

Développement I (Algorithmique et Java)

DEV 1 - DEV

2018-2019

A.HALLAL
C.LEIGNEL
DP.BISCHOP
J.BELEHO
M.CODUTTI
N.PETTIAUX
N.RICHARD
P.BETTENS



Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

Rue Royale 67 1000 Bruxelles
+32 (0)2 219 15 46
esi@he2b.be

Ce syllabus a été écrit initialement par Stefan Monabaliu pour un cours de « Logique et techniques de programmation ». Les grilles de cours ont été revues notamment suite au décret paysage et le cours d'aujourd'hui est un cours de développement qui s'intitule simplement « Développement I ».

Ce syllabus a été adapté par Catherine Leruste, Laurent Beeckmans, Marco Codutti et Pierre Bettens au fil de ses versions.

Document produit avec L^AT_EX.
Version du 17 septembre 2018.

Crédits

 Icône « Book dictionary » de Thomas Elbig¹ de The noun project²

 Icône « Maze » de Gilbert Bages³ de The noun project

 Icône « Attention » de Vineet Kumar Thakur⁴ de The noun project

 Icône « Card » de Alexander Skowalsky⁵ de The noun project

 Icône « Stop » de Gregor Cresnar⁶ de The noun project

 Icône « Coffee » de Luis Prado⁷ de The noun project



Ce document est mis à disposition selon les termes de la Licence Creative Commons Attribution - Partage dans les Mêmes Conditions 4.0 International.
<https://creativecommons.org/licenses/by-sa/4.0/deed.fr>

Les autorisations au-delà du champ de cette licence peuvent être demandées à
esi-dev1-list@he2b.be.

-
1. <https://thenounproject.com/dergraph>
 2. <https://thenounproject.com>
 3. <https://thenounproject.com/gilbertbages>
 4. <https://thenounproject.com/vkvineet>
 5. <https://thenounproject.com/sandorsz>
 6. <https://thenounproject.com/grega.cresnar>
 7. <https://thenounproject.com/Luis>

Table des matières

I	Introduction générale	7
1	Résoudre des problèmes	9
1.1	La notion de problème	9
1.2	Procédure de résolution	11
1.3	Ressources	14
2	Les algorithmes et les programmes	15
2.1	Algorithmes et programmes	15
2.2	Les phases d'élaboration d'un programme	18
2.3	Conclusion	20
2.4	Ressources	20
II	Les bases de l'algorithmique et de la programmation	21
3	Spécifier le problème	23
3.1	Déterminer les données et le résultat	23
3.2	Les noms	24
3.3	Les types	25
3.4	Résumés	27
3.5	Exemples numériques	28
4	Premiers algorithmes	29
4.1	Exercice résolu : un problème simple	31
4.2	Décomposer les calculs ; variables et assignation	35
4.3	Quelques difficultés liées au calcul	39
4.4	Des algorithmes et des programmes de qualité	46
4.5	Améliorer la lisibilité	49
4.6	Appel d'algorithme, appel de méthode	53
4.7	Interagir avec l'utilisateur	54
5	Premiers programmes	57
5.1	Introduction	57
5.2	Environnement de développement	61
5.3	La grammaire du langage	63
6	Une question de choix	65
6.1	Le si (<i>if-then</i>)	65
6.2	Le si-sinon (<i>if-then-else</i>)	68
6.3	Le si-sinon-si	71
6.4	Expression booléenne	73

6.5	Le selon-que (<i>switch</i>)	74
7	Module et références	77
7.1	Décomposer le problème	77
7.2	Exemple	78
7.3	Paramètres et valeur de retour	79
7.4	Type primitif et type référence	82
7.5	Résumons	86
8	Un travail répétitif	89
8.1	La notion de travail répétitif	89
8.2	Une même instruction, des effets différents	90
8.3	Le « tant que » (<i>while</i>)	92
8.4	Le « pour » (<i>for</i>)	94
8.5	« faire – tant que »	98
8.6	Quel type de boucle choisir ?	99
8.7	Exercices résolus	100
III	Les tableaux	109
9	Les tableaux	111
9.1	Utilité des tableaux	112
9.2	Définitions	115
9.3	Déclaration - création - initialisation	115
9.4	Tableau et paramètres	118
9.5	Parcours d'un tableau	121
9.6	Taille logique et taille physique	122
9.7	Les tableaux ont comme premier indice 0	123
10	Algorithmes d'insertions et de recherches	125
10.1	Insertion dans un tableau non trié	126
10.2	Insertion dans un tableau trié	130
10.3	La recherche dichotomique	135
10.4	Introduction à la complexité	139
11	Algorithmes de tris	143
11.1	Motivation	143
11.2	Tri par insertion	145
11.3	Tri par sélection des minima successifs	147
11.4	Tri bulle	149
11.5	Cas particuliers	150
11.6	Références	151
IV	Compléments	153
12	Les chaînes et les String	155
12.1	Longueur	156
12.2	Chaîne et caractère	156
12.3	Le contenu d'une chaîne	157

12.4	La concaténation	158
12.5	Manipuler les caractères	159
12.6	L'alphabet	161
12.7	Chaîne et nombre	162
12.8	Extraction de sous-chaînes	162
12.9	Recherche de sous-chaîne	163
13	Les variables structurées et les classes	165
13.1	Structure et classe	166
13.2	Déclaration d'une variable	167
13.3	Utilisation des variables de type structuré	168
13.4	Remarque	171
V	Les annexes	173
A	Exercices	175
A.1	Exercices : spécifier le problème	175
A.2	Exercices : premiers algorithmes et programmes	178
A.3	Exercices : tracer un algorithme	178
A.4	Exercices : division entière, quotient et reste	180
A.5	Exercices : structures alternatives (choix)	183
A.6	Exercices : modules et méthodes	187
A.7	Exercices : structures itératives	190
A.8	Exercices : tableaux	196
A.9	Exercices : chaîne et String	201
A.10	Exercices : structures et classes	203
A.11	Exercices récapitulatifs	204
B	Les fiches	211
	Un calcul simple	212
	Un calcul complexe	213
	Un nombre pair	215
	Maximum de deux nombres	216
	Passage d'un tableau en paramètre	217
	Parcours complet d'un tableau	219
	Parcours partiel d'un tableau	220
	Maximum dans un tableau	222
	Tableau non trié	223
	Tableau trié	225
	Recherche dichotomique	228
	Tri d'un tableau	229
C	Bonnes pratiques	233
C.1	Tester un booléen	233
C.2	Assigner un booléen	234
C.3	Assigner une valeur en fonction d'une condition	235
C.4	Interrompre une boucle for	235
C.5	Plusieurs return	236

Première partie

Introduction générale

Chapitre 1

Résoudre des problèmes

« *L’algorithmique est le permis de conduire de l’informatique. Sans elle, il n’est pas concevable d’exploiter sans risque un ordinateur.* »⁸

Ce chapitre a pour but de vous faire comprendre ce qu’est une *procédure de résolution de problèmes*.

Contenu

1.1	La notion de problème	9
1.1.1	Préliminaires : utilité de l’ordinateur	9
1.1.2	Poser le problème	10
1.2	Procédure de résolution	11
1.2.1	Chronologie des opérations	11
1.2.2	Les opérations élémentaires	11
1.2.3	Les opérations bien définies	12
1.2.4	Opérations soumises à une condition	13
1.2.5	Opérations à répéter	13
1.2.6	À propos des données	13
1.3	Ressources	14

1.1 La notion de problème

1.1.1 Préliminaires : utilité de l’ordinateur

L’ordinateur est une machine. Mais une machine intéressante dans la mesure où elle est destinée d’une part, à nous décharger d’une multitude de tâches peu valorisantes, rébarbatives telles que le travail administratif répétitif, mais surtout parce qu’elle est capable de nous aider, voire nous remplacer, dans des tâches plus ardues qu’il

⁸. [CORMEN e.a., Algorithmique, Paris, Edit. Dunod, 2010, (Cours, exercices et problèmes), p. V]

nous serait impossible de résoudre sans son existence (conquête spatiale, prévision météorologique, jeux vidéo...).

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

1.1.2 Poser le problème

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

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

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

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

étant donné [la situation de départ] **on demande** [l'objectif]

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

Exercice Un problème flou.

Soit le problème suivant : « Calculer la moyenne de nombres entiers. ».

Ce problème est-il bien posé ?

Expliquez pourquoi cet énoncé n'est pas bon.

Proposez un énoncé qui soit acceptable.

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

1.2 Procédure de résolution

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

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

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

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

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

Traduite en termes informatiques, une telle procédure d'exécutions d'actions dans un contexte précis, sera appelée un **algorithme**. Nous y reviendrons et le définirons tout à fait précisément plus loin, dans notre contexte.

1.2.1 Chronologie des opérations

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

1.2.2 Les opérations élémentaires

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

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

1.2.3 Les opérations bien définies

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

Des instructions floues de ce genre sont dangereuses à faire figurer dans une bonne marche à suivre car elles font appel à une appréciation arbitraire de l'exécutant. Le résultat obtenu risque d'être imprévisible d'une exécution à l'autre. De plus, les termes du type *environ, beaucoup, pas trop et à peu près* sont intraduisibles et proscrites au niveau d'un langage informatique!⁹

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

Afin de mettre en évidence la difficulté d'écrire une marche à suivre claire et non ambiguë, l'expérience suivante a été menée et vous pouvez la reproduire.

Expérience. Le dessin.

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

1. Chaque personne prend une feuille de papier et y dessine quelque chose en quelques traits précis. Le dessin ne doit pas être trop compliqué ; on ne teste pas ici vos talents de dessinateur ! (ça peut être une maison, une voiture...)
2. Sur une **autre** feuille de papier, chacun rédige des instructions permettant de reproduire fidèlement son propre dessin. Attention ! Il est important de ne **jamais faire référence à la signification du dessin**. Ainsi, on peut écrire : « dessine un rond » mais certainement pas : « dessine une roue ».
3. Chacun cache à présent son propre dessin et échange sa feuille d'instructions avec celle de quelqu'un d'autre.
4. Chacun s'efforce ensuite de reproduire le dessin d'un autre en suivant **scrupuleusement** les instructions indiquées sur la feuille reçue en échange, **sans**

9. Toute personne intéressée découvrira dans la littérature spécialisée que même les procédures de génération de nombres aléatoires sont elles aussi issues d'algorithmes mathématiques tout à fait déterministes. Des algorithmes dont la sortie ne dépend que des paramètres, c'est-à-dire qui, étant donné des entrées données, fournira systématiquement la même sortie.

tenter d'initiative (par exemple en croyant avoir compris ce qu'il faut dessiner).

5. Nous examinerons enfin les différences entre l'original et la reproduction et nous tenterons de comprendre pourquoi elles se sont produites (par imprécision des instructions ou par mauvaise interprétation de celles-ci par le dessinateur...)

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



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

1.2.4 Opérations soumises à une condition

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

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

1.2.5 Opérations à répéter

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

Dans certains cas, le nombre de répétitions est connu à l'avance (*répéter 10 fois*) ou déterminé par une durée (*faire cuire pendant 30 minutes*) et dans d'autres cas il est inconnu. Dans ce cas, la fin de la période de répétition d'un bloc d'opérations dépend alors de la réalisation d'une condition (*lancer le dé jusqu'à ce qu'il tombe sur 6*, *faire chauffer jusqu'à évaporation complète*...).

En informatique, lorsqu'il y a répétition organisée, on parle de **boucle**. On en retrouve beaucoup, sous différentes formes, dans les codes informatiques.

L'exemple ci-dessous permet d'illustrer le danger de boucle infinie, due à une mauvaise formulation de la condition d'arrêt. Si l'on demande de *lancer le dé — de 6 faces — jusqu'à ce que la valeur obtenue soit 7*, une personne dotée d'intelligence comprend que la condition est impossible à réaliser, mais un robot appliquant cette directive à la lettre lancera le dé éternellement.

1.2.6 À propos des données

Les types d'objets figurant dans les diverses procédures de résolution sont fonction du cadre dans lequel s'inscrivent ces procédures, du domaine d'application de ces

marches à suivre. Par exemple, pour une recette de cuisine, ce sont les ingrédients. Pour un jeu de construction ce sont les briques.

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

1.3 Ressources

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

- ▷ Les Sépas 18 - Les algorithmes¹⁰
- ▷ Les Sépas 11 - Un bug¹¹
- ▷ Le crépier psycho-rigide comme algorithme¹²
- ▷ Le baseball multicolore comme algorithme¹³
- ▷ Le jeu de Nim comme algorithme¹⁴
- ▷ Code Studio¹⁵, un site d'apprentissage et d'initiation à l'algorithmique et la notion de programme.

10. <https://www.youtube.com/watch?v=hG9Jty7P6Es>

11. <https://www.youtube.com/watch?v=deIOGV5sWTY>

12. <https://pixees.fr/?p=446>

13. <https://pixees.fr/?p=450>

14. <https://pixees.fr/?p=443>

15. <http://studio.code.org/>

Chapitre 2

Les algorithmes et les programmes

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

Contenu

2.1	Algorithmes et programmes	15
2.1.1	Algorithme	15
2.1.2	Programme	16
2.1.3	Les constituants principaux de l'ordinateur	17
2.1.4	Exécution d'un programme	18
2.2	Les phases d'élaboration d'un programme	18
2.3	Conclusion	20
2.4	Ressources	20

2.1 Algorithmes et programmes

Décrivons la différence entre un algorithme et un programme et comment un ordinateur peut exécuter un programme.

2.1.1 Algorithme

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

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



Comme toute marche à suivre, un algorithme doit s'exprimer dans un certain langage — à priori le langage naturel — mais il y a d'autres possibilités : ordino-gramme, arbre programmatique, pseudo-code. . .

L'algorithme est destiné à être compris par une personne et à être traduit — ce qui n'est pas immédiat — en un programme.

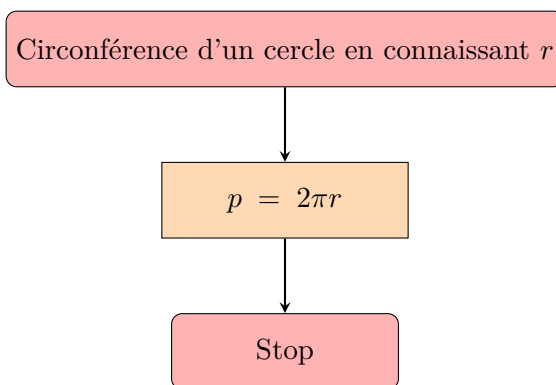
Exemple simple

Calculer la circonférence d'un cercle.

Un algorithme en **langage naturel** pourrait être : en connaissant le rayon, le calcul de la circonférence se fait grâce à

$$p = 2\pi r$$

En utilisant un **organigramme** l'algorithme peut se décrire comme suit (et nous y reviendrons plus longuement plus tard) :



ORGANIGRAMME 1 – Illustration d'un organigramme

En **pseudo-code** cette fois, l'algorithme aurait l'allure suivante :

```

algorithm circleCircumference(radius : real) → real
  return 2 π radius
  
```

pseudocode

2.1.2 Programme



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

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

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

Un exemple simple (suite)

En Java, le programme aurait l'allure suivante mais ne serait pas utilisable en l'état. Il lui manque une méthode principale. Nous y reviendrons :

```
1 public class CircleCircumference{  
    public static double circleCircumference(double radius){  
        return 2 * Math.PI * radius;  
4    }  
}
```

java

2.1.3 Les constituants principaux de l'ordinateur

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

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

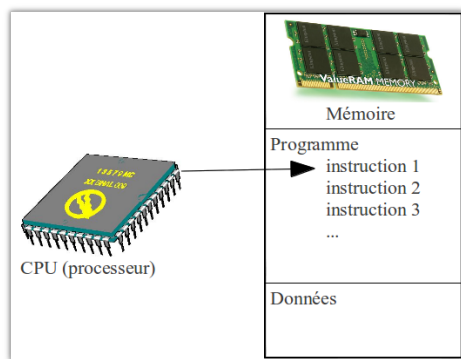
- ▷ **l'organe de contrôle** : c'est le cerveau de l'ordinateur. Il est l'organisateur, le contrôleur suprême de l'ensemble. Il assume l'enchaînement des opérations élémentaires. Il s'occupe également d'organiser l'exécution effective de ces opérations élémentaires reprises dans les programmes.
- ▷ **l'organe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

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

écrit en langage de programmation plus ou moins « évolué » (c'est-à-dire plus ou moins proche du langage naturel).

2.1.4 Exécution d'un programme

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



Comment fonctionne le processeur ?

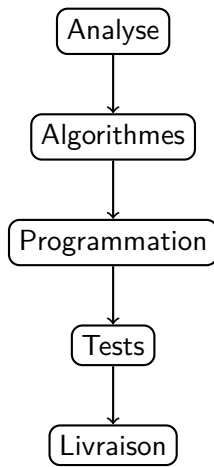
De façon très simplifiée, les étapes suivantes ont lieu :

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

Nous voyons bien qu'il s'agit d'un travail automatique ne laissant aucune place à l'initiative !

2.2 Les phases d'élaboration d'un programme

Voyons pour résumer un schéma **simplifié** des différentes phases nécessaires au développement d'un programme.



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

Notons que ce processus n'est pas linéaire. À chaque phase, on pourra détecter des erreurs, imprécisions ou oublis des phases précédentes et revenir en arrière. Un problème peut être amélioré, complexifié, modifié ou étendu.

Par exemple : Imaginons que je veuille préparer un repas pour mes invités et invitées. Je désire leur proposer une lasagne et décide de suivre une recette. À la dernière minute, j'apprend que l'une des personnes invitées est allergique au lait. Plusieurs choix s'offrent à moi :

- ▷ je modifie ma recette et remplace le lait par du lait de soja ;
- ▷ je change de plat ;
- ▷ je le notifie qu'il ou elle n'est plus le bienvenu ou qu'il recevra une tartine.

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

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

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

à un autre est un peu moins facile à cause des particularités propres à chaque langage.

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

2.3 Conclusion

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

Le but de ce premier cours de **développement** est de mêler l'apprentissage des algorithmes et du langage Java comme premier langage.

Nous tenterons :

- ▷ de définir une bonne démarche d'élaboration d'algorithmes (apprentissage de la **logique** de programmation) ;
- ▷ comprendre et apprendre les algorithmes classiques qui ont fait leurs preuves. Pouvoir les utiliser en les adaptant pour résoudre nos problèmes concrets.
- ▷ traduire ces algorithmes en langage Java et les « faire tourner », c'est-à-dire les implémenter sur une machine ; éditer le code, le compiler, l'exécuter...

Le tout devrait avoir pour résultat l'élaboration de *bons programmes*, c'est-à-dire *des programmes dont il est facile de se persuader qu'ils sont corrects* et des programmes dont la maintenance est la plus aisée possible. Dans ce sens, ce cours est bien un premier cours de développement.

Les matières non traitées dans cette première approche du développement le seront, d'abord dans le cours de Développement II (DEV2) et ensuite, Développement III, IV... en fonction du cursus.

2.4 Ressources

Pour prolonger votre réflexion sur les notions vues dans ce chapitre, nous vous proposons quelques ressources en ligne :

- ▷ C'est pas sorcier ! Ordinateur, tout un programme¹⁶

16. <https://www.youtube.com/watch?v=c96KP5jZVYk>

Deuxième partie

Les bases de l'algorithmique et de la programmation

Chapitre 3

Spécifier le problème

Comme nous l'avons dit, un problème ne sera véritablement bien spécifié que s'il s'inscrit dans le schéma suivant :

étant donné [les données] **on demande** [résultat]

La première étape dans la résolution d'un problème est de préciser ce problème à partir de l'énoncé, c-à-d de déterminer et préciser les données et le résultat.

Contenu

3.1	Déterminer les données et le résultat	23
3.2	Les noms	24
3.3	Les types	25
3.3.1	Il n'y a pas d'unité	26
3.3.2	Préciser les valeurs possibles	26
3.3.3	Le type des données complexes	27
3.3.4	Exercice	27
3.4	Résumés	27
3.4.1	Résumé graphique	28
3.4.2	Résumé textuel	28
3.5	Exemples numériques	28

3.1 Déterminer les données et le résultat

La toute première étape est de parvenir à extraire d'un énoncé de problème, quelles sont les données et quel est le résultat attendu¹⁷. Dans la suite, nous utiliserons un exemple très simple afin d'illustrer les concepts qui nous intéressent.

17. Plaçons-nous pour le moment dans le cadre de problèmes où il y a exactement un résultat.

Exemple. Soit l'énoncé suivant : « Calculer la surface d'un rectangle à partir de sa longueur et sa largeur ».

Quelles sont les données ? Il y en a deux :

- ▷ la longueur du rectangle ;
- ▷ sa largeur.

Quel est le résultat attendu ? la surface du rectangle.

3.2 Les noms

Pour identifier clairement chaque **donnée** et pouvoir y faire référence dans le futur algorithme et dans le programme nous devons lui attribuer un **nom**¹⁸. Il est important de bien choisir les noms. Le but est de trouver un nom qui soit suffisamment court, tout en restant explicite et ne prêtant pas à confusion.

Exemple. Quel nom choisir pour la longueur d'un rectangle ?

On peut envisager les noms suivants :

- ▷ `length` est probablement le plus approprié.
- ▷ `rectangleLength` peut se justifier pour éviter toute ambiguïté avec une autre longueur.
- ▷ `len` peut être admis si le contexte permet de comprendre immédiatement l'abréviation.
- ▷ `l` est à proscrire car pas assez explicite.
- ▷ `theLengthOfMyRectangle` est inutilement long.
- ▷ `foo` (truc en anglais) ou `tmp` ne sont pas de bons choix car ils n'ont aucun lien avec la donnée.

Les noms donnés à chaque donnée seront directement associés à une **variable** lorsque l'algorithme sera traduit en un programme dans un langage de programmation.



Variable : Emplacement mémoire nommé pouvant contenir une valeur. Cette valeur peut être remplacée par une autre — elle est variable — au fil de l'exécution du programme.

Nous allons également donner un **nom à l'algorithme** de résolution du problème. Cela permettra d'y faire référence dans les explications mais également de l'utiliser dans d'autres algorithmes. Généralement, un nom d'algorithme est :

- ▷ soit un verbe indiquant ce que fait l'algorithme ;
- ▷ soit un nom indiquant le résultat fourni.

18. Dans ces notes, nous nous efforcerons de choisir des noms en anglais, mais vous pouvez très bien choisir des noms français si vous ne vous sentez pas encore suffisamment à l'aise avec l'anglais.

Exemple. Quel nom choisir pour l'algorithme qui calcule la surface d'un rectangle ?

On peut envisager le verbe `computeRectangleArea` ou le nom `rectangleArea` (notre préféré). On pourrait aussi simplifier en `area` s'il est évident que le problème traite des rectangles.

Les noms donnés aux algorithmes deviendront généralement les noms des programmes lorsque ces algorithmes seront implémentés.

Notons que les langages de programmation imposent certaines limitations (parfois différentes d'un langage à l'autre) ce qui peut nécessiter une modification du nom lors de la traduction de l'algorithme en un programme. À chaque langage de programmation sont également associées des **conventions d'écriture** que les développeurs respectent. Bien que nous y reviendrons plus en détail dans la section 4.5, notons déjà que la simple convention de nom de méthode change d'un langage à l'autre.

Par exemple pour *rectangle area*, nous utiliserons `rectangleArea` en langage Java mais `rectangle_area` en langage C, C++ et Python.

3.3 Les types

Nous allons également attribuer un **type** à chaque donnée ainsi qu'au résultat. Le **type** décrit la nature de son contenu, quelles valeurs elle peut prendre.

Certains langages imposent et vérifient le type de chaque variable tandis que d'autres non. En Java, chaque variable et chaque donnée ont un type.

Dans un premier temps, nous utiliserons ces **types**¹⁹ :

Algo	Java	
integer	int	pour les nombres entiers
real	double	pour les nombres réels
string	String ²⁰	pour les chaînes de caractères, les textes par exemple : "Bonjour", "Bonjour le monde!", "a", "..."
boolean	boolean	quand la valeur ne peut être que <code>true</code> (vrai) ou <code>false</code> (faux)

¹⁹. Écrire ces différents types en français n'est pas une faute. Dans ces notes nous utiliserons l'anglais

²⁰. Notez la majuscule, indispensable en Java

Exemples.

- ▷ Pour la longueur, la largeur et la surface d'un rectangle, on prendra un réel (le type `real`).
- ▷ Pour le nom d'une personne, on choisira une chaîne de caractère (le type `string`).
- ▷ Pour l'âge d'une personne, un entier est indiqué (`integer`).
- ▷ Pour décrire si un étudiant est doubleur ou pas, un booléen (`boolean`) est adapté.
- ▷ Pour représenter un mois, on préférera souvent un entier donnant le numéro du mois (par ex : 3 pour le mois de mars) plutôt qu'une chaîne (par ex : "mars") car les manipulations, les calculs seront plus simples.

3.3.1 Il n'y a pas d'unité

Un type numérique indique que les valeurs possibles seront des nombres. Il n'y a là aucune notion d'unité. Ainsi, la longueur d'un rectangle, un réel, peut valoir 2.5 mais certainement pas 2.5 cm . Si cette unité a de l'importance, il faut la spécifier dans le nom de la donnée ou en commentaire.

Exemple. Faut-il préciser les unités pour les dimensions d'un rectangle ?

Si la longueur d'un rectangle vaut 6, on ne peut pas dire s'il s'agit de centimètres, de mètres ou encore de kilomètres. Pour notre problème de calcul de la surface, ce n'est pas important ; la surface n'aura pas d'unité non plus.

Notre seule contrainte est que la longueur et la largeur soient exprimées dans la même unité.

Si, par contre, il est important de préciser que la longueur est donnée en centimètres, on pourrait l'expliciter en la nommant `lengthCm`.

3.3.2 Préciser les valeurs possibles

Nous aurions pu introduire un seul type numérique mais nous avons choisi de distinguer les entiers et les réels. Pourquoi ? Préciser qu'une donnée ne peut prendre que des valeurs entières (par exemple dans le cas d'un numéro de mois) aide le lecteur à mieux la comprendre. Nous allons aussi pouvoir définir des opérations propres aux entiers (le reste d'une division par exemple). Enfin, pour des raisons techniques, beaucoup de langages font cette distinction.

Même ainsi, le type choisi n'est pas toujours assez précis. Souvent, la donnée ne pourra prendre que certaines valeurs.

Exemples.

- ▷ Un âge est un entier qui ne peut pas être négatif.
- ▷ Un mois est un entier compris entre 1 et 12.

Ces précisions pourront être données en commentaire pour aider à mieux comprendre le problème et sa solution.

3.3.3 Le type des données complexes

Parfois, aucun des types disponibles ne permet de représenter la donnée. Il faut alors la décomposer.

Exemple. Quel type choisir pour la date de naissance d'une personne ?

On pourrait la représenter dans une chaîne (par ex : "17/3/1985") mais cela rendrait difficile le traitement, les calculs (par exemple, déterminer le numéro du mois). Le mieux est sans doute de la décomposer en trois parties : le jour, le mois et l'année, tous des entiers.

Plus loin dans le cours, nous verrons qu'il est possible de définir de nouveaux types de données grâce aux *structures* pour les algorithmes et aux *classes* pour les langages orientés objets tels que Java.

Nous pourrions alors définir et utiliser un type `Date` et il ne sera plus nécessaire de décomposer une date en trois morceaux bien que le type sera bien entendu composé de trois valeurs.

3.3.4 Exercice

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

- ▷ le prix d'un produit en grande surface ?
- ▷ la taille de l'écran de votre ordinateur ?
- ▷ votre nom ?
- ▷ votre adresse ?
- ▷ le pourcentage de remise proposé pour un produit ?
- ▷ une date du calendrier ?
- ▷ un moment dans la journée ?

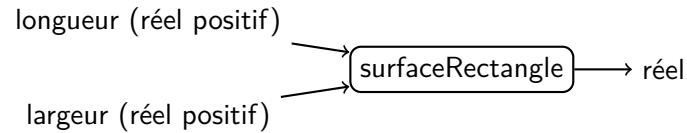
3.4 Résumés

Toutes les informations déjà collectées sur le problème peuvent être résumées.

3.4.1 Résumé graphique

Représentation graphique du problème.

Exemple. Pour le problème, de la surface du rectangle, on fera le schéma suivant :



3.4.2 Résumé textuel

Résumé textuel du problème, indiquant clairement les données et les résultats recherchés.

Exemple.

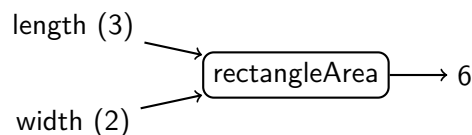
Données	longueur (un réel positif) largeur (un réel positif)
Résultat	la surface du rectangle

3.5 Exemples numériques

Une dernière étape pour vérifier que le problème est bien compris est de donner quelques exemples numériques. Il est possible de les spécifier en français, via un graphique ou via une notation compacte.

Exemples. Voici différentes façons de présenter des exemples numériques pour le problème de calcul de la surface d'un rectangle :

- ▷ En français : si la longueur du rectangle vaut 3 et sa largeur vaut 2, alors sa surface vaut 6.
- ▷ Via un schéma :



- ▷ En notation compacte : `rectangleArea(3, 2)` donne/vaut 6.

Chapitre 4

Premiers algorithmes

Dans le chapitre précédent, vous avez appris à analyser un problème et à clairement le spécifier. Il est temps d'écrire des solutions. Pour cela, nous allons devoir trouver comment passer des données au résultat et l'exprimer dans un langage compris de tous.

En fonction de la difficulté du problème et de notre sensibilité, nous pouvons représenter un algorithme de plusieurs manières ; en français (ou en anglais), en pseudo-code ou avec un organigramme.

Contenu

4.1	Exercice résolu : un problème simple	31
4.1.1	Trouver l'algorithme	31
4.1.2	Vérifier l'algorithme	33
4.1.3	Implémenter l'algorithme en Java	33
4.1.4	Résolution d'un second problème simple	34
4.2	Décomposer les calculs ; variables et assignation	35
4.2.1	Les variables	35
4.2.2	L'assignation	37
4.2.3	Tracer un algorithme	38
4.2.4	Exercice résolu : durée du trajet	39
4.3	Quelques difficultés liées au calcul	39
4.3.1	Un peu de vocabulaire	40
4.3.2	Les comparaisons et les assignations de variables booléennes	41
4.3.3	Les opérations logiques	41
4.3.4	La division entière et le reste	44
4.3.5	Le hasard et les nombres aléatoires	45
4.4	Des algorithmes et des programmes de qualité	46
4.4.1	L'efficacité	47
4.4.2	La lisibilité	47
4.4.3	La rapidité	48
4.4.4	La taille	48
4.4.5	Conclusion	48
4.5	Améliorer la lisibilité	49
4.5.1	Mise en page des algorithmes	49
4.5.2	Écriture des programmes	50
4.5.3	Choix des noms	50
4.5.4	Les commentaires	50
4.5.5	Constantes	52
4.6	Appel d'algorithme, appel de méthode	53
4.7	Interagir avec l'utilisateur	54
4.7.1	Afficher un résultat	54
4.7.2	Demander des valeurs	55
4.7.3	Préférer les paramètres	55

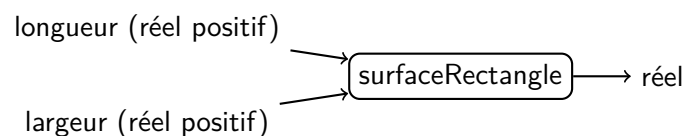
4.1 Exercice résolu : un problème simple

4.1.1 Trouver l'algorithme

Illustrons notre propos sur l'exemple qui a servi de fil conducteur tout au long du chapitre précédent. Rappelons l'énoncé et l'analyse qui en a été faite.

Problème. Calculer la surface d'un rectangle à partir de sa longueur et sa largeur.

Analyse. Nous sommes arrivés à la spécification suivante :



ou encore :

Données	longueur (un réel positif) largeur (un réel positif)
Résultat	la surface du rectangle

Exemples.

▷ rectangleArea(3,2) donne 6 ;

▷ rectangleArea(3.5,1) donne 3.5.

ou

Données		
L	3	3.5
l	2	1
Résultat	6	3.5

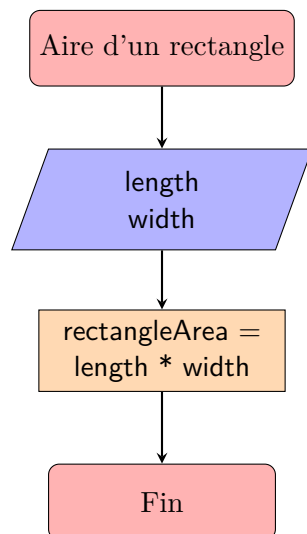
Comment résoudre ce problème ? La toute première étape est de comprendre le lien entre les données et le résultat. Ici, on va se baser sur la formule de la surface :

$$\text{surface} = \text{longueur} * \text{largeur}$$

La surface s'obtient donc en multipliant la longueur par la largeur ²¹.

Un **organigramme** d'une solution aura cette allure :

21. Trouver la bonne formule n'est pas toujours facile. Dans votre vie professionnelle, vous devrez parfois écrire un algorithme pour un domaine que vous connaissez peu, voire pas du tout. Il vous faudra alors chercher de l'aide, demander à des experts du domaine. Dans ce cours, nous essaierons de nous concentrer sur des problèmes simples voire très simples.

ORGANIGRAMME 2 – Solution de `rectangleArea`

Un **pseudo-code** d'une solution s'écrit :

```

algorithm rectangleArea(length, width : reals) → real
  return length * width
  
```

pseudocode

Le mot **algorithm** et l'**indentation** — soulignée par une ligne verticale — permettent de délimiter l'algorithme. La première ligne est appelée **l'entête** de l'algorithme. On y retrouve :

- ▷ le nom de l'algorithme,
- ▷ une déclaration des données, qu'on appellera ici les **paramètres**,
- ▷ le type du résultat.

Les paramètres recevront des valeurs concrètes au **début** de l'exécution de l'algorithme.

L'instruction **return** permet d'indiquer la valeur du résultat, ce que l'algorithme *retourne*. Si on spécifie une formule, un calcul, c'est le **résultat** (on dit l'*évaluation*) de ce calcul qui est retourné et **pas la formule**.

Pour indiquer le calcul à faire, écrivez-le, naturellement comme vous le feriez en mathématique.

Pour demander l'**exécution** d'un algorithme (on dit aussi *appeler*) il suffit d'indiquer son nom et les valeurs concrètes à donner aux paramètres. Ainsi, `rectangleArea(6,3)` fait appel à l'algorithme correspondant pour calculer la surface d'un rectangle dont la longueur est 6 et la largeur est 3.

Dans ces notes, nous utiliserons « \rightarrow » pour montrer ce que retourne l'algorithme mais « : » ou un simple « \square »²² font l'affaire.

22. Ce symbole signifie : « ici se trouve un espace »

4.1.2 Vérifier l'algorithme

Une étape importante, après l'écriture d'un algorithme est la vérification de sa validité. Il est important d'exécuter l'algorithme avec des exemples numériques et vérifier que chaque réponse fournie est correcte.

Pour tester un algorithme il faut — même si ça paraît étrange — **éteindre son cerveau**. Il faut agir comme une machine et exécuter **ce qui est écrit** pas ce que l'on voulait écrire ou ce que l'on pensait avoir écrit ou encore ce qu'il est censé faire. Cela demande un peu de pratique.

Exemple. Vérifions notre solution pour le calcul de la surface du rectangle en reprenant les exemples choisis.

test n°	longueur	largeur	réponse attendue	réponse fournie	
1	3	2	6	6	✓
2	3.5	1	3.5	3.5	✓

4.1.3 Implémenter l'algorithme en Java

Maintenant que l'algorithme est écrit, il reste à l'implémenter en Java. Une solution en Java s'écrit :

```
public class RectangleArea{  
    public static double rectangleArea(double length, double width){  
3        return length * width;  
    }  
}  
6
```

java

Telle quelle la solution n'est pas fonctionnelle en ce sens que l'exécution du programme ne montrera aucun résultat à l'écran.

Le **point d'entrée** d'un programme en Java est la méthode **main**. Cette méthode est la première qui sera exécutée. Elle est obligatoire si l'on veut pouvoir exécuter le programme.

Au minimum, si l'on veut calculer et afficher l'aire d'un rectangle de longueur 3 et de largeur 2, il faudrait plutôt écrire :

```

public class RectangleArea{
    public static double rectangleArea(double length, double width){
3      return length * width;
    }

    6  public static void main(String[] args){
        System.out.println(rectangleArea(3,2));
    }
    9 }

```

java

**Attention :**

Écrire une solution complète d'un exercice c'est :

- ▷ spécifier le problème ;
- ▷ fournir des exemples ;
- ▷ écrire un algorithme ;
- ▷ vérifier les exemples ;
- ▷ écrire un programme correspondant à l'algorithme ;
- ▷ tester le programme et constater qu'il fournit bien les résultats attendus

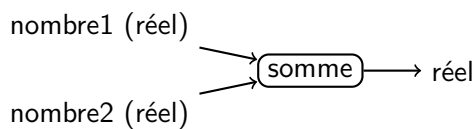
4.1.4 Résolution d'un second problème simple



Vous pouvez vous baser sur la fiche 1 page 212 qui résume la résolution du calcul de la surface d'un rectangle, depuis l'analyse de l'énoncé jusqu'à l'algorithme et à sa vérification.

Problème. Calculer la somme de deux nombres donnés.

Analyse.



ou :

Données	nombre 1 (un nombre réel quelconque) nombre 2 (un autre nombre réel)
Résultat	la somme des deux nombres

Algorithme L'algorithme s'écrit simplement :

```
algorithm addition(number1, number2 : reals) → real
    return number1 + number2
```

pseudocode

Il est aussi assez facile de vérifier qu'il fournit bien les bonnes réponses pour les exemples choisis.

test n°	nombre1	nombre2	réponse attendue	réponse fournie	
1	3	2	5	5	✓
2	-3	2	-1	-1	✓
3	3	2.5	5.5	5.5	✓
4	-2.5	2.5	0	0	✓

Programme La traduction en un programme Java est très semblable au premier exemple.

```
public class Addition{
2   public static double add(double number1, double number2){
        return number1 + number2;
    }
5 }
```

java

4.2 Décomposer les calculs ; variables et assignation

Pour pouvoir décomposer un calcul un peu long en plusieurs étapes nous devons introduire deux nouvelles notions : les *variables locales* et *l'assignation*.

Par exemple le calcul d'un prix TTC (toutes taxes comprises) d'un ensemble de produits dont on connaît la quantité et le prix unitaire hors taxe, peut se décomposer en le calcul du prix unitaire TTC et, ensuite, en le calcul du prix total.

4.2.1 Les variables

Une **variable** est une zone mémoire munie d'un nom et qui contiendra une valeur d'un type donné. Cette valeur *peut* évoluer au fil de l'avancement de l'algorithme ou du programme. Elle sert à retenir des étapes intermédiaires de calculs.

Les variables peuvent être **locales** ou **globales**.



- ▷ Une **variable locale** n'est connue et utilisable qu'au sein de l'algorithme où elle est déclarée.
En Java, une variable locale n'est connue que dans le *bloc* d'instructions dans lequel elle est déclarée. Un bloc d'instructions est un ensemble d'instructions délimitées par une paire d'accolades.
- ▷ Une **variable globale** est connue dans tous les algorithmes d'un même problème.
En Java, une variable globale est connue de toutes les classes et les méthodes d'un même projet. Nous y reviendrons.

Dans nos algorithmes et dans nos programmes nos variables seront toujours locales.

