au moins 2 éléments égaux.

24. 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 éléments de type Couleur. Écrire un algorithme qui reçoit ces tableaux en paramètres et affiche 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.