



Haute École de Bruxelles
École Supérieure d'Informatique
Bachelor en Informatique

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

DEV 1
Algorithmique
2015

Activité d'apprentissage enseignée par :

<i>M. Codutti</i>	<i>G. Cuvelier</i>	<i>A. Hallal</i>
<i>C. Leruste</i>	<i>E. Levy</i>	<i>N. Pettiaux</i>

Document produit avec L^AT_EX.
Version du 2 septembre 2016.



Ce document est distribué sous licence Creative Commons
Paternité - Partage à l'Identique 2.0 Belgique
(<http://creativecommons.org/licenses/by-sa/2.0/be/>).
Les autorisations au-delà du champ de cette licence
peuvent être demandées à `esi-dev1-list@heb.be`.

Table des matières

I	Introduction aux algorithmes	7
1	Résoudre des problèmes	9
1.1	La notion de problème	9
1.2	Procédure de résolution	10
1.3	Ressources	13
2	Une approche ludique : Code Studio	15
3	Les algorithmes informatiques	17
3.1	Algorithmes et programmes	17
3.2	Les phases d'élaboration d'un programme	19
3.3	Conclusion	19
3.4	Ressources	20
II	Les bases de l'algorithmique	21
4	Spécifier le problème	23
4.1	Déterminer les données et le résultat	23
4.2	Les noms	23
4.3	Les types	24
4.4	Résumé graphique	26
4.5	Exemples numériques	26
4.6	Exercices	27
5	Premiers algorithmes	31
5.1	Un problème simple	31
5.2	Décomposer les calculs	34
5.3	Quelques difficultés liées au calcul	38
5.4	Des algorithmes de qualité	44
5.5	Améliorer la lisibilité d'un algorithme	46
5.6	Interagir avec l'utilisateur	48
6	Une question de choix	51
6.1	Le si	51
6.2	Le si-sinon	53
6.3	Le si-sinon-si	55
6.4	Le selon-que	56
6.5	Exercices de synthèse	58
7	Décomposer le problème	59
7.1	Motivation	59
7.2	Exemple	59
7.3	Les paramètres	60
7.4	La valeur de retour	63
7.5	Résumons	63

7.6 Exercices	66
8 Un travail répétitif	69
8.1 La notion de travail répétitif	69
8.2 Une même instruction, des effets différents	70
8.3 « tant que »	71
8.4 « pour »	73
8.5 « faire – tant que »	75
8.6 Quel type de boucle choisir ?	77
8.7 Acquisition de données multiples	77
8.8 Les suites	80
8.9 Exercices récapitulatifs	81
 III Les tableaux	 85
9 Les tableaux	87
9.1 Utilité des tableaux	87
9.2 Définitions	89
9.3 Déclaration	90
9.4 Utilisation	90
9.5 Initialisation compacte	90
9.6 Tableau et paramètres	91
9.7 Parcours d'un tableau	93
9.8 Taille logique et taille physique	94
9.9 Des tableaux qui ne commencent pas à 0	95
10 Gérer les données dans un tableau	99
10.1 Données non triées	99
10.2 Données triées	101
10.3 La recherche dichotomique	104
10.4 Introduction à la complexité	106
11 Le tri	109
11.1 Motivation	109
11.2 Tri par insertion	111
11.3 Tri par sélection des minima successifs	112
11.4 Tri bulle	113
11.5 Cas particuliers	114
11.6 Références	115
12 Exercices sur les tableaux	117
 IV Compléments	 121
13 Les chaines	123
13.1 Introduction	123
13.2 Longueur	123
13.3 Chaine et caractère	123
13.4 Le contenu d'une chaine	124
13.5 La concaténation	124
13.6 Manipuler les caractères	124
13.7 L'alphabet	125
13.8 Chaine et nombre	125
13.9 Extraction de sous-chaines	126
13.10 Recherche de sous-chaine	126