Pour être utilisable, une variable doit être *déclarée* au début de l'algorithme. La **déclaration** d'une variable est l'instruction qui définit son nom et son type. On pourrait écrire :

longueur et largeur seront les noms de deux objets destinés à recevoir
les longueur et largeur du rectangle, c'est-à-dire des nombres à valeurs réelles.

pseudocode

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

réels length, width

pseudocode

Certains langages de programmation imposent que les variables soient déclarées — c'est le cas de Java — et le vérifient tandis que d'autres permettent d'utiliser des variables sans les déclarer.

Dans ce premier cours de développement, nous déclarerons toujours nos variables avant de les utiliser. Principalement dans un souci de lisibilité et également car ceci évite des erreurs lors de l'élaboration des algorithmes et des programmes.

En langage Java, cela donne :

```
double length, width;
```

java


Par convention, Java privilégie une déclaration par ligne, comme ceci :

```
1 double length;  
double width;
```

java

Pour choisir le nom d'une variable, les règles sont les mêmes que pour les données d'un problème.

4.2.2 L'assignation

L'**assignation** (on dit aussi *affectation interne*) est une instruction qui donne une valeur à une variable ou la modifie. 

Cette instruction est probablement la plus importante car c'est ce qui permet de retenir les résultats de calculs intermédiaires.

En algorithmique, nous écrivons :


```
variableName = expression
```

pseudocode

En Java, nous utiliserons exactement le même symbole = :

```
variableName = <une expression>
```

java

Une **expression** est un calcul faisant intervenir des variables, des valeurs explicites et des opérateurs (comme +, -, <...). Une expression a une **valeur**. 

Exemples. Quelques assignations correctes en algorithmique :

```
denRes = den1 * den2
count = count + 1
average = (number1 + number2) / 2
isALowerthanB = a < b           // pour une variable logique
aString = "hello"
aString = hello                 // quelle différence avec la précédente ?
```

pseudocode

Et d'autres qui ne le sont pas :

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



pseudocode

Et ces même exemples — corrects — en Java :

```
1 denRes = den1 * den2;
  count = count + 1;
  average = (number1 + number2) / 2;
4 isALowerthanB = a < b;
  aString = "hello";
  aString = hello;
```

7

java

Remarques

- ▷ Une assignation n'est ni une égalité, ni une définition.

Ainsi, l'assignation $\text{cpt} = \text{cpt} + 1$ ne veut pas dire que cpt et $\text{cpt} + 1$ sont égaux, ce qui est mathématiquement faux mais que la *nouvelle* valeur de cpt doit être calculée en ajoutant 1 à sa valeur actuelle.

Ce calcul doit être effectué au moment de l'exécution de cette instruction.

- ▷ Seules les variables déclarées peuvent être affectées.
- ▷ Toutes les variables apparaissant dans une expression doivent avoir été affectées préalablement. Le contraire provoquerait une erreur, un arrêt de l'algorithme ou du programme.
- ▷ La valeur affectée à une variable doit être compatible avec son type. Pas question de mettre une chaîne dans une variable booléenne.
- ▷ Certaines personnes utilisent \leftarrow comme symbole d'assignation pour les algorithmes. C'est très bien aussi.

Nous utiliserons dans ce cours le symbole $=$ afin d'être plus proche du langage.

4.2.3 Tracer un algorithme

Pour vérifier qu'un algorithme est correct, le développeur ou la développeuse sera souvent amené · e à le tracer.

Tracer un algorithme ou un programme consiste à suivre l'évolution des variables à chaque étape ou à chaque instruction et à vérifier qu'elles contiennent bien à tout moment la valeur attendue.

Dans le cadre de nos algorithmes, ce traçage se fait sur papier comme dans les exemples ci-dessous. Par contre dans la panoplie des outils de développement liés à un langage de programmation apparaît toujours un débogueur. Un débogueur est un outil aidant le ou la développeur · se à trouver ses erreurs. La première manière de faire étant de tracer son code. Il est donc tout à fait possible d'exécuter un programme instruction par instruction et de voir le contenu des variables à chaque étape. Nous ne pouvons que vous encourager à le faire.

Exemple. Traçons des extraits d'algorithmes.

1: integer a, b, c
 2: a = 12
 3: b = 5
 4: c = a - b
 5: a = a + c
 6: b = a

pseudocode

#	a	b	c
1	indéfini	indéfini	indéfini
2	12		
3		5	
4			7
5	19		
6		19	

<pre> 1: integer a, b, c 2: a = 12 3: c = a - b 4: d = c - 2 </pre> <p><i>pseudocode</i></p>	#	a	b	c
	1	indéfini	indéfini	indéfini
	2	12		
	3			???
	4			???

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

4.2.4 Exercice résolu : durée du trajet

Savoir, face à un cas concret, s'il est préférable de décomposer le calcul ou pas, n'est pas toujours évident. La section 4.5 page 49 sur la lisibilité vous apportera des arguments qui permettront de trancher.

L'exercice suivant est suffisamment complexe pour mériter une décomposition du calcul.

Exercice : Durée du trajet Étant donné la vitesse moyenne non nulle en **m/s** d'un véhicule et la distance parcourue en **km** par ce véhicule, calculer la durée en secondes du trajet de ce véhicule.

Nous vous proposons de rédiger une solution complète.

Vous pouvez vérifier ensuite sur la fiche 2 page 213 qui présente cette solution.

4.3 Quelques difficultés liées au calcul

Vous êtes habitués à effectuer des calculs. L'expérience nous montre toutefois que certains calculs vous posent des difficultés. Soit parce que ce sont des opérations que vous utilisez peu, soit parce que vous n'avez pas l'habitude de les voir comme des calculs. Citons :

- ▷ assigner des valeurs booléennes en fonction de comparaisons ;
- ▷ manipuler les opérateurs logiques ;
- ▷ utiliser la division entière et le reste.

Parfois, le problème se situe au niveau de la compréhension du vocabulaire. Examinons ces situations une à une en fournissant des exemples et des exercices pour que cela devienne naturel.

4.3.1 Un peu de vocabulaire

La première difficulté que rencontre un ou une apprenant·e est généralement liée au vocabulaire utilisé. Qu'est-ce qu'un opérateur ? Un opérande ? Rappelons et fixons ces notions.



expression

Une expression indique un calcul à effectuer (par exemple : $(a + b) * c$). Une fois le calcul effectué (on dit qu'on *évalue* l'expression), on obtient une valeur, d'un certain type. Une expression est composée d'opérandes et d'opérateurs, elle a une valeur et un type.

opérateur

Un opérateur est ce qui désigne une opération. Exemple : $+$ désigne l'addition.

opérande

Un opérande est ce sur quoi porte l'opération. Exemple : dans l'expression $a+b$, a et b sont les opérandes. Un opérande peut être une sous-expression. Exemple : dans l'expression $(a+b) * c$, $(a+b)$ est l'opérande de gauche de l'opérateur $*$.

unaire, binaire, ternaire

Un opérateur qui agit sur deux opérandes (le plus fréquent) est qualifié de binaire. On rencontre aussi des opérateurs unaires (ex : le $-$ dans l'expression $-a$). En Java, vous rencontrerez aussi un opérateur ternaire (3 opérandes) mais ils sont plus rares.

littéral

Un littéral est une valeur notée explicitement (comme 12, 34.4, "bonjour")

priorité

Les opérateurs sont classés par priorité. Cela permet de savoir dans quel ordre les exécuter. Par exemple, la multiplication est prioritaire par rapport à l'addition. C'est pourquoi l'expression $a + b * c$ est équivalente à $a + (b * c)$ et pas à $(a + b) * c$. Les parenthèses permettent de modifier ou de souligner la priorité.

Exercice : analyse d'expression Voici une série d'expressions. Nous vous proposons d'identifier tous les opérateurs et leurs opérandes, d'indiquer si les opérateurs sont unaires ou binaires et d'identifier les littéraux.

Nous vous proposons aussi de fournir une version de l'expression avec le moins de parenthèses possibles et une autre avec un maximum de parenthèses (tout en respectant le sens de l'expression bien sûr et sans mettre de parenthèses redondantes).

▷ $a+1$

▷ $(a+b)*12-4*(-a-b)$

▷ $a+(b*12)-4*-a$

4.3.2 Les comparaisons et les assignations de variables booléennes

$3 + 1$ est un calcul dont le résultat est 4, un entier. C'est sans doute évident.

$1 < 3$ est aussi un calcul dont le résultat est un *booléen*, vrai en l'occurrence. Ce résultat peut être assigné à une variable booléenne.

Exemples. Voici quelques assignations correctes

```
boolean positive, adult, successful, perfect
positive = nb > 0           // positive est mis à vrai si le nb est positive
adult = age ≥ 21           // adult est vrai si l'âge est 21 ou plus
successful = cote ≥ 10      // successful est mis à vrai si la cote est
                           // supérieure ou égale à 10
perfect = errors == 0       // c'est perfect si le nombre de fautes est 0
```

pseudocode

En Java

```
boolean positive,
2  adult,
   successful,
   perfect;
5 positive = nb > 0;
  adult = age >= 21;
  successful = code >= 10;
8 perfect = errors == 0;
```

java

Remarque Pour tester l'égalité, nous utilisons le symbole `==` afin de le différencier du symbole `=` qui est le symbole de d'assignation. C'est la même chose en Java et dans beaucoup d'autres langages.

Exercices : écrire des expressions booléennes Pour chacune des phrases suivantes, écrivez l'assignation qui lui correspond.

- ▷ La variable booléenne *négatif* doit indiquer si le nombre *montant* est négatif.
- ▷ Un groupe est complet s'il contient exactement 20 personnes.
- ▷ Un algorithme est considéré comme long si le nombre de lignes dépasse 20.
- ▷ Un étudiant a *la plus grande distinction* si sa cote est de 18/20 ou plus.

4.3.3 Les opérations logiques

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

opérateur	nom	description
NON	négation	vrai devient faux et inversement
AND	conjonction logique	vrai si les 2 conditions sont vraies
OR	disjonction logique	vrai si au moins une des 2 conditions est vraie

Remarque Nous utilisons AND et OR mais nous acceptons et comprenons ET et OU.

Ce tableau se résume en *tables de vérité* comme suit :

a	b	a ET b
V	V	V
V	F	F
F	V	F
F	F	F

a	b	a OU b
V	V	V
V	F	V
F	V	V
F	F	F

a	NON a
V	F
F	V

Ces opérateurs peuvent intervenir dans des expressions booléennes.

Exemples

- ▷ tarifPlein = $18 \leq \text{âge} \text{ ET } \text{âge} < 60$
- ▷ distinction = $14 \leq \text{cote} \text{ ET } \text{cote} < 16$
- ▷ nbA3chiffres = $100 \leq \text{nb} \text{ ET } \text{nb} \leq 999$
- ▷ tarifRéduit = NON tarifPlein
- ▷ tarifRéduit = NON ($18 \leq \text{âge} \text{ ET } \text{âge} < 60$)
- ▷ tarifRéduit = $\text{âge} < 18 \text{ OU } 60 \leq \text{âge}$

En Java, le AND (ET) s'écrit `&&`²³ et le OR (OU), `||`²⁴.

Écrire des calculs utilisant ces opérateurs n'est pas facile car le français nous induit souvent en erreur en nous poussant à utiliser un ET pour un OU et inversement ou bien à utiliser des raccourcis d'écriture ambigus²⁵.

Par exemple, ne pas écrire : `tarifRéduit = âge < 18 OU ≥ 60`

Loi de De Morgan. Cette loi s'énonce :

« La négation d'une conjonction (AND) de deux propositions est la disjonction (OR) des deux négations. De même, la négation de d'une disjonction de deux propositions est la conjonction des deux négations. »

23. & est le caractère *ampersand* (esperluette en français)

24. | est le caractère *pipe* (barre verticale en français)

25. Vous noterez que le nombre de "et" et de "ou" dans cette phrase ne facilite pas sa compréhension ;)

... et s'écrit plus simplement :

$$\text{NON } (a \text{ ET } b) \Leftrightarrow \text{NON } a \text{ OU NON } b$$

$$\text{NON } (a \text{ OU } b) \Leftrightarrow \text{NON } a \text{ ET NON } b$$

Par exemple, ces trois propositions sont équivalentes :

$$\text{tarifRéduit} = \text{NON } (18 \leq \text{âge} \text{ AND } \text{âge} < 60)$$

$$\text{tarifRéduit} = (\text{NON } 18 \leq \text{âge}) \text{ OR } (\text{NON } \text{âge} < 60)$$

$$\text{tarifRéduit} = \text{âge} < 18 \text{ OR } 60 \leq \text{âge}$$

Priorités et parenthèses. L'opérateur NON est prioritaire sur les opérateurs AND et OR.

L'opérateur AND est prioritaire sur le OR.

Ainsi l'expression : NON a OR b AND c doit se comprendre : (NON a) OR (b AND c). Il est toujours possible d'ajouter des parenthèse pour aider à la compréhension.

Évaluation paresseuse L'évaluation paresseuse (*lazy evaluation*) ou évaluation court-circuitée est une évaluation qui s'arrête dès que le résultat est connu.

Les opérateurs AND et OR sont des opérateurs court-circuités. En particulier, si la première partie d'un AND est fausse, il n'est pas nécessaire de regarder la deuxième opérande ; le résultat sera faux. De même si la première partie d'un OR est vraie ; le résultat sera vrai quelle que soit la valeur de la deuxième partie.

Cette manière de fonctionner permet de gagner du temps dans l'évaluation et permet aussi d'éviter des erreurs. Ceci est vrai dans les algorithmes et — surtout — dans beaucoup de langages de programmation. Java utilise cette évaluation paresseuse.

Exemples.

```
ok = 1/b < 0.1      // provoque une erreur et un arrêt de l'algorithme si b=0.
ok = b≠0 ET 1/b < 0.1 // donne la valeur faux à ok si b=0 (court-circuit).
ok = 1/b < 0.1 ET b≠0 // provoque une erreur et un arrêt si b=0.
```

pseudocode

Cette propriété sera abondamment utilisée dans le parcours de tableaux par exemple.

Exercice : simplifier des expressions booléennes Voici quelques assignations correctes du point de vue de la syntaxe mais contenant des lourdeurs d'écriture. Trouvez des expressions plus simples qui auront un effet équivalent.

- ▷ beautifulBoolean = adulte == vrai
- ▷ beautifulBoolean = adulte == faux
- ▷ beautifulBoolean = etudiant == vrai AND jeune == faux
- ▷ beautifulBoolean = NON (adulte == vrai) AND NON (adulte == faux)

▷ nbA3chiffres = NON (nb < 100 OR nb ≥ 1000)

Exercice : expressions logiques Pour chacune des phrases suivantes, écrivez l'assignation qui lui correspond.

- ▷ J'irai au cinéma si le film me plaît et que j'ai 20€ en poche.
- ▷ Je n'irai pas au cinéma si je n'ai pas 20€ en poche.
- ▷ Je broserai le premier cours de la journée s'il commence à 8h et aussi si je n'ai pas dormi mes 8h.

4.3.4 La division entière et le reste



La **division entière** consiste à effectuer une division en ne gardant que la partie entière du résultat. Le **reste** de la division entière de a par b est ce qui n'a pas été repris dans la division... ce qu'il reste.

Lorsque l'on fait une division entière, le **dividende** D est divisé par le **diviseur** d pour donner un **quotient** q et, éventuellement, un **reste** r .

$$D = d * q + r$$

$$\frac{D}{d} = q$$

et, éventuellement un reste r

Dans nos algorithmes, nous noterons la division entière DIV et le reste MOD. a DIV b est le quotient de la division et a MOD b — nous dirons *a modulo b* — est le reste de cette division.

En Java, la division se note $/$ et le reste $\%$.

Par exemple, imaginons qu'une classe comprenne 14 étudiants et étudiantes qu'il faut réunir par 3 dans le cadre d'un travail de groupe. Il est possible de former 4 groupes mais il restera 2 étudiants et étudiantes ne pouvant former un groupe complet. C'est le reste de la division de 14 par 3.

Exemples :

- | | |
|------------------|------------------|
| ▷ 7 DIV 2 vaut 3 | ▷ 7 MOD 2 vaut 1 |
| ▷ 8 DIV 2 vaut 4 | ▷ 8 MOD 2 vaut 0 |
| ▷ 6 DIV 6 vaut 1 | ▷ 6 MOD 6 vaut 0 |
| ▷ 6 DIV 7 vaut 0 | ▷ 6 MOD 7 vaut 6 |

Utilité - Tester la divisibilité

Les deux opérateurs MOD et DIV, respectivement $/$ et $\%$, permettent de tester si un nombre est un multiple d'un autre.

Si l'on veut par exemple savoir si un nombre est **pair** — pour rappel, un nombre pair est un multiple de 2 — il suffit de vérifier que le reste de la division par 2 est nul.

$$\text{nb pair} \equiv \text{nb divisible par 2} \equiv \text{nb MOD } 2 = 0$$

En supposant que `isOdd` (*odd* pour pair et *even* pour impair) est une variable booléenne, nous pourrions donc écrire : `isOdd = nb MOD 2 == 0`.

Et, en langage java :

```
boolean isOdd;
isOdd = nb % 2 == 0;
3
```

java

Utilité - Extraire les chiffres d'un nombre

Faisons une petite expérience numérique.

calcul	résultat	calcul	résultat
65536 MOD 10	6	65536 DIV 10	6553
65536 MOD 100	36	65536 DIV 100	655
65536 MOD 1000	536	65536 DIV 1000	65
65536 MOD 10000	5536	65536 DIV 10000	6

Nous voyons que les divisions entières (DIV) et les restes (MOD) avec des puissances de 10 permettent de conserver les chiffres de droite (division entière, MOD) ou d'enlever les chiffres de droite (DIV). Combinés, ils permettent d'extraire n'importe quel chiffre d'un nombre.

Exemple : $(65536 \text{ DIV } 100) \text{ MOD } 10 = 5$.

4.3.5 Le hasard et les nombres aléatoires

Il existe de nombreuses applications qui font intervenir le hasard.

Par exemple dans les jeux où il est nécessaire de mélanger des cartes, lancer des dés, faire apparaître des ennemis de façon aléatoire...

Le vrai hasard n'existe pas en informatique puisqu'il s'agit de suivre des étapes précises dans un ordre fixé et que la machine est déterministe. Pourtant, on peut concevoir des algorithmes et des programmes qui *simulent* le hasard²⁶. À partir

²⁶. Pour être précis, nous devrions dire pseudo-hasard ou algorithmes et programmes pseudo-aléatoires.

d'un nombre donné²⁷ (appelé *graine* ou *seed* en anglais) ils fournissent une suite de nombres qui *ont l'air* aléatoires.

Concevoir de tels algorithmes est très compliqué et dépasse largement le cadre de ce cours. Heureusement, la plupart des langages informatiques proposent de base une façon d'obtenir un tel nombre aléatoire.

En Java, il suffit d'écrire²⁸

```
int number = Math.random();
```

java

pour obtenir un nombre (pseudo-)aléatoire compris entre 0 et 1 strictement. C'est-à-dire strictement plus petit que 1.

Dans nos algorithmes, nous pouvons toujours supposer qu'il existe un tel algorithme sans devoir l'écrire.

random → réel (entre 0 inclus et 1 exclu)

À partir de cet algorithme et de cette méthode découlent ces deux autres « algorithmes » :

▷ un entier entre 0 inclus et n exclu

```
algorithm random(n : integer) → integer
| return random() * n                // le nombre réel est tronqué
```

pseudocode

▷ un entier entre min et max inclus

```
algorithm random(min : integer, max : integer) → integer
| return random(max-min+1) + min
```

pseudocode

4.4 Des algorithmes et des programmes de qualité

Dans la section précédente, nous avons vu qu'il est possible de décomposer un calcul en étapes. Mais quand faut-il le faire ? Ou, pour poser la question autrement :

²⁷. Ce nombre peut être fixé ou généré à partir de l'environnement (par exemple, l'horloge interne).

²⁸. Nous verrons plus tard qu'il existe une autre manière de faire en utilisant la classe `Random`

Puisqu'il existe plusieurs algorithmes qui résolvent un problème, lequel préférer ? Quelle traduction dans un langage de programmation (Java) privilégier ?

Répondre à cette question, c'est se demander ce qui fait la qualité d'un algorithme ou d'un programme informatique. Quels sont les critères qui permettent de juger ?

C'est un vaste sujet mais nous voudrions aborder les principaux.

La connaissance de l'algorithmique permet d'écrire des algorithmes de qualité. La connaissance d'un langage — qui est une autre compétence — permet d'écrire des programmes de qualité. Les qualités de l'un n'étant pas nécessairement celles de l'autre.

4.4.1 L'efficacité

L'**efficacité** désigne le fait que l'algorithme (le programme) résout²⁹ bien le problème donné. C'est un minimum !

4.4.2 La lisibilité

La **lisibilité** indique si une personne qui lit l'algorithme ou le programme peut facilement percevoir comment il fonctionne. C'est crucial car un algorithme ou un programme est **souvent lu** par de nombreuses personnes :

- ▷ celles qui doivent se convaincre de sa validité avant de passer à la programmation ;
- ▷ celles qui doivent trouver les causes d'une erreur lorsque celle-ci a été rencontrée³⁰ ;
- ▷ celles qui doivent faire évoluer l'algorithme ou le programme suite à une modification du problème ;
- ▷ et, accessoirement, celles qui doivent le coter ;)

C'est un critère **très important** qu'il ne faut surtout pas sous-évaluer. Vous en ferez d'ailleurs l'amère expérience : si vous négligez la lisibilité de votre algorithme ou de votre programme, vous-même ne le comprendrez plus quand vous le relirez quelque temps plus tard !

Comparer la lisibilité de deux algorithmes ou de deux programmes n'est pas une tâche évidente car c'est une notion subjective. Il faut se demander quelle version va être le plus facilement comprise par la majorité des lecteurs. La section 4.5 page 49 explique ce qui peut être fait pour rendre ses algorithmes plus lisibles. À ces recommandations s'ajouteront les **conventions d'écriture** propres à chaque langage qu'il faudra aussi respecter.

29. À ne pas confondre avec *l'efficacité* qui indique qu'il est économe en ressources.

30. On parle du processus de *déverminage* (ou *debugging* en anglais).

4.4.3 La rapidité

La **rapidité** indique si l'algorithme ou le programme permet d'arriver plus ou moins vite au résultat.

C'est un critère qui est souvent sur-évalué, essentiellement pour deux raisons.

- ▷ Il est trompeur. On peut croire une version plus rapide alors qu'il n'en est rien. Par exemple, on peut se dire que décomposer un calcul ralentit un programme puisqu'il doit gérer des variables intermédiaires. Ce n'est pas forcément le cas. Les compilateurs modernes sont capables de nombreuses prouesses pour optimiser le code et fournir un résultat aussi rapide qu'avec un calcul non décomposé.
- ▷ L'expérience montre que la recherche de rapidité mène souvent à des algorithmes moins lisibles. Or la lisibilité doit être privilégiée à la rapidité car sinon il sera impossible de corriger et/ou de faire évoluer l'algorithme.

Ce critère est un cas particulier de l'*efficience* qui traite de la gestion économe des ressources. Nous reparlerons de rapidité dans le chapitre consacré à la *complexité* des algorithmes.

Le langage Java est réputé lent. C'est une affirmation péremptoire et peut-être périmée. Bien sûr, comme le *bytecode* est interprété (ces notions sont expliquées dans la section 5.1.1 p. 58) , il est nécessaire de charger la machine virtuelle. La gestion de la mémoire prise en charge par le langage a aussi un certain cout. Hormis ces deux aspects, la rapidité d'un programme dépend surtout de la qualité du code... et donc du développeur.

L'important, dans ce premier cours de développement est d'écrire des programmes lisibles et respectant les conventions d'écriture.

4.4.4 La taille

Nous voyons parfois des étudiants et des étudiantes contentes d'avoir pu écrire un algorithme en moins de lignes. Ce critère n'a **aucune importance** ; un algorithme plus court n'est pas nécessairement plus rapide ni plus lisible.

Lors de la traduction en langage Java, certaines formes d'écriture — plus courtes — seront privilégiées. Nous y reviendrons et — au fur et à mesure de l'apprentissage — elles deviendront vite évidentes.

4.4.5 Conclusion

Tous ces critères n'ont pas le même poids. Le point le plus important est bien sûr d'écrire des algorithmes et des programmes corrects mais ne vous arrêtez pas là ! Demandez-vous s'il n'est pas possible de le retravailler pour améliorer sa lisibilité ³¹.

31. On appelle *refactorisation* l'opération qui consiste à modifier un algorithme ou un code sans changer ce qu'il fait dans le but, notamment, de le rendre plus lisible.

4.5 Améliorer la lisibilité

Comme nous venons de le voir, la lisibilité est une qualité essentielle que doivent avoir nos algorithmes et nos programmes. Qu'est ce qui permet d'améliorer la lisibilité d'un algorithme ?

4.5.1 Mise en page des algorithmes

Il y a d'abord la **mise en page** qui aide le lecteur à avoir une meilleure vue d'ensemble de l'algorithme, à en repérer rapidement la structure générale. Ainsi, dans ce syllabus :

- ▷ Les mots imposés ou, tout au moins importants, sont mis en évidence (en gras³²).
- ▷ Une seule instruction se trouve par ligne.
- ▷ Les instructions à l'intérieur de l'algorithme sont *indentées* (décalées vers la droite). On indentera également les instructions à l'intérieur des choix et des boucles.
- ▷ Des lignes verticales relient le début et la fin de quelque chose. Ici, un algorithme mais on pourra l'utiliser également pour les choix et les boucles.

Exemples à ne pas suivre.

```
algorithm duréeTrajet(vitesseMS, distanceKM : réels) → réel
réel distanceM
distanceM = 1000 * distanceKM
return distanceM / vitesseMS
```



pseudocode

```
algorithm duréeTrajet(vitesseMS, distanceKM : réels) → réel
  réel distanceM
  distanceM = 1000 * distanceKM    retourner distanceM / vitesseMS
```



pseudocode

Il faudra préférer

```
algorithm duréeTrajet(vitesseMS, distanceKM : réels) → réel
  réel distanceM
  distanceM = 1000 * distanceKM
  return distanceM / vitesseMS
```

pseudocode

³². Difficile de mettre en gras avec un bic. Dans une version écrite vous pouvez : souligner ou surligner le mot, l'écrire en majuscule ou le mettre en couleur.

4.5.2 Écriture des programmes

Pour l'écriture d'un programme, les règles sont semblables. L'utilisation d'un éditeur de code³³ ou d'un IDE (Environnement de Développement Intégré)³⁴ aide à l'écriture de programmes lisibles et respectant les conventions.

- ▷ Le code est indenté comme les algorithmes.
- ▷ Une seule instruction par ligne.
- ▷ La longueur des lignes n'excède pas 80 caractères³⁵.
- ▷ S'il est nécessaire de couper une ligne trop longue, celle-ci est coupée avant un opérateur ou après une virgule.

```
1 public static boolean iDoNothing(double beautifulDouble,  
    int firstInteger, boolean isItReallyTrue){  
    double notSoBeautiful = beautifulDouble * firstInteger;  
4    return notSoBeautiful > 100 && notSoBeautiful < 10000  
        || isItReallyTrue;  
    }  
7
```

java

4.5.3 Choix des noms

Il y a, ensuite, l'écriture des instructions elles-mêmes. Ainsi :

- ▷ Il faut choisir soigneusement les noms (d'algorithmes, de paramètres, de variables...)
- ▷ Il faut décomposer (ou au contraire fusionner) des calculs pour arriver au résultat jugé le plus lisible.
- ▷ Il est également possible d'introduire des commentaires et/ou des constantes. Deux concepts que nous allons développer maintenant.

4.5.4 Les commentaires



Commenter un algorithme signifie lui ajouter du texte explicatif destiné au **lecteur** pour l'aider à mieux comprendre le fonctionnement de l'algorithme. Un commentaire n'est pas utilisé par celui qui exécute l'algorithme ; il ne modifie pas ce que l'algorithme fait.

Habituellement, on distingue deux sortes de commentaires :

33. Un éditeur de code est un programme aidant à l'édition de code qu'il ne faut pas confondre avec un éditeur de texte. Des éditeurs de codes connus ; *gVim*, *Notepad++*

34. Des IDE connus ; *Netbeans*, *Eclipse*...

35. Ce n'est pas la largeur de l'écran qui détermine la longueur des lignes. La limite à 80 caractères permet un code plus lisible. Elle permet aussi de pouvoir lire plus vite le code en limitant le mouvement des yeux de gauche à droite.

- ▷ Ceux placés **au-dessus** de l'algorithme ou du programme qui expliquent **ce qu'il fait** et dans quelles **conditions** il fonctionne (les contraintes sur les paramètres).

C'est la documentation **documentation**.

- ▷ Ceux placés **dans** l'algorithme ou le programme qui expliquent **comment** il le fait.

Commenter correctement un programme est une tâche qui n'est pas évidente et qu'il faut travailler. Il faut arriver à apporter au lecteur une information **utile** qui n'apparaît pas directement dans le code. Par exemple, il est contre-productif de répéter ce que l'instruction dit déjà. Il faut supposer que le lecteur connaît le langage et l'algorithmique.

Dans nos algorithmes, les commentaires commencent par `//`.

En langage Java, il y a trois manières de commenter :

1. `//` en début de ligne ;
2. le commentaire sur plusieurs lignes en commençant par `/*` et en le terminant par `*/` ;
`/* <commentaire> */`
3. le commentaire sur plusieurs lignes en commençant par `/**` et en le terminant par `*/`. Dans ce cas, c'est un commentaire destiné à la documentation *javadoc*. Nous y reviendrons.

Voici quelques mauvais commentaires

```
// Exemples de mauvais commentaires
real length
sum = 0
```

```
// La longueur est un réel .
// On initialise la somme à 0
```



pseudocode

```
double length    // La longueur est un réel
2 /* La somme est initialisée à 0.
   Ce sera toujours le cas. */
int sum = 0;
5
```

java

Remarque Un excès de commentaires peut être le révélateur des problèmes de lisibilité du code lui-même. Par exemple, un choix judicieux de noms de variables peut s'avérer bien plus efficace que des commentaires. Ainsi, l'instruction

```
newCapital = oldCapital * (1 + rate / 100)
```

pseudocode

dépourvue de commentaires est bien préférable aux lignes suivantes :

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



pseudocode

Pour résumer :

N'hésitez pas à documenter votre programme pour expliquer ce qu'il fait et à le retravailler pour que tout commentaire à l'intérieur de l'algorithme devienne superflu.

Exemples. Voici comment on pourrait documenter un de nos algorithmes.

```
// Calcule la surface d'un rectangle dont on donne la largeur et la longueur.
// On considère que les données ne sont pas négatives.
algorithm rectangleArea(length, width : reals) → real
    return length * width
```

pseudocode

```
1 /*
   * Calcule la surface d'un rectangle dont on donne la largeur et
   * la longueur.
4  *
   * Les données ne sont pas négatives.
   */
7 public static double rectangleArea (double length, double width){
    return length * width;
}
10
```

java

4.5.5 Constantes

Une **constante** est une information pour laquelle nom, type et valeur sont figés.

L'usage est d'écrire les constantes en majuscules.

Il est inutile de spécifier leur type — mais c'est autorisé — celui-ci étant défini implicitement par la valeur de la constante. L'utilisation de constantes dans vos algorithmes présente les avantages suivants :

- ▷ Une meilleure lisibilité du code, pour autant que vous lui trouviez un nom explicite.
- ▷ Une plus grande facilité pour modifier le code si la constante vient à changer (modification légale du seuil de réussite par exemple).

Exemples

```

constante PI = 3,1415
constante SEUIL_RÉUSSITE = 10
constante ESI = "École Supérieure d'Informatique"

```

pseudocode

```

public static final double PI = 3.1415;
2 public static final int PASS_LEVEL = 10;
public static final String ESI = "École_supérieure_d'Informatique";

```

java

Remarque En Java, définir une constante π est un mauvais choix. Cette constante existe déjà. Elle est accessible directement par `Math.PI`.

Exercice Utiliser une constante. Trouvez un algorithme que vous avez écrit où l'utilisation de constante pourrait améliorer la lisibilité de votre solution.

4.6 Appel d'algorithme, appel de méthode

Reprenons l'algorithme `rectangleArea` qui nous a souvent servi d'exemple. Il permet de calculer la surface d'un rectangle dont on connaît la longueur et la largeur. Mais d'où viennent les données ? Et que faire du résultat ?

Tout d'abord, un algorithme ou un programme peut utiliser (on dit **appeler**) un autre algorithme ou programme.

Cet autre algorithme doit exister *quelque part* : sur la même page, une autre page, un autre document, peu importe.

En Java, la contrainte est un peu plus forte. Les programmes ne sont pas écrits sur une feuille mais rassemblés dans un ou plusieurs fichiers. Dans un premier temps, nous écrirons nos programmes dans un seul fichier. Ce fichier représentera une *classe*.

▷ Les instructions qui définissent une classe sont :

```

public class MyClass{
2  // statements
}

```

java

▷ La classe `MyClass` doit se trouver dans le fichier `MyClass.java`

Pour pouvoir appeler une méthode en Java, il faudra qu'elle existe dans la même classe³⁶.

³⁶. Nous relâcherons assez vite cette contrainte

L'appel d'un algorithme s'écrit ainsi :

```
area = rectangleArea(122,3.78)
```

```
// On appelle l'algorithme
```

pseudocode

L'appel d'une méthode s'écrit ainsi :

```
area = rectangleArea(122, 3.78);
```

2

java

L'appel d'un algorithme est considéré comme une expression, un calcul qui, comme toute expression, possède une valeur (la valeur retournée) et peut intervenir dans un calcul plus grand, être assignée à une variable...

4.7 Interagir avec l'utilisateur

4.7.1 Afficher un résultat

Un programme concret (en Java par exemple) qui permet de calculer des surfaces de rectangles devra communiquer le résultat à l'utilisateur du programme. Nous allons renseigner un affichage par la commande **print**. Ce qui donne :

```
print rectangleArea(122, 3.78)
```

pseudocode

L'instruction **print** signifie que l'algorithme doit, à cet endroit de l'algorithme communiquer une information à l'utilisateur. La façon dont il va communiquer cette information (à l'écran dans une application texte, via une application graphique, sur un cadran de calculatrice ou de montre, sur une feuille de papier imprimée, via un synthétiseur vocal...) ne nous intéresse pas ici.

En langage Java, un affichage sur la *sortie standard*, la console ou encore dans le terminal se fait comme suit :

```
1 System.out.println(rectangleArea(122, 3.78));
```

java

Les affichages et les demandes peuvent aussi se faire à l'aide d'un organigramme et d'un parallélogramme.

Afficher rectangleArea(122, 3.78)

4.7.2 Demander des valeurs

Il serait maintenant intéressant de demander à l'utilisateur ce que valent la longueur et la largeur. C'est le but de la commande **read**.

```
read length  
read width  
print rectangleArea(length, width)
```

pseudocode

L'instruction **read** signifie que l'utilisateur va, à cet endroit de l'algorithme, être sollicité pour donner une valeur qui sera affectée à une variable. À nouveau, la façon dont il va indiquer cette valeur (au clavier dans une application texte, via un champ de saisie ou une liste déroulante dans une application graphique, via une interface tactile, via des boutons physiques, via la reconnaissance vocale...) ne nous intéresse pas ici.

En langage Java, trois instructions seront nécessaires pour pouvoir faire une lecture au clavier pour un programme s'exécutant dans la console³⁷.

```
1 import java.util.Scanner;  
  // ...  
  Scanner keyboard = new Scanner(System.in);  
4 // ...  
  double length = keyboard.nextDouble();
```

java

4.7.3 Préférer les paramètres

Un algorithme avec paramètres est toujours plus intéressant qu'un algorithme qui demande les données et affiche le résultat car il peut être utilisé (appelé) dans un autre algorithme pour résoudre une partie du problème. C'est exactement pareil pour un programme ; on privilégiera des méthodes recevant des valeurs en arguments que des méthodes qui demandent les données à l'utilisateur. Cette demande sera faite à un autre moment.

L'exemple du rectangle pourrait s'écrire de manière un peu plus complète comme suit :

37. Pour une application graphique, c'est encore un peu plus compliqué.

```
// Calcule la surface d'un rectangle dont on donne la largeur et la longueur.  
// On considère que les données ne sont pas négatives.  
algorithm rectangleArea(length, width : reals) → real  
|   return length* width
```

```
algorithm AreaTest()  
|   reals length, width  
|   read length, width  
|   print rectangleArea(length, width)
```

pseudocode

Et voici une traduction en langage Java :

```
import java.util.Scanner;  
  
3 public class AreaTest {  
    /*  
     * Calcule la surface d'un rectangle dont on donne la largeur et  
6     * la longueur.  
     *  
     * Les données ne sont pas négatives.  
9     */  
    public static double rectangleArea(double length, double width){  
        return length * width;  
12    }  
  
    public static void main(String[] args){  
15        double length;  
        double width;  
        Scanner keyboard = new Scanner(System.in);  
18  
        System.out.println("Entrez la longueur:");  
        length = keyboard.nextDouble();  
21        System.out.println("Entrez la largeur:");  
        width = keyboard.nextDouble();  
  
24        System.out.println("Surface:" + rectangleArea(length, width));  
    }  
27 }
```

java

Chapitre 5

Premiers programmes

La traduction d'un algorithme en un programme est une première étape de développement. La réalisation — c'est-à-dire l'exécution du programme sur une machine — est une seconde étape très importante. Que faut-il faire en plus de la traduction de l'algorithme pour que le programme fonctionne sur un ordinateur ?

Contenu

5.1	Introduction	57
5.1.1	Compilation - interprétation	58
5.2	Environnement de développement	61
5.2.1	La base	61
5.2.2	Écrire le code	61
5.2.3	Exécuter le programme	62
5.2.4	Hello world	62
5.3	La grammaire du langage	63

5.1 Introduction

Il existe beaucoup de langages de programmation. Ces langages peuvent être rassemblés par classes de langages en fonction de leurs spécificités. Une classe de langages est adaptée à une classe de problèmes. Les langages — comme les problèmes — évoluent au fur et à mesure du temps.

langage machine

le langage machine est le langage de plus bas niveau. Il est exécutable par la machine et incompréhensible par l'humain sans effort ;

langage assembleur

là où tout est représenté par des nombres en langage machine, le langage assembleur propose une première couche d'abstraction : les instructions sont des mots : MOV, JMP...

langage de haut niveau

les langages de haut niveau sont destinés aux développeurs, le niveau d'abstraction est grand. Ils proposent des instructions, des variables, des structures de contrôle... tout ça sera traduit pour être compris par la machine.

Par exemple, *Fortran*, *COBOL*, *Pascal*, *C*...

langage orienté objets

là où les langages de haut niveau étaient plus orientés sur les problèmes à résoudre

— Que faut-il faire ?

les langages orienté objets s'intéressent d'abord aux données

— Quelles sont les données que nous avons et que devons nous fournir ?

C'est sans doute la classe de langages la plus répandue.

Par exemple, *C++*, *Java*, *C#*, *Python*, *Go*, *Ruby*, *VB.NET*, *Vala*, *Objective C*, *Eiffel*, *Ada*, *PHP*, *Smalltalk*, *Scala*...

langage fonctionnel

là où les langages orienté objets s'intéressent aux changements d'état des objets lorsqu'ils évoluent au fur et à mesure du programme, la programmation fonctionnelle consiste à exprimer le problème à résoudre en terme de fonctions (mathématiques). C'est une autre façon de programmer. Si elle est peu répandue, elle n'est pourtant pas récente.

Par exemple, *Lisp*, *Common Lisp*, *Haskell*, *Scala*...

Dans ce cours, nous nous intéressons au langage orienté objets, Java.

La langage Java est un langage strict. Il impose, par exemple, que l'on déclare les variables que l'on utilise et il vérifie que les données assignées aux variables soient du « bon » type. Il possède un *garbage collector* (ramasse-miettes en français³⁸) qui gère la mémoire à la place du développeur. Ces aspects facilitent l'acquisition de bonnes pratiques de développement.

C'est également un langage très présent dans l'industrie. Il est donc bien adapté pour des étudiants entamant un *bachelor* professionnalisant.

Certaines personnes lui reprocheront sa lenteur et sa syntaxe qui peut paraître lourde au premier abord tandis que d'autres rétorqueront que la lenteur n'est due qu'à la mauvaise qualité du code.

Vous aurez tout le loisir de vous faire votre propre idée.

5.1.1 Compilation - interprétation

L'ordinateur ne comprenant que le langage machine, toutes les instructions devront, à un moment ou à un autre, être traduites en langage machine. Pour certains langages, la traduction se fait une fois pour toutes. Ils sont compilés. Pour d'autres la traduction se fait au *fil de l'eau*, pendant l'exécution. Ils sont interprétés.

38. Vous noterez la traduction.

Définition

Un langage est **compilé** (voir fig. 5.1) si le code source est traduit d'une traite. Un nouveau fichier contenant le code exécutable est créé.

Ces langages nécessitent un **compilateur**.

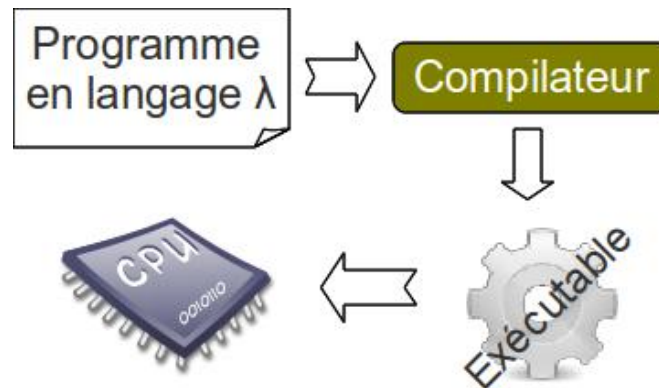


FIGURE 5.1 – Langage compilé

Définition

Un langage est **interprété** (voir fig. 5.2) si le code source est traduit instruction par instruction. Aucun nouveau fichier n'est créé et les instructions sont traduites à chaque exécution du programme.

Ces langages nécessitent un **interpréteur**³⁹.

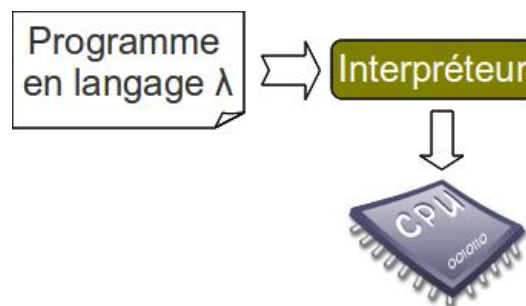


FIGURE 5.2 – Langage interprété

Les deux approches ont leurs avantages et leurs inconvénients.

- ▷ Dès lors qu'une machine possède l'interpréteur, le développeur peut diffuser son code et le code pourra être exécuté sur la machine... quel que soit son OS (*operating system*, système d'exploitation).
Le programme fonctionnera sans modification supplémentaire sur une machine MS Windows, Linux ou Mac OS.
- ▷ Avec une approche « interprétée », pour diffuser un programme, il faut diffuser le code source. S'il s'agit d'un langage compilé, il est possible de ne distribuer que le binaire / exécutable.

39. Certaines personnes préfèrent parler d'un interprète mais le terme prête à confusion.

Par contre, il existe des « *décompilateurs* » qui peuvent reconstruire le code source à partir du binaire / exécutable.

- ▷ La phase de compilation permet de détecter beaucoup de (petites) erreurs avant d'exécuter le programme.
- ▷ L'exécution du binaire / exécutable est plus rapide que l'interprétation du code source.



Java a une approche mixte, il est compilé et ensuite interprété. Le code source est compilé et produit un *bytecode* sauvegardé dans un nouveau fichier. Le *bytecode* est ensuite interprété par une machine virtuelle, la *jvm* (*Java Virtual Machine*).

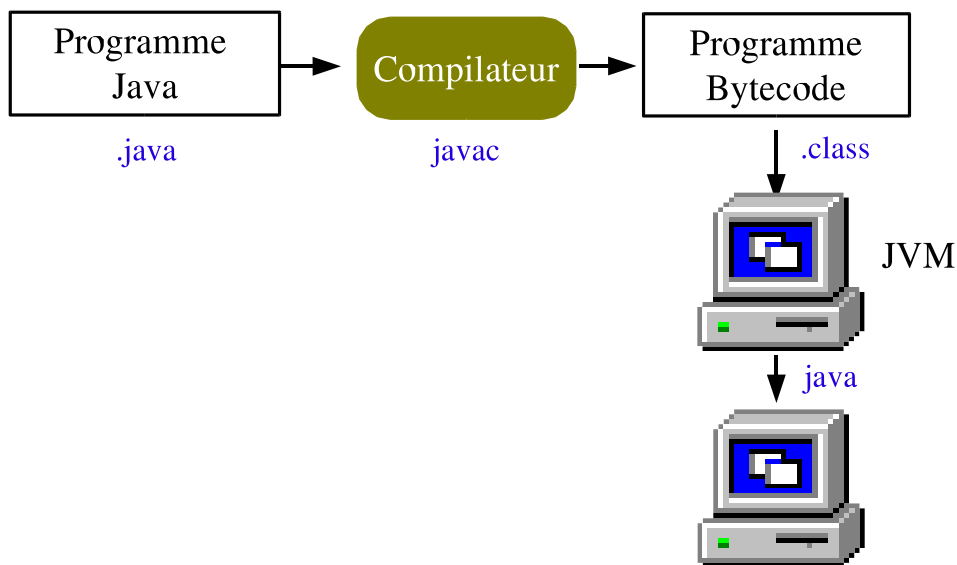


FIGURE 5.3 – Java est compilé et ensuite interprété

Le **code source** du programme est écrit dans un fichier texte dont l'extension sera `.java`. C'est le fichier que manipule le développeur. Le code source est la suite l'instructions java.

À l'aide du **compilateur** — le programme `javac` — le fichier est compilé. Un nouveau fichier ayant comme extension `.class`, est créé contenant le *bytecode*. Ce fichier est destiné à la machine virtuelle. Voici l'instruction pour compiler le programme.

```
$  
javac MyProgram.java
```

terminal

C'est la machine virtuelle — le programme `java` — qui est l'interpréteur du bytecode. C'est ce programme qui permet d'exécuter le... programme.

```
$  
java MyProgram
```

terminal

Remarques Notez que `javac` prend le nom d'un fichier en paramètre (avec son extension) alors que `java` prend le nom d'un programme — nous dirons une classe bientôt — en paramètre (sans extension donc).

5.2 Environnement de développement

L'environnement de développement représente l'ensemble des outils nécessaires au développement, à l'écriture des programmes.

5.2.1 La base

Il est évidemment nécessaire d'avoir un **ordinateur** et de le connaître un tant soit peu. Voici quelques compétences nécessaires :

- ▷ manipuler des fichiers : les déplacer, les trouver, les renommer...
- ▷ ouvrir un terminal ;
- ▷ connaître son clavier et particulièrement où se trouvent les caractères spéciaux. Par exemple : `{ } [] ; < > " ' ;`

Par contre, que l'OS utilisé soit MS Windows, linux ou Mac OS importe peu pour l'apprentissage de l'algorithmique et du développement en Java.

5.2.2 Écrire le code

Pour **écrire le code**, deux approches sont possibles : l'utilisation d'un éditeur de code ou d'un IDE (EDI, Environnement de Développement Intégré).

1. éditeur de code

Un éditeur de code est un éditeur de texte — à ne pas confondre avec un traitement de texte — augmenté c'est-à-dire offrant des fonctionnalités supplémentaires. Citons par exemple :

- ▷ la coloration syntaxique. Le programme reconnaît les mots clés du langage, les structures et les écrit en couleur pour accroître la lisibilité du code ;
- ▷ l'indentation automatique et la réindentation du code ;
- ▷ une certaine autocomplétion pour les mots connus du langage et les variables.

Exemples d'éditeurs de code toutes plateformes, licences et prix confondus : *(g)Vim*, *Notepad++*, *Atom*, *SublimeText*...

Exemples d'éditeurs de texte inutile pour le développement : *nano*, *Notepad*, *Mousepad*.

2. IDE

Un environnement de développement intégré est un programme servant à écrire les programmes. En plus d'un éditeur de code, un IDE compile en arrière plan,

a un débogueur, organise les fichiers, crée des fichiers (en partie) pré-complétés sur base de *templates*...

Exemples d'IDE : *Netbeans*, *Eclipse*...

Connaitre et utiliser correctement un éditeur de code et un IDE sont deux compétences essentielles d'un bon développeur.

5.2.3 Exécuter le programme

Quand le code est écrit, il faut le compiler puis l'exécuter. Les deux programmes, `javac` et `java`, font partie d'un ensemble de programmes fournis avec le langage Java. Cet ensemble de programmes nécessaires au développement en Java s'appelle **JDK** : *Java Development Kit*

Il en existe deux :

1. Java est — actuellement — la propriété d'Oracle et c'est Oracle qui fournit le JDK officiel.
Télécharger Java chez Oracle ⁴⁰
2. OpenJDK est une alternative libre au JDK officiel Java.
Télécharger OpenJDK ⁴¹

Remarque Il ne faut pas confondre JDK et JRE. Un JRE pour *Java Runtime Environment*, est l'ensemble de programmes — il y en a moins — nécessaires à l'**exécution** de programmes Java. Dans ces programmes se trouve une machine virtuelle java (*JVM*) pour l'interprétation des programmes java.

Vous avez probablement déjà un — voire plusieurs — JRE sur votre machine.

5.2.4 Hello world

Une fois les armes fourbies, il est temps d'écrire son premier programme. Et, selon la tradition, nous allons écrire un programme qui affiche *Hello world*.

Un programme Java s'écrit dans une **classe**. Cette classe porte un nom et ce nom doit être le même que celui du fichier qui la contient. Nous allons appeler notre première classe **Hello**.

Grâce à mon éditeur de code, je crée le fichier **Hello.java** ⁴² qui contient :

40. <http://www.oracle.com/technetwork/java/javase/downloads>

41. <http://openjdk.java.net/>

42. Si vous êtes un utilisateur MS Windows, désactivez la propriété «*Hide extension for know files types*». Si vous ne le faites pas, vous allez — probablement — voir **Hello.java** alors que le fichier que vous créez est **Hello.java.txt**.

```
public class Hello{  
2  public static void main (String[] args) {  
    System.out.println("Hello_world");  
  }  
5 }
```

java

Je compile ma classe :

```
$  
javac Hello.java
```

terminal

Un fichier `Hello.class` apparaît dans mon répertoire courant. C'est le *bytecode* de ma classe. J'exécute ma classe :

```
$  
java Hello
```

terminal

... et je vois apparaître dans mon terminal les mots **Hello world**.

5.3 La grammaire du langage

Un langage de programmation, dès lors qu'il est compilé et exécuté par une machine, répond à des règles très strictes. En tout cas, ces règles doivent être non ambiguës.

Cet ensemble de règles est appelé la **grammaire du langage** et se trouve dans *The Java Language Specification*, un ouvrage reprenant toute la spécification du langage Java. Nous allons y faire référence régulièrement dans ces notes.

JLS Télécharger *The Java Language Specification* ⁴³

Les **symboles terminaux** de la grammaire, ceux que l'on peut retrouver en l'état dans le code sont écrit en police à chasse fixe comme ça. Les **règles de production**, qui sont définies ailleurs dans la grammaire, sont écrites en italiques *comme ça*.

Dans ces notes, nous utiliserons une **grammaire simplifiée** par soucis de simplification pour une première approche du développement sans jamais le préciser. Celles et ceux qui veulent aller plus loin sont invités à faire référence à JLS pour la grammaire complète.

Nous avons dit que le type entier en Java était `int` et que les nombres réels étaient déclarés grâce au mot clé `double`. Si nous avions voulu utiliser la grammaire (simplifiée) nous aurions pu écrire :

43. <https://docs.oracle.com/javase/specs/>

```
NumericType:  
  IntegralType  
  FloatingPointType
```

```
IntegralType:  
  int
```

```
FloatingPointType:  
  double
```

Ce qui signifie qu'il existe deux sortes de types numériques : les nombres entiers et les nombres à virgule flottante. Pour les nombres entiers, il s'agit du type `int`. C'est un symbole terminal qui peut se retrouver tel quel dans un code. Pour les nombres réels — nous avons dit *pseudo-réels* — ou « à virgule flottante », il s'agit du type `double`.

Nous sommes incomplets à ce stade. Les personnes curieuses peuvent aller voir la grammaire complète de ces règles dans JLS10⁴⁴ p. 42.

44. <https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf>

Chapitre 6

Une question de choix

Dans la section ?? (p. ??) nous proposons une initiation ludique aux algorithmes. Vous avez eu l'occasion d'y aborder les alternatives. Par exemple, vous avez indiqué au zombie quelque chose comme : « S'il existe un chemin à gauche alors tourner à gauche ».

Les **alternatives** permettent de n'exécuter des instructions que si une certaine *condition* est vérifiée. Avec le zombie, par exemple, vous testiez son environnement ; dans nos algorithmes et dans nos programmes, nous allons tester les données.

Les algorithmes et les programmes vus jusqu'à présent ne proposent qu'un seul « chemin », une seule « histoire ». À chaque exécution de l'algorithme, les mêmes instructions s'exécutent dans le même ordre. Les alternatives permettent de créer des histoires différentes, d'adapter les instructions aux valeurs concrètes des données.

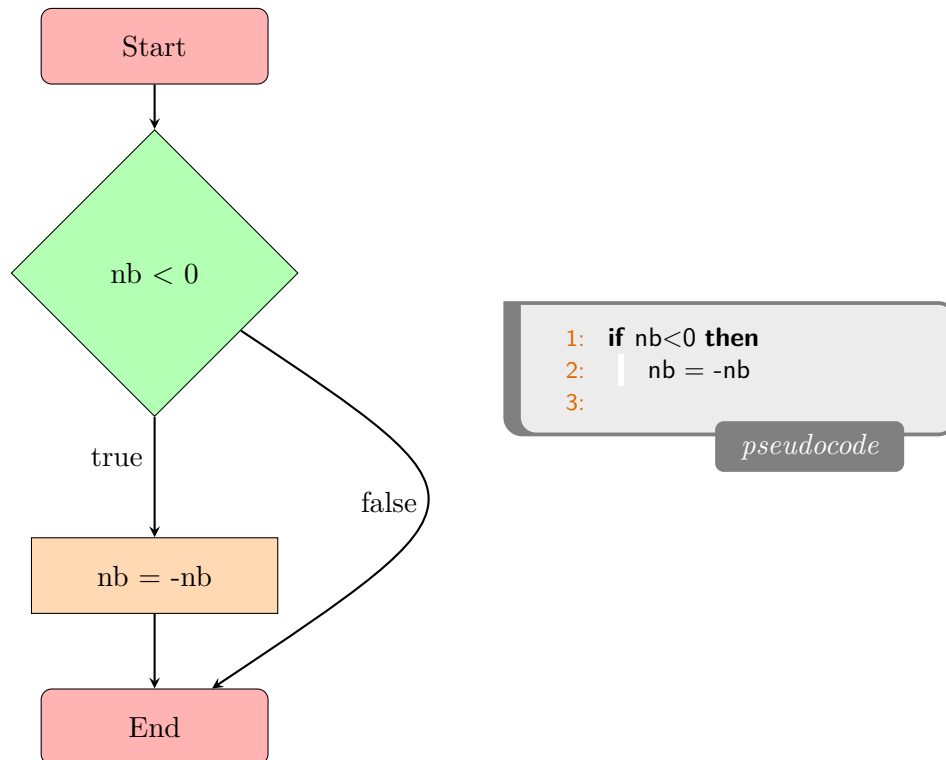
Contenu

6.1	Le si (<i>if-then</i>)	65
6.2	Le si-sinon (<i>if-then-else</i>)	68
6.3	Le si-sinon-si	71
6.4	Expression booléenne	73
6.5	Le selon-que (<i>switch</i>)	74

6.1 Le si (*if-then*)

Il existe des situations où des instructions ne doivent pas toujours être exécutées et un test va nous permettre de le savoir.

Exemple. Supposons que la variable `nb` contienne un nombre positif ou négatif. Et supposons que l'on veuille le rendre positif. Il faudra tester son signe et, s'il est négatif, l'inverser. Par contre, s'il est positif, il n'y a rien à faire. Voici comment écrire, graphiquement et en pseudocode, un algorithme :



Traçons l'algorithme dans deux cas différents pour bien illustrer son déroulement.

#	nb	test
1	-3	vrai
2	3	

#	nb	test
1	3	faux

La condition peut être n'importe quelle expression (calcul) dont le résultat est un booléen (vrai ou faux).



Remarque Attention, la confusion est fréquente.

Un « si » n'est pas une règle que l'ordinateur doit apprendre et exécuter à chaque fois que l'occasion se présente. La condition n'est testée que lorsqu'on arrive à cet endroit de l'algorithme.

En langage Java, le *si* s'écrit comme suit :

```
if ( Expression ) Statement
```

1. *Expression* représente une expression booléenne. c'est-à-dire ayant comme valeur **true** ou **false**.
2. *Statement* représente une instruction ou un **bloc** d'instructions. Un bloc d'instructions est toujours délimités par une paire d'accolades.

Le test précédent pourrait s'écrire comme suit :

```
1 if (nb < 0)
    nb = -nb;
```

java

Pour s'éviter des erreurs, nous utiliserons toujours le bloc d'instructions et nous écrirons :

```
if (nb < 0){
    nb = -nb;
3 }
```

java

Exercice de compréhension Tracez l'algorithme ou le programme avec les valeurs fournies et donnez la valeur de retour.

```
algorithm exercice(a, b : entiers) → entier
    entier c
    c = 2 * a
    if c > b then
        c = c - b
    return c
```

pseudocode

```
public static int exercice(int a, int b){
2   int c;
    c = 2 * a;
    if (c > b){
5       c = c-b;
    }
    return c;
8 }
```

java

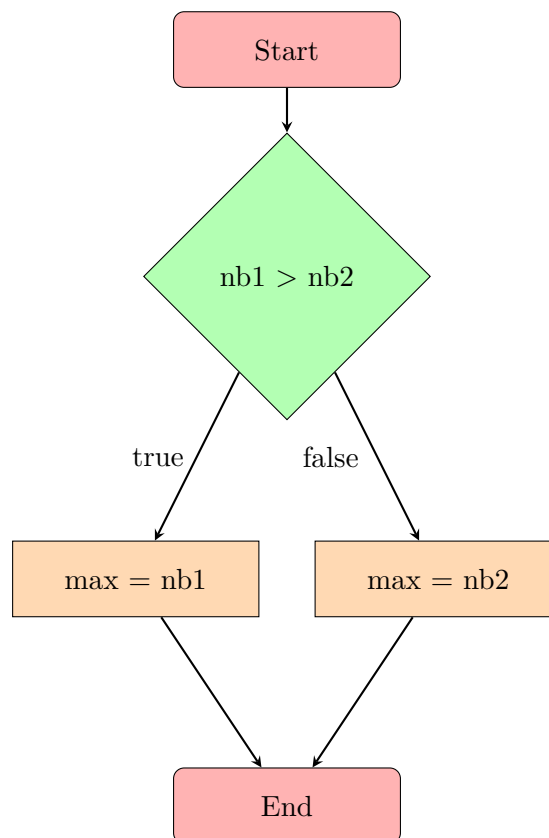
▷ exercice(2, 5) = _____

▷ exercice(4, 1) = _____

6.2 Le si-sinon (*if-then-else*)

La construction si-sinon permet d'exécuter certaines instructions ou d'autres en fonction d'un test. Pour illustrer cette instruction, nous allons nous pencher sur un grand classique, la recherche de maximum.

Exemple. Pour déterminer le maximum de deux nombres, c'est-à-dire la plus grande des deux valeurs, il y aura deux chemins possibles. Le maximum devra prendre la valeur du premier nombre ou du second selon que le premier est plus grand que le second ou pas.



```
1: if nb1>nb2 then
2:   max = nb1
3: else
4:   max = nb2
5:
```

pseudocode

Traçons-le dans différentes situations.

#	nb1	nb2	max	test
	3	2	indéfini	
1			indéfini	vrai
2			3	

#	nb1	nb2	max	test
	4	42	indéfini	
1			indéfini	faux
4			42	

Le cas où les deux nombres sont égaux est également géré.

#	nb1	nb2	max	test
	4	4	indéfini	
1			indéfini	faux
4			4	

En langage Java, le *si-sinon* s'écrit :

```
IfThenElseStatement:
  if ( Expression )
    Statement
  else
    Statement
```

Le test précédent peut donc s'écrire en supposant les variables déclarées :

```
if (nb1 > nb2){
    max = nb1;
3 } else {
    max = nb2;
}
6
```

java

Exercice de compréhension Tracez ces algorithmes ou programmes avec les valeurs fournies et donnez la valeur de retour.

```
algorithm exercice(a, b : integer) → integer
  integer c
  if a > b then
    c = a DIV b
  else
    c = b MOD a
  return c
```

pseudocode

```

public static int exercice(int a, int b){
    int c;
3   if (a > b){
        c = a/b;
    } else {
6       c = b/a;
    }
9   }

```

java

▷ exercice(2, 3) = ____

▷ exercice(4, 1) = ____

```

algorithm exercice(x1, x2 : integer) → integer
    boolean ok
    ok = x1 > x2
    if ok then
        | ok = ok ET x1 == 4
    else
        | ok = ok OU x2 == 3

    if ok then
        | x1 = x1 * 1000

    return x1 + x2

```

pseudocode

```

public static int exercice(int x1, int x2){
    boolean ok;
3   ok = x1 > x2;
    if (ok){
        ok = ok && x1 == 4;
6   } else {
        ok = ok || x2 == 3;
    }
9   return x1 + x2;
}

```

java

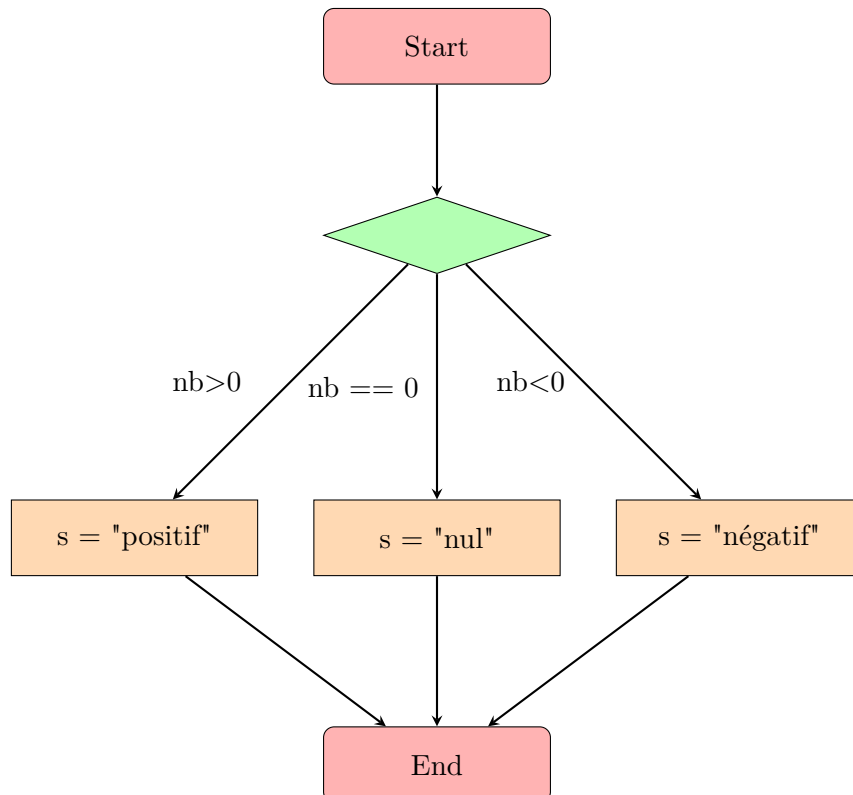
▷ exercice(2, 3) = ____

▷ exercice(4, 1) = ____

6.3 Le si-sinon-si

Avec cette construction, il est possible d'indiquer à un endroit de l'algorithme plus de deux chemins possibles. Partons à nouveau d'un exemple pour illustrer cette instruction.

Exemple. On voudrait mettre dans la chaîne **signe** la valeur "positif", "négatif" ou "nul" selon qu'un nombre donné est positif, négatif ou nul.



```

1: if nb>0 then
2:   signe = "positif"
3: else if nb==0 then
4:   signe = "nul"
5: else
6:   signe = "négatif"
7:
  
```

pseudocode

Traçons-le

#	nb	signe	test
	2	indéfini	
1			vrai
2		"positif"	

#	nb	signe	test
	0	indéfini	
1			faux
3			vrai
4		"nul"	

#	nb	signe	test
	-5	indéfini	
1			faux
3			faux
6		"négatif"	

En langage Java, il n'y a pas de structure particulière pour ce test. Le if-then-else fait bien l'affaire. Seule l'indentation change un peu pour plus de lisibilité. Ce test peut donc s'écrire — en supposant toujours que les variables sont déclarées — comme ceci :

```

1 if (nb > 0){
    s = "positif";
  } else if (nb == 0) {
4   s = "nul";
  } else {
    s = "négatif";
7 }

```

java

Remarques.

- ▷ Pour le dernier cas, on se contente d'un **sinon** sans indiquer la condition ; ce serait inutile, elle serait toujours vraie.
- ▷ Le **si** et le **si-sinon** peuvent être vus comme des cas particuliers du **si-sinon-si**.
- ▷ On pourrait écrire la même chose avec des **si-sinon** imbriqués mais le **si-sinon-si** est plus lisible.

```

if nb>0 then
  signe = "positif"
else
  if nb=0 then
    signe = "nul"
  else
    signe = "négatif"

```

pseudocode

- ▷ Lorsqu'une condition est testée, on sait que toutes celles au-dessus se sont avérées fausses. Cela permet parfois de simplifier la condition.

Exemple. Supposons que le prix unitaire d'un produit (`prixUnitaire`) dépende de la quantité achetée (`quantité`). En dessous de 10 unités, on le paie 10€ l'unité. De 10 à 99 unités, on le paie 8€ l'unité. À partir de 100 unités, on paie 6€ l'unité.


```

if quantité<10 then
  prixUnitaire = 10
else if quantité<100 then           // On sait que ce n'est pas <10
                                     // inutile de le tester
  prixUnitaire = 8
else
  prixUnitaire = 6

```

pseudocode

6.4 Expression booléenne

Nous avons dit que dans un test, l'*expression* était une expression booléenne et nous avons vu quelques opérateurs intervenants dans ces expressions. Revenons plus en détail sur ce concept.

Dénifition. Une expression booléenne est une expression — c'est-à-dire le résultat d'un calcul — dont la valeur est booléenne : `true` ou `false`.



Une telle expression se compose grâce :

1. aux opérateurs relationnels (*relational operator* ou *comparators*) ;
Un opérateur relationnel est un opérateur dont la valeur est booléenne et les opérandes numériques.

RelationalOperator:
(one of)
< > <= >=

2. aux opérateurs d'égalité (*equality operators*) ;
Un d'égalité est un opérateur dont la valeur est booléenne et les opérandes de même type (à conversion près).

EqualityOperator:
(one of)
== !=

3. au complément logique (*logical complement operator*) et aux opérateurs conditionnels (*conditionals operators*) ;
Le complément logique et les opérateurs conditionnels sont des opérateurs dont la valeur est booléenne et le ou les opérandes également booléens.
Le `&&` est prioritaire sur le `||`.

LogicalComplementOperator:

!

ConditionalOperator:

(one of)

|| &&

6.5 Le selon-que (*switch*)

Cette nouvelle instruction permet d'écrire plus lisiblement *certaines* **si-sinon-si**, plus précisément quand le choix d'une branche dépend de la valeur précise d'une variable (ou d'une expression).

Exemple. Imaginons qu'une variable (`dayNumber`) contienne un numéro de jour de la semaine et qu'on veuille mettre dans une variable (`dayName`) le nom du jour correspondant ("lundi" pour 1, "mardi" pour 2...)

Une solution avec un **si-sinon-si** est possible mais le **selon-que** (*switch*) est plus lisible.

```
switch dayNumber
  case 1: dayName = "lundi"
  case 2: dayName = "mardi"
  case 3: dayName = "mercredi"
  case 4: dayName = "jeudi"
  case 5: dayName = "vendredi"
  case 6: dayName = "samedi"
  case 7: dayName = "dimanche"
```

pseudocode

remplace avantageusement

```
if dayNumber=1 then
  dayName = "lundi"
else if dayNumber=2 then
  dayName = "mardi"
else if dayNumber=3 then
  dayName = "mercredi"
else if dayNumber=4 then
  dayName = "jeudi"
else if dayNumber=5 then
  dayName = "vendredi"
else if dayNumber=6 then
  dayName = "samedi"
else
  dayName = "dimanche"
```



pseudocode

En langage Java, le *switch* s'écrit (grammaire simplifiée) :

SwitchStatement:
`switch (Expression) SwitchBlock`

SwitchBlock:
`SwitchLabels Statement`

SwitchLabel:
`case ConstantExpression:`
`default:`

- ▷ l'expression ne peut pas être de n'importe quel type. À ce stade, elle peut être, un entier ou une chaîne ;
- ▷ *Statement* peut être une instruction ou plusieurs ;
- ▷ *SwitchLabels* (avec un *s*) se sont plusieurs « **case** »
- ▷ le *switch* en java peut être vu comme un « *saut* au bon label ». Dès lors que l'instruction a trouvé le bon *case*, l'exécution des instructions continue. Il faut explicitement demander de sortir du *switch* en utilisant l'instruction **break**

Le *switch* précédent s'écrit :

```
1 switch (dayNumber) {  
    case 1:  
        dayName = "lundi";  
4     break;  
    case 2:  
        dayName = "mardi";  
7     break;  
    case 3:  
        dayName = "mercredi";  
10    break;  
    case 4:  
        dayName = "jeudi";  
13    break;  
    case 5:  
        dayName = "vendredi";  
16    break;  
    case 6:  
        dayName = "samedi";  
19    break;  
    case 7:  
        dayName = "dimanche";  
22 }
```

java

Remarques.

- ▷ Il peut y avoir plusieurs valeurs pour un cas donné.

- ▷ Il peut y avoir un cas par défaut, **default** qui sera exécuté si la valeur n'est pas reprise par ailleurs.

La syntaxe générale est :

```
switch expression
| case liste1 de valeurs séparées par des virgules :
|   Instructions
| case liste2 de valeurs séparées par des virgules :
|   Instructions
|   ...
| case listek de valeurs séparées par des virgules :
|   Instructions
| case default :
|   Instructions
```

pseudocode

Chapitre 7

Module et références

Contenu

7.1	Décomposer le problème	77
7.2	Exemple	78
7.3	Paramètres et valeur de retour	79
7.3.1	Le paramètre en entrée	80
7.3.2	Le paramètre en entrée-sortie	81
7.3.3	La valeur de retour	82
7.4	Type primitif et type référence	82
7.4.1	Type primitif	83
7.4.2	Type référence	84
7.4.3	Les paramètres en Java	85
7.5	Résumons	86

7.1 Décomposer le problème

Jusqu'à présent, les problèmes que nous avons abordés étaient relativement petits. Nous avons pu les résoudre avec un algorithme d'un seul tenant.

Dans la réalité, les problèmes sont plus conséquents et il devient nécessaire de les décomposer en sous-problèmes. On parle d'une *approche modulaire*. Les avantages d'une telle décomposition sont multiples.

- ▷ **Cela permet de libérer l'esprit.** L'esprit humain ne peut pas traiter trop d'informations à la fois (*surcharge cognitive*). Lorsqu'un sous-problème est résolu, il peut se libérer l'esprit et attaquer un autre sous-problème.
- ▷ **On peut réutiliser ce qui a été fait.** Si un même sous-problème apparaît plusieurs fois dans un problème ou à travers plusieurs problèmes, il est plus efficace de le résoudre une fois et de réutiliser la solution.
- ▷ **On accroît la lisibilité.** Si, un algorithme, appelle un autre algorithme pour résoudre un sous-problème, le lecteur ou la lectrice verra un nom d'algorithme qui peut être plus parlant que les instructions qui se cachent derrière, même s'il

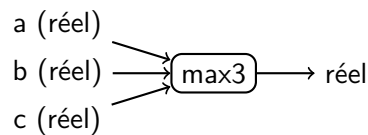
y en a peu. Par exemple, `dizaine(nb)` est plus parlant que `nb MOD 100 DIV 10` pour calculer les dizaines d'un nombre.

Parmi les autres avantages, que vous pourrez moins percevoir en début d'apprentissage, citons la possibilité de répartir le travail dans une équipe.

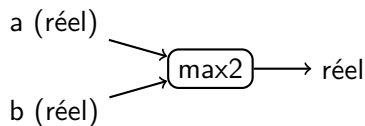
Un algorithme qui résout une partie de problème est parfois appelé **fonction**, **procédure**, **méthode** ou encore **module** en fonction du langage et du contexte. Il y a quelques nuances mais elles importent peu ici.

7.2 Exemple

Illustrons l'approche modulaire sur le calcul du maximum de 3 nombres.



Commençons par écrire la solution du problème plus simple : le maximum de 2 nombres.



```

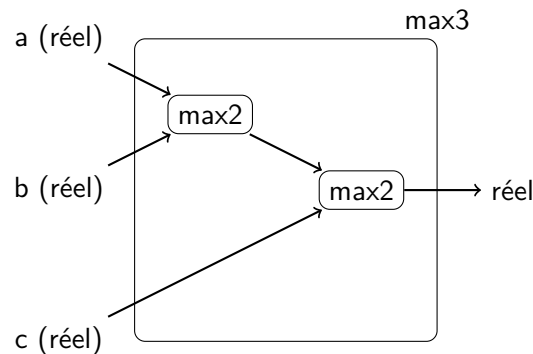
algorithm max2(a : real, b : real)
  → real
  real max
  if a > b then
    max = a
  else
    max = b
  return max
  
```

pseudocode

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

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

Elle s'illustre comme suit :



Sur base de cette idée, on voit que calculer le maximum de trois nombres peut se faire en calculant deux fois le maximum de deux nombres. On ne va évidemment pas *recopier*⁴⁵ dans notre solution ce qu'on a écrit pour le maximum de deux nombres ; on va plutôt y faire référence, c'est-à-dire appeler l'algorithme `max2`. Ce qui donne :

```

algorithm max3(a : real, b : real, c : real) → real
    reals maxab, max
    maxab = max2(a,b)
    max = max2(maxab,c)
    return max

```

pseudocode

qui peut se simplifier en :

```

algorithm max3(a,b,c : reals) → real
    return max2( max2(a,b) ,c)

```

pseudocode

7.3 Paramètres et valeur de retour

Jusqu'à présent, nous avons considéré que les paramètres d'un algorithme (ou *module*) correspondent à ses données et que le résultat, unique, est retourné.

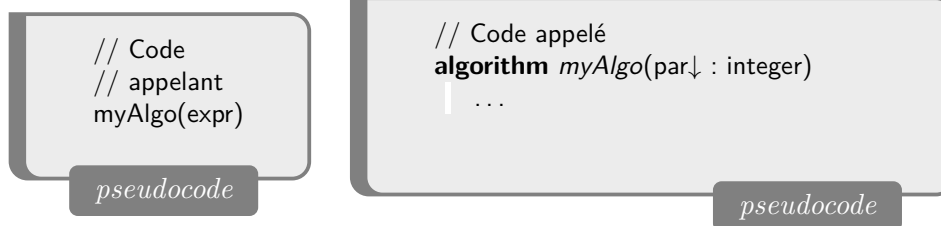
Il s'agit d'une situation fréquente mais pas obligatoire que nous pouvons généraliser. En algorithmique nous allons traiter avec trois sortes de paramètres. Nous verrons ensuite que tous les langages n'acceptent pas tous les types de paramètres.

45. Cette approche serait fastidieuse, engendrerait de nombreuses erreurs lors du recopiage et serait difficile à lire. Même le copier/coller n'est pas une bonne solution. Il diminue la lisibilité et rend la refactorisation et l'évolution des algorithmes et des programmes plus compliquées.

7.3.1 Le paramètre en entrée

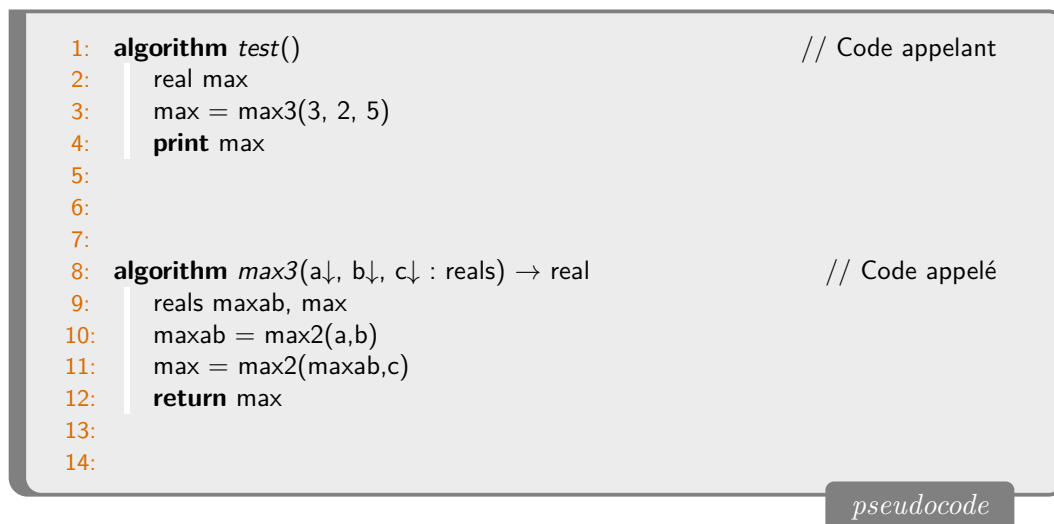
Le paramètre en **entrée** est ce que nous connaissons déjà. Il correspond à une donnée de l'algorithme. Une valeur va lui être attribuée en début d'algorithme et elle ne sera pas modifiée. On pourra faire suivre le nom du paramètre d'une flèche vers le bas (\downarrow) pour rappeler son rôle mais ce n'est pas obligatoire lorsqu'il n'y a pas d'ambiguïté.

Lors de l'appel, c'est une **valeur** qui est fournie ou, plus généralement une expression dont la valeur sera donnée au paramètre. Voici un cas général de paramètre en entrée.



C'est comme si l'algorithme `myAlgo` commençait par l'affectation `par = expr`.

Exemple. Reprenons l'exemple de `max3` en ajoutant un petit test.



Traçons son exécution.

	test	max3				
#	max	a	b	c	maxab	max
2	indéfini					
3,7		3	2	5		
8					indéfini	indéfini
9					3	indéfini
10						5
11,3	5					

Notez bien : Dans cet exemple, on trouve deux fois la variable `max`. Il s'agit bien de deux variables **différentes** ; l'une est définie et connue dans `test` ; l'autre l'est dans `max3`.

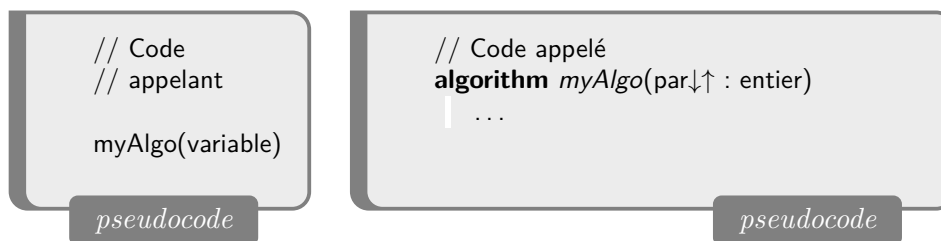
7.3.2 Le paramètre en entrée-sortie

Le paramètre en **entrée-sortie** est un paramètre tel que :

- ▷ l'algorithme reçoit une valeur en entrée ;
- ▷ le paramètre peut être modifié.

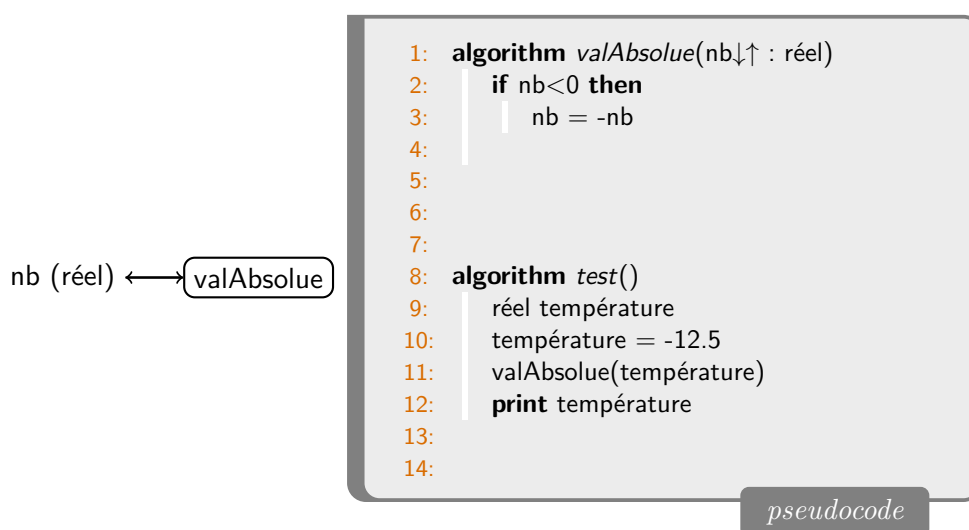
Cela signifie que l'algorithme a pour but de modifier le paramètre. Un tel paramètre sera suivi d'une double flèche ($\downarrow\uparrow$).

C'est une **une variable** qui doit être passée en paramètre. Sa valeur est donnée au paramètre au début de l'algorithme. À la fin de l'algorithme, la variable reçoit la valeur du paramètre. Voici un cas général de paramètre en sortie.



C'est comme si, dans le code appelé, il y avait une première ligne pour donner sa valeur au paramètre (`par = variable`) et une dernière ligne pour effectuer l'assignation opposée (`variable = par`). Il n'y a pas de **return**.

Exemple. Nous avons vu un algorithme qui retourne la valeur absolue d'un nombre. Nous pourrions imaginer une variante qui **modifie** le nombre reçu. En voici le schéma et la solution avec un appel possible :



Traçons-le.

	test	valAbsolue	
#	température	nb	test
8	indéfini		
9	-12.5		
10, 1		-12.5	
2			vrai
3		12.5	
5, 10	12.5		

Remarque Nous verrons dans la section 7.4 que le langage Java ne permet pas de traduire ces algorithmes en l'état.

7.3.3 La valeur de retour

La valeur de retour correspond au résultat de l'algorithme.

```
// Code
// appelant
integer var
var = myAlgo()
myAlgo()
```

pseudocode

```
// Code appelé
algorithm myAlgo() → integer
| ...
```

pseudocode

Remarques

- ▷ Cette valeur de retour est optionnelle, un algorithme peut ne rien retourner. Un algorithme qui ne **retourne** rien (pas de \rightarrow) n'a pas de valeur ; il ne peut pas apparaître dans une expression ou être assigné à une variable.
- ▷ Le fait que la valeur de retour soit unique peut sembler rédhibitoire et c'est vrai. Ceci dit nous verrons qu'il existe plusieurs méthodes pour s'en sortir.

7.4 Type primitif et type référence

Dans nos algorithmes nous traitons avec des paramètres en entrée, en entrée/sortie et des valeurs de retour, qu'en est-il dans les langages de programmation ?

- ▷ Tous acceptent d'avoir une valeur de retour unique.
- ▷ Tous acceptent des paramètres en entrée.
- ▷ Certains et sous certaines conditions acceptent des paramètres en entrée/sortie.

Intéressons nous au langage Java en commençant par faire un petit détour sur les notions de **type primitif** et **type référence**.

7.4.1 Type primitif

Définition : Une variable de type primitif est une variable qui contient directement la valeur qui lui est assignée. Cette variable a une taille fixe qui dépend de son type. L'emplacement mémoire qui lui est attribué se trouve sur la pile (*stack*⁴⁶).



Par exemple, une variable de type `int` a une taille de 4 *bytes* (32 bits). Toujours.

Une variable `i` de type primitif et contenant la valeur 7 peut se représenter comme ci-contre.

`i`

`7`

Il existe, en Java, 8 types primitifs : des types primitifs numériques entiers, numériques à virgule flottante, les caractères et les booléens. Voici ce que dit la grammaire :

PrimitiveType:

NumericType

`boolean`

NumericType:

IntegralType

FloatingPointType

IntegralType:

(one of)

`byte short int long char`

FloatingPointType:

(one of)

`float double`

Chaque type a une taille déterminée.

Les entiers sont codés en notation en complément à deux, excepté le type `char` qui est un entier non-signé de 16 bits représentant le code Unicode codé en UTF-16 du caractère⁴⁷.

Voici les tailles et les intervalles.

46. Lorsqu'un programme s'exécute, le système lui attribue plusieurs emplacements mémoire : un contenant les instructions et deux qui contiendront les variables du programme. La pile (*stack*) et le tas (*heap*).

47. Pour en savoir plus sur l'Unicode, UTF8, UTF16 et UTF32, lire « Unicode, UTF8, UTF16, UTF32... et tutti quanti »
<http://namok.be/blog/?post/2009/11/30/unicode-UTF8-UTF16-UTF32-et-tutti-quanti>

type	taille (<i>byte</i>)	taille (<i>bit</i>)	intervalle
byte	1	8	$[-128, 127]$ $[-2^8, 2^8 - 1]$
short	2	16	$[-32\,768, 32\,767]$ $[-2^{16}, 2^{16} - 1]$
int	4	32	$[2\,147\,483\,648, 2\,147\,483\,647]$ $[-2^{32}, 2^{32} - 1]$
long	8	64	$[9\,223\,372\,036\,854\,775\,808, 9\,223\,372\,036\,854\,775\,807]$ $[-2^{64}, 2^{64} - 1]$
char	2	16	$[0, 65\,535]$ $[0, 2^{16} - 1]$

Les nombres pseudo-réel, ou encore les nombres à virgule flottante, sont codés suivant la norme IEEE 754⁴⁸. Selon cette norme, un nombre est représenté avec un signe, une mantisse et un exposant. Le tout en base 2. Un bit est utilisé pour le signe.

$$\text{nombre} = \text{signe} \text{ mantisse}_2 2^{\text{exposant}_2}$$

type	taille (<i>bit</i>)	exposant	mantisse
float	32	8	23
double	64	11	52

Pour les booléens, bien qu'un bit suffirait, la taille dépend de l'architecture et de la *jvm*.

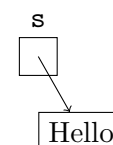
7.4.2 Type référence



Définition : Une variable de type référence est une variable qui ne contient pas directement la valeur qui lui est assignée. Elle contient une adresse mémoire désignant l'endroit où est — ou sera — stockée la valeur. L'emplacement mémoire attribué à la variable se trouve sur la pile et a la même taille pour toutes les variables de type référence tandis que l'emplacement mémoire qui contiendra effectivement la valeur sera attribué sur le tas (*heap*).

String est un type référence.

Une variable **s** de type référence et contenant la valeur "Hello" peut se représenter comme ci-contre.

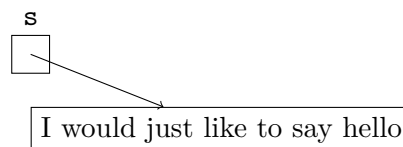


48. https://fr.wikipedia.org/wiki/IEEE_754

La même variable `s` peut recevoir une autre valeur, par exemple beaucoup plus grande `I would just like to say hello`.

```
1 String s = "Hello";  
  s = "I_would_just_like_to_say_hello";
```

java



Remarque. Même si nous ne connaissons actuellement qu'un seul type référence, ce seront les types les plus répandus. En effet, les types primitifs en Java sont au nombre de 8 comme nous l'avons vu tandis qu'il existe des types références prédéfinis (comme `String`) et tous les types références définis par le développeur.

Les tableaux sont aussi des types références comme nous le verrons plus tard.

7.4.3 Les paramètres en Java

Dans nos algorithmes nous avons : une valeur de retour, des paramètres en entrée et des paramètres en entrée/sortie. L'ensemble étant optionnel. Qu'en est-il pour le langage Java ?

Tout comme en algorithmique, les méthodes Java permettent de retourner une valeur grâce au `return Expression` de fin de méthode. Une méthode peut ne rien retourner. Dans ce cas, il suffit de fermer l'accolade ouverte en début de méthode.

Les paramètres en Java se passent **par valeur**. En ce sens, c'est équivalent aux paramètres en entrée.

Traduisons l'exemple de la section 7.3.1 (p.80) en Java. Les valeurs 3, 2 et 5 sont passées en argument à la méthode `max3`. Les variables `a`, `b` et `c`, locales à la méthode `max3`, reçoivent les trois valeurs.

```
public class Test{  
    public static void main(String[] args){  
3        double max;  
        max = max3(3,2,5);  
        System.out.println("Maximum:␣" + max);  
6    }  
  
    public double max2(double a, double b){  
9        //  
    }  
  
12    public double max3(double a, double b, double c){  
        double maxab;  
        double max;  
15    maxab = max2(a,b);  
        max = max2(maxab, c);  
        return max;  
18    }  
}
```

java

Il n'y a pas de paramètre en entrée/sortie en Java puisque tous les passages de paramètres se font par valeur. Si le paramètre est de type référence, la valeur que reçoit la méthode est la valeur de la référence. Même s'il n'est pas possible de modifier le paramètre reçu, il sera possible de modifier l'objet ou le tableau **référéncé** par la valeur reçue en paramètre. En ce sens, c'est un peu un passage de paramètre en entrée/sortie.

7.5 Résumons

Reprenons tout ce que nous venons de voir avec un exemple d'algorithme qui possède tous les types de paramètres.

```
1: algorithm testDivision()
2:   integers nb1, nb2, quotient, remainder
3:   nb1 = 5
4:   nb2 = 3
5:   quotient = division(nb1, nb2, remainder)
6:   print quotient
7:   print remainder
8:   nb1 = 7
9:   nb2 = 9
10:  quotient = division(nb1, nb2, remainder)
11:  print quotient
12:  print remainder
13:
14:
15: algorithm division( dividend↓ : integer, divisor↓ : integer,
16:   remainder↑ : integer) → integer
17:   dividend = nb1
18:   divisor = nb2
19:   remainder = remainder
20:
21:   // Le code proprement dit de l'algorithme
22:   integer quotient
23:   quotient = dividend DIV divisor
24:   remainder = dividend MOD divisor
25:
26:   remainder = remainder
27:   return quotient
28:
29:
```

pseudocode

Traçons-le avec : *quotient* **q**, *dividend* **D**, *divisor* **d**, *remainder* **r**.

	testDivision				division			
#	nb1	nb2	q	r	D	d	r	q
3	5							
4		3						
5, 15					5	3		
23								1
24							2	
27, 5			1	2				
8	7							
9		9						
10, 15					7	9		
23								0
24							7	
27, 10			0	7				

Pour mieux se comprendre, il est utile d'introduire un peu de vocabulaire.



Les paramètres déclarés dans l'entête d'un algorithme sont appelés **paramètres formels**. Les paramètres donnés à l'appel de l'algorithme sont appelés **paramètres effectifs**.

Les instructions en gris dans l'exemple ne sont pas écrites mais c'est comme si elles étaient présentes pour initialiser les paramètres formels \downarrow et $\downarrow\uparrow$ en début d'algorithme et pour donner des valeurs aux paramètres effectifs $\downarrow\uparrow$ en fin d'algorithme.

À la fin de l'algorithme, c'est comme si la valeur retournée *remplaçait* l'appel. Dans notre exemple, c'est donc cette valeur retournée qui sera affichée.

Chapitre 8

Un travail répétitif

Les ordinateurs révèlent tout leur potentiel dans leur capacité à répéter inlassablement les mêmes tâches. Vous avez pu appréhender les boucles lors de votre initiation sur le site `code.org`. Voyons comment utiliser à bon escient des boucles dans nos codes.

Conseil pédagogique. D'expérience, nous savons que ce chapitre est difficile à appréhender et qu'au fil des pages, la matière se complexifie. C'est en restant assidus et assidues ; en faisant les exercices et en demandant un retour sur ses solutions aux enseignants et enseignantes que l'on met toutes les chances de son côté. Si l'on se sent perdu, il ne faut pas hésiter à demander de l'aide.



Contenu

8.1	La notion de travail répétitif	89
8.2	Une même instruction, des effets différents	90
8.2.1	Exemple - Afficher les nombres de 1 à 5	91
8.3	Le « tant que » (<i>while</i>)	92
8.3.1	Exemple - Afficher les nombres de 1 à 5	92
8.3.2	Exemple - Généralisation à n nombres	93
8.4	Le « pour » (<i>for</i>)	94
8.4.1	Exemples	95
8.4.2	Un pas négatif	96
8.4.3	Remarques	97
8.4.4	Exemple – Afficher uniquement les nombres pairs	97
8.5	« faire – tant que »	98
8.5.1	Exemple	98
8.6	Quel type de boucle choisir ?	99
8.7	Exercices résolus	100
8.7.1	Saisie des données par l'utilisateur	100
8.7.2	Les suites	105

8.1 La notion de travail répétitif

Lorsque l'on veut (faire) effectuer un travail répétitif, il faut indiquer deux choses :

- ▷ le travail à répéter ;
- ▷ quand s'arrêter ou quand continuer ou encore, combien de fois faire le travail.

Examinons quelques exemples pour fixer notre propos.

Exemple 1. Pour traiter des dossiers, nous pouvons dire « tant qu'il reste un dossier à traiter, le traiter » ou encore « traiter un dossier puis passer au suivant jusqu'à ce qu'il n'en reste plus à traiter ».

- ▷ La tâche à répéter est : « traiter un dossier ».
- ▷ Quand continuer : « s'il reste encore un dossier à traiter ».

Nous aurions pu dire de manière semblable :

- ▷ La tâche à répéter est : « traiter un dossier ».
- ▷ Quand s'arrêter : « dès qu'il n'y a plus de dossier à traiter ».

Exemple 2. Pour calculer la cote finale de tous les étudiants et toutes les étudiantes, nous dirions quelque chose du genre « Pour tout étudiant, calculer sa cote ».

- ▷ La tâche à répéter est : « calculer la cote d'un · e étudiant · e ».
- ▷ Combien de fois faire le travail : « autant qu'il y a d'étudiant · es »

Il faut le faire pour tous les étudiants et les étudiantes. Nous pourrions être plus précis et dire qu'il faut commencer au premier, passer à chaque fois au suivant et s'arrêter lorsque c'est terminé avec le dernier.

Exemple 3. Pour afficher tous les nombres de 1 à 100, nous dirions : « Pour tous les nombres de 1 à 100, afficher le nombre ».

- ▷ La tâche à répéter est : « afficher un nombre ».
- ▷ Nous indiquons qu'il faut le faire pour tous les nombres de 1 à 100. Il faut commencer à 1, passer au suivant et s'arrêter après avoir affiché 100.

8.2 Une même instruction, des effets différents

Comprenez bien que c'est toujours la même tâche qui est exécutée mais pas avec le même effet à chaque fois. Ainsi, c'est un dossier qui est traité mais à chaque fois un différent ; c'est un nombre qui est affiché mais à chaque fois un différent.

Par exemple, la tâche à répéter pour afficher des nombres ne peut pas être **afficher 1** ni **afficher 2** ni... Par contre, on pourra utiliser l'instruction **afficher nb** si on s'arrange pour que la variable **nb** s'adapte à chaque passage dans la boucle.

De façon générale, pour obtenir un travail répétitif, il faut trouver une formulation de la tâche qui va produire un effet différent à chaque fois.

8.2.1 Exemple - Afficher les nombres de 1 à 5

Si nous voulions un algorithme qui affiche les nombres de 1 à 5 sans utiliser de boucle, nous pourrions écrire :

```
print 1  
print 2  
print 3  
print 4  
print 5
```

pseudocode

Ces cinq instructions sont proches mais pas tout-à-fait identiques. En l'état, nous ne pouvons pas encore en faire une boucle⁴⁹ ; il va falloir ruser. Nous pouvons obtenir le même résultat avec l'algorithme suivant :

```
nb = 1  
print nb  
nb = 2  
print nb  
nb = 3  
print nb  
nb = 4  
print nb  
nb = 5  
print nb
```

pseudocode

ou encore

```
1: nb = 1  
2: print nb  
3: nb = nb + 1  
4: print nb  
5: nb = nb + 1  
6: print nb  
7: nb = nb + 1  
8: print nb  
9: nb = nb + 1  
10: print nb  
11: nb = nb + 1
```

pseudocode

Il est plus compliqué, mais cette fois les lignes 2 et 3 se répètent exactement. D'ailleurs, la dernière ligne ne sert à rien d'autre qu'à obtenir cinq copies identiques. Le travail à répéter est donc :

```
print nb  
nb = nb + 1
```

pseudocode

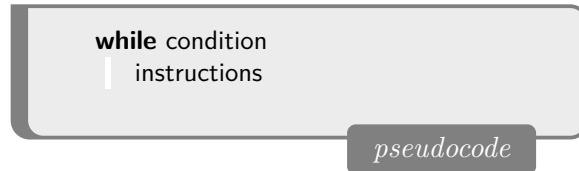
Cette tâche doit être effectuée cinq fois dans notre exemple. Il existe plusieurs structures répétitives qui vont se distinguer par la façon dont on va contrôler le nombre de répétitions. Voyons-les une à une⁵⁰.

49. Vous vous dites peut-être que ce code est simple ; inutile d'en faire une boucle. Ce n'est qu'un exemple. Que feriez-vous s'il fallait afficher les nombres de 1 à 1000 ?

50. Nous ne verrons pas de structure de type **répéter 5 fois**... Elle est simple à comprendre mais pas souvent adaptée au problème à résoudre.

8.3 Le « tant que » (*while*)

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



Comme pour la structure if, la **condition** est une expression à valeur booléenne.

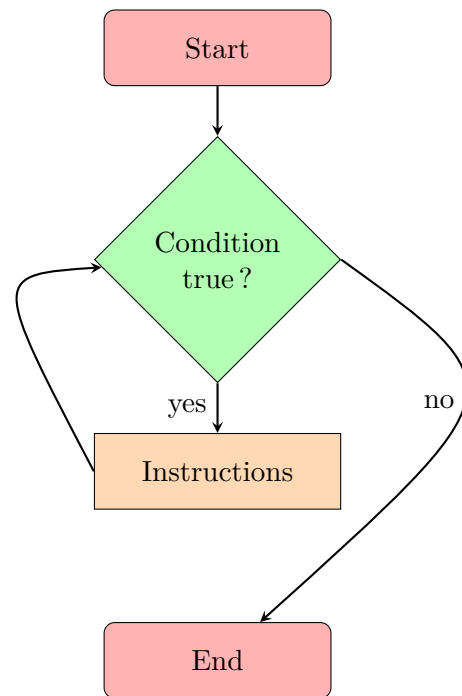
Dans ce type de structure, il faut qu'il y ait dans la séquence d'instructions du **while** c'est-à-dire le bloc indenté au moins une instruction qui modifie une des composantes de la condition de telle manière qu'elle puisse devenir **fausse** à un moment donné. Dans le cas contraire, la condition reste indéfiniment vraie et la boucle va tourner sans fin, c'est ce qu'on appelle une **boucle infinie**.

L'organigramme ci-contre décrit le déroulement de cette structure. On remarquera que si la condition est fausse dès le début, la tâche n'est jamais exécutée.

En langage Java, voici l'instruction *while* qui correspond exactement au « tant que » des algorithmes. *Expression* est une expression booléenne comme nous l'avons vu (cf. section 6.4 p. 73) et *Statement* peut être une seule instruction ou un bloc d'instructions délimitées par des accolades.

```

while ( Expression )
    Statement
  
```



8.3.1 Exemple - Afficher les nombres de 1 à 5

Reprenons notre exemple d'affichage des nombres de 1 à 5. Pour rappel, la tâche à répéter est :



La condition va se baser sur la valeur de **nb**. On continue tant que le nombre n'a pas dépassé 5. Ce qui donne (en n'oubliant pas l'initialisation de **nb**) :

```
1: algorithm compteur5()
2:   entier nb
3:   nb = 1
4:   while nb ≤ 5
5:     print nb
6:     nb = nb + 1
7:   print "nb vaut : ", nb
9:
10:
```

pseudocode

```
1 public static void count5(){
2   int nb = 1;
3   while (nb <= 5){
4     System.out.println(nb);
5     nb = nb + 1;
6   }
7   System.out.println(
8     "nb_vaut:" + nb);
9 }
10
```

java

#	nb	condition	affichage
2	indéfini		
3	1		
4		vrai	
5			1
6	2		
4		vrai	
5			2
6	3		
4		vrai	
5			3
6	4		
4		vrai	
5			4
6	5		
4		vrai	
5			5
6	6		
4		faux	
8			nb vaut 6

8.3.2 Exemple - Généralisation à n nombres

Généraliser l’algorithme et le programme pour qu’ils reçoivent le nombre d’itérations en paramètres se fait simplement comme suit :

```
algorithm compteur(n : integer)
  integer nb
  nb = 1
  while nb ≤ n
    print nb
    nb = nb + 1
```

pseudocode

```

public static void count(int n){
2   int i = 1;
   while (i <= n){
       System.out.println(i);
5       i = i + 1;
   }
   System.out.println(
8       "i_vaut: " + i);
}

```

java

8.4 Le « pour » (*for*)

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

```

for variable from début to fin by pas
    instructions

```

pseudocode

Dans ce type de structure, **début**, **fin** et **pas** peuvent être des constantes, des variables ou des expressions entières.

Le **pas** est facultatif, et généralement omis (dans ce cas, sa valeur par défaut est 1).

La boucle s'arrête lorsque la variable dépasse la valeur de fin.

La variable de contrôle ne servant que pour la boucle et étant forcément entière, nous allons considérer qu'il n'est pas nécessaire de la déclarer et qu'elle n'est pas utilisable en dehors de la boucle⁵¹.

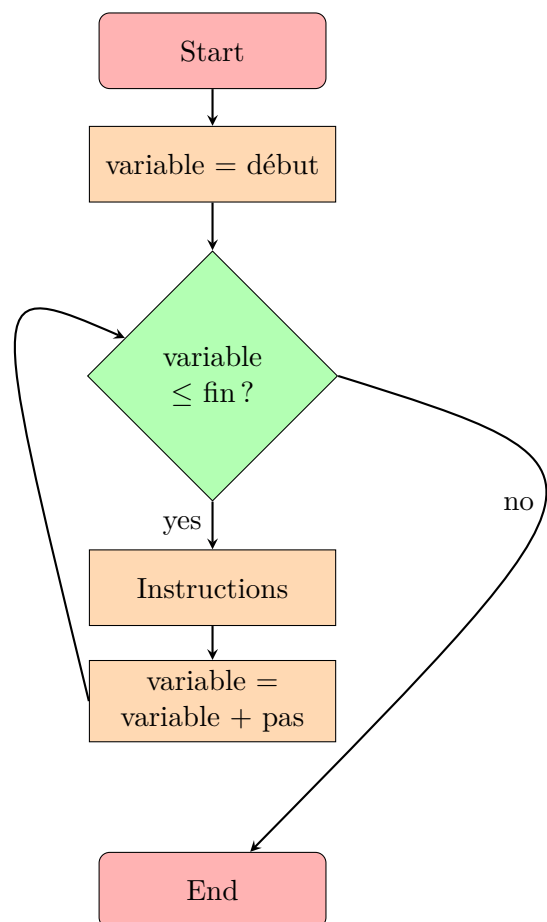
En langage Java, il existe une instruction *for* également. Elle est un peu plus large que celle-ci. Voici sa grammaire (simplifiée).

ForStatement:

```

for ( ForInit; Expression; ForUpdate)
    Statement

```



51. De nombreux langages ne le permettent d'ailleurs pas ou ont un comportement indéterminé si on le fait.

ForInit

initialise la variable ;
Par exemple `variable = début`.
Il sera nécessaire de la déclarer.

Expression

est la condition de fin ;

ForUpdate

est l'incrément, le passage à l'itération suivante ;
Par exemple `variable = variable + pas`

Une traduction stricte du « pour » s'écrit en langage Java :

```
for (int i = début; i < fin; i = i + pas){  
  // statement  
}
```

java

Nous verrons plus loin qu'il existe également un *enhanced for* — appelé *foreach* — permettant une écriture très simple pour le parcours de certaines collections.

8.4.1 Exemples

Reprenons notre exemple d'affichage des nombres de 1 à 5. Voici la solution avec un **for** et le traçage correspondant.

```
1: algorithm compterJusque5()  
2:   // par défaut le pas est de 1  
3:   for nb from 1 to 5  
4:     print nb  
5:  
6:   print "nb n'existe plus"  
7:  
8:
```

pseudocode

```
public static void countTo5({  
  for (int nb=1; nb<= 5; nb = nb+1){  
    System.out.println(i);  
  }  
  System.out.println(  
    "nb_n'existe_plus");  
}
```

java

#	nb	cond.	affichage
3	1	vrai	
4			1
3	2	vrai	
4			2
3	3	vrai	
4			3
3	4	vrai	
4			4
3	5	vrai	
4			5
3	6	faux	
6	#	#	nb n'existe plus

Pour généraliser à n nombres, nous pouvons écrire :

```
algorithm compterJusqueN(n : entier)
  for nb from 1 to n
    print nb
```

pseudocode

```
1 public static void countToN(int n){
  for (int i = 1; i <= n; i = i + 1){
    System.out.println(i);
4  }
}
```

java

Remarques

- ▷ En langage Java, l'indice de la boucle est souvent noté i .
- ▷ Lorsqu'il n'est pas explicitement demandé le contraire, l'indice d'une boucle commence à 0.

8.4.2 Un pas négatif

Le pas est parfois négatif, dans le cas d'un compte à rebours, par exemple. Dans ce cas, la boucle s'arrête lorsque la variable prend une valeur plus petite que la valeur de fin et il faut faire attention à bien formuler la condition.

Exemple : Compte à rebours à partir de n .

```
algorithm compterÀReboursDécroissant(n : entier)
  for nb from n to 1 by -1
    print nb
  print "Partez!"
```

pseudocode

```
public static void countTo1(int n){
  for (int i = n; i > 0; i = i - 1){
3    System.out.print(i);
  }
  System.out.println("Go_!");
6 }
```

java

8.4.3 Remarques

1. Il faut veiller à la cohérence de cette structure.

Nous pouvons considérer pour nos algorithmes qu'au cas (à éviter) où **début** est strictement supérieur à **fin** et le **pas** est positif, la séquence d'instructions n'est jamais exécutée (mais ce n'est pas le cas dans tous les langages de programmation!). Idem si **début** est strictement inférieur à **fin** mais avec un **pas** négatif.

2. Il est important de ne pas modifier les variables de contrôle **début**, **fin** ou **pas** ni de modifier la **variable** au sein de la boucle pour garder une boucle lisible.



8.4.4 Exemple – Afficher uniquement les nombres pairs

Cette fois-ci, nous allons écrire une boucle affichant uniquement les nombres **pairs** jusqu'à la limite n .

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

Notez que n peut être impair. Si n vaut 11, l'affichage est le même que pour 10. Avec un « pour », une solution est :

```
algorithm afficherPair(n : entier)
  for nb from 2 to n by 2
    print nb
```

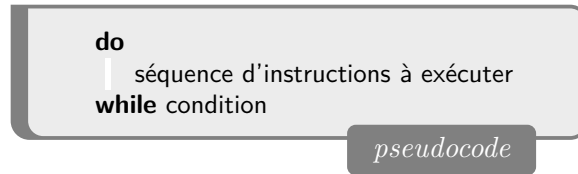
pseudocode

```
public static void printOdd(int n){
  for (int i=2; i <= n; i = i + 2){
3    System.out.println(i);
  }
}
6
```

java

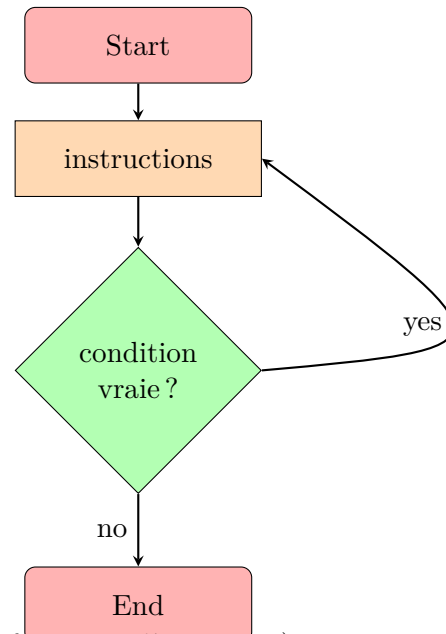
8.5 « faire - tant que »

Cette structure est très proche du « faire - tant que » à ceci près que le test est fait à la fin et pas au début⁵². La tâche est donc toujours exécutée au moins une fois.



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

En langage Java, il existe une instruction *do-while*, « faire-tant que » (et pas d'instruction « faire-jusqu'à ce que »).



```

DoStatement:
do
    Statement
while ( Expression );
  
```

8.5.1 Exemple

Reprenons notre exemple d'affichage des nombres de 1 à 5. Voici la solution et le traçage correspondant.

⁵². Certains langages introduisent aussi (ou à la place) un **faire jusqu'à ce que**. Dans ce cas, la boucle continue lorsque le test est **faux** et s'arrête lorsqu'il est vrai.

```
1: algorithm compteur5()
2:   entier nb
3:   nb = 1
4:   do
5:     print nb
6:     nb = nb + 1
7:   while nb ≤ 5
8:   print "nb vaut ", nb
9:
10:
```

pseudocode

```
public static void count5(){
    int nb = 1;
3   do {
        System.out.println(nb);
        nb = nb + 1;
6   } while (nb <= 5);
    System.out.println("nb_
        vaut_" + nb);
9   }
```

java

#	nb	condition	affichage
2	indéfini		
3	1		
5			1
6	2		
7		vrai	
5			2
6	3		
7		vrai	
5			3
6	4		
7		vrai	
5			4
6	5		
7		vrai	
5			5
6	6		
7		faux	
8			nb vaut 6

8.6 Quel type de boucle choisir ?

En pratique, il est possible d'utiliser systématiquement la boucle **tant que** qui peut s'adapter à toutes les situations.

Cependant, il est plus clair d'utiliser la boucle **pour** dans les cas où le nombre d'itérations est fixé et connu à l'avance. Par là, nous voulons dire que le nombre d'itérations est déterminé au début de la boucle.

La boucle **faire-tant que** convient quant à elle dans les cas où le contenu de la boucle doit être parcouru au moins une fois, alors que dans **tant que**, le nombre de parcours peut être nul si la condition initiale est fausse.

La figure fig.?? ci-dessous propose un récapitulatif.

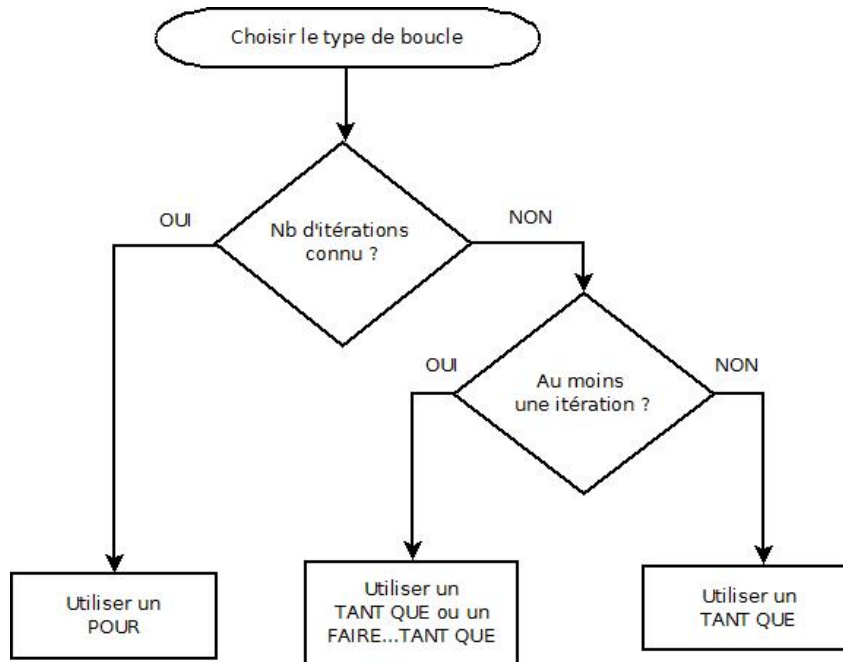


FIGURE 8.1 – Récapitulatif, choix d'une structure répétitive

8.7 Exercices résolus

8.7.1 Saisie des données par l'utilisateur

Il existe des problèmes où l'algorithme doit demander une série de valeurs à l'utilisateur pour pouvoir les traiter. Par exemple, les sommer, en faire la moyenne, calculer la plus grande. . .

Dans ce genre de problème, l'algorithme ou le programme stocke chaque valeur donnée par l'utilisateur dans une seule et même variable et la traite avant de passer à la suivante. Prenons un exemple concret pour mieux comprendre.

Écrire un algorithme et un programme qui calcule et retourne la somme d'une série de nombres donnés par l'utilisateur.

Il faut d'abord se demander comment l'utilisateur va pouvoir indiquer combien de nombres il faut additionner ou quand est-ce que le dernier nombre à additionner a été entré. Voyons quelques possibilités.

Variante 1 : le nombre de valeurs est connu

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

```

// Lit des valeurs entières et retourne la somme des valeurs lues.
algorithm sum() → integer                                // Variante 1
| integer nValues                                         // nombre de valeurs à additionner
| integer value                                           // un des termes de l'addition
| integer sum                                             // la somme
| sum = 0                                                 // la somme se construit petit à petit.
|                                                         // 0 au départ

| read nValues
| for i from 1 to nValues
| | read value
| | sum = sum + value

| return sum

```

pseudocode

En langage Java, l'habitude est de commencer à compter à partir de 0. Là où nous avons l'habitude de compter de 1 à n , nous compterons toujours de 0 à $n - 1$. Plutôt que d'avoir comme condition de fin $i \leq n - 1$, nous écrirons $i < n$.

Le programme Java aura l'allure suivante :

```

public static int sum(){
    int nValues;
    3 int value;
    int sum = 0;
    Scanner keyboard = new Scanner(System.in); // nécessite un import
    6
    nValues = keyboard.nextInt();
    for (int i = 0; i < nValues; i = i + 1){
    9     value = keyboard.nextInt();
        sum = sum + value;
    }
    12 return sum;
}

```

java

Remarque Dans les exemples, nous utilisons des lectures au clavier très rudimentaires. Par exemple :

```

1 value = keyboard.nextInt();

```

java

Cette instruction, lorsqu'elle est exécutée, se traduit par un curseur qui clignote en attendant un nombre entier. Ce serait mieux de demander à l'utilisateur au préalable d'entrer un entier en lui écrivant un message. Plutôt comme suit :

```
1 System.out.print("Entrer_un_entier:");  
  value = keyboard.nextInt();
```

java

`nextInt` lit un entier au clavier. Si ce n'est pas un entier qui est entré... une erreur survient et le programme plante lamentablement. Pour éviter ceci, nous pouvons demander au programme de vérifier que l'utilisateur ou l'utilisatrice entre bien un entier, sinon, lui signaler. Justement la classe `Scanner` propose une méthode `hasNextInt`. Nous pourrions alors écrire ceci :

```
System.out.print("Entrer_un_entier:");  
while (!keyboard.hasNextInt()){  
3   keyboard.next();  
}  
value = keyboard.nextInt();  
6
```

java

Lors de l'appel à la méthode `hasNextInt`, l'utilisateur va devoir entrer une valeur qui sera mémorisée (*bufferisée*) et la méthode retourne `true` si l'utilisateur a entré un entier, `false` sinon.

- ▷ Si c'est `true`, c'est un entier. Le programme n'entre pas dans la boucle et se rend directement à l'instruction suivante. Cette instruction consomme la valeur lue précédemment et la place dans la variable `value`.
- ▷ Si c'est `false`, ce n'est pas un entier. Le programme entre dans la boucle et l'instruction `keyboard.next()` consomme la valeur lue précédemment... et n'en fait rien. Elle vide simplement le *buffer*.

Le programme retourne au test et réexécute `hasNextInt` qui lit une valeur au clavier et la place dans le *buffer*. Et ainsi de suite.

Variante 2 : stop ou encore

Après chaque nombre, l'algorithme ou le programme demande à l'utilisateur s'il y a encore un nombre à additionner.

Dans ce cas, il faut chercher une solution différente car le nombre de valeurs à additionner — et donc le nombre d'exécution de la boucle — n'est pas connu au départ. Il faudra utiliser un « tant que » ou un « faire-tant que ».

Si nous demandons en fin de boucle s'il reste encore un nombre à additionner l'algorithme aura l'allure suivante :

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
algorithm sum() → integer // Variante 2a
    boolean hasMore // est-ce qu'il reste encore une valeur à additionner ?
    integer value // un des termes de l'addition
    integer sum // la somme
    sum = 0
    do
        read value
        sum = sum + value
        read hasMore
    while hasMore
    return sum
```

pseudocode

En langage Java, nous utiliserons naturellement un *do-while*. Voici :

```
public static int sum(){
    boolean hasMore;
3   int value;
    int sum = 0;
    Scanner keyboard = new Scanner(System.in); // nécessite un import
6
    do {
        value = keyboard.nextInt();
9       sum = sum + value;
        hasMore = keyboard.nextBoolean();
    } while (hasMore);
12  return sum;
}
```

java

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

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
algorithm sum() → integer // Variante 2b
    booléen hasMore // est-ce qu'il reste hasMore une value à additionner ?
    integer value // un des termes de l'addition
    integer sum // la sum
    sum = 0
    read hasMore
    while hasMore
        read value
        sum = sum + value
        read hasMore

    return sum
```

pseudocode

En Java, ça donne :

```
1 public static int sum(){
    boolean hasMore;
    int value;
4   int sum = 0;
    Scanner keyboard = new Scanner(System.in); // nécessite un import

7   hasMore = keyboard.nextBoolean();
    while (hasMore){
        value = keyboard.nextInt();
10    sum = sum + value;
        hasMore = keyboard.nextBoolean();
    }
13  return sum;
}
```

java

Variante 3 : valeur sentinelle

L'utilisateur entre une valeur spéciale pour indiquer la fin. Il s'agit d'une valeur **sentinelle**.

Cette solution n'est envisageable que si cette valeur **sentinelle** ne fait pas partie de l'ensemble des valeurs possibles.

Dans notre exemple si on veut additionner des nombres positifs uniquement, la valeur -1 peut servir de valeur sentinelle. Mais sans limite sur les nombres à additionner (positifs, négatifs ou nuls), il n'est pas possible de choisir une sentinelle.

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


```

// Lit des valeurs entières positives et retourne la somme des valeurs lues.
// La valeur sentinelle est -1.
algorithm sumPositives() → integer                                // Variante 3
    integer value                                                    // un des termes de l'addition
    integer sum                                                       // la sum
    sum = 0
    read value
    while value ≠ -1
        sum = sum + value
        read value                                                    // remarquer l'endroit où on lit une value.
    return sum

```

pseudocode

Et en Java, ça se traduit immédiatement en :

```

public static int sum(){
    int value;
    3 int sum = 0;
    Scanner keyboard = new Scanner(System.in); // nécessite un import

    6 value = keyboard.nextInt();
    while (value != -1){
        sum = sum + value;
    9 value = keyboard.nextInt();
    }
    return sum;
12 }

```

java

8.7.2 Les suites

Nous avons vu quelques exemples d'algorithmes qui affichent une suite de nombres (par exemple, afficher les nombres pairs). Nous avons pu les résoudre facilement avec un **for** en choisissant judicieusement les valeurs de début et de fin ainsi que le pas.

Ce n'est pas toujours aussi simple. Nous allons voir deux exemples plus complexes et les solutions qui les accompagnent. Elles pourront se généraliser à beaucoup d'autres exemples.

Exemple 1 - Afficher les carrés

Nous voulons afficher les n premiers nombres carrés parfaits : 1, 4, 9, 16, 25...

Si l'on veut savoir quel est le 7^e nombre à afficher, c'est 7^2 , soit 49". Plus généralement, le nombre à afficher lors du i^e passage dans la boucle est i^2 .

étape	1	2	3	4	5	6	7	8
valeur à afficher	1	4	9	16	25	36	49	64

L'algorithme qui en découle est :

```

algorithm suiteCarrés(n : entier)
  for i from 1 to n
    print  $i^2$ 

```

pseudocode

Dans cette solution, la variable de contrôle compte simplement le nombre d'itérations. Le calcul du nombre à afficher se fait en fonction de cette variable de contrôle (ici le carré convient). Par une vieille habitude des programmeurs⁵³, une variable de contrôle qui se contente de compter les passages dans la boucle est souvent nommée i . Elle est appelée « indice » de parcours de la boucle.

Cette solution peut être utilisée chaque fois qu'on peut calculer le nombre à afficher en fonction de i .

Cet exemple illustre un premier modèle important d'algorithme de génération de suite, modèle dans lequel l'élément à la i^{e} place, souvent appelé le i^{e} terme, peut être déterminé directement au départ d'une fonction de l'indice i .

$$i \longrightarrow f(i)$$

Exemple 2 - Une suite un peu plus complexe

Écrire un algorithme qui affiche les n premiers nombres de la suite : 1, 2, 4, 7, 11, 16...

Comme nous pouvons le constater, à chaque étape on ajoute un peu plus au nombre précédent.

$$1 \xrightarrow{+1} 2 \xrightarrow{+2} 4 \xrightarrow{+3} 7 \xrightarrow{+4} 11 \xrightarrow{+5} 16 \dots$$

Ici, difficile de partir de la solution de l'exemple précédent car il n'est pas facile de trouver la fonction $f(i)$ qui permet de calculer le nombre à afficher en fonction de i .

Par contre, il est assez simple de calculer ce nombre en fonction du précédent.

$$\text{nb à afficher} = \text{nb affiché juste avant} + i$$

Sauf pour le premier, qui ne peut pas être calculé en fonction du précédent.

⁵³. Née avec le langage FORTRAN où la variable i était par défaut une variable entière.

Mathématiquement, cette suite s'écrirait

$$x_0 = 1$$

$$x_i = x_{i-1} + i$$

Une solution élégante et facilement adaptable à d'autres situations est :

```
algorithm suite(n : integer)
  integer val
  val = 1re valeur à afficher
  for i from 1 to n
    print val
    val = la valeur suivante calculée à partir de
          la valeur courante et/ou de précédentes valeurs
```

pseudocode

qui, dans notre exemple précis, devient :

```
// affiche la suite 1 2 4 7 11 16 ...
algorithm suite(n : integer)
  integer val
  val = 1
  for i from 1 to n
    print val
    val = val + i
```

pseudocode

et, en langage Java, nous pourrions écrire :

```
public static void suite(int n){
2  int value = 1;
  for (int i=1; i <= n; i = i + 1){
    System.out.println(value);
5    value = value + 1;
  }
}
8
```

java

Cet exemple illustre un second modèle de génération de suite dans lequel l'élément à la i^e place, le i^e terme, est exprimé en fonction du ou des précédents termes. Ce genre de suite, s'appelle « suite récurrente ».

Troisième partie

Les tableaux

Chapitre 9

Les tableaux

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

Contenu

9.1	Utilité des tableaux	112
9.2	Définitions	115
9.3	Déclaration - création - initialisation	115
9.4	Tableau et paramètres	118
9.4.1	Un tableau comme paramètre en entrée	118
9.4.2	Un tableau comme paramètre en entrée-sortie	118
9.4.3	Un tableau comme paramètre en langage Java	119
9.4.4	Un tableau comme type de retour	119
9.4.5	Paramétrer la taille	120
9.5	Parcours d'un tableau	121
9.6	Taille logique et taille physique	122
9.7	Les tableaux ont comme premier indice 0	123

9.1 Utilité des tableaux

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

Exemple. Statistiques de ventes.

Un gérant d'une entreprise commerciale souhaite connaître l'impact d'une journée de promotion publicitaire sur la vente de dix de ses produits.

Pour ce faire, les numéros de ces produits (numérotés de 0 à 9 pour simplifier) ainsi que les quantités vendues pendant cette journée de promotion sont encodés au fur et à mesure de leurs ventes. En fin de journée, le vendeur entrera la valeur -1 pour signaler la fin de l'introduction des données. Ensuite, les statistiques des ventes seront affichées.

La démarche générale se décompose en trois parties :

- ▷ le traitement de début de journée, qui consiste essentiellement à mettre les compteurs des quantités vendues pour chaque produit à 0 ;
- ▷ le traitement itératif durant toute la journée : au fur et à mesure des ventes, il convient de les enregistrer, c'est-à-dire d'ajouter au compteur des ventes d'un produit la quantité vendue de ce produit ; ce traitement itératif s'interrompra lorsque la valeur -1 sera introduite ;
- ▷ le traitement final, consistant à communiquer les valeurs des compteurs pour chaque produit.

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


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

    entiers cpt0, cpt1, cpt2, cpt3, cpt4, cpt5, cpt6, cpt7, cpt8, cpt9
    entiers numéroProduit, quantité

    cpt0 = 0
    cpt1 = 0
    cpt2 = 0
    cpt3 = 0
    cpt4 = 0
    cpt5 = 0
    cpt6 = 0
    cpt7 = 0
    cpt8 = 0
    cpt9 = 0

    print "Introduisez le numéro du produit :"
    read numéroProduit

    while numéroProduit  $\neq$  -1

        print "Introduisez la quantité vendue :"
        read quantité

        switch numéroProduit vaut
            0 : cpt0 = cpt0 + quantité
            1 : cpt1 = cpt1 + quantité
            2 : cpt2 = cpt2 + quantité
            3 : cpt3 = cpt3 + quantité
            4 : cpt4 = cpt4 + quantité
            5 : cpt5 = cpt5 + quantité
            6 : cpt6 = cpt6 + quantité
            7 : cpt7 = cpt7 + quantité
            8 : cpt8 = cpt8 + quantité
            9 : cpt9 = cpt9 + quantité

        print "Introduisez le numéro du produit :"
        read numéroProduit

    print "quantité vendue de produit 0 :", cpt0
    print "quantité vendue de produit 1 :", cpt1
    print "quantité vendue de produit 2 :", cpt2
    print "quantité vendue de produit 3 :", cpt3
    print "quantité vendue de produit 4 :", cpt4
    print "quantité vendue de produit 5 :", cpt5
    print "quantité vendue de produit 6 :", cpt6
    print "quantité vendue de produit 7 :", cpt7
    print "quantité vendue de produit 8 :", cpt8
    print "quantité vendue de produit 9 :", cpt9
```

Que se passerait-il si le nombre de produits à traiter était de 20, voire 1000 ? Une simplification de l'écriture s'impose. La solution est apportée par un nouveau type de variables : les **variables indicées** ou **tableaux** (*arrays* en anglais).

Au lieu d'avoir à manier dix compteurs distincts (`cpt0`, `cpt1`, etc.), nous allons envisager une seule « grande » variable `cpt` compartimentée en dix « cases » ou « sous-variables » (appelées aussi les « éléments » du tableau). Elles se distingueront les unes des autres par un numéro (un « indice ») : `cpt0` deviendrait ainsi `cpt[0]`, `cpt1` deviendrait `cpt[1]`, et ainsi de suite jusqu'à `cpt9` qui deviendrait `cpt[9]`.

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

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

Voici la version de notre solution avec tableau.

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

    array of 10 entiers cpt
    entiers i, numéroProduit, quantité

    for i from 0 to 9
        cpt[i] = 0

    print "Introduisez le numéro du produit : "
    read numéroProduit

    while numéroProduit ≠ -1

        print "Introduisez la quantité vendue : "
        read quantité

        cpt[numéroProduit] = cpt[numéroProduit] + quantité

        print "Introduisez le numéro du produit : "
        read numéroProduit

    for i from 0 to 9
        print "quantité vendue de produit ", i, " : ", cpt[i]
```

pseudocode

9.2 Définitions

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



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

La **taille** d'un tableau est le nombre de ses éléments. Attention, la taille d'un tableau ne peut pas être modifiée pendant son utilisation, elle est déterminée à la création du tableau.

Un tableau est de type **référence**.

9.3 Déclaration - création - initialisation

Afin de pouvoir utiliser un tableau, il sera nécessaire de distinguer :



- ▷ la **déclaration** qui consiste à signaler qu'une variable est un tableau. Dans un programme, cette déclaration a pour effet de réserver un emplacement mémoire qui contiendra une référence vers les cases du tableau.
- ▷ la **création** du tableau consiste à réserver l'emplacement mémoire qui contiendra les différentes valeurs. Pour le créer, il faut connaître sa taille.
- ▷ l'**initialisation** du tableau consiste à donner ses premières valeurs aux différents éléments du tableau.

Nous supposons dans la suite que **size** est le nombre d'éléments et **Type** est le type des éléments que l'on trouvera dans le tableau.

Dans nos algorithmes, pour déclarer un tableau, nous écrirons :

```
array of Type Identifier
```

```
// Par exemple  
array of integer myArray
```

pseudocode

La création d'un tableau s'écrira :

```
name = array of size Type
```

```
// Par exemple  
myArray = array of 10 integer
```

pseudocode

Pour déclarer et créer un tableau, nous écrirons en choisissant la formulation longue ou courte :

```
array of Type Identifier = array of size Type
array of size Type Identifier
```

```
// Par exemple
array of real myArray = array of 10 real
array of 1000 integer otherArray
```

pseudocode

Tous les types sont permis mais tous les éléments sont du même type.

En langage Java, c'est assez semblable. Les caractères associés aux tableaux, sont la paire de crochets ; []. Pour déclarer un tableau, nous écrirons :

ArrayDeclarationExpression:
Type[] *Identifier*

Pour créer un tableau, préalablement déclarer, nous écrirons :

ArrayCreationExpression:
 new *Type*[*Expression*]

L'accès aux éléments d'un tableau se fait grâce aux crochets pour nos algorithmes, en langage Java et dans beaucoup d'autres langages. Comme suit :

```
myArray[i] // élément du tableau 'myArray' à l'indice i
```

pseudocode

La première case du tableau porte l'indice 0⁵⁴, la dernière l'indice `size-1`. C'est considéré comme une erreur d'indiquer un indice qui ne correspond pas à une case du tableau (trop petit ou trop grand). Par exemple, si on déclare et crée le tableau `myArray` :

```
array of 100 integers myArray
```

pseudocode

il est interdit d'utiliser `myArray[-1]` ou `myArray[100]`.

L'initialisation d'un tableau, c'est-à-dire l'assignation de ses premières valeurs, peut se faire élément par élément en utilisant les crochets ou en utilisant une notation compacte entre accolades .

54. Nous considérons que la première case du tableau porte le numéro 0 comme c'est le cas dans beaucoup de langages de programmation (comme Java par exemple). Plus loin, nous verrons une notation alternative qui permet de choisir un autre numéro de début pour le tableau, plus naturel pour certains problèmes.

```
myArray = {2, 3, 7}           // équivalent à myArray[0] = 2, myArray[1] = 3,
                               // myArray[2] = 7
myArray = {0, ..., 0}        // tous les éléments du tableau sont initialisés à 0
```

pseudocode

Il est également possible d'utiliser cette initialisation « entre accolade » en langage Java. Elle s'appelle *ArrayInitializer*. Par contre, seule la première forme est autorisée.

ArrayInitializer:

{ *[VariableInitializerList]* }

VariableInitializerList:

Expression [, *Expression*]

Exemples

```
array of 3 integers myArray           // déclaration et création
myArray = {2, 3, 7}                   // initialisation
```

pseudocode

```
1  int[] myArray;
   myArray = new int[3];
   myArray[0] = 2;
4  myArray[1] = 3;
   myArray[2] = 7;

7  myArray = new int[] {-1, 3, 12, 5};

   int[] myArray2 = {3, 14, 15, 92};
10
```

java

ligne 7

Pour utiliser un *ArrayInitializer*, il faut créer et initialiser le tableau dans la même instruction. Dans ce cas, le langage interdit de renseigner le nombre d'éléments du tableau... le compilateur peut le déduire.

Cette instruction crée un nouveau tableau qui écrase l'autre.

ligne 9

Lorsque l'on initialise le tableau avec un *ArrayInitializer* lors de la déclaration, la partie création peut être omise.

Remarque. Chaque élément d'un tableau doit être manié avec la même précaution qu'une variable simple, c'est-à-dire qu'on ne peut utiliser un élément du tableau qui n'aurait pas été préalablement affecté ou initialisé.

9.4 Tableau et paramètres

Le type *tableau* étant un type à part entière, il est tout-à-fait éligible comme type pour les paramètres et la valeur de retour d'un algorithme. De manière générale là où pouvait apparaître un type (entier, pseudo-réel...) peut apparaître un tableau. Toujours. Voyons cela en détail.

Nous avons vu dans la section 7.3 p. 79 les différentes sortes de paramètres. Voyons ce qu'il en est pour chacun d'eux lorsque ce paramètre est un tableau.

9.4.1 Un tableau comme paramètre en entrée

↓ : indique que l'algorithme va consulter les valeurs du tableau reçu en paramètre. Les éléments doivent donc avoir été initialisés avant d'appeler l'algorithme. Exemple :

```
// Affiche les éléments d'un tableau
algorithm display(myArray↓ : array of 10 integers)
  for i from 0 to 9
  |   print myArray[i]

// Utilisations possible
array of 10 integers aArray
aArray = {2,3,5,7,11,13,17,19,23,29}
display(aArray)
```

pseudocode

Rappelons qu'il s'agit du passage par défaut si aucune flèche n'est indiquée.

9.4.2 Un tableau comme paramètre en entrée-sortie

↓↑ : indique que l'algorithme va consulter/modifier les valeurs du tableau reçu en paramètre. Exemple :

```
// Inverse le signe des éléments du tableau
algorithm reverseSigne(myArray↓↑ : array of 10 integers)
  for i from 0 to 9
  |   myArray[i] = -myArray[i]

// Utilisation possible
array of integers aArray
aArray = array of integers {2,-3,5,-7,11,13,17,-19,23,29}
reverseSigne(aArray)
```

pseudocode

9.4.3 Un tableau comme paramètre en langage Java

En langage Java, le passage de paramètres se fait toujours par valeur. Comme un tableau est de type référence, c'est bien la valeur de la référence qui est passée en paramètre. En ce sens, il sera possible de modifier les valeurs d'un tableau reçu mais pas de remplacer le tableau reçu par un autre. Exemples :

```

1  public static void reverseSigne(int[] is){
    for (int i=0; i < is.length; i = i + 1){
        is[i] = -is[i];
4   }
    }

7  // Utilisation
   int[] myArray = {2,-3,5,-7,11,13,17,-19,23,29};
   reverseSigne(myArray);

```

java

```

    public static void fail(double[] ds){
        ds = new double[] {2, 2.61};
3   }

6  // Utilisation
   double[] myArray = {1, 3.14};
   fail(myArray);
9  // 3.14 et pas 2.61
   System.out.println("Value:␣" + myArray[1]);

```

java



9.4.4 Un tableau comme type de retour

Comme pour n'importe quel autre type, un algorithme ou un programme peut retourner un tableau. Ce sera à lui de le déclarer et de lui donner des valeurs. Exemple :

```

// Retourne un tableau initialisé de 10 entiers
algorithm create() → array of integers
    array of 10 integers myArray
    for i from 0 to 9
        | myArray[i] = i
    return myArray

// Utilisation possible
array of integers aArray
aArray = create()

```

pseudocode

```
1 public int[] create(){
    is[] = new int[10];
    for (int i=0; i < 10; i = i + 1){
4       is[i] = i;
    }
    return is;
7 }

10 // Utilisation possible
    int[] aArray = create();
```

java

9.4.5 Paramétrer la taille

Un tableau connaît sa taille dès lors qu'elle existe c'est-à-dire qu'il est créé. Elle est accessible grâce au mot `length` comme suit :

```
array of 10 reals myArray
integer size = myArray.length

array of reals otherArray = {3.14, 2.71}
integer otherSize = otherArray.length
```

pseudocode

```
1 double[] myArray;
  int size = myArray.length;

4 double[] otherArray = {3.14, 2.71};
  int otherSize = otherArray.length;
```

java

Un tableau peut être passé en paramètre à un algorithme ou à une méthode et ce, quelle que soit la taille du tableau passé en paramètre. L'algorithme ou la méthode aura comme paramètre un tableau qui connaît sa taille. Lorsque l'algorithme ou la méthode recevra le paramètre effectif, la taille sera déterminée.

Exemple :

```
// Affiche un tableau de taille quelconque
algorithm display(myArray : array of integers)
  print "Tableau de ", myArray.length, " éléments".
  for i from 0 to myArray.length - 1 // Attention ! le dernier élément //
  est à l'indice taille -1
    print myArray[i]

// Utilisation possible
array of 2 entiers entiers
entiers[0] = 56
entiers[1] = -3
afficher(entiers)
entiers = array of integers {6, 12, 18}
afficher(entiers)
```

pseudocode

La fiche 5 page 217 récapitule tout ça.

9.5 Parcours d'un tableau

Dans la plupart des problèmes que vous rencontrerez vous serez amenés à parcourir un tableau. Il est important de maîtriser ce parcours. Examinons les situations courantes et voyons quelles solutions conviennent.

Parcours complet

Nous avons déjà vu dans les exemples comment parcourir tous les éléments d'un tableau. Une boucle **pour** fait généralement l'affaire. La fiche 6 page 219 décrit comment afficher tous les éléments d'un tableau. Il est possible de faire autre chose avec ces éléments : les sommer, les comparer...

Parcours partiel

Parfois, il n'est pas nécessaire de parcourir le tableau jusqu'au bout car un arrêt prématuré survient dès qu'une condition est remplie.

Par exemple : recherche d'un élément et arrêt dès qu'il est trouvé ; vérification que toutes les valeurs sont non nulles mais l'on vient d'en trouver une ; vérification que la suite des éléments est positives alors que l'on vient d'en trouver un négatif ou nul.

Pour résoudre ce problème, la structure **tant que** avec un test d'arrêt fait bien l'affaire.

La fiche 7 page 220 détaille un parcours partiel.

9.6 Taille logique et taille physique

Parfois, la taille du tableau n'est pas connue à l'avance.

Imaginons, par exemple, qu'il s'agisse de demander des valeurs à l'utilisateur et de les stocker dans un tableau pour un traitement ultérieur. Supposons que l'utilisateur va indiquer la fin des données par une valeur sentinelle. Impossible de savoir, à priori, combien de valeurs il va entrer et, par conséquent, la taille à donner au tableau.

Une solution est de créer un tableau suffisamment grand pour tous les cas⁵⁵ quitte à n'en n'utiliser qu'une partie.

Mais comment savoir quelle est la partie du tableau qui est effectivement utilisée ?

Comprenez bien qu'il **n'y a pas de concept de case vide**. Rien ne différencie une case utilisée d'une case non utilisée. La manière de s'en sortir est de stocker les valeurs dans la partie basse du tableau (à gauche) et de retenir dans une variable le nombre de cases effectivement utilisées.



La **taille physique** d'un tableau est le nombre de cases qu'il contient. Sa **taille logique** est le nombre de cases actuellement utilisées.

Exemple. Voici un tableau qui ne contient pour l'instant que quatre nombres.

10	4	3	7	?	?	?	?	?	?
----	---	---	---	---	---	---	---	---	---

taillePhysique = 10 tailleLogique = 4

Les cases indiquées par "?" ne sont pas vides ; elles peuvent être non initialisées (et il n'est pas possible de tester si une case est non initialisée) ou bien contenir une valeur qui n'est pas pertinente.

Exemple. L'algorithme suivant demande des valeurs positives à l'utilisateur et les stocke dans un tableau (maximum 1000). Toute valeur négative ou nulle est une valeur sentinelle.

⁵⁵. En tout cas, suffisamment grand pour tous les cas qu'on accepte de prendre en compte ; il faudra bien fixer une limite.

```

algorithm stockValues()
    array of 1000 integers myArray
    integer nbValues                                // C'est la taille logique
    integer value
    nbValues = 0
    read value
    while value > 0 AND nbValues < myArray.length
        myArray[nbValues] = value
        nbValues = nbValues + 1
        read value

    if value > 0 then                                // Pourquoi tester value et pas nbValues ?
        print "La limite physique a été atteinte"

```

pseudocode

Lors de l'implémentation de l'algorithme dans un langage de programmation, si la limite physique du tableau est atteinte, une erreur d'exécution va survenir. Une telle erreur survient dès que le programme essaie d'accéder à une case mémoire « qui n'existe pas ». Traduisez l'exemple en Java sans le if final et regardez l'erreur générée. Nous y reviendrons au laboratoire.

9.7 Les tableaux ont comme premier indice 0

Les tableaux ont comme premier indice 0, Un tableau de taille n a toujours ses indices allant de 0 à $n-1$.

Parfois le problème à résoudre donne envie d'avoir des indices dont le premier ne commence pas à 0. Par exemple allant de 1 à 7. Dans ces cas, il est toujours possible de s'arranger pour commencer à partir de 0.

Exemple. Imaginons l'algorithme qui permet de convertir un numéro de jour en son intitulé : 1 donne "lundi", 2 donne "mardi", ... Une solution possible, sans tableau, serait :

```

algorithm dayName(dayNumber : integer) → String
    switch dayNumber
        case 1: return "lundi"
        case 2: return "mardi"
        case 3: return "mercredi"
        case 4: return "jeudi"
        case 5: return "vendredi"
        case 6: return "samedi"
        case 7: return "dimanche"

```

pseudocode

Mais une solution plus élégante passe par l'utilisation d'un tableau. Appelons-le `dayString` et stockons-y les noms des jours comme illustré ci-dessous.

1	2	3	4	5	6	7
"lundi"	"mardi"	"mercredi"	"jeudi"	"vendredi"	"samedi"	"dimanche"

Si le tableau commence à l'indice 1 jusqu'à l'indice 7, pour obtenir le nom d'un jour, il suffirait d'écrire : `dayString[dayNumber]`.

Comme les tableaux commencent toujours à l'indice 0, il faudra plutôt considérer le tableau ci-dessous et écrire `dayString[dayNumber-1]`.

0	1	2	3	4	5	6
"lundi"	"mardi"	"mercredi"	"jeudi"	"vendredi"	"samedi"	"dimanche"

Chapitre 10

Algorithmes d'insertions et de recherches

Souvent lorsque les applications stockent des données. Ces données changent. Il est nécessaire d'en ajouter, de modifier certaines, d'en rechercher, d'en supprimer.

Contenu

10.1	Insertion dans un tableau non trié	126
10.1.1	Inscription	126
10.1.2	Vérifier une inscription	127
10.1.3	Désinscription	128
10.1.4	Exercices	129
10.2	Insertion dans un tableau trié	130
10.2.1	Rechercher la position d'une inscription	130
10.2.2	Inscrire un étudiant	132
10.2.3	Désinscription	134
10.2.4	Intérêt de trier les valeurs ?	135
10.3	La recherche dichotomique	135
10.4	Introduction à la complexité	139
10.4.1	Une approche pratique : la simulation numérique	139
10.4.2	Une approche théorique : la complexité	139

Exemple Imaginons qu'une école organise un événement libre et gratuit pour les étudiants et les étudiantes. Mais pour y assister, il est nécessaire de s'inscrire. Nous voulons fournir ce qu'il faut pour :

- ▷ inscrire un ou une étudiant · e ;
- ▷ vérifier l'inscription ;
- ▷ désinscrire ;
- ▷ afficher la liste des personnes inscrites.

Nous allons, pour l'exemple, stocker les numéros d'étudiants et d'étudiantes dans un tableau⁵⁶. Comme le tableau ne sera généralement pas rempli, nous allons créer un tableau « trop grand » et gérer sa taille logique. Nous avons deux variables :

- ▷ `registered` : le tableau des numéros d'étudiants
- ▷ `nRegistered` : le nombre d'étudiant · es actuellement inscrit · es (la taille logique du tableau)

Nous allons envisager deux cas :

1. les numéros seront placés dans le tableau sans ordre imposé ;
2. les numéros seront placés dans l'ordre.

10.1 Insertion dans un tableau non trié

Les valeurs sont stockées dans le tableau dans un ordre quelconque. Par exemple :

`registered` =

42010	41390	43342	42424	?	?	?	?	...
-------	-------	-------	-------	---	---	---	---	-----

`nRegistered` = 4

Il y a pour le moment quatre personnes inscrites dont les numéros sont ceux repris dans les quatre premières cases du tableau.

10.1.1 Inscription

Pour inscrire un étudiant ou une étudiante, il suffit de l'ajouter à la suite dans le tableau. Par exemple, en partant de la situation décrite ci-dessus, l'inscription de l'étudiant numéro 42123 aboutit à la situation suivante :

`registered` =

42010	41390	43342	42424	42123	?	?	?	...
-------	-------	-------	-------	-------	---	---	---	-----

`nRegistered` = 5

⁵⁶. Ce n'est bien sûr pas la bonne manière de faire pour une application réelle pour laquelle nous utiliserions probablement une base de données... mais c'est une autre histoire... et un autre cours.

L'algorithme est assez simple :

```
// Inscrit un étudiant de numéro donné.  
algorithm register(registered $\downarrow\uparrow$  : array of integers, nRegistered $\downarrow\uparrow$  : integer,  
                  number $\downarrow$  : integer)  
    // On peut imaginer vérifier que l'étudiant n'est pas déjà inscrit  
    // On peut imaginer vérifier qu'il reste de la place  
    // (c-à-d que le tableau n'est pas plein)  
    registered[nRegistered] = number  
    nRegistered = nRegistered + 1
```

pseudocode

En langage Java, il n'est pas possible de passer `nRegistered` en entrée sortie puisque le passage de paramètre se fait par valeur et que `int` est un type primitif. Il faudra que ce soit en valeur de retour. Nous pourrions écrire :

```
public static int register(int[] registered, int nRegistered,  
    int number){  
3    // idem  
    registered[nRegistered] = number;  
    return nRegistered + 1;  
6 }
```

java

Dans ce cas, c'est bien le code appelant qui devra maintenir la valeur de la taille logique à jour.

10.1.2 Vérifier une inscription

Pour vérifier si un étudiant est déjà inscrit, il faut le rechercher dans la partie utilisée du tableau. Cette recherche se fait via un parcours avec sortie prématurée.

L'algorithme ne va pas se contenter de retourner un booléen précisant si le numéro est trouvé mais va retourner l'indice dans le tableau de ce numéro, -1 sinon.

```
// Vérifie si un étudiant est inscrit et donne sa position (-1 si non inscrit)
algorithm check(registered↓ : array of n integers, nRegistered↓ : integer,
               number↓ : integer) → integer
    integer i
    i = 0
    while i < nRegistered ET registered[i] ≠ number
    |   i = i + 1

    if i < nRegistered then
    |   return i
    else
    |   return -1
```

pseudocode

La traduction en langage Java ne pose aucune difficulté et est laissée en exercice.

La fiche 9 page 223 reprend cet algorithme dans un cadre plus général.

10.1.3 Désinscription

Pour désinscrire un étudiant, il faut le trouver dans le tableau et l'enlever. Attention, il ne peut pas y avoir de trou dans un tableau (il n'y a pas de case vide). Le plus simple est d'y déplacer le dernier élément. Par exemple, reprenons la situation après inscription de l'étudiant numéro 42123.

registered =								
42010	41390	43342	42424	42123	?	?	?	...
nRegistered = 5								

La désinscription de l'étudiant numéro 41390 donnerait ce qui suit.

Nous avons volontairement indiqué le 42123 en 5^e position. Il est toujours là mais ne sera pas considéré par les algorithmes puisque cette case est au-delà de la taille logique (donnée par la variable nRegistered).

registered =								
42010	42123	43342	42424	42123	?	?	?	...
nRegistered = 4								

L'algorithme est assez simple à écrire en utilisant la recherche écrite juste avant :


```
// Désinscrit l'étudiant donné
algorithm unsubscribe( registered $\downarrow\uparrow$  : array of integers,
                      nRegistered $\downarrow\uparrow$  : integer, number $\downarrow$  : integer )
    integer pos
    pos = check(registered, nRegistered, number)
    // On pourrait vérifier et générer une erreur si l'étudiant n'est pas inscrit
    registered[pos] = registered[nRegistered-1]
    nRegistered = nRegistered - 1
```

pseudocode

Comme précédemment, en langage Java, il n'est pas possible de passer `nRegistered` en entrée sortie. Il faudra que ce soit en valeur de retour. Nous pourrions écrire :

```
public static int unsubscribe(int[] registered,
2   int nRegistered, int number){
    int pos = check(registered, nRegistered, number);
    registered[pos] = registered[nRegistered - 1];
5   return nRegistered - 1;
}
```

java

10.1.4 Exercices

1. Éviter la double inscription.
Comment modifier l'algorithme d'inscription pour s'assurer qu'un étudiant ne s'inscrive pas deux fois ?
2. Limiter le nombre d'inscriptions.
Comment modifier l'algorithme d'inscription pour refuser une inscription si le nombre maximal de participant est atteint en supposant que ce maximum est égal à la taille physique du tableau ?
3. Vérifier la désinscription.
Que se passerait-il avec l'algorithme de désinscription tel qu'il est si on demande à désinscrire un étudiant non inscrit ? Que suggérez-vous comme changement ?
4. Optimiser la désinscription.
Dans l'algorithme de désinscription tel qu'il est, voyez-vous un cas où on effectue une opération inutile ? Avez-vous une proposition pour l'éviter ?

10.2 Insertion dans un tableau trié

Imaginons à présent que nous maintenons un ordre dans les données du tableau. En reprenant l'exemple utilisé pour les données non triées, nous avons :

registered =								
41390	42010	42424	43342	?	?	?	?	...
nRegistered = 4								

- Qu'est-ce que ça change au niveau des algorithmes ?
- Beaucoup !

Par exemple, plus question de placer un nouvel inscrit à la suite des autres ; il faut trouver sa place.

10.2.1 Rechercher la position d'une inscription

Tous les algorithmes que nous allons voir dans le cadre de données triées ont une partie en commun : la recherche de la position du numéro, s'il est présent ou de la position où il aurait dû être s'il est absent. Commençons par cela.

L'algorithme est assez proche de celui de la vérification d'une inscription vu précédemment si ce n'est que l'algorithme peut s'arrêter dès qu'un numéro plus grand est trouvé. Nous allons décomposer l'algorithme en deux parties : la première pour la recherche de la position à laquelle devrait se trouver le numéro et la seconde pour vérifier que c'est bien le bon numéro qui se trouve à cette position.

```
// Recherche un étudiant.
// return : la position où a été trouvé l'étudiant ou -1 s'il est pas inscrit
algorithm find( registered↓ : array of integers, nRegistered↓ : integer,
               number↓ : integer ) → integer
    integer pos = findPosition(registered, nRegistered, number)
    if registered[pos] == number then
        return pos
    else
        return -1

// Recherche une position
// return : la position a laquelle est ou devrait être le numéro
algorithm findPosition( registered↓ : array of integers, nRegistered↓ : integer,
                      number↓ : integer ) → integer
    integer pos = 0
    while pos < nRegistered AND registered[pos] < number
        pos = pos + 1
    return pos
```

pseudocode

Comprenez-vous pourquoi :

- ▷ `registered[nRegistered] < number` contient un '`<`' dans la condition du tant que et pas un '`≠`' ?
- ▷ la condition du if est composée de deux parties et utilise un `=` ?

La liste étant triée il faut s'arrêter de chercher dès que la valeur n'est plus inférieure à la valeur cherchée. Dès qu'elle est plus grande c'est que la valeur **n'est pas** dans la liste.

De même pour la condition du if, il faut d'abord tester si la recherche n'est pas arrivée à la fin du nombre d'inscrits et d'inscrites. Ensuite, il faut voir si le numéro à la position trouvée est bien le bon.

La traduction en Java est immédiate. Le if tel que présenté dans l'algorithme peut se réécrire grâce à l'opérateur ternaire de Java. Profitons en pour le présenter.

ConditionalExpression:

Condition ? Expression : Expression

Si la condition est vraie, l'expression prend la première valeur et si la condition est fausse, ce sera la seconde valeur. Par exemple : `heure < 12 ? "matin" : "soir"` vaut "soir" si `heure` vaut 15.

```

2  /**
   * Recherche une inscription dans le tableau.
   *
   * @param registered le tableau d'inscrit·es
5  * @param nRegistered le nombre d'inscrit·es
   * @param number le numéro à chercher
   * @return la position dans le tableau, -1 si non présent
8  */
   public static int find(int[] registered, int nRegistered,
       int number){
11     int pos = findPosition(registered, nRegistered, number);
       return registered[pos] == number ? pos : -1;
   }
14
15  /**
   * Recherche la position que le nombre soit présent ou pas.
17  * @param registered le tableau d'inscrit·es
   * @param nRegistered le nombre d'inscrit·es
   * @param number le numéro à chercher
20  * @return la position à laquelle est ou devrait être le numéro.
   */
   public static int findPosition(int[] registered, int nRegistered,
23     int number){
       int pos = 0;
       while (pos < nRegistered && registered[pos] < number){
26         pos = pos + 1;
       }
       return pos;
29  }

```

java

10.2.2 Inscrire un étudiant

L'inscription d'un étudiant se fait en trois étapes :

1. recherche de l'étudiant via l'algorithme vu à l'instant, ce qui nous donne la place où le placer ;
Nous ne vérifions pas si l'étudiant ou l'étudiante est déjà présent · e dans le tableau.
2. libération de la place pour l'y mettre ;
Cette opération n'est pas triviale. Si vous tenez des cartes en main, il est facile d'insérer une nouvelle carte à n'importe quelle position. Avec un tableau, il en va tout autrement ; pour insérer un élément à un endroit donné, il faut décaler tous les suivants d'une position sur la droite.
3. placer l'élément à la position libérée.

Ce qui nous donne :

```
// Inscrit un étudiant de numéro donné.
algorithm subscribe( registered↑↓ : array of integers, nRegistered↑↓ : integer,
                    number↓ : integer )

    integer pos
    pos = findPosition(registered, nRegistered, number)
    shiftRight(registered, pos, nRegistered-1)
    registered[pos] = number
    nRegistered = nRegistered + 1

// Décale d'une position à droite les éléments entre la position début et fin
algorithm shiftRight( tab↑↓ : array of integers, begin↓ : integer, end↓ : integer )
    for i from end to begin by -1
    |   tab[i+1] = tab[i]
```

pseudocode

En langage Java, la traduction est immédiate :

```
/**
 * Inscrit un ou une étudiant.e.
3  *
 * @param registered le tableau d'inscrit.es
 * @param nRegistered le nombre d'inscrit.es
6  * @param number le numéro à chercher
 */
public static void subscribe(int[] registred, int nRegistered,
9     int number){
    int pos = findPosition(registered, nRegistered, number);
    shiftRight(registred,pos, nRegistered);
12    registered[pos] = number;
    nRegistered = nRegistered + 1;
}
15
/**
 * Décale les éléments du tableau de begin à end un à droite.
18 * @param tab le tableau
 * @param begin l'indice de départ
 * @param end l'indice de fin
21 */
public shiftRight(int[] tab, int begin, int end){
    for (int i = end, i >= begin; i = i - 1){
24        tab[i + 1] = tab[i];
    }
}
27
```

java

10.2.3 Désinscription

Ici, il va falloir décaler vers la gauche les éléments qui suivent celui à supprimer.

```
// désinscrit l'étudiant donné
algorithm unsubscribe( registered $\downarrow\uparrow$  : array of n integers, nRegistered $\downarrow\uparrow$  : integer,
                                number $\downarrow$  : integer )

    integer pos
    pos = findPosition(registered, nRegistered, number)
    shiftLeft(registered, pos, nRegistered-1)
    nRegistered = nRegistered - 1

// Décale d'une position à gauche les éléments entre la position début et fin
algorithm shiftLeft( tab $\downarrow\uparrow$  : array of n integers,
                                begin $\downarrow$  : integer, end $\downarrow$  : integer )

    for i from begin to end
    |   tab[i-1] = tab[i]
```

pseudocode

La fiche 10 page 225 reprend cet algorithme dans un cadre plus général.

En langage Java, la traduction est immédiate :

```

/**
 * Désinscrit un ou une étudiant·e.
3  *
 * @param registered le tableau d'inscrit·es
 * @param nRegistered le nombre d'inscrit·es
6  * @param number le numéro à chercher
 */
public static void subscribe(int[] registred, int nRegistered,
9      int number){
    int pos = findPosition(registered, nRegistered, number);
    shiftLeft(registred,pos, nRegistered);
12  nRegistered = nRegistered - 1;
}

15 /**
 * Décale les éléments du tableau de begin à end un à gauche
 * (et écrase donc le premier).
18 * @param tab le tableau
 * @param begin l'indice de départ
 * @param end l'indice de fin
21 */
public shiftLeft(int[] tab, int begin, int end){
    for (int i = begin, i <= end; i = i + 1){
24        tab[i - 1] = tab[i];
    }
}
27

```

java

10.2.4 Intérêt de trier les valeurs ?

Est-ce que trier les valeurs est vraiment intéressant ?

La recherche est un peu plus rapide. En moyenne deux fois plus rapide si l'élément ne s'y trouve pas. Par contre, l'ajout et la suppression d'un élément sont plus lents. Les algorithmes sont plus complexes à écrire et à comprendre. Le gain ne paraît pas évident et en effet, en l'état, il ne l'est pas.

Mais c'est sans compter un autre algorithme de recherche, beaucoup plus rapide, la recherche dichotomique, que nous allons voir maintenant.

10.3 La recherche dichotomique

La recherche dichotomique est un algorithme de recherche d'une valeur dans un tableau trié. Il a pour essence de réduire à chaque étape la taille de l'ensemble de recherche de moitié, jusqu'à ce qu'il ne reste qu'un seul élément dont la valeur devrait être celle recherchée, sauf bien entendu en cas d'inexistence de cette valeur dans l'ensemble de départ.

Pour l'expliquer, nous allons considérer un tableau d'entiers complètement rempli. Il sera facile d'adapter la solution à un tableau partiellement rempli (avec une taille

logique) ou un tableau contenant tout autre type de valeurs qui peut s'ordonner.

Description de l'algorithme

Soit *value* la valeur recherchée dans une zone délimitée par les indices *leftIndex* et *rightIndex*. Il faut commencer par déterminer l'élément médian, c'est-à-dire celui qui se trouve « au milieu » de la zone de recherche⁵⁷ ; son indice sera déterminé par la formule

$$\text{medianIndex} = (\text{leftIndex} + \text{rightIndex}) \text{ DIV } 2$$



L'algorithme compare alors *value* avec la valeur de cet élément médian ; il est possible que ce soit la valeur cherchée ; sinon, il partage la zone de recherche en deux parties : une qui **ne contient certainement pas** la valeur cherchée et une qui **pourrait la contenir**. C'est cette deuxième partie qui devient la nouvelle zone de recherche. Il adapte *leftIndex* ou *rightIndex* en conséquence. L'algorithme réitère le processus jusqu'à ce que la valeur cherchée soit trouvée ou que la zone de recherche soit réduite à sa plus simple expression, c'est-à-dire un seul élément.

Exemple de recherche fructueuse

Supposons que l'on recherche la valeur **23** dans un tableau de 20 entiers. La zone ombrée représente à chaque fois la partie de recherche, qui est au départ le tableau trié dans son entièreté. Au départ, *leftIndex* vaut 0 et *rightIndex* vaut 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Première étape : $\text{medianIndex} = (0 + 19) \text{ DIV } 2$, c'est-à-dire 9.

Comme la valeur en position 9 est 15, la valeur cherchée doit se trouver à sa droite, et on prend comme nouvelle zone de recherche celle délimitée par *leftIndex* qui vaut 10 et *rightIndex* qui vaut 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Deuxième étape : $\text{medianIndex} = (10 + 19) \text{ DIV } 2$, c'est-à-dire 14.

Comme on y trouve la valeur 25, on garde les éléments situés à la gauche de celui-ci ; la nouvelle zone de recherche est [10, 13].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Troisième étape : $\text{medianIndex} = (10 + 13) \text{ DIV } 2$, c'est-à-dire 11 où se trouve l'élément 20. La zone de recherche devient *leftIndex* vaut 12 et *rightIndex* vaut 13.

⁵⁷. Cet élément médian n'est pas tout à fait au milieu dans le cas d'une zone contenant un nombre pair d'éléments.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Quatrième étape : $\text{medianIndex} = (12 + 13) \text{ DIV } 2$, c'est-à-dire 12 où se trouve la valeur cherchée ; la recherche est terminée.

Exemple de recherche infructueuse

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Arrivé à ce stade, la zone de recherche s'est réduite à un seul élément. Ce n'est pas celui qu'on recherche mais c'est à cet endroit qu'il se serait trouvé ; c'est donc là qu'on pourra éventuellement l'insérer. Le paramètre de sortie prend la valeur 5.

Algorithme

```

algorithm dichotomousResearch(
    myArray↓ : array of integers, value↓ : integer, pos↓↑ : integer ) → boolean
    integer rightIndex, leftIndex, medianIndex
    integer candidate
    boolean isFind

    leftIndex = 0
    rightIndex = myArray.length - 1
    isFind = false

    while NON isFind AND leftIndex ≤ rightIndex
        medianIndex = (leftIndex + rightIndex) DIV 2
        candidate = myArray[medianIndex]
        if candidate == value then
            isFind = vrai
        else if candidate < value then
            leftIndex = medianIndex + 1           // on garde la partie droite
        else
            rightIndex = medianIndex - 1         // on garde la partie gauche

    if isFind then
        pos = medianIndex
    else
        pos = leftIndex           // dans le cas où la valeur n'est pas trouvée,
        // on vérifiera que leftIndex donne la valeur où elle pourrait être insérée.

    return isFind

```

pseudocode

La traduction de l'algorithme en Java est laissé à titre d'exercice. La difficulté réside dans le choix de la valeur de retour et du paramètre en entrée/sortie **pos**. En Java, il faudra choisir de ne pas obtenir un booléen mais bien l'indice de la valeur si elle est trouvée et -1 sinon. Nous ne pourrions pas avoir recevoir l'indice « à insérer » dans le cas où la valeur ne se trouve pas dans le tableau.

10.4 Introduction à la complexité

L'algorithme de recherche dichotomique est beaucoup plus rapide que l'algorithme de recherche linéaire.

Mais qu'est-ce que cela veut dire exactement ?

En est-on sûr ?

Comment le mesure-t-on ?

Quels critères utilise-t-on ?

De façon générale, comment comparer la vitesse de deux algorithmes différents qui résolvent le même problème.

10.4.1 Une approche pratique : la simulation numérique

Une manière naïve de comparer la rapidité des algorithmes serait de les traduire dans un langage de programmation, de les exécuter et de comparer les temps d'exécution.

Cette technique pose toutefois quelques problèmes :

- ▷ il faut que les programmes soient exécutés dans des environnements strictement identiques ce qui n'est pas toujours le cas ou facile à vérifier ;
- ▷ certains environnements peuvent être favorables à un algorithme par rapport à l'autre ce qui ne serait pas le cas d'un autre environnement ;
Par exemple, certaines architectures sont plus rapides dans les calculs entiers mais moins dans les calculs flottants.
- ▷ le test ne porte que sur un (voir quelques uns) jeu(x) de tests.
Comment en tirer un enseignement général ?
Que se passerait-il avec des données plus importantes ?
Avec des données différentes ?

En fait, un chiffre précis ne nous intéresse pas. Ce qui est vraiment intéressant, c'est de savoir comment l'algorithme va se comporter avec de grandes données. C'est ce qu'apporte l'approche suivante.

10.4.2 Une approche théorique : la complexité

Le principe de cette approche est de déduire de l'algorithme le nombre d'opérations de base à effectuer en fonction de la **taille** du problème à résoudre et en déduire comment il va se comporter sur de « gros » problèmes ?

Dans le cas de la recherche dans un tableau, la taille du problème est la taille du tableau dans lequel l'élément est cherché. Nous pouvons considérer que l'opération de base est la comparaison avec un élément du tableau.

La question est alors la suivante : pour un tableau de taille n , à combien de comparaisons faut-il procéder pour trouver l'élément (ou se rendre compte de son absence) ?

Pour la recherche linéaire

Cela dépend évidemment de la position de la valeur à trouver. Dans le meilleur des cas c'est 1, dans le pire c'est n mais on peut dire, qu'en moyenne, cela entraîne « $n/2$ » comparaisons (que ce soit pour la trouver ou se rendre compte de son absence).

Pour la recherche dichotomique

Ici, la zone de recherche est divisée par 2, à chaque étape. Imaginons une liste de 64 éléments : après 6 étapes, on arrive à un élément isolé. Pour une liste de taille n , on peut en déduire que le nombre de comparaisons est au maximum l'exposant qu'il faut donner à 2 pour obtenir n , soit « $\log_2(n)$ ».

Comparaisons

Voici un tableau comparatif du nombre de comparaisons en fonction de la taille n

n	10	100	1000	10.000	100.000	1 million
recherche linéaire	5	50	500	5.000	50.000	500.000
recherche dichotomique	4	7	10	14	17	20

Nous pouvons voir que c'est surtout pour des grands problèmes que la recherche dichotomique montre toute son efficacité. Et nous voyons aussi que l'important pour mesurer la complexité est l'ordre de grandeur du nombre de comparaisons.

Nous dirons que la recherche simple est un algorithme de complexité linéaire (c'est-à-dire que le nombre d'opérations est de l'ordre de n ou proportionnel à n ou encore que doubler la taille du problème fait doubler aussi le temps de calcul). Ceci se note en langage plus mathématique : $O(n)$ (prononcé « grand O de n »).

Pour la recherche dichotomique, la complexité est logarithmique. Elle se note $O(\log_2(n))$ ce qui veut dire que doubler la taille du problème augmente d'une seule itération la boucle. C'est un fameux gain.

Comparons les complexités les plus courantes.

n	10	100	1000	10^4	10^5	10^6	10^9
$O(1)$	1	1	1	1	1	1	1
$O(\log_2(n))$	4	7	10	14	17	20	30
$O(n)$	10	100	1000	10^4	10^5	10^6	10^9
$O(n^2)$	100	10^4	10^6	10^8	10^{10}	10^{12}	10^{18}
$O(n^3)$	1000	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{27}
$O(2^n)$	1024	10^{30}	10^{301}	10^{3010}	10^{30102}	10^{301029}	$10^{301029995}$

Nous pouvons en conclure que si trouver un algorithme qui résout un problème est sans doute bien, trouver un algorithme qui a une complexité exponentielle ne sert à rien en pratique.

Par exemple un algorithme de recherche de complexité exponentielle, exécuté sur une machine pouvant effectuer un milliard de comparaisons par secondes, prendrait

plus de dix mille milliards d'années pour trouver une valeur dans un tableau de 100 éléments.

Chapitre 11

Algorithmes de tris

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

Contenu

11.1	Motivation	143
11.2	Tri par insertion	145
11.3	Tri par sélection des minima successifs	147
11.4	Tri bulle	149
11.5	Cas particuliers	150
11.6	Références	151

11.1 Motivation

Dans le chapitre précédent, nous avons vu que la recherche d'information est beaucoup plus rapide dans un tableau trié grâce à l'algorithme de recherche dichotomique. Nous avons vu comment **garder** un ensemble trié en insérant toute nouvelle valeur au *bon* endroit.

Dans certaines situations, nous serons amené à devoir trier toute une collection de valeurs. Par exemple un tableau.

1. D'abord les situations impliquant le classement total d'un ensemble de données « brutes », c'est-à-dire complètement désordonnées.

Prenons pour exemple les feuilles récoltées en vrac à l'issue d'un examen ; il y a peu de chances que celles-ci soient remises à l'examineur de manière ordonnée ; celui-ci devra donc procéder au tri de l'ensemble des copies, par exemple par ordre alphabétique des noms des étudiants ou des étudiantes, ou par numéro de groupe, etc.

2. Enfin, les situations qui consistent à devoir retrier des données préalablement ordonnées sur un autre critère.

Prenons l'exemple d'un paquet de copies d'examen déjà triées sur l'ordre alphabétique des noms des étudiants et des étudiantes, et qu'il faut retrier cette fois-ci sur les numéros de groupe. Il est clair qu'une méthode efficace veillera à conserver l'ordre alphabétique déjà présent dans la première situation afin que les copies apparaissent dans cet ordre dans chacun des groupes.

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

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

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

- ▷ le nombre de tests effectués (comparaisons de valeurs) et ;
- ▷ le nombre de transferts de valeurs réalisés.

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



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

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

11.2 Tri par insertion

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

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

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

Prenons comme exemple un tableau `tab` de 20 entiers. Au départ, le sous-tableau trié est formé du premier élément, `tab[0]`, qui vaut 20 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	12	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

L'étape suivante consiste à insérer `tab[1]`, qui vaut 12, dans ce sous-tableau de taille 2 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
12	20	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

Ensuite, c'est au tour de `tab[2]`, qui vaut 18, d'être inséré :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
12	18	20	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

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

Algorithme. L'algorithme combine la recherche de la position d'insertion et le décalage concomitant de valeurs.

```
// Trie le tableau reçu en paramètre (via un tri par insertion).
algorithm insertionSort(myArray↓↑ : array of n integers)
    integers i, j, value
    for i from 1 to n-1
        value = myArray[i]
        // recherche de l'endroit où insérer value dans le
        // sous-tableau trié et décalage simultané des éléments
        j = i - 1
        while j ≥ 0 ET value < myArray[j]
            myArray[j+1] = myArray[j]
            j = j - 1
        myArray[j+1] = value
```

pseudocode

L'algorithme se traduit en Java sans aucune difficulté.

```
/**
 * Tri par insertion.
3  *
 * @param myArray tableau à trier
 */
6 public static insertionSort(int[] myArray){
    int j, value;
    for (int i = 1; i < myArray.length; i = i + 1){
9        value = myArray[i];
        j = i - 1;
        while (j >= 0 && value < myArray[j]){
12            myArray[j + 1] = myArray[j];
            j = j - 1;
        }
15        myArray[j + 1] = value;
    }
}
18
```

java

11.3 Tri par sélection des minima successifs

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

Description de l'algorithme. Prenons par exemple un tableau de 20 entiers.

La première étape consiste à rechercher la valeur minimale du tableau. Il s'agit de l'élément d'indice 9 et de valeur 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	52	61	47	82	64	95	66	84	17	32	24	46	48	75	55	19	61	21	30

Celui-ci devrait donc apparaître en 1^{re} position du tableau. Or cette position est occupée par la valeur 20. Comment procéder dès lors pour ne perdre aucune valeur et sans faire appel à un second tableau ? La solution est simple, il suffit d'échanger le contenu des deux éléments d'indices 0 et 9 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
17	52	61	47	82	64	95	66	84	20	32	24	46	48	75	55	19	61	21	30

Le tableau se subdivise à présent en deux sous-tableaux, un sous-tableau déjà trié (pour le moment réduit au seul élément $tab[0]$) et le sous-tableau des autres valeurs non encore triées (de l'indice 1 à 19). On recommence ce processus dans ce second sous-tableau : le minimum est à présent l'élément d'indice 16 et de valeur 19. Celui-ci viendra donc en 2^e position, échangeant sa place avec la valeur 52 qui s'y trouvait :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
17	19	61	47	82	64	95	66	84	20	32	24	46	48	75	55	52	61	21	30

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

Et ainsi de suite, jusqu'à obtenir un sous-tableau trié comprenant tout le tableau. En pratique l'arrêt se fait une étape avant car lorsque le sous-tableau non trié n'est plus que de taille 1, il contient nécessairement le maximum de l'ensemble.

Algorithme.

```

// Trie le tableau reçu en paramètre (via un tri par sélection des minima successifs).
algorithm selectionSort(myArray↓↑ : array of integers)
    integer i, minIndex
    for i from 0 to myArray.length - 2      // i correspond à l'étape      // de
l'algorithme
        minIndex = searchMinIndex( myArray, i, myArray.length - 1 )
        swap( myArray[i], myArray[minIndex] )

// Retourne l'indice du minimum entre les indices début et fin du tableau reçu.
algorithm searchMinIndex( myArray↓ : array of integers,
                           début↓, fin↓ : integers) → integer

    integers minIndex, i
    minIndex = début
    for i from début+1 to fin
        if myArray[i] < myArray[minIndex] then
            minIndex = i

    return minIndex

// Échange le contenu de 2 variables.
algorithm swap(a↓↑, b↓↑ : integers)
    integer aux
    aux = a
    a = b
    b = aux

```

pseudocode

Les deux premiers algorithmes se traduisent sans difficulté en Java. Le troisième ne peut pas se traduire en l'état. Il faudra soit échanger les valeurs dans la méthode `selectionSort`, soit écrire une méthode `swap` recevant en paramètre, un tableau et deux indices.

11.4 Tri bulle

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

Description de l'algorithme Prenons pour exemple un tableau de taille 14. En partant de la fin du tableau, on le parcourt vers la gauche en comparant chaque couple de valeurs consécutives. Quand deux valeurs sont dans le désordre, on les permute. Le premier parcours s'achève lorsqu'on arrive à l'élément d'indice 0 qui contient alors la « bulle », c'est-à-dire la plus petite valeur du tableau, soit 1 dans l'exemple :

0	1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	12	15	4	8	1	7	12	11	3	6	5	4
10	5	12	15	4	8	1	7	12	11	3	6	4	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	3	11	4	6	5
10	5	12	15	4	8	1	7	3	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	1	8	3	7	12	11	4	6	5
10	5	12	15	1	4	8	3	7	12	11	4	6	5
10	5	12	1	15	4	8	3	7	12	11	4	6	5
10	5	1	12	15	4	8	3	7	12	11	4	6	5
10	1	5	12	15	4	8	3	7	12	11	4	6	5
1	10	5	12	15	4	8	3	7	12	11	4	6	5

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

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

Et ainsi de suite. Au total, il y aura $n - 1$ étapes.

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

```
// Trie le tableau reçu en paramètre (via un tri bulle).
algorithm bubbleSort(myArray $\downarrow\uparrow$  : array of integers)
    integers bubbleIndex, i
    for bubbleIndex from 0 to myArray.length - 2
        for i from myArray.length - 2 to bubbleIndex by -1
            if myArray[i] > myArray[i + 1] then
                swap( myArray[i], myArray[i + 1] )    // voir algorithme précédent
```

pseudocode

Hormis la méthode `swap`, la traduction en langage Java est triviale.

11.5 Cas particuliers

Tri partiel

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

Sélection des meilleurs

Une autre situation courante est la suivante : un ensemble de valeurs sont traitées (lues, calculées...) et il ne faut garder que les k plus petites (ou plus grandes).

L'algorithme est plus simple à écrire si on utilise une position supplémentaire en fin de tableau. Notons aussi qu'il faut tenir compte du cas où il y a moins de valeurs que le nombre de valeurs voulues ; c'est pourquoi on ajoute une variable indiquant le nombre exact de valeurs dans le tableau.

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

```
algorithm bests(smallers $\downarrow\uparrow$  : array of k+1 integers, )  $\rightarrow$  nbValues  
    integer val  
    nbValues = 0  
    read val  
    while val  $\geq$  0  
        insert( val, smallers, nbValues )  
    read val
```

pseudocode

```
algorithm insert(val $\downarrow$  : integer, smallers $\downarrow\uparrow$  : array of k+1 integers,  
                nbValues $\downarrow\uparrow$  : integer)  
    integer i  
    i = nbValues - 1  
    while i  $\geq$  0 AND val < smallers[i]  
        smallers[i + 1] = smallers[i]  
        i = i - 1  
  
    smallers[i + 1] = val  
    if nbValues < k then  
        nbValues = nbValues + 1
```

pseudocode

11.6 Références

▷ http://interstices.info/jcms/c_6973/les-algorithmes-de-tri

Ce site permet de visualiser les méthodes de tris en fonctionnement. Il présente tous les tris vus en première plus quelques autres comme le « tri rapide » (« quicksort ») que vous verrez en deuxième année.

Quatrième partie

Compléments

Chapitre 12

Les chaînes et les `String`

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

Très tôt dans ce cours, nous avons introduit le type `chaîne` mais nous n'en avons encore fait que des usages basiques, essentiellement des affichages. Il est temps d'aller plus loin et de les *manipuler*, c'est-à-dire d'examiner et de modifier, déplacer... le contenu de chaînes. Procédons par étapes.

Contenu

12.1	Longueur	156
12.2	Chaîne et caractère	156
12.3	Le contenu d'une chaîne	157
12.4	La concaténation	158
12.5	Manipuler les caractères	159
12.6	L'alphabet	161
12.7	Chaîne et nombre	162
12.8	Extraction de sous-chaînes	162
12.9	Recherche de sous-chaîne	163

12.1 Longueur



La **longueur** d'une chaîne est le nombre de caractères qu'elle contient. Pour la connaître on utilisera la notation pointée `unechaine.length`.

Par exemple :

- ▷ `"Bonjour".length` vaut 7.
- ▷ `"Une chaîne".length` vaut 10 (l'espace compte pour 1).
- ▷ `"A".length` vaut 1.
- ▷ `"".length` est égal à 0.

En langage Java également, une chaîne connaît sa taille :

```
"Bonjour".length() // vaut 7
```

java

Remarque

- ▷ Notez la présence des parenthèses en Java.
- ▷ Nous n'écrirons pas de parenthèse dans nos algorithmes mais ce n'est pas une faute d'en mettre.

12.2 Chaîne et caractère

Certains langages, comme Java, distinguent clairement le type chaîne (**String**) et le type caractère (**char**). Ainsi, en Java, le littéral `'A'` représente un caractère qui est différent du littéral `"A"` qui représente une chaîne (composée d'un seul caractère). Le type **String** est un type référence tandis que le type **char** est un type primitif.

Cette distinction est essentiellement dictée par des détails d'implémentation ; un caractère pouvant se représenter de façon plus économe qu'une chaîne. En langage Java toujours essayer de comparer — par un `"A" == 'A'` — une chaîne et un caractère génère une erreur de compilation.

D'autres langages, comme Python, n'ont pas de type caractère spécifique. D'autres encore, comme C, ont un type caractère mais pas de type chaîne à proprement parler ; ils sont vus comme des tableaux de caractères.

Dans nos algorithmes — mais pas dans nos programmes Java — nous introduirons un type **character** (caractère) mais sans le distinguer clairement d'une chaîne. Ainsi, un caractère pourra être utilisé comme une chaîne et une chaîne d'un seul caractère pourra être considéré comme un caractère. En somme, **character** n'est qu'un raccourci pour dire : *une chaîne de longueur 1*.

12.3 Le contenu d'une chaîne

Pour pouvoir manipuler une chaîne, il faut pouvoir accéder aux caractères qui la composent. Comme il s'agit d'une opération de base souvent utilisée, nous allons introduire une écriture compacte, empruntée aux tableaux : `nomChaine[i]` désigne le i^{e} caractère de la chaîne `nomChaine` (en commençant à 1).

Par exemple :

```
string text = "Hello"
print text[5]           // Affiche "o"
text[2] = "a"           // texte vaut "Hallo" ;
                        // un caractère doit être remplacé par un seul caractère.
```

pseudocode

Exemple. Écrivons un algorithme qui vérifie si un mot donné contient une lettre donnée.

```
algorithm contains(word : string, letter : character) → boolean
    entier i
    i = 1
    while i ≤ word.length AND word[i] ≠ letter
    |   i = i + 1
    return i ≤ word.length
```

pseudocode

En langage Java, l'API propose une méthode — `char charAt(int)` — permettant d'accéder à une lettre d'un mot et la retournant. Ce qui se traduit en langage Java

```
1 public static boolean contains(String word, char letter){
    int i = 1;
    while (i <= word.length() && word.charAt(i) != letter){
4       i = i + 1;
    }
    return i <= word.length();
7 }
```

java

12.4 La concaténation

Il est fréquent de devoir rassembler plusieurs chaînes pour former une seule chaîne plus grande, il s'agit de l'opération de **concaténation**. Introduisons également une notation compacte, le `+`.

Par exemple :

```
string texte
texte = "al" + "go" + "rithmique"    // assigne la chaîne "algorithmique"
                                     // à la variable texte.
```

pseudocode

Exemple. Écrivons un algorithme qui inverse toutes les lettres d'un mot. Ainsi, "algo" deviendra "ogla".

```
algorithm mirror(↓word : string) → string
    string mirror
    mirror = ""
    for i from 1 to word.length
        | mirror = word[i] + mirror
    return mirror
```

pseudocode

Cet algorithme se traduit immédiatement en langage Java par :

```
1 public static String mirror(String word){
    String mirror = "";
    for (int i = 0; i < word.length(); i = i + 1){
4         mirror = word[i] + mirror;
    }
    return mirror;
7 }
```

java

L'instruction de la ligne 4 concatène un caractère (`word[i]`) avec une chaîne (`mirror`) ce qui n'est, à priori, pas possible. Dans ce cas, Java convertit le caractère en une chaîne avant d'effectuer la concaténation de manière tout à fait transparente.

Dans cette instruction à la ligne 4 toujours, il y a création d'une nouvelle chaîne de caractère à chaque itération de la boucle. Si l'on veut gagner un peu de place en mémoire, il est possible de créer une chaîne de la bonne taille dès le début et d'y placer les caractères dans l'ordre qui nous convient au fur et à mesure. Ce serait mieux n'est-ce pas ?

En langage Java, ce n'est pas immédiat de remplacer un caractère par un autre dans une chaîne. Ceci parce que `character` et `String` sont deux types différents. De plus,

l'un est de type primitif comme nous l'avons déjà dit et l'autre de type référence.

Java — comme tous les autres langages d'ailleurs — propose en plus des instructions élémentaires telles que `if` et `while` par exemple, une API (*application programming interface*). Une API est un ensemble de classes et de méthodes, de manière générale ; de services, qu'offre le langage pour aider le développeur. Nous reviendrons abondamment sur ces concepts dans les autres cours de développement.

Dans le cas qui nous occupe, une manière élégante de résoudre le problème, est l'utilisation de `StringBuilder` qui, comme son nom l'indique, permet de manipuler plus finement une chaîne de caractères. Nous pouvons écrire :

```
1 public static String mirror(String word){
    StringBuilder sb = new StringBuilder(word);
    for (int i = 0; i < word.length(); i = i + 1){
4      sb.setCharAt(word.length() - i - 1, word.charAt(i));
    }
    return sb.toString();
7 }
```

java

- ▷ instruction 2 : création d'un espace de la même longueur que la chaîne `word` ;
- ▷ instruction 4 : placement du caractère à la position d'indice i dans `word` à la position d'indice « taille du mot - i - 1 » dans le `StringBuilder`.

12.5 Manipuler les caractères

Voici quelques algorithmes et méthodes de l'API Java pratiques pour la manipulation de chaînes.

Nous pourrions écrire ces algorithmes et ces méthodes de l'API mais ce serait long et fastidieux. Dans nos exercices, nous supposons que les algorithmes existent et pour chacun d'eux, nous montrerons comment les écrire en langage Java en donnant un exemple simple.

isLetter(c : character) → boolean

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

`boolean Character.isLetter(char c)`

```
1 Character.isLetter('#'); // false
   char c = 'A';
   Character.isLetter(c);  // true
4
```

java

isLowerCase(c : character) → boolean

Permet de savoir si le caractère **est une lettre minuscule**.

`boolean Character.isLowerCase(char c)`

```
Character.isLowerCase('A'); // false
2 char c = 'a';
Character.isLowerCase(c); // true
```

java

isUpperCase(c : character) → boolean

Permet de savoir si le caractère **est une lettre majuscule**.

boolean Character.toUpperCase(char c)

```
Character.isUpperCase('a'); // false
2 char c = 'A';
Character.isUpperCase(c); // true
```

java

isDigit(c : character) → boolean

Permet de savoir si un caractère **est un chiffre**. Elle retourne vrai pour les dix caractères '0', '1', '2', '3', '4', '5', '6', '7', '8' et '9' (et aussi pour les chiffres dans d'autres langues, cfr. javadoc⁵⁸).

```
Character.isDigit('a'); // false
2 char c = '1';
Character.isDigit(c); // true
```

java

toUpperCase(s : string) → string

Retourne une chaîne où toutes les lettres du texte ont été converties en majuscules.

En langage Java, c'est la chaîne qui est convertie.

String.toUpperCase()

```
String s = "hello";
2 s.toUpperCase();
System.out.println(s); // affiche HELLO
```

java

toLowerCase(s : string) → string

Retourne une chaîne où toutes les lettres du texte ont été converties en minuscules.

En langage Java, c'est la chaîne qui est convertie.

String.toLowerCase()

```
String s = "WORLD";
2 s.toLowerCase();
System.out.println(s); // affiche world
```

java

58. <https://docs.oracle.com/javase/9/docs/api/java/lang/Character.html#isDigit-char->

12.6 L'alphabet

Il peut aussi être pratique de connaître la position d'une lettre dans l'alphabet pour résoudre des exercices. Ceci se fera à l'aide de la fonction :

letterIndex(c : character) → integer

Retourne toujours un entier entre 1 et 26.

Par exemple `letterIndex("E")` donnera 5, ainsi que `letterIndex("e")`.

Cette fonction traite donc de la même manière les majuscules et les minuscules. `letterIndex` retournera aussi 5 pour les caractères "é", "è", "ê", "ë"...).

En langage Java, il est assez simple de connaître la position d'une lettre dans l'alphabet dès lors que l'on sait qu'à chaque lettre est associée un code (Unicode) et que ces codes se suivent. La table unicode contenant les lettres de l'alphabet⁵⁹ nous montre que le code associé à la lettre 'A' est 0x0041 ou 65 en base 10, 'B' 0x0042 et ainsi de suite.

Cette table nous montre également que les lettres minuscules arrivent ensuite ; 'a' a comme code 0x0061 ou 97 en base 10, 'b' 0x0062...

Retourner la position d'une lettre devient aussi simple qu'une soustraction. En supposant qu'il ne faille pas vérifier que le caractère reçu est bien une lettre, nous pourrions écrire :

```
public static int letterIndex(char c){
2   return Character.toLowerCase(c) - 0x61 + 1;
}
```

java

Pour traiter les caractères accentués, c'est plus complexe et ça sort un peu du cadre d'un premier cours de développement. Bien sûr, nous pouvons nous en sortir avec une série de if de la forme `if(c == 'é' || c == 'è'...)` mais ce serait fastidieux.

L'API Java peut faire ce travail pour nous si c'est demandé gentiment. Il existe une classe de l'API pour « normaliser » les caractères d'une chaîne. Cette normalisation consiste à vérifier que les caractères accentués le sont sous la forme « lettre-accent ». C'est-à-dire qu'un à est bien codé a'. Dans ce cas, `replaceAll` peut supprimer les accents et la méthode devient⁶⁰ :

```
public static int letterIndex(char c){
2   c = Normalizer
      .normalize(""+c, Normalizer.Form.NFD)
      .replaceAll("[\u0300-\u036F]", "")
5   .charAt(0);
      return Character.toLowerCase(c) - 0x61 + 1;
}
8
```

java

... mais ça sort un peu du cadre de ce premier cours de développement.

59. <https://www.unicode.org/charts/PDF/U0000.pdf>

60. Un `import java.text.Normalizer` est nécessaire.

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

indexToUpperChar(*n* : integer) → character

Retourne la forme majuscule de la *n*^e lettre de l'alphabet (où *n* sera obligatoirement compris entre 1 et 26). Par exemple, `lettreMaj(13)` retourne "M".

En langage Java, c'est immédiat par `return 0x41 + n`.

indexToLowerChar(*n* : integer) → character

Idem pour les minuscules.

En langage Java, c'est immédiat par `return 0x61 + n`.

12.7 Chaîne et nombre

Si une chaîne contient un nombre, il doit être possible de le convertir, et inversement.

toString(*n* : real) → string

Transforme un nombre en chaîne. Ex : `chaîne(42)` retourne la chaîne "42" et `chaîne(3,14)` donnera "3,14".

En langage Java, cette conversion est immédiate et se fait de manière transparente.

toReal(*s* : string) → real

Transforme une chaîne contenant des caractères numériques en nombre. Ainsi, `nombre("3,14")` retournera 3,14. C'est une erreur de l'utiliser avec une chaîne qui ne représente pas un nombre.

En langage Java, pour chaque type, il existe une méthode qui fait cette transformation. Pour le type `double`, il suffit d'écrire :

```
1 String s = "3.14";
   double d = Double.parseDouble(s);
```

java

12.8 Extraction de sous-chaînes

substring(*s* : string, beginIndex : integer, endIndex : integer) → string

Permet d'extraire une sous-chaîne d'une chaîne à partir d'une position donnée jusqu'à une autre. `beginIndex` est la position de départ et `endIndex - 1` la position de fin... ce qui implique que `endIndex - beginIndex` est la longueur de la sous chaîne.

En langage Java, il existe également une méthode `substring` :

```
String s = "Karine, Jenny et Vicky paraissent belles";
System.out.println(s.substring(8,40));
3 // Jenny et Vicky paraissent belles
```

java

Il faut bien sûr être vigilant pour ne pas sortir des bornes de la chaîne sinon, une erreur est générée.

Cette fonction est très utile pour sélectionner des portions d'une chaîne contenant des informations codées sous un certain format. Prenons par exemple une date stockée dans une chaîne *stringDate* de format "JJ/MM/AAAA" (de longueur 10). Pour extraire :

- ▷ le jour, `substring(stringDate, 0, 2)` ;
- ▷ le mois, `substring(stringDate, 3, 5)` ;
- ▷ le jour, `substring(stringDate, 6, 10)` ;

12.9 Recherche de sous-chaîne

indexOf(s : string, subs : string) → integer

Permet de savoir si une sous-chaîne donnée est présente dans une chaîne donnée. Elle permet d'éviter d'écrire le code correspondant à une recherche. La valeur de l'entier renvoyé est la position où commence la sous-chaîne recherchée. Si la sous-chaîne ne s'y trouve pas, la fonction retourne -1.

En langage Java, il existe également une méthode `indexOf` :

```
String s = "Des_3_nombrils,"  
2  + "c'est_Karine_la_plus_intelligente.";   
s.indexOf("Karine"); // 22
```

java

Remarque Pour simplement savoir si la sous-chaîne est présente sans demander sa position, la fonction `contains` retourne un booléen et prend les mêmes paramètres.

Chapitre 13

Les variables structurées et les classes

Dans la résolution de problème, l'écriture d'algorithmes et de programmes nous ne pourrions pas nous contenter des types de données simples. Nous allons devoir manipuler des données plus complexes. C'est là qu'interviennent les structures en algorithmique et les classes en langage orienté objet.

Comme un tableau, une structure permet de stocker plusieurs valeurs. Mais, à l'inverse des tableaux, chaque valeur peut être de type différent et est désignée par un nom et pas par un numéro.

Cela permet de créer un nouveau type permettant de représenter des données qui ne peuvent s'inscrire dans les types simples déjà vus. Par exemple :

- ▷ Une **date** est composée de trois éléments (le jour, le mois, l'année). Le mois peut s'exprimer par un entier (15/10/2015) ou par une chaîne (15 octobre 2015).
- ▷ Un **moment** de la journée est un triple d'entiers (heures, minutes, secondes).
- ▷ La localisation d'un **point** dans un plan nécessite la connaissance de deux coordonnées cartésiennes (l'abscisse x et l'ordonnée y) ou polaires (le rayon r et l'angle θ).
- ▷ Une **adresse** est composée de plusieurs données : un nom de rue, un numéro de maison, parfois un numéro de boîte postale, un code postal, le nom de la localité, un nom ou code de pays pour une adresse à l'étranger...

Contenu

13.1	Structure et classe	166
13.1.1	Définition d'une structure	166
13.1.2	Définition d'une classe	167
13.2	Déclaration d'une variable	167
13.3	Utilisation des variables de type structuré	168
13.4	Remarque	171

13.1 Structure et classe

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

Notez qu'un attribut d'une structure peut lui-même être une structure. Par exemple, une **carte d'identité** inclut parmi ses informations une **date** de naissance, l'**adresse** de son propriétaire. . .

Les langages orientés objet permettent de déclarer leurs propres types. Cette déclaration se fera au sein d'une **classe**.

Comme nous l'avons déjà vu lorsque nous écrivions notre premier programme (cfr. section 5.2.4 p 62) chaque programme Java se trouve dans une classe. Nous allons faire cohabiter des fichiers Java contenant une méthode principale (**main**) et des méthodes statiques et d'autres fichiers Java définissant une classe qui représente un objet, une structure. Cette section ne fera qu'effleurer la notion d'objet que nous détaillerons dans le cours DEV2 « Développement II ».

13.1.1 Définition d'une structure



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

```
structure StructureName
    type1 fieldName1
    type2 fieldName2
    ...
    typeN fieldNameN
```

pseudocode

fieldName1, ..., fieldNameN sont les noms des différents attributs (ou champs) de la structure, et type1, ..., typeN les types de ces attributs. Tous les types peuvent être envisagés : les types « simples » (integer, real, boolean, string), les tableaux et même d'autres types structurés dont la structure aura été préalablement définie.

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

```
structure Date
    integer day
    integer month
    integer year
```

pseudocode

```
structure Moment
    integer hour
    integer minute
    integer second
```

pseudocode

```
structure Point
```

```
  real x
  real y
```

pseudocode

```
structure Student
```

```
  string name
  integer matricule
  string section
```

pseudocode

13.1.2 Définition d'une classe

Sans surprise, le mot clé utilisé sera `class` et une classe se définit comme suit (dans une version très simplifiée de la grammaire) :

ClassDeclaration:

```
[public] class Identifier
      ClassBody
```

ClassBody:

```
FieldDeclaration
...

```

FieldDeclaration est une série de déclarations habituelles de variables de la forme `Type Identifier ;`.

Définir une classe `Date` équivalente à la structure ci-dessus, aura l'allure suivante :

```
public class Date{
2   int day;
    int month;
    int year;
5 }
```

java

13.2 Déclaration d'une variable

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

```
Date birthday, dDay
Moment departure, arrival, aBeautifulMoment
Point a, b, gravity
```

pseudocode

```

Date birthday;
Date Dday;
3 Moment departure, arrival, aBeautifullMoment;
Point a, b;
Point gravity;
6

```

java

13.3 Utilisation des variables de type structuré

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

```
variableName.fieldName
```

pseudocode

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

```

birthday.day = 15
birthday.month = 10
birthday.year = 2014
arrival.hour = departure.hour + 2
gravity.x = (a.x + b.x) / 2

```

pseudocode

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

```
arrival = departure
```

pseudocode

qui résume les trois instructions suivantes :

```

arrival.heure = departure.heure
arrival.minute = departure.minute
arrival.seconde = departure.seconde

```

pseudocode

Par facilité d'écriture, l'assignation peut se faire en une seule fois via des « {} ».

Exemple :


```
birthday = {19, 5, 1955}
```

pseudocode

Une variable structurée peut aussi être le paramètre ou la valeur de retour d'un algorithme. Par exemple, l'entête d'un algorithme renvoyant le nombre de secondes écoulées entre une heure de départ et d'arrivée serait :

```
algorithm secondsElapsed(departure↓, arrival↓ : Moment) → integer
```

pseudocode

On pourra aussi lire ou afficher une variable structurée⁶¹.

```
read aMoment
print aMoment
```

pseudocode

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

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

Et en langage Java ?

Pour une classe définie comme dans la section précédente, Java permet de déclarer une variable de ce type et d'accéder aux attributs avec une notation pointée.

```
Date birthday = new Date();
birthday.day = 15;
3 // ...
```

java

Notez la présence de **new Date()**. Instruction obligatoire, le **constructeur** réserve un emplacement mémoire destiné à recevoir les valeurs de chaque attribut. En langage orienté objet, nous disons que le constructeur, crée — ou instancie — l'objet.

Nous verrons en DEV2 « Développement II » que de donner l'accès public — c'est-à-dire à tous — aux attributs via `birthday.day` par exemple) n'est pas la bonne pratique. Il vaut mieux mettre en œuvre **l'encapsulation** qui consiste à rendre les attributs privés (inaccessibles en dehors de la classe) et de fournir des méthodes — appelées accesseurs — donnant ces accès.

Java permet d'utiliser une variable de manière globale puisque c'est une variable de type référence. Nous pourrions bien sûr écrire :

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

```
arrival = departure;
```

2

java

Notez bien que cette assignation est une copie de référence et que les deux variables références le même objet. Modifier l'un, modifie l'autre. En ce sens, c'est différent de ce que nous faisons dans nos algorithmes.

L'assignation globale telle que présentée précédemment dans les algorithmes ne pourra se faire qu'en écrivant explicitement un constructeur⁶² recevant les valeurs utiles en paramètres. Un **constructeur** :

- ▷ a le même nom que la classe ;
- ▷ n'a pas de type de retour ;
- ▷ construit l'objet ; réserve l'emplacement mémoire nécessaire ;
- ▷ est appelé grâce au mot clé **new**

La classe `Date`, précédemment écrite, pourrait s'écrire :

```
1 public class Date{
    int day;
    int month;
4   int year;

    public Date(){
7       // permet d'instancier une date sans valeurs de départ
    }

10  public Date(int aDay, int aMonth, int aYear){
        day = aDay;
        month = aMonth;
13     year = aYear;
    }
}
16
```

java

... et l'assignation globale :

```
Date birthday = new Date{19, 5, 1955};
```

2

java

Une classe définie par l'utilisateur — comme la classe `Date` — peut être un paramètre d'une méthode ou un type de retour.

Pour la lecture, il faudra lire chaque attribut séparément et créer l'objet — grâce au constructeur — ensuite.

Pour afficher la valeur d'un objet, il est nécessaire :

⁶². Spoiler. Votre IDE peut vous aider dans cette tâche.

là où l'on veut l'affichage

de faire une simple demande d'affichage d'une variable

```
1 System.out.println(birthday);
```

java

dans la classe

d'ajouter une méthode `toString` (et il faut impérativement respecter le nom)⁶³

```
// ...
2 public String toString(){
    return day + "/" + month + "/" + year;
}
5
```

java

Nous avons montré ici, ce qu'il faut faire en langage Java pour pouvoir utiliser l'équivalent des structures en algorithmique. La notion de classe dans les langages orientés objets est beaucoup plus large et s'accompagne de concepts essentiels de programmation tels que l'encapsulation, le polymorphisme et l'héritage. Ces concepts seront abordés en DEV2 - « Développement II ».

13.4 Remarque

Tous les exemples ci-dessus présentent des champs simples (entier ou chaîne) mais il n'y a en fait aucune contrainte : un champ peut-être un tableau ou même une autre structure.

Par exemple, la structure suivante définit une personne qui possède, en plus de son nom, une date de naissance (la structure `Date`) et 3 numéros de téléphone (un tableau de 3 chaînes).

```
structure Person
    string name
    Date birthday
    array of 3 string phoneNumbers
```

pseudocode

Et voici un exemple d'utilisation.

```
Person me
me.birthday.day = 23
me.phoneNumbers[0] = "0444/42.42.42"
```

pseudocode

En langage Java, cet exemple peut être immédiatement traduit par :

⁶³. Comme pour le constructeur, votre IDE peut vous aider.

```
1 public class Person{
    String name;
    Date birthday;
4   String[] phoneNumbers;

    public Person(){
7       birthday = new Date();
        phoneNumbers = new String(3);
    }
10 }
```

java

Notez bien la nécessité de créer le tableau et l'objet date dans le constructeur.

L'exemple d'utilisation devient :

```
1 Person me = new Person();
  me.birthday.day = 23;
  me.phoneNumbers[0] = "0444/42.42.42";
4
```

java

À nouveau, nous verrons les bonnes pratiques dans le prochain cours de développement.

Cinquième partie

Les annexes

Annexe A

Exercices

Contenu

A.1	Exercices : spécifier le problème	175
A.2	Exercices : premiers algorithmes et programmes	178
A.3	Exercices : tracer un algorithme	178
A.4	Exercices : division entière, quotient et reste	180
A.5	Exercices : structures alternatives (choix)	183
A.6	Exercices : modules et méthodes	187
A.7	Exercices : structures itératives	190
A.7.1	while	190
A.7.2	for	191
A.7.3	do while	192
A.7.4	Exercices récapitulatifs sur les structures répétitives	192
A.8	Exercices : tableaux	196
A.9	Exercices : chaîne et String	201
A.10	Exercices : structures et classes	203
A.11	Exercices récapitulatifs	204
A.11.1	AEBBCLRS	204
A.11.2	Serpents et échelles	206
A.11.3	Les algorithmes en maternelle	208

A.1 Exercices : spécifier le problème

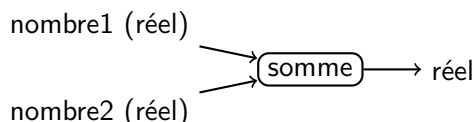
Pour ces premiers exercices, nous vous demandons d'imiter la démarche décrite dans le chapitre 1, à savoir :

- ▷ déterminer quelles sont les données ; leur donner un nom et un type ;
- ▷ déterminer quel est le type du résultat ;
- ▷ déterminer un nom pertinent pour l'algorithme ;
- ▷ fournir un résumé graphique ;
- ▷ donner des exemples.

1 Somme de 2 nombres

Calculer la somme de deux nombres donnés.

Solution.⁶⁴ Il y a ici clairement 2 données. Comme elles n'ont pas de rôle précis, on peut les appeler simplement **nombre1** et **nombre2** (**nb1** et **nb2** sont aussi de bons choix). L'énoncé ne dit pas si les nombres sont entiers ou pas ; restons le plus général possible en prenant des réels. Le résultat sera de même type que les données. Le nom de l'algorithme pourrait être simplement **somme**. Ce qui donne :



Et voici quelques exemples numériques : **somme(3, 2)** donne 5 **somme(-3, 2)** donne -1 **somme(3, 2.5)** donne 5.5 **somme(-2.5, 2.5)** donne 0.

2 Moyenne de 2 nombres

Calculer la moyenne de deux nombres donnés.

3 Surface d'un triangle

Calculer la surface d'un triangle connaissant sa base et sa hauteur.

4 Périmètre d'un cercle

Calculer le périmètre d'un cercle dont on donne le rayon.

5 Surface d'un cercle

Calculer la surface d'un cercle dont on donne le rayon.

6 TVA

Si on donne un prix hors TVA, il faut lui ajouter 21% pour obtenir le prix TTC. Écrire un algorithme qui permet de passer du prix HTVA au prix TTC.

7 Les intérêts

Calculer les intérêts reçus après 1 an pour un montant placé en banque à du 2% d'intérêt.

8 Placement

Étant donné le montant d'un capital placé (en €) et le taux d'intérêt annuel (en %), calculer la nouvelle valeur de ce capital après un an.

⁶⁴. Nous allons de temps en temps fournir des solutions. En algorithmique, il y a souvent **plusieurs** solutions possibles. Ce n'est donc pas parce que vous avez trouvé une autre façon de faire qu'elle est fausse. Mais il peut y avoir des solutions **meilleures** que d'autres ; n'hésitez jamais à montrer la vôtre à votre professeur pour avoir son avis.

9 Prix TTC

Étant donné le prix unitaire d'un produit (hors TVA), le taux de TVA (en %) et la quantité de produit vendue à un client, calculer le prix total à payer par ce client.

10 Durée de trajet

Étant donné la vitesse moyenne en **m/s** d'un véhicule et la distance parcourue en **km** par ce véhicule, calculer la durée en secondes du trajet de ce véhicule.

11 Allure et vitesse

L'allure d'un coureur est le temps qu'il met pour parcourir 1 km (par exemple, 4'37"). On voudrait calculer sa vitesse (en km/h) à partir de son allure. Par exemple, la vitesse d'un coureur ayant une allure de 4'37" est de 13 km/h.

12 Somme des chiffres

Calculer la somme des chiffres d'un nombre entier de 3 chiffres.

13 Conversion HMS en secondes

Étant donné un moment dans la journée donné par trois nombres, à savoir, heure, minute et seconde, calculer le nombre de secondes écoulées depuis minuit.

14 Conversion secondes en heures

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "heure".

Ex : 10000 secondes donnera 2 heures.

15 Conversion secondes en minutes

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "minute".

Ex : 10000 secondes donnera 46 minutes.

16 Conversion secondes en secondes

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "seconde".

Ex : 10000 secondes donnera 40 secondes.

17 Cote moyenne

Étant donné les résultats (cote entière sur 20) de trois examens passés par un étudiant (exprimés par six nombres, à savoir, la cote et la pondération de chaque examen), calculer la moyenne globale exprimée en pourcentage.

A.2 Exercices : premiers algorithmes et programmes

Dans ces exercices, nous vous proposons d'écrire un algorithme et de le traduire en un programme Java.

18 Moyenne de 2 nombres

Calculer la moyenne de deux nombres donnés.

19 Surface d'un triangle

Calculer la surface d'un triangle connaissant sa base et sa hauteur.

20 Périmètre d'un cercle

Calculer le périmètre d'un cercle dont on donne le rayon.

21 Surface d'un cercle

Calculer la surface d'un cercle dont on donne le rayon.

22 TVA

Si on donne un prix hors TVA, il faut lui ajouter 21% pour obtenir le prix TTC. Écrire un algorithme qui permet de passer du prix HTVA au prix TTC.

23 Les intérêts

Calculer les intérêts reçus après 1 an pour un montant placé en banque à du 2% d'intérêt.

24 Placement

Étant donné le montant d'un capital placé (en €) et le taux d'intérêt annuel (en %), calculer la nouvelle valeur de ce capital après un an.

25 Conversion HMS en secondes

Étant donné un moment dans la journée donné par trois nombres, à savoir, heure, minute et seconde, calculer le nombre de secondes écoulées depuis minuit.

26 Prix TTC

Étant donné le prix unitaire d'un produit (hors TVA), le taux de TVA (en %) et la quantité de produit vendue à un client, calculer le prix total à payer par ce client.

A.3 Exercices : tracer un algorithme

Nous vous proposons de tracer vos algorithmes comme présenté dans le chapitre ?? dans la section 4.2.3, page 38.

27 Tracer des bouts de code

Suivez l'évolution des variables pour les bouts d'algorithmes donnés.

```

1: entiers a, b, c
2: a = 42
3: b = 24
4: c = a + b
5: c = c - 1
6: a = 2 * b
7: c = c + 1

```

pseudocode

#	a	b	c
1			
2			
3			
4			
5			
6			
7			

```

1: entiers a, b, c
2: a = 2
3: b = a3
4: c = b - a2
5: a =  $\sqrt{c}$ 
6: a = a / a

```

pseudocode

#	a	b	c
1			
2			
3			
4			
5			
6			

28 Calcul de vitesse

Soit le problème suivant : « Calculer la vitesse (en km/h) d'un véhicule dont on donne la durée du parcours (en secondes) et la distance parcourue (en mètres). ».

Voici *une* solution :

```

1: algorithm vitesseKMh(distanceM, duréeS : réels) → réel
2:   réels distanceKM, duréeH
3:   distanceKM = 1000 * distanceM
4:   duréeH = 3600 * duréeS
5:   return  $\frac{\text{distanceKM}}{\text{duréeH}}$ 
6:
7:

```

pseudocode

L'algorithme, s'il est correct, devrait donner une vitesse de 1 km/h pour une distance de 1000 mètres et une durée de 3600 secondes. Testez cet algorithme avec cet exemple.

#	
1	
2	
3	
4	
5	

Si vous trouvez qu'il n'est pas correct, voyez ce qu'il faudrait changer pour le corriger.

29 Allure et vitesse

L'allure d'un coureur est le temps qu'il met pour parcourir 1 km (par exemple, 4'37"). On voudrait calculer sa vitesse (en km/h) à partir de son allure. Par exemple, la vitesse d'un coureur ayant une allure de 4'37" est de 12,996389892 km/h.

30 Cote moyenne

Étant donné les résultats (cote entière sur 20) de trois examens passés par un étudiant (exprimés par six nombres, à savoir, la cote et la pondération de chaque examen), calculer la moyenne globale exprimée en pourcentage.

A.4 Exercices : division entière, quotient et reste

31 Calculs

Voici quelques petits calculs à compléter faisant intervenir la division entière et le reste. Par exemple : "14 DIV 3 = 4 reste 2" signifie que $14 \text{ DIV } 3 = 4$ et $14 \text{ MOD } 3 = 2$.

$$\triangleright 11 \text{ DIV } 3 = \text{___} \text{ reste } \text{___}$$

$$\triangleright 11 \text{ DIV } \text{___} = 2 \text{ reste } 3$$

$$\triangleright 3 \text{ DIV } 11 = \text{___} \text{ reste } \text{___}$$

$$\triangleright \text{___} \text{ DIV } 3 = 3 \text{ reste } 1$$

32 Les prix ronds

Voici un algorithme qui reçoit une somme d'argent exprimée en centimes et qui calcule le nombre (entier) de centimes qu'il faudrait ajouter à la somme pour tomber sur un prix rond en euros. Testez-le avec des valeurs numériques. Est-il correct ?

```

algorithm versPrixRond(prixCentimes : entier) → entier
|   return 100 - (prixCentimes MOD 100)

```

pseudocode

test n°	prixCentimes	réponse correcte	valeur retournée	Correct ?
1	130	70		
2	40	60		
3	99	1		
4	100	0		

Remarque Les exercices qui suivent n'ont pas tous été déjà analysés et ils demandent des calculs faisant intervenir des divisions entières, des restes et/ou des expressions booléennes. Comme d'habitude, écrivez la spécification si ça n'a pas encore été fait, donnez des exemples, rédigez un algorithme et vérifiez-le.

33 Nombre multiple de 5

Calculer si un nombre entier donné est un multiple de 5.

Solution. Dans ce problème, il y a une donnée, le nombre à tester. La réponse est un booléen qui est à vrai si le nombre donné est un multiple de 5.

nombre (entier) \longrightarrow multiple5 \longrightarrow booléen

Exemples.

▷ multiple5(4) ▷ multiple5(15) ▷ multiple5(0) ▷ multiple5(-10)
 donne faux donne vrai donne vrai donne vrai

La technique pour vérifier si un nombre est un multiple de 5 est de vérifier que le reste de la division par 5 donne 0. Ce qui donne :

```

1: algorithm multiple5(nombre : entier)  $\rightarrow$  booléen
2:   return nombre MOD 5 == 0
3:
4:

```

pseudocode

Vérifions sur nos exemples :

test n°	nombre	réponse correcte	valeur retournée	Correct ?
1	4	faux	faux	✓
2	15	vrai	vrai	✓
3	0	vrai	vrai	✓
4	-10	vrai	vrai	✓

34 Nombre entier positif se terminant par un 0

Calculer si un nombre donné se termine par un 0.

35 Les centaines

Calculer la partie *centaine* d'un nombre entier positif quelconque.

36 Somme des chiffres

Calculer la somme des chiffres d'un nombre entier positif inférieur à 1000.

37 Conversion secondes en heures

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "heure".

Ex : 10000 secondes donnera 2 heures.

Aide : L'heure n'est qu'un nombre exprimé en base 60 !

38 Conversion secondes en minutes

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "minute".

Ex : 10000 secondes donnera 46 minutes.

39 Conversion secondes en secondes

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "seconde".

Ex : 10000 secondes donnera 40 secondes.

40 Un double aux dés

Écrire un algorithme qui simule le lancer de deux dés et indique s'il y a eu un double (les deux dés montrant une face identique).

41 Année bissextile

Écrire un algorithme qui vérifie si une année est bissextile.

Pour rappel, les années bissextiles sont les années multiples de 4. Font exception, les multiples de 100 (sauf les multiples de 400 qui sont bien bissextiles). Ainsi 2012 et 2400 sont bissextiles mais pas 2010 ni 2100.

42 Conversion en heures-minutes-secondes

Écrire un algorithme qui permet à l'utilisateur de donner le nombre de secondes écoulées depuis minuit et qui affiche le moment de la journée correspopcnt en heures-minutes-secondes. Par exemple, si on est 3726 secondes après minuit alors il est 1h2'6".

A.5 Exercices : structures alternatives (choix)

43 Compréhension

Tracez cet algorithme avec les valeurs fournies et donnez la valeur de retour.

```
algorithm exerciceA(a, b : entiers) → entier
    entier c
    c = 2 * a
    if c > b then
        c = c - b
    return c
```

pseudocode

▷ exerciceA(2, 5) = ____

▷ exerciceA(4, 1) = ____

44 Simplification d'algorithmes

Voici quelques extraits d'algorithmes corrects du point de vue de la syntaxe mais contenant des lourdeurs d'écriture. Simplifiez-les.

```
if ok = vrai then
    print nombre
```

pseudocode

```
if ok = faux then
    print nombre
```

pseudocode

```
if ok = vrai OU ok = faux then
    print nombre
```

pseudocode

```
if ok = vrai ET ok = faux then
    print nombre
```

pseudocode

45 Compréhension

Tracez ces algorithmes avec les valeurs fournies et donnez la valeur de retour.

```

algorithm exerciceB(b, a : entiers) → entier
    entier c
    if a > b then
        | c = a DIV b
    else
        | c = b MOD a

    return c

```

pseudocode

▷ exerciceB(2, 3) = ____

▷ exerciceB(4, 1) = ____

```

algorithm exerciceC(x1, x2 : entiers) → entier
    booléen ok
    ok = x1 > x2
    if ok then
        | ok = ok ET x1 = 4
    else
        | ok = ok OU x2 = 3

    if ok then
        | x1 = x1 * 1000

    return x1 + x2

```

pseudocode

▷ exerciceC(2, 3) = ____

▷ exerciceC(4, 1) = ____

46 Simplification d'algorithmes

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

```

if condition then
    | ok = vrai
else
    | ok = faux

```

pseudocode

```

if a > b then
    | ok = faux
else
    | if a ≤ b then
        | ok = vrai

```

pseudocode

47 Maximum de 2 nombres

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

Solution. Une solution complète est disponible dans la fiche 4 page 216.

48 Calcul de salaire

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

49 Fonction de Syracuse

Écrire un algorithme qui, étant donné un entier n quelconque, retourne le résultat de la fonction $f(n) = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n + 1 & \text{si } n \text{ est impair} \end{cases}$

50 Tarif réduit ou pas

Dans une salle de cinéma, le tarif plein pour une place est de 8€. Les personnes ayant droit au tarif réduit payent 7€. Écrire un algorithme qui reçoit un booléen indiquant si la personne peut bénéficier du tarif réduit et qui retourne le prix à payer.

51 Maximum de 3 nombres

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

52 Le signe

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

53 Le type de triangle

Écrire un algorithme qui indique si un triangle dont on donne les longueurs de ces 3 cotés est : équilatéral (tous égaux), isocèle (2 égaux) ou quelconque.

54 Dés identiques

Écrire un algorithme qui lance trois dés et indique si on a obtenu 3 dés de valeur identique, 2 ou aucun.

55 Grade

Écrire un algorithme qui retourne le grade d'un étudiant suivant la moyenne qu'il a obtenue.

Un étudiant ayant obtenu

- ▷ moins de 50% n'a pas réussi ;
- ▷ de 50% inclus à 60% exclu a réussi ;
- ▷ de 60% inclus à 70% exclu a une satisfaction ;
- ▷ de 70% inclus à 80% exclu a une distinction ;
- ▷ de 80% inclus à 90% exclu a une grande distinction ;
- ▷ de 90% inclus à 100% inclus a la plus grande distinction.

56 Numéro du jour

Écrire un algorithme qui retourne le numéro du jour de la semaine reçu en paramètre (1 pour "lundi", 2 pour "mardi"...).

57 Tirer une carte

Écrire un algorithme qui affiche l'intitulé d'une carte tirée au hasard dans un paquet de 52 cartes. Par exemple, "As de cœur", "3 de pique", "Valet de carreau" ou encore "Roi de trèfle".

Remarque. Il est plus facile de déterminer séparément chacune des deux caractéristiques de la carte : couleur et valeur.

58 Nombre de jours dans un mois

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

59 Réussir DEV1

Pour réussir l'UE (unité d'enseignement) DEV1, il faut que la cote attribuée à cette UE soit supérieure ou égale à 50%. Cette cote tient compte de votre examen intégré et de vos interrogations. Écrire un algorithme qui reçoit la cote finale (sur 100) d'un étudiant pour l'UE DEV1 et qui indique si l'étudiant a réussi cette UE.

60 La fourchette

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

61 Le prix des photocopies

Un magasin de photocopies facture 0,10 € les dix premières photocopies, 0,09 € les vingt suivantes et 0,08 € au-delà. Écrivez un algorithme qui reçoit le nombre de photocopies effectuées et qui affiche la facture correspondante.

62 Le stationnement alternatif

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

A.6 Exercices : modules et méthodes**63 Tracer des algorithmes**

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

```

algorithm ex1()
  entiers x, y
  addition(3, 4, x)
  print x
  x = 3
  y = 5
  addition(x, y, y)
  print y

```

```

algorithm addition(a↓, b↓, c↑ : entiers)
  entier somme
  somme = a + b
  c = somme

```

pseudocode

```

algorithm ex2()
  entiers a, b
  addition(3, 4, a) // voir ci-dessus
  print a
  a = 3
  b = 5
  soustraction(b, a, b)
  print b

```

```

algorithm soustraction(a↓, b↓, c↑ : entiers)
  c = a - b

```

pseudocode

```
algorithm ex3()  
  entiers a, b, c  
  calcul(3, 4, c)  
  print c  
  a = 3  
  b = 4  
  c = 5  
  calcul(b, c, a)  
  print a, b, c
```

```
algorithm calcul( $a \downarrow, b \downarrow, c \uparrow$  : entiers)  
  a = 2 * a  
  b = 3 * b  
  c = a + b
```

pseudocode

```
algorithm ex4()  
  entiers a, b, c  
  a = 3  
  b = 4  
  c = f(b)  
  print c  
  calcul2(a, b, c)  
  print a, b, c
```

```
algorithm calcul2( $a \downarrow, b \downarrow, c \uparrow$  : entiers)  
  a = f(a)  
  c = 3 * b  
  c = a + c
```

```
algorithm f( $a \downarrow$  : entier)  $\rightarrow$  entier  
  entier b  
  b = 2 * a + 1  
  return b
```

pseudocode

64 Appels de module

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

algorithm PGCD($a \downarrow, b \downarrow$: entiers) \rightarrow entier

pseudocode

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

pseudocode

65 Maximum de 4 nombres

Écrivez un algorithme qui calcule le maximum de 4 nombres.

66 Écart entre 2 durées

Étant donné deux durées données chacune par trois nombres (heure, minute, seconde), écrire un algorithme qui calcule le délai écoulé entre ces deux durées en heure(s), minute(s), seconde(s) sachant que la deuxième durée donnée est plus petite que la première.

67 Tirer une carte

L'exercice suivant a déjà été résolu. Refaites une solution modulaire.

Écrire un algorithme qui affiche l'intitulé d'une carte tirée au hasard dans un paquet de 52 cartes. Par exemple, "As de cœur", "3 de pique", "Valet de carreau" ou encore "Roi de trèfle".

68 Nombre de jours dans un mois

Écrire un algorithme qui donne le nombre de jours dans un mois. Il reçoit en paramètre le numéro du mois (1 pour janvier...) ainsi que l'année. Pour le mois de février, il faudra répondre 28 ou 29 selon que l'année fournie est bissextile ou pas. Vous devez réutiliser au maximum ce que vous avez déjà fait lors d'exercices précédents (cf. exercice 41 page 182).

69 Valider une date

Écrire un algorithme qui valide une date donnée par trois entiers : l'année, le mois et le jour. Vous devez réutiliser au maximum ce que vous avez déjà fait lors d'exercices précédents.

70 Généraliser un algorithme

Dans l'exercice 33 page 181, nous avons écrit un algorithme pour tester si un nombre est divisible par 5. Si on vous demande à présent un algorithme pour tester si un nombre est divisible par 3, vous le feriez sans peine. Idem pour tester la divisibilité par 2, 4, 6, 7, 8, 9... mais vous vous lasseriez bien vite.

N'est-il pas possible d'écrire un seul algorithme, plus général, qui résolve tous ces problèmes d'un coup ?

A.7 Exercices : structures itératives**A.7.1 while****71 Compréhension d'algorithmes**

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

algorithm *boucle1()*

```
entier x
x = 0
while x < 12
    x = x + 2

print x
```

pseudocode

algorithm *boucle2()*

```
booléen ok
entier x
ok = vrai
x = 5
while ok
    x = x + 7
    ok = x MOD 11 ≠ 0

print x
```

pseudocode

algorithm *boucle3()*

```
booléen ok
entiers cpt, x
x = 10
cpt = 0
ok = vrai
while ok ET cpt < 3
    if x MOD 2 = 0 then
        x = x + 1
        ok = x < 20
    else
        x = x + 3
        cpt = cpt + 1

print x
```

pseudocode

72 Afficher des nombres

En utilisant un **while**, écrire un algorithme qui reçoit un entier n positif et affiche

- a) les nombres de 1 à n ;
- b) les nombres de 1 à n en ordre décroissant ;
- c) les nombres impairs de 1 à n ;
- d) les nombres de $-n$ à n ;
- e) les multiples de 5 de 1 à n ;
- f) les multiples de n de 1 à 100.

A.7.2 for**73 Compréhension d'algorithmes**

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

```

algorithm boucle5()
    entier x
    booléen ok
    x = 3
    ok = vrai
    for i from 1 to 5
        x = x + i
        ok = ok ET (x MOD 2 = 0)

    if ok then
        print x
    else
        print 2 * x

```

pseudocode

```

algorithm boucle6()
    entiers fin
    for i from 1 to 3
        fin = 6 * i - 11
        for j from 1 to fin by 3
            print 10 * i + j

```

pseudocode

74 Afficher des nombres

Reprenons un exercice déjà donné avec le **while**. En utilisant un **for**, écrire un algorithme qui reçoit un entier n positif et affiche

- a) les nombres de 1 à n ;
- b) les nombres de 1 à n en ordre décroissant ;
- c) les nombres de $-n$ à n ;
- d) les multiples de 5 de 1 à n ;
- e) les multiples de n de 1 à 100.

A.7.3 do while

75 Compréhension d'algorithmes

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

```

algorithm boucle4()
  booléens pair, grand
  entiers p, x
  x = 1
  p = 1
  do
    p = 2 * p
    x = x + p
    pair = x MOD 2 = 0
    grand = x > 15
  while NON pair ET NON grand
  print x

```

pseudocode

76 Afficher des nombres

Reprenons un exercice déjà fait avec le **while** et le **for** en utilisant cette fois un **do while**. Écrire un algorithme qui reçoit un entier n positif et affiche

- a) les nombres de 1 à n ;
- b) les nombres de 1 à n en ordre décroissant ;
- c) les nombres de $-n$ à n ;
- d) les multiples de 5 de 1 à n ;
- e) les multiples de n de 1 à 100.

A.7.4 Exercices récapitulatifs sur les structures répétitives

77 Afficher les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières à l'utilisateur et qui affiche celles qui sont impaires. L'algorithme commence par demander à l'utilisateur le nombre de valeurs qu'il désire donner.

78 Compter les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières à l'utilisateur et qui lui affiche le nombre de valeurs impaires qu'il a donné. Après chaque valeur entrée, l'algorithme demande à l'utilisateur s'il y en a encore d'autres.

79 Choix de la valeur sentinelle

Quelle valeur sentinelle prendrait-on pour additionner une série de cotes d'interrogations ? Une série de températures ?

80 Afficher les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières non nulles à l'utilisateur et qui affiche celles qui sont impaires. La fin des données sera signalée par la valeur sentinelle 0.

81 Compter le nombre de réussites

Écrire un algorithme qui demande une série de cotes (entières, sur 20) à l'utilisateur et qui affiche le pourcentage de réussites. La fin des données sera signalée par une valeur sentinelle que vous pouvez choisir.

82 Suites

Écrire les algorithmes qui affichent les n premiers termes des suites suivantes. À vous de voir quel est, parmi les 2 modèles décrits dans les exemples précédents, le modèle de solution le plus pcpté.

- a) -1, -2, -3, -4, -5, -6...
- b) 1, 3, 6, 10, 15, 21, 28...
- c) 1, 0, 1, 0, 1, 0, 1, 0...
- d) 1, 2, 0, 1, 2, 0, 1, 2...
- e) 1, 2, 3, 1, 2, 3, 1, 2...
- f) 1, 2, 3, 2, 1, 2, 3, 2...

Remarque Pour les exercices qui suivent, nous vous donnons peu d'indications sur la solution à mettre en œuvre. À vous de jouer...

83 Lire un nombre

Écrire un algorithme qui demande à l'utilisateur un nombre entre 1 et n et le retourne. Si la valeur donnée n'est pas dans l'intervalle souhaité, l'utilisateur est invité à rentrer une nouvelle valeur jusqu'à ce qu'elle soit correcte.

algorithm *lireNb*(n : entier) \rightarrow entier

pseudocode

Solution. Si la valeur donnée par l'utilisateur était toujours correcte, il suffirait d'écrire :

```
algorithm lireNb(n : entier) → entier
    entier val
    read val
    return val
```

pseudocode

Pour tenir compte des entrées invalides, il faut ajouter une boucle qui prévient que la valeur est mauvaise et la redemande. Et ceci, tant que c'est incorrect. Ce qui donne :

```
algorithm lireNb(n : entier) → entier
    entier val
    read val
    while val < 1 OU val > n
        print "valeur incorrecte"
        read val
    return val
```

pseudocode

La solution qu'on obtient est proche d'une lecture répétée avec valeur sentinelle. Dans ce cas, la valeur sentinelle est toute valeur correcte.

84 Lancé de dés

Écrire un algorithme qui lance n fois un dé et compte le nombre de fois qu'une certaine valeur est obtenue.

```
algorithm lancerDé(n : entier, val : entier) → entier
```

pseudocode

85 Factorielle

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

Par convention, $0! = 1$.

86 Produit de 2 nombres

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

87 Table de multiplication

Écrire un algorithme qui affiche la table de multiplication des nombres de 1 à 10 (cf. l'exemple ci-contre).

Attention ! Nous sommes en algorithmique, ne vous préoccupez pas de la mise en page de ce qui est affiché. Ce sera une question importante quand vous traduirez l'algorithme dans un langage de programmation mais pas ici.

```

1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
2 x 1 = 2
...
10 x 9 = 90
10 x 10 = 100

```

88 Double 6

Écrire un algorithme qui lance de façon répétée deux dés. Il s'arrête lorsqu'il obtient un double 6 et retourne le nombre de lancers effectués.

89 Nombre premier

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

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

90 Nombres premiers

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

91 Somme de chiffres

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

92 Nombre parfait

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

Par exemple, 6 est parfait car $6 = 1 + 2 + 3$. De même, 28 est parfait car $28 = 1 + 2 + 4 + 7 + 14$.

93 Décomposition en facteurs premiers

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

94 Nombre miroir

Le miroir d'un nombre est le nombre obtenu en lisant les chiffres de droite à gauche. Ainsi le nombre miroir de 4209 est 9024. Écrire un algorithme qui calcule le miroir d'un nombre entier positif donné.

95 Palindrome

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

96 Jeu de la fourchette

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

A.8 Exercices : tableaux**97 Déclarer et initialiser**

Écrire un algorithme qui déclare un tableau de 100 chaînes et met votre nom dans la 3^e case du tableau.

98 Initialiser un tableau à son indice

Écrire un algorithme qui déclare un tableau de 100 entiers et initialise chaque élément à la valeur de son indice. Ainsi, la case numéro i contiendra la valeur i .

99 Initialiser un tableau aux valeurs de 1 à 100

Écrire un algorithme qui déclare un tableau de 100 entiers et y met les nombres de 1 à 100.

100 Compréhension

Expliquez la différence entre $\text{tab}[i] = \text{tab}[i+1]$ et $\text{tab}[i] = \text{tab}[i] + 1$.

101 Trouver les entêtes

Écrire les entêtes (et uniquement les entêtes) des algorithmes qui résolvent les problèmes suivants :

- Écrire un algorithme qui inverse le signe de tous les éléments négatifs dans un tableau d'entiers.
- Écrire un algorithme qui donne le nombre d'éléments négatifs dans un tableau d'entiers.
- Écrire un algorithme qui détermine si un tableau d'entiers contient au moins un nombre négatif.
- Écrire un algorithme qui détermine si un tableau de chaînes contient une chaîne donnée en paramètre.

- e) Écrire un algorithme qui détermine si un tableau de chaînes contient au moins deux occurrences de la même chaîne, quelle qu'elle soit.
- f) Écrire un algorithme qui retourne un tableau donnant les n premiers nombres premiers, où n est un paramètre de l'algorithme.
- g) Écrire un algorithme qui reçoit un tableau d'entiers et retourne un tableau de booléens de la même taille où la case i indique si oui ou non le nombre reçu dans la case i est strictement positif.

102 Inverser le signe des éléments

Écrire un algorithme qui inverse le signe de tous les éléments négatifs dans un tableau d'entiers.

103 Somme

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

104 Nombre d'éléments d'un tableau

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

105 Compter les éléments négatifs

Écrire un algorithme qui donne le nombre d'éléments négatifs dans un tableau d'entiers.

106 Y a-t-il une copie parfaite ?

Écrire un algorithme qui reçoit en paramètre le tableau `cotes` de n entiers représentant les cotes des étudiants et qui retourne un booléen indiquant s'il contient **au moins** une fois la valeur 20.

107 Y a-t-il une seule copie parfaite ?

Écrire un algorithme qui reçoit en paramètre le tableau `cotes` de n entiers représentant les cotes des étudiants et qui retourne un booléen indiquant s'il contient **exactement** une fois la valeur 20.

108 Lancers de deux dés

Écrire un algorithme qui lance n fois deux dés et compte le nombre de fois que chaque somme apparaît.

algorithm *lancers2Dés*(n : entier) \rightarrow array [2 à 12] de entiers

pseudocode

109 Nombre de jours dans un mois

Écrire un algorithme qui reçoit un numéro de mois (de 1 à 12) ainsi qu'une année et donne le nombre de jours dans ce mois (en tenant compte des années bissextiles). N'hésitez pas à réutiliser des algorithmes déjà écrits.

110 Éviter la double inscription

Comment modifier l'algorithme d'inscription pour s'assurer qu'un étudiant ne s'inscrive pas deux fois ?

111 Limite au nombre d'inscriptions

Comment modifier l'algorithme d'inscription pour refuser une inscription si le nombre maximal de participants est atteint en supposant que ce maximum est égal à la taille physique du tableau ?

112 Vérifier la désinscription

Que se passerait-il avec l'algorithme de désinscription tel qu'il est si on demande à désinscrire un étudiant non inscrit ? Que suggérez-vous comme changement ?

113 Calcul de complexités

Quelle est la complexité d'un algorithme qui :

- a) recherche le maximum d'un tableau de n éléments ?
- b) remplace par 0 toutes les occurrences du maximum d'un tableau de n éléments ?
- c) vérifie si un tableau contient deux éléments égaux ?
- d) vérifie si les éléments d'un tableau forment un palindrome ?
- e) cherche un élément dans un tableau en essayant des cases au hasard jusqu'à le trouver ?

114 Gestion des données

Comparer les algorithmes d'ajout, de recherche et de suppression de valeurs dans le cas d'un tableau trié et d'un tableau non trié.

115 Réflexion

L'algorithme de recherche dichotomique est-il toujours à préférer à l'algorithme de recherche linéaire ?



Remarque Voici quelques exercices qui reprennent les différentes notions vues sur les tableaux. Pour chacun d'entre-eux :

- ▷ demandez-vous quel est le problème le plus proche déjà rencontré et voyez comment adapter la solution ;
- ▷ indiquez la complexité de votre solution ;
- ▷ décomposez votre solution en plusieurs algorithmes si ça peut améliorer la lisibilité ;
- ▷ testez votre solution dans le cas général et dans les éventuels cas particuliers.

116 Renverser un tableau

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

117 Tableau symétrique ?

Écrire un algorithme qui reçoit en paramètre le tableau `tabChaines` de n chaînes 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.

118 Maximum

Écrire un algorithme qui reçoit en paramètre le tableau `tabEnt` de n entiers et qui retourne la plus **grande** valeur de ce tableau.

119 Minimum

Écrire un algorithme qui reçoit en paramètre le tableau `tabEnt` de n entiers et qui retourne la plus **petite** valeur de ce tableau. Idem pour le minimum.

120 Indice du maximum/minimum

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

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

121 Tableau ordonné ?

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

122 Remplir un tableau

On souhaite remplir un tableau de 20 éléments avec les entiers de 1 à 5, chaque nombre étant répété quatre fois. On voudrait deux variantes :

- a) D'abord une version où les nombres identiques sont groupés : d'abord tous les 1 puis tous les 2, ...

1	1	1	1	2	2	2	2	3	3	3	3	4	4
								4	4	5	5	5	5

- b) Ensuite une version où on trouve dans le tableau les valeurs 1, 2, 3, 4, 5 qui se suivent, quatre fois.

1	2	3	4	5	1	2	3	4	5	1	2	3	4
								5	1	2	3	4	5

Remarque : Il existe de nombreuses façons de résoudre ce problème. On peut par exemple utiliser deux boucles imbriquées. On peut aussi adapter un algorithme de génération de suites.

123 Positions du minimum

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

- Écrire une première version « classique » avec deux parcours de tableau
- Écrire une deuxième version qui ne parcourt qu'une seule fois **cotes** en stockant dans un deuxième tableau (de quelle taille ?) les indices du plus petit élément rencontrés (ce tableau étant à chaque fois réinitialisé lorsqu'un nouveau minimum est rencontré)
- Écrire une troisième version qui **retourne** le tableau contenant les indices. Écrire également un algorithme qui appelle cette version puis affiche les indices reçus.

124 Occurrence des chiffres

Écrire un algorithme qui reçoit un nombre entier positif ou nul en paramètre et qui retourne un tableau de 10 entiers indiquant, pour chacun de ses chiffres, le nombre de fois qu'il apparaît dans ce nombre. Ainsi, pour le nombre 10502851125, on retournera le tableau {2,3,2,0,0,3,0,0,1,0}.

125 Les doublons

Écrire un algorithme qui vérifie si un tableau de chaînes contient au moins 2 éléments égaux.

126 Le crible d'Ératosthène

« Le crible d'Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné N . L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers. On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant. On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant, car dans ce cas, tous les non-premiers ont déjà été rayés précédemment. » (source : Wikipédia)

Le tableau dont il est question peut être un simple tableau de booléens ; le booléen en position "i" indiquant si le nombre "i" est premier ou pas.

Écrire un algorithme qui reçoit un entier n et affiche tous les entiers premiers de 1 à n .

127 Mastermind

Dans le jeu du Mastermind, un joueur A doit trouver une combinaison de n 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.

Les propositions du joueur A, ainsi que la combinaison secrète du joueur B sont contenues dans des tableaux de n composantes de type chaîne.

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

```
algorithm testerProposition( proposition↓, solution↓ : array of n chaînes,
                             bienPlacés↑, malPlacés↑ : entiers)
```

pseudocode

A.9 Exercices : chaîne et String**128 Calcul de fraction**

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

Solution. Voici une solution possible.

```

algorithm calculerFraction(fraction : string) → real
    integer posFraction, numérateur, dénominateur
    posFraction = position(fraction, "/")
    numérateur = nombre(souschaîne(fraction, 1, posFraction-1))
    dénominateur = nombre(souschaîne(
        fraction,
        posFraction+1,
        long(fraction)-posFraction))
    return numérateur / dénominateur

```

pseudocode

129 Conversion de nom

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

130 Gauche et droite

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

131 Grammaire

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

132 Normaliser une chaîne

Écrire un module qui reçoit une chaîne et retourne une autre chaîne, version normalisée de la première. Par normalisée, on entend : enlever tout ce qui n'est pas une lettre et tout mettre en majuscule.

Exemple : "Le <COBOL>, c'est la santé!" devient "LECOBOLCESTLASANTE".

133 Les palindromes

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

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

134 Le chiffre de César

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

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

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

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

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

135 Remplacement de sous-chaînes

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

A.10 Exercices : structures et classes

Dans ces exercices nous supposons que vous avez au préalable écrit les structures et classes `Moment`, `Point`... telles que décrites dans le chapitre 13 p 165.

136 Conversion moment-secondes

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

137 Conversion secondes-moment

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

138 Temps écoulé entre 2 moments

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

139 Milieu de 2 points

Écrire un module recevant deux points du plan et qui retourne le point situé au milieu des deux.

140 Distance entre 2 points

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

141 Un cercle

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

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

142 Un rectangle

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

- a) un module calculant le périmètre d'un rectangle reçu en paramètre ;
- b) un module calculant la surface d'un rectangle reçu en paramètre ;
- c) un module recevant en paramètre un rectangle R et les coordonnées d'un point P, et renvoyant **vrai** si et seulement si le point P est à l'intérieur du rectangle R ;

143 Validation de date

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

A.11 Exercices récapitulatifs

La plupart des exercices qui suivent sont inspirés d'anciens examens. Ils sont assez conséquents et reprennent la plupart des notions vues dans ce cours.

A.11.1 AEBBCLRS

Le Scrabble est un jeu de lettres, où le but est de former des mots ayant le score le plus élevé possible. Pour calculer le score d'un mot, une valeur est associée à chaque lettre de l'alphabet. Par exemple la lettre A vaut 1, V vaut 4, J vaut 8, ... Le score d'un mot est la somme de la valeur de ses lettres⁶⁵.

65. Dans le vrai jeu c'est un peu plus compliqué.

Par exemple le score du mot **JAVA** est $8 + 1 + 4 + 1 = 14$.

Les joueurs tirent chacun 7 lettres qu'ils placent sur un *chevalet*. Le but du jeu est de former des mots à partir des lettres du chevalet et de les placer sur le plateau de jeu.

Valeur des lettres

Écrire un algorithme, `valeurLettre`, qui reçoit un caractère et retourne sa valeur. Les valeurs des lettres sont les suivantes :

- ▷ A,E,I,L,N,O,R,S,T,U : 1 point
- ▷ D,G,M : 2 points
- ▷ B,C,P : 3 points
- ▷ F,H,V : 4 points
- ▷ J,Q : 8 points
- ▷ K,W,X,Y,Z : 10 points

L'algorithme lance une erreur avec un message adéquat si le caractère passé en paramètre n'est pas une lettre majuscule.

Score d'un mot

Écrire un algorithme, `scoreMot`, qui reçoit une chaîne de caractères, `mot`, et retourne la valeur de ce mot, c'est-à-dire la somme de la valeur de chacune de ses lettres.

Exemple : si l'algorithme reçoit le mot **JAVA**, il retourne 14 ($=8+1+4+1$).

Mot possible

Écrire un algorithme, `motPossible`, qui reçoit un tableau de caractères, `chevalet`, et une chaîne de caractères, `mot`, et retourne vrai si les lettres du chevalet permettent d'obtenir le mot donné, et retourne faux dans le cas contraire.

Exemple : si l'algorithme reçoit le tableau `[A, O, V, G, G, J, L]` et le mot `"ALGO"`, l'algorithme retourne vrai. Par contre si le mot donné est `"JAVA"`, alors l'algorithme retourne faux car la lettre **A** n'apparaît pas 2 fois dans le chevalet.

Aide : Une solution possible est de faire une copie du chevalet et pour chaque lettre du mot de vérifier que la lettre s'y trouve, si elle s'y trouve de la remplacer par un symbole (par exemple un '-'). Pour cela on définit un algorithme `copieChevalet` permettant de copier un chevalet et un algorithme `indiceLettre` permettant de connaître l'indice d'une lettre dans un chevalet (cet algorithme retourne -1 si la lettre ne s'y trouve pas).

Meilleur mot

Écrire un algorithme, `meilleurMot`, qui reçoit :

- ▷ un tableau de caractères, `chevalet`, qui contient les lettres disponibles pour former un mot ;
- ▷ un tableau de chaînes de caractères, `dico`, qui contient par exemple tous les mots du dictionnaire.

L'algorithme retourne le mot du dictionnaire que l'on peut obtenir avec les lettres du `chevalet` et qui a le score le plus élevé. Si aucun mot n'est possible avec le `chevalet`, l'algorithme retourne un mot vide "".

A.11.2 Serpents et échelles

Serpents et échelles est un jeu de société populaire, se jouant à plusieurs, consistant à déplacer les jetons sur un tableau de cases avec un dé en essayant de monter les échelles et en évitant de trébucher sur les serpents.

Pour l'histoire, le jeu peut être perçu comme une représentation d'un chemin spirituel que les humains prennent pour atteindre le ciel. Avec des bons gestes, le chemin est raccourci (ce que symbolisent les échelles), tandis qu'avec de mauvais gestes, le chemin est allongé (d'où vient le symbolisme des serpents).

Extrait de Wikipedia.

Représentation du chemin

Nous représenterons le chemin par un tableau d'entiers. Chaque case du tableau contiendra une valeur :

- ▷ positive pour représenter une échelle et permettre d'avancer plus vite ;
- ▷ négative pour représenter le serpent et ralentir (voire reculer).

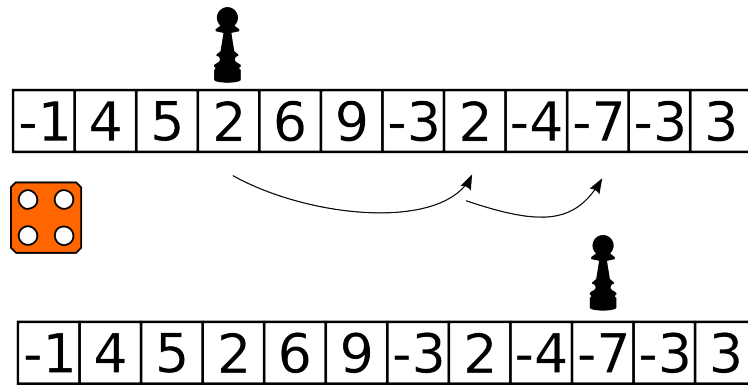
Règles du jeu

Pour jouer, le joueur qui a la main lance le dé et avance de la valeur indiquée par le dé. À partir de cette nouvelle position, il vérifie la valeur de la case sur laquelle il se trouve pour continuer à avancer ou au contraire reculer. Si la valeur est positive, il avance et si elle est négative, il recule de la valeur contenue dans la case.

Si la case est déjà occupée, le joueur se placera sur la case suivante.

Le joueur passe la main au joueur suivant.

Exemple. Le joueur se trouvant à la position 3 lance le dé. Il obtient la valeur 4. Il avance de 4 cases et se retrouve sur la case en position 7 qui contient la valeur 2. Il avance encore de 2 cases ... et termine donc son tour en position 9. Il passe alors la main au joueur suivant.



Le premier joueur à atteindre ou dépasser la dernière case a gagné. Nous supposons que cette dernière case est en position 42. Nous supposons également que la première et la dernière case n'ont ni échelle ni serpent (valeur nulle).

Positions des joueurs

Les positions des joueurs seront mémorisées dans un tableau d'entiers. Toutes les cases de ce tableau sont initialisées à 0.

Le plateau de jeu, un tableau d'entiers s'appellera **chemin**.

Le tableau d'entiers contenant la position courante des joueurs s'appellera **positionsJoueurs**.

S'il y a 4 joueurs, le tableau *positionsJoueurs* contient [2,5,1,7] alors :

- ▷ le joueur 0 est en position 2 sur le chemin ;
- ▷ le joueur 1 est en position 5 sur le chemin ;
- ▷ le joueur 2 est en position 1 sur le chemin ;
- ▷ le joueur 3 est en position 7 sur le chemin ;

Le premier jour, initialiser le chemin

Écrivez un algorithme

algorithm *créerChemin*($n \downarrow$: entier) \rightarrow array of n entier

pseudocode

Cet algorithme *créerChemin* créera un tableau d'entiers dont les valeurs seront des valeurs aléatoires comprises strictement entre -10 et 10 (inclus).

Nous supposons que l'entier n est un naturel strictement supérieur à 0. Vous ne devez pas le vérifier.

Qui est en tête ?

Écrivez un algorithme

algorithm *enPremièrePosition*(positionsJoueurs↓ : array of m entiers) → entier

pseudocode

Cet algorithme retourne le numéro du joueur en tête de la course. Il recherche donc le maximum dans le tableau passé en argument et retourne l'indice de ce maximum.

Jouer un tour de jeu

Cet algorithme permettra de jouer un tour de jeu pour un joueur donné. Jouer un tour de jeu consiste à le faire avancer ou reculer (c'est-à-dire changer sa position) du bon nombre de cases.

algorithm *jouer*(chemin↓ : array of n entiers, positionsJoueurs↓↑ : array of m entiers, joueurCourant↓ : entier, valeurDé↓ : entier)

pseudocode

Cet algorithme fait changer de position le joueur d'indice *joueurCourant* de la valeur donnée par le dé (*valeurDé*) en respectant les règles du jeu.

Cet algorithme met donc à jour le tableau *positionsJoueurs*.

Comme le joueur ne peut se placer sur une case déjà occupée, il peut être utile d'écrire un module *estOccupé* précisant si la case est libre ou occupée.

A.11.3 Les algorithmes en maternelle

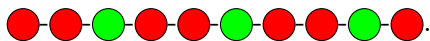
En maternelle, déjà, les enfants font des algorithmes mais il s'agit d'une chose un peu différente. On leur propose un collier⁶⁶ dessiné sur une feuille de papier et ils doivent le colorier en répétant un *motif* donné⁶⁷, une séquence précise de couleurs.

Par exemple, on donne ce collier



qui est pré-colorié avec deux perles rouges et une perle verte, c'est le motif.

Le résultat attendu est :



Comme on peut le constater sur cet exemple, un motif peut comporter plusieurs perles de la même couleur et, en fin de collier, il est possible qu'on ne puisse appliquer qu'une partie du motif.

Nous allons représenter chaque perle par un caractère indiquant sa couleur et un collier comme un tableau de caractères. Une perle non coloriée sera indiquée par un point ('.').

Par exemple, le collier ci-dessus serait représenté ainsi :

66. Par exemple, mais ça peut aussi être une chenille, un escargot, une simple suite de cases...

67. Ce qu'ils appellent un *algorithme*.

'R'	'R'	'V'	'.'	'.'	'.'	'.'	'.'	'.'	'.'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Créer un collier

Écrivez un algorithme **créerCollier** qui reçoit une taille et crée un collier de cette taille où toutes les perles sont non coloriées (rappel : une perle non coloriée est représentée par un point).

On suppose que la taille reçue est bien un entier non négatif.

Créer un motif

Écrivez un algorithme **créerMotif** qui reçoit un collier dont aucune perle n'est coloriée et qui colorie les premières en fonction des indications de l'utilisateur.

Concrètement, l'utilisateur entre les couleurs des perles en spécifiant à chaque fois, après, s'il y a encore une perle à colorier. Dans notre exemple de la première page, l'utilisateur entrerait successivement : 'R', vrai, 'R', vrai, 'V', faux.

L'algorithme doit vérifier que l'utilisateur ne demande pas à colorier plus de perles qu'il n'y en a dans le collier.

Taille du motif

Écrivez un algorithme **tailleMotif** qui reçoit un collier dont seulement les premières perles sont coloriées (c'est le motif qu'il faudra suivre) et qui donne la taille de ce motif. Dans l'exemple de la première page, il faudra retourner la valeur 3. Votre algorithme doit générer une erreur si il n'y a pas de motif à suivre.

Suivre un motif

Écrivez un algorithme **colorier** qui reçoit un collier dont seulement les premières perles sont coloriées (c'est le motif qu'il faut suivre) et qui colorie le reste du collier en suivant ce motif.

Vérifier un collier

Écrivez un algorithme **vérifier** qui reçoit un collier complètement colorié et la taille du motif de départ et qui vérifie si le collier respecte ce motif.

Trouver le motif

Écrivez un algorithme **trouverMotif** qui reçoit un collier complètement colorié et qui détermine la taille du motif. C'est la plus petite séquence qui se répète. Comme cas extrême, ce pourrait être le collier tout entier.

Aide : vous avez déjà tout pour que cet exercice soit facile.

Annexe **B**

Les fiches

Vous trouverez ici toutes les fiches des algorithmes analysés dans ce cours.

1	Un calcul simple	212
2	Un calcul complexe	213
3	Un nombre pair	215
4	Maximum de deux nombres	216
5	Passage d'un tableau en paramètre	217
6	Parcours complet d'un tableau	219
7	Parcours partiel d'un tableau	220
8	Maximum dans un tableau	222
9	Tableau non trié	223
10	Tableau trié	225
11	Recherche dichotomique	228
12	Tri d'un tableau	229

Fiche n° 1 – Un calcul simple

Le problème

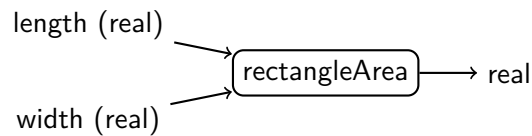
Calculer la surface d'un rectangle à partir de sa longueur et sa largeur.

Spécification

Données (toutes réelles et non négatives) :

- ▷ la longueur du rectangle ;
- ▷ sa largeur.

Résultat : un réel représentant la surface du rectangle.



Exemples

- ▷ `rectangleArea(3, 2)` donne 6
- ▷ `rectangleArea(3.5, 1)` donne 3.5

Solution

La surface d'un rectangle est obtenue en multipliant la largeur par la longueur.

$$\text{surface} = \text{longueur} * \text{largeur}$$

```

algorithm rectangleArea(length, width : real) → real
  return length * width
  
```

pseudocode

Vérification

test n°	longueur	largeur	réponse attendue	réponse fournie	
1	3	2	6	6	✓
2	3.5	1	3.5	3.5	✓

Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois que la réponse s'obtient par un calcul simple sur les données. Si le calcul est plus complexe, il peut être utile de le décomposer pour accroître la lisibilité (cf. fiche 2 page ci-contre)

Fiche n° 2 – Un calcul complexe

Problème

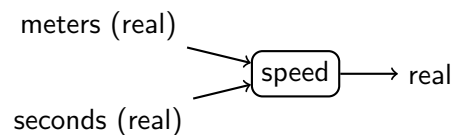
Calculer la vitesse moyenne (en km/h) d'un véhicule dont on donne la distance parcourue (en mètres) et la durée du parcours (en secondes).

Spécification

Données (toutes réelles et non négatives) :

- ▷ la distance parcourue par le véhicule (en m) ;
- ▷ la durée du parcours (en s).

Résultat : la vitesse moyenne du véhicule (en km/h).



Exemples

- ▷ `speed(100,10)` donne 36
- ▷ `speed(10000,3600)` donne 10

Solution

La vitesse moyenne est liée à la distance et à la durée par la formule :

$$\text{vitesse} = \frac{\text{distance}}{\text{durée}}$$

pour autant que les unités soient cohérentes.

Ainsi pour obtenir une vitesse en km/h, il faut convertir la distance en kilomètres et la durée en heures. Ce qui donne :

```

algorithm speed(meters, seconds : real) → real
  real km, hours
  km = meters / 1000
  hours = seconds / 3600
  return km / hours
  
```

pseudocode

Vérification

test n°	dist. (m)	durée (s)	rép. attendue	rép. fournie	
1	100	10	36	36	✓
2	10000	3600	10	10	✓

Solution Java

```
public static double speed(double meters, double seconds){  
2  double km;  
   double hours;  
   km = meters / 1000;  
5  hours = seconds / 3600;  
   return km / hours;  
}
```

java

Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois que la réponse s'obtient par un calcul complexe sur les données qu'il est bon de décomposer pour aider à sa lecture. Si le calcul est plutôt simple, on peut le garder en une seule expression (cf. fiche 1 page 212).

Fiche n° 3 – Un nombre pair

Problème

Un nombre reçu en paramètre est-il pair⁶⁸ ?

Spécification

Données : le nombre entier dont on veut savoir si il est pair.

Résultat : un booléen à *vrai* si le *nombre* est pair et *faux* sinon.

number (integer) \longrightarrow isEven \longrightarrow boolean

Exemples

- ▷ isEven(2016) donne *vrai*
- ▷ isEven(2015) donne *faux*

Solution

Un nombre est pair si il est multiple de 2. C'est-à-dire si le reste de sa division par 2 vaut 0.

```
algorithm isEven(number : integer)  $\rightarrow$  boolean
|   return number MOD 2 == 0
```

pseudocode

Attention à éviter les mauvaises écritures expliquées à l'annexe C.2 page 234 : pas de **si-sinon**.

Quand l'utiliser ?

À chaque fois qu'un résultat booléen dépend d'un calcul simple. Si le calcul est plus compliqué, on peut le décomposer comme indiqué dans la fiche 2 page 213.

On peut également s'inspirer de cette solution quand il faut donner sa valeur à une variable booléenne.

⁶⁸. *Even* en anglais et *odd* pour impair.

Fiche n° 4 – Maximum de deux nombres

Problème

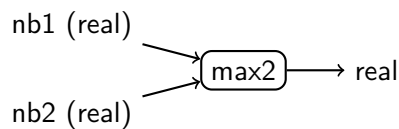
Quel est le maximum de deux nombres ?

Analyse

Voilà un classique de l'algorithmique. Attention ! On ne veut pas savoir *lequel* est le plus grand mais juste la valeur. Il n'y a donc pas d'ambiguïté si les deux nombres sont égaux.

Données : deux nombres réels.

Résultat : un réel contenant la plus grande des deux valeurs données.



Exemples

▷ $\text{max2}(-3, 4)$ donne 4 ▷ $\text{max2}(7, 4)$ donne 7 ▷ $\text{max2}(4, 4)$ donne 4

Solution

```

algorithm max2(nb1 : real, nb2 : real) → real
  real max
  if nb1 > nb2 then
    | max = nb1
  else
    | max = nb2
  return max
  
```

pseudocode

Attention à éviter les mauvaises écritures expliquées à l'annexe C.3 page 235.

Quand l'utiliser ?

Cet algorithme peut bien sûr être facilement adapté à la recherche du minimum.

Fiche n° 5 – Passage d'un tableau en paramètre

Problème

Le type *tableau* étant un type à part entière, il est tout-à-fait éligible comme type pour les paramètres et la valeur de retour d'un algorithme.

Passer un tableau en paramètre

- ▷ ↓ : indique que l'algorithme va consulter les valeurs du tableau reçu en paramètre. Les éléments doivent donc avoir été initialisés avant d'appeler l'algorithme. Exemple :

```
// Affiche les éléments d'un tableau de n entiers
algorithm display(is↓ : array of integer)
    for i from 0 to is.length - 1
        print is[i]

// Utilisation possible
array of 10 entiers myIntegers
myIntegers = {2,3,5,7,11,13,17,19,23,29}
afficher(myIntegers)
```

pseudocode

Rappel C'est le passage de paramètre par défaut si aucune flèche n'est indiquée.

- ▷ ↓↑ : indique que l'algorithme va consulter/modifier les valeurs du tableau reçu en paramètre. Exemple :

```
// Inverse le signe des éléments d'un tableau de n entiers
algorithm oppositeValues(is↓↑ : array of integer)
    for i from 0 to is.length - 1
        is[i] = -is[i]

// Utilisation possible
array of 5 entiers myIntegers
myIntegers = {2,-3,5,-7,11}
oppositeValues(myIntegers)
```

pseudocode

Rappel Java : Un tableau Java étant un type référence, c'est toujours la valeur de la référence qui est passée en paramètre. En ce sens, un passage en entrée ou en entrée / sortie est identique en Java et le tableau doit toujours exister c'est-à-dire avoir été créé au préalable.

Retourner un tableau

Comme pour n'importe quel autre type, un algorithme peut retourner un tableau. Ce sera à lui de le déclarer et de lui donner des valeurs.

Exemple :

```
// Crée un tableau d'entiers de taille n, l'initialise à 0 et le retourne.  
algorithm create(n : integer) → array of n integer  
| array of n integer is  
| for i from 0 to n-1  
| | is[i] = 0  
  
| return is  
  
  
// Utilisation possible  
algorithm test()  
| array of integer myIntegers  
| myIntegers = create(20)  
| display(myIntegers)
```

pseudocode

Fiche n° 6 – Parcours complet d'un tableau

Problème

Afficher tous les éléments d'un tableau d'entiers.

Spécification

- ▷ **Données** : le tableau à afficher
- ▷ **Résultat** : aucun.
- ▷ **Affiche** : les éléments du tableau, dans l'ordre.

Solution

Puisqu'on parcourt tout le tableau, on peut utiliser une boucle *pour* (for).

```
algorithm display(is : array of integer)
  for i from 0 to is.length - 1
    print is[i]
```

pseudocode

Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois qu'il est nécessaire d'examiner **tous** les éléments d'un tableau, quel que soit le traitement voulu : les afficher, les sommer, les comparer...

Fiche n° 7 – Parcours partiel d'un tableau

Problème

Recherche d'un zéro dans un tableau.

Spécification

Données : le tableau à tester

Résultat : un booléen à vrai si il existe une valeur nulle dans le tableau.

is (tableau d'entiers) \longrightarrow containsNulValue \longrightarrow booléen

Solution

Contrairement au parcours complet (cf. fiche 6 page précédente) nous allons utiliser un *tant que* (while) car nous voulons arrêter l'algorithme dès que la valeur cherchée est trouvée.

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

[A] Sans variable booléenne

```

algorithm containsNulValue(is : array of integer)  $\rightarrow$  boolean
    integer i
    i = 0
    while i < is.length AND is[i]  $\neq$  0
        i = i + 1

    return i < n
    // Si i < n -> arrêt prématuré
    // c-à-d que l'on a trouvé un 0.

```

pseudocode

Il faut être attentif à **ne pas inverser** les deux parties du test. Il faut absolument vérifier que l'indice est bon avant de tester la valeur à cet indice. Revoyez la notion de court-circuit (4.3.3 page 43).

[B] Avec variable booléenne

```

algorithm containsNullValue(is : array of integer) → boolean
    integer i
    boolean isZero
    isZero = false
    i = 0
    while i < is.length AND NON isZero
        isZero = is[i] == 0
        i = i + 1
    return isZero

```

pseudocode

Au sortir de la boucle, l'indice *i* ne désigne pas l'élément qui nous a permis d'arrêter mais le suivant. Si nécessaire, on peut remplacer l'intérieur de la boucle par :

```

if is[i] == 0 then
    isZero = true
else
    i = i + 1

```

pseudocode

Dans notre exemple, nous cherchons un élément particulier (un 0). Dans le cas où l'on vérifie si tous les éléments possèdent une certaine propriété (être positifs par exemple), nous veillerons à adapter le nom du booléen et son utilisation (par exemple un booléen appelé *areAllPositives*, initialisé à *true* avec un *...AND areAllPositives* dans le test.

Dans tous les cas, faites attention à ne pas utiliser *is[i]* si vous n'êtes pas sûr du *i*. C'est particulièrement vrai après la boucle.

Attention aussi à éviter les mauvaises pratiques expliquées à l'annexe C.4 page 235.

Quand l'utiliser ?

Ce type de solution peut être utilisé pour tout parcours d'un tableau où un arrêt prématuré est possible.

- ▷ Est-ce que tous les éléments sont positifs ?
- ▷ Est-ce que les éléments sont triés ?
- ▷ Est-ce qu'un élément précis est présent ?
- ▷ ...

Fiche n° 8 – Maximum dans un tableau

Problème

Trouver la valeur maximale dans un tableau d'entiers.

Spécification

Données : le tableau à analyser

Résultat : la valeur du maximum

Solution

Il faut veiller à initialiser le maximum avec la première valeur du tableau pour parcourir le reste du tableau à la recherche d'une valeur plus grande.

```
algorithm max(is : array of integer) → integer
    integer max
    max = is[0]
    for i from 1 to is.length - 1
        if is[i] > max then
            max = is[i]
    return max
```

pseudocode

Variante

Si c'est la position et non la valeur qui nous intéresse, une solution serait :

```
algorithm max(is : array of integer) → integer
    integer maxIndex
    maxIndex = 0
    for i from 1 to is.length - 1
        if is[i] > is[maxIndex] then
            maxIndex = i
    return maxIndex
```

pseudocode

Fiche n° 9 – Tableau non trié

Problème

Rechercher, ajouter, supprimer des données non triées dans un tableau d'entiers non triés.

Rechercher

Retourner l'indice d'une donnée trouvée dans un tableau non trié ou -1 si elle n'est pas trouvée.

Nous supposons que le tableau n'est pas rempli, l'algorithme recevra donc une valeur représentant le nombre d'éléments du tableau. Cette valeur est inférieure à la taille du tableau bien sûr.

Données : le tableau à analyser, le nombre d'éléments dans ce tableau (taille logique), la valeur à rechercher

Résultat : la position de l'élément si il est dans le tableau et -1 sinon

```
// Vérifie si un nombre est dans un tableau et donne sa position (-1 sinon)
algorithm indexValue(is↓ : array of integers,
                    nbValues↓ : integer, value↓ : integer) → integer
    integer i
    i = 0
    while i < nbValues AND is[i] ≠ value
        i = i + 1

    if i < nbValues then
        return i
    else
        return -1
```

pseudocode

Ajouter

Ajouter une donnée non encore présente dans le tableau de données non triées

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à ajouter

Résultat : le tableau reçu est modifié en lui ajoutant la valeur si elle n'y était pas déjà

```
// Ajoute un nombre non encore présente dans le isleau.
algorithm add(is $\downarrow\uparrow$  : array of integer,
              nValues $\downarrow\uparrow$  : integer, value $\downarrow$  : integer)
    if nValues < is.length AND indexValue(is, nValues, value)  $\neq$  -1 then
        is[nValues] = value
        nValues = nValues + 1
```

pseudocode

Remarque Si l'algorithme précédent se traduit immédiatement en langage Java, ce n'est pas le cas de celui-ci. Pourquoi ?

Le passage de paramètre se faisant par valeur (cfr. chapitre 10 p. 125), la taille logique du tableau doit être retournée et maintenue à jour par le code appelant. En langage Java, cet algorithme aurait l'allure suivante :

```
public static int add(int[] is, int nValues, int value){
2   if (nValues < is.length()
      && indexValue(is, nValues, value) != -1){
      is[nValues] = value;
5   nValues = nValues + 1;
      }
      return nValues;
8 }
```

java

Supprimer

Supprimer une donnée d'un tableau de données non triées

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à supprimer

Résultat : le tableau reçu est modifié en lui supprimant la valeur

```
// Supprime un nombre donné dans le tableau.
algorithm delete( is $\downarrow\uparrow$  : array of n integer,
                 nValues $\downarrow\uparrow$  : integer, value $\downarrow$  : integer )
    integer index
    index = indexValue(is, nValues, value)
    if index  $\neq$  -1 then
        is[index] = is[nValues-1]
        nValues = nValues - 1
```

pseudocode

Fiche n° 10 – Tableau trié

Problème

Rechercher, ajouter, supprimer des données triées dans un tableau d'entiers triés.

Rechercher

Rechercher la position où a été trouvé l'élément ou la position où il aurait dû être

Données : le tableau à analyser, le nombre d'éléments dans ce tableau, la valeur à rechercher

Résultat : la position où a été trouvée la valeur ou la position où elle aurait dû être

```
// Recherche un élément.
// - index : indique la position où a été trouvée la valeur
// ou la position où elle aurait dû être
algorithm findIndex( is↓ : array of integer, nValues↓ : integer,
                    value↓ : integer, ) → index integer
    integer index = 0
    while index < nValues AND is[index] < value
    |   index = index + 1

    return index
```

pseudocode

Vérifier

Rechercher l'indice d'une donnée trouvée dans un tableau trié ou -1 si elle n'est pas trouvée

Données : le tableau à analyser, le nombre d'éléments dans ce tableau, la valeur à rechercher

Résultat : la position de l'élément si il est dans le tableau et -1 sinon

Cette opération est triviale.

```
// Vérifie si un nombre est dans un tableau d'entiers trié
// et donne sa position (-1 si non présent)
algorithm verify( is↓ : array of integers, nValues↓ : integer,
                                     value↓ : integer ) → integer
    integer index
    boolean isFound
    index = findIndex( is, nValues, value)
    if is[index] == value then
        return index
    else
        return -1
```

pseudocode

Ajouter

Ajouter une donnée non encore présente dans le tableau de données triées.

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à ajouter

Résultat : le tableau reçu est modifié en lui ajoutant la valeur si elle n'y était pas déjà

```
// Ajouter un nombre donné.
algorithm add( is↑↓ : array of integer, nValues↓ : integer,
                                     value↓ : integer ) → integer
    integer index
    boolean isFound
    if verify(is, nValues, value) == -1 then
        index = findIndex(is, nValues, value)
        shiftRight(is, index, nValues)
        is[index] = value
        nValues = nValues + 1
    return nValues

// Décale d'une position à droite les éléments
// entre la position début et fin
algorithm shiftRight( is↑↓ : array of integer, begin↓ : integer, end↓ : integer )
    for i from end to begin by -1
        is[i+1] = is[i]
```

pseudocode

Supprimer

Supprimer une donnée d'un tableau de données triées

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à supprimer

Résultat : le tableau reçu est modifié en lui supprimant la valeur

```
// supprimer l'élément donné
algorithm delete( is↓↑ : array of integer, nValues↓↑ : integer,
                value↓ : integer ) → integer
    integer index
    boolean isFound
    if verify(is, nValues, value) ≠ -1 then
        index = findIndex(is, nValues, value)
        shiftLeft(is, index + 1, nValues)
        nValues = nValues - 1
    return nValues

// Décale d'une position à gauche les éléments
// entre la position début et fin
algorithm shiftLeft( tab↓↑ : array of integers, begin↓ : integer, end↓ : integer )
    for i from begin to end
        is[i-1] = is[i]
```

pseudocode

Fiche n° 11 – Recherche dichotomique

Problème

Trouver rapidement la position d'une valeur donnée dans un tableau **trié** d'entiers. Si la valeur n'est pas présente, on donnera la position où elle aurait du se trouver.

Spécification

Données : le tableau à analyser et la valeur recherchée

Résultat : un booléen indiquant si la valeur a été trouvée et un entier indiquant soit la position où la valeur a été trouvée soit la position où elle aurait du être.

Solution

L'algorithme rapide que nous avons vu est la recherche dichotomique.

```

algorithm dichotomousResearch(
    myArray↓ : array of integers, value↓ : integer, pos↓↑ : integer ) → boolean
    integer rightIndex, leftIndex, medianIndex
    integer candidate
    boolean isFind

    leftIndex = 0
    rightIndex = myArray.length - 1
    isFind = false

    while NON isFind AND leftIndex ≤ rightIndex
        medianIndex = (leftIndex + rightIndex) DIV 2
        candidate = myArray[medianIndex]
        if candidate == value then
            isFind = vrai
        else if candidate < value then
            leftIndex = medianIndex + 1           // on garde la partie droite
        else
            rightIndex = medianIndex - 1          // on garde la partie gauche

    if isFind then
        pos = medianIndex
    else
        pos = leftIndex           // dans le cas où la valeur n'est pas trouvée,
        // on vérifiera que leftIndex donne la valeur où elle pourrait être insérée.

    return isFind
  
```

pseudocode

Fiche n° 12 – Tri d'un tableau

Trier un tableau d'entiers par ordre croissant.

Spécification

Données : le tableau non trié, à trier.

Résultat : ce même tableau trié.

Solution

Le tri par insertion

```
// Trie le tableau reçu en paramètre (via un tri par insertion).
algorithm insertionSort(myArray↓↑ : array of integers)
    integers i, j, value
    for i from 1 to myArray.length - 1
        value = myArray[i]
        // recherche de l'endroit où insérer value dans le
        // sous-tableau trié et décalage simultané des éléments
        j = i - 1
        while j ≥ 0 AND value < myArray[j]
            myArray[j+1] = myArray[j]
            j = j - 1
        myArray[j+1] = value
```

pseudocode

```
/**
 * Tri par insertion.
3  * @param myArray tableau à trier
 */
public static insertionSort(int[] myArray){
6  int j, value;
  for (int i = 1; i < myArray.length; i = i + 1){
    value = myArray[i];
9    j = i - 1;
    while (j >= 0 && value < myArray[j]){
      myArray[j + 1] = myArray[j];
12   j = j - 1;
    }
    myArray[j + 1] = value;
15  }
}
```

java

Le tri par sélection des minima successifs

```
// Trie le tableau reçu en paramètre (via un tri par sélection des minima successifs).
algorithm selectionSort(myArray↓↑ : array of integers)
    integer i, minIndex
    for i from 0 to myArray.length - 2      // i correspond à l'étape      // de
l'algorithme
        minIndex = searchMinIndex( myArray, i, myArray.length - 1 )
        swap( myArray[i], myArray[minIndex] )

// Retourne l'indice du minimum entre les indices début et fin du tableau reçu.
algorithm searchMinIndex( myArray↓ : array of integers,
                        début↓, fin↓ : integers) → integer
    integers minIndex, i
    minIndex = début
    for i from début+1 to fin
        if myArray[i] < myArray[minIndex] then
            minIndex = i

    return minIndex

// Échange le contenu de 2 variables.
algorithm swap(a↓↑, b↓↑ : integers)
    integer aux
    aux = a
    a = b
    b = aux
```

pseudocode

Le tri bulle

```
// Trie le tableau reçu en paramètre (via un tri bulle).  
algorithm bubbleSort(myArray $\downarrow\uparrow$  : array of integers)  
  integers bubbleIndex, i  
  for bubbleIndex from 0 to myArray.length - 2  
    for i from myArray.length - 2 to bubbleIndex by -1  
      if myArray[i] > myArray[i + 1] then  
        swap( myArray[i], myArray[i + 1] )    // voir algorithme précédent
```

pseudocode

Annexe C

Bonnes pratiques

Ce chapitre recense quelques bonnes pratiques d'écriture d'algorithme et de code. Il montre également des manières d'écrire du code peu lisibles et donc à éviter, voire à proscrire. Rappelons que le principal critère qui nous guide dans ce chapitre est la lisibilité et donc la facilité de compréhension à la lecture.

Les bonnes pratiques d'écriture ont également un aspect culturel et dépendant d'un langage. En effet, certaines communautés de programmeurs habitués à un langage adoptent des styles de programmation qui ne sont pas partagés par d'autres programmeurs, habitués à un autre langage. Nous présentons ici des bonnes pratiques d'algorithmique générale et des bonnes pratiques liées au langage Java.

Contenu

C.1	Tester un booléen	233
C.2	Assigner un booléen	234
C.3	Assigner une valeur en fonction d'une condition	235
C.4	Interrompre une boucle <code>for</code>	235
C.5	Plusieurs <code>return</code>	236

C.1 Tester un booléen

Lorsqu'il s'agit de comparer deux nombres dans la condition d'un **if**, nous écrivons assez naturellement : **if** (`nb1<nb2`) ...

Il ne vous viendrait pas à l'idée d'écrire : **if** (`nb1<nb2 == true`) ...

Avec un booléen, les choses sont moins simples. Par exemple, pour exécuter un traitement si la variable `adult` est vraie, certaines personnes programmeuses débutantes écrivent ceci :

```
if adult == true then
  ...
```

*pseudocode*

Cette écriture est correcte mais inutilement lourde. La variable étant déjà un booléen qui vaut vrai ou faux, il suffit d'écrire :

```
if adult then
  ...
```

pseudocode

De même, si le test doit être vrai quand la variable booléenne est fausse, il suffit d'écrire :

```
if NON adult then
  ...
```

pseudocode

C'est une bonne pratique d'utiliser cette deuxième écriture.

C.2 Assigner un booléen

Pour assigner une valeur booléenne à une variable, il n'est pas utile d'utiliser un if-else, une assignation directe fait l'affaire.

Par exemple, si la variable booléenne `isNul` doit valoir `true` si un nombre est égal à 0 et `false` sinon, nous pourrions écrire le bout de code ci-dessous.

```
if nb == 0 then
  isNul = true
else
  isNul = false
```

*pseudocode*

Ce code est correct mais inutilement compliqué. Puisque on assigne `true` si la condition est vraie et `false` si la condition est fausse, il suffit d'assigner la condition. Ce qui donne :

```
isNul = nb == 0
```

pseudocode

Beaucoup plus court et plus lisible n'est-ce-pas ?

C.3 Assigner une valeur en fonction d'une condition

Examinons l'exemple ci-contre qui assigne un prix `price` qui doit être différent en fonction du droit ou non à un tarif réduit.

```
if isDiscount then
    price = 8
else
    price = 12
```

pseudocode

Nous voyons souvent l'extrait de code plus court. Donc mieux

Cette manière d'écrire comporte effectivement (un peu) moins de lignes mais ce critère n'a que très peu d'importance. Cette version n'est pas plus rapide (elle est même plus lente en cas de tarif réduit) et elle est moins lisible car le lecteur croit d'abord que le tarif est toujours de 12 avant de constater qu'il peut être différent.

```
price = 12
if isDiscount then
    price = 8
```



pseudocode

Privilégiez la première version.

C.4 Interrompre une boucle for

Certains développeurs et développeuses débutantes⁶⁹ interrompent anticipativement une boucle `for` en modifiant l'indice. C'est une mauvaise idée.

Dans l'exemple ci-dessous, l'indice `i` reçoit une « grande valeur » pour sortir de la boucle `for`.

```
for i from 1 to n
    ...
    if aCondition then
        i = n+1 // Pour interrompre la boucle
    ...
```



pseudocode

Cette manière de faire est à proscrire absolument et ce, quelle qu'en soit la raison. Ce n'est pas une bonne idée en terme de lisibilité et dans certains langages c'est interdit par le compilateur ou cela ne fait pas ce qui est attendu.

⁶⁹. En essayant d'avoir une écriture plus inclusive, je m'autorise un accord de proximité.

Si vous devez interrompre un parcours répétitif, lorsqu'une condition est remplie, nous insistons pour que vous utilisiez une boucle **while** et l'utilisation d'un booléen pour contrôler la fin.

Par exemple :

```
i = 1
isFinished = false
while i ≤ n AND NON isFinished
    ...
    if aCondition then
        isFinished = false
    ...
    i = i+1
```

pseudocode

Si le test est la première instruction de la boucle et que la condition est courte, on peut aussi se passer de la variable booléenne.

```
i = 1
while i ≤ n AND NON aCondition
    ...
    i = i+1
```

pseudocode

Si vous tenez absolument à interrompre la boucle **for**, vous pouvez utiliser un **return** comme expliqué au point suivant.

C.5 Plusieurs **return**

Plus tôt dans ce cours nous avons énoncé comme règle :

« Un seul **return** par algorithme ».

Cette règle n'est pas partagée par tous les développeurs et développeuses. Il existe un courant qui tolère plusieurs **return** dans certains cas précis.

En voici quelques uns.

Un cas concret est l'arrêt anticipé d'une boucle. Certains écrivent :

```
for i from 1 to n
  if ... then
    return something

return something else
```

pseudocode

Une autre situation est lorsque la valeur à retourner dépend d'un test. On voit alors des bouts de code qui ressemblent à :

```
if ... then
  return quelquechose
else
  return autrechose
```

pseudocode

Ces écritures seront tolérées dans ce cours. Elles sont à utiliser avec parcimonie et ne se justifient que dans des algorithmes assez courts.

Index

- affectation interne, 37
- afficher, 54
- aléatoire, 45
- algorithme, 15
- alternatives, 65
- AND, 41
- api, 159
- appel, 53
- assignation, 37

- binaire, 40
- bloc, 36, 66

- caractère, 156
- chaîne, 155
- chaîne (longueur d'une), 156
- classe, 53, 166
- commentaires, 50
- comparaisons, 41
- comparators, 73
- compilation, 59
- concaténation, 158
- conditionals operators, 73
- constantes, 52
- constructeur, 170
- court-circuit, 43
- création, 115

- débogueur, 38
- déclaration, 36, 115
- décomposer le code, 77
- demander, 55
- DIV, 44
- division entière, 44
- do-while, 98

- efficacité, 47
- encapsulation, 169
- entête, 32
- environnement, 61

- ET, 41
- expression, 37, 40
- expression booléenne, 73

- fonction, 78
- for, 94

- grammaire, 63

- hasard, 45

- ide, 61
- if, 65
- if-else, 68
- implémenter un algorithme, 33
- indentation, 49
- initialisation, 116
- interprétation, 59

- javadoc, 51
- JDK, 62
- jls, 63
- JRE, 62
- jvm, 60

- lisibilité, 47, 49
- littéral, 40

- méthode, 78
- majeur, 144
- mineur, 144
- MOD, 44
- module, 78, 79
- modulo, 44
- Morgan, 42
- mots-clés, 49

- noms, 24
- NON, 41

- opérande, 40

- opérateur, 40
- opérateurs logiques, 41
- OR, 41
- OU, 41

- paramètres, 79
- paramètres effectifs, 88
- paramètres formels, 88
- point d'entrée, 33
- pour, 94
- primitif, 77
- priorité, 40
- procédure, 78
- programme, 16

- référence, 115
- rapidité, 48
- read, 55
- recherche dichotomique, 135
- reference, 77
- return, 32

- selon-que, 74
- si, 65
- si-sinon, 68
- si-sinon-si, 71
- structure, 166
- switch, 74

- tableau, 114, 115
- tableau (indice), 115
- tableau (taille), 115
- tables de vérité, 42
- taille logique, 122
- taille physique, 122
- tant que, 92
- ternaire, 40, 131
- tracer, 38
- tri, 143
- type primitif, 82
- type référence, 82
- types, 25

- unaire, 40

- vérifier un algorithme, 33
- valeur sentinelle, 104
- variable, 24, 35
- variable globale, 36
- variable locale, 36

- while, 92