13.11 Exercices	126
14 Les variables structurées	129
14.1 Le type structuré	129
14.2 Définition d'une structure	129
14.3 Déclaration d'une variable de type structuré	130
14.4 Utilisation des variables de type structuré	130
14.5 Exemple d'algorithme	131
14.6 Remarque sur les champs	131
14.7 Exercices sur les structures	132
V Conclusion	133
15 Exercices récapitulatifs	135
15.1 AEBBCLRS	135
15.2 Serpents et échelles	136
15.3 Les algorithmes en maternelle	138
VI Les annexes	141
A Les fiches	143
Un calcul simple	144
Un calcul complexe	145
Un nombre pair	146
Maximum de deux nombres	147
Passage d'un tableau en paramètre	148
Parcours complet d'un tableau	150
Parcours partiel d'un tableau	151
Maximum dans un tableau	153
Tableau non trié	154
Tableau trié	155
Recherche dichotomique	157
Tri d'un tableau	158
B Bonnes (et mauvaises) pratiques	161
B.1 Tester un booléen	161
B.2 Assigner un booléen	161
B.3 Assigner une valeur en fonction d'une condition	162
B.4 Interrompre une boucle pour	162
B.5 Plusieurs retourner	163
C Le LDA	165
D Aide mémoire	169

Première partie

Introduction aux algorithmes

1	Résoudre des problèmes	9
2	Une approche ludique : Code Studio	15
3	Les algorithmes informatiques	17

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. »¹

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



La notion de problème

Préliminaires : utilité de l’ordinateur

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

En première 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 on souhaite répondre soit **accessible à la raison**.

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,

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

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 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...*

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.

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 oui ou non en fonction de la réalisation d'une condition).

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.

Les opérations bien définies

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

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

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

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

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

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



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

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

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.

Opérations à répéter

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

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

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 jusqu'à ce que la valeur obtenue soit 7*, un humain doté d'intelligence comprend que la condition est impossible à réaliser, mais un robot appliquant cette directive à la lettre lancera le dé perpétuellement.

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.

Ressources

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

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

Chapitre 2

Une approche ludique : Code Studio



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

Sur le site <http://studio.code.org/> nous avons sélectionné pour vous :

- ▷ **L'heure de code** : <http://studio.code.org/hoc/1>.
Un survol des notions fondamentales en une heure au travers de vidéos explicatives et d'exercices interactifs.
- ▷ **Cours d'introduction** : <http://studio.code.org/s/20-hour>.
Un cours de 20 heures destiné aux adolescents. Il reprend et approfondi les éléments effleurés dans « L'heure de code »

Nous vous conseillons de créer un compte sur le site ainsi vous pourrez retenir votre progression et reprendre rapidement votre travail là où vous l'avez interrompu.

Votre professeur va vous guider dans votre apprentissage pendant le cours et vous pourrez approfondir à la maison.

Chapitre 3

Les algorithmes informatiques

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



Algorithmes et programmes

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

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 : ordinogramme, arbre programmatique, pseudo-code ou LDA (langage de description d'algorithmes) que nous allons utiliser dans le cadre de ce cours.

Programme

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



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

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

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).

Exécution d'un programme

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



Comment fonctionne le processeur ?

De façon très simplifiée, on passe par les étapes suivantes :

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

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

Les phases d'élaboration d'un programme

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



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

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

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

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

- ▷ Passer par une phase « algorithmique » permet de séparer deux difficultés : quelle est la marche à suivre ? Et comment l'exprimer dans le langage de programmation choisi ? Le langage que nous allons utiliser en algorithmique est plus souple et plus général que le langage Java par exemple (où il faut être précis au « ; » près).
- ▷ De plus, un algorithme écrit facilite le dialogue dans une équipe de développement. « J'ai écrit un algorithme pour résoudre le problème qui nous occupe. Qu'en pensez-vous ? Pensez-vous qu'il est correct ? Avez-vous une meilleure idée ? ». L'algorithme est plus adapté à la communication car plus lisible.
- ▷ Enfin, si l'algorithme est 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 que pour être directement programmés. Mais qu'est-ce qu'un problème simple ? Cela va évidemment changer tout au long de votre apprentissage. Un problème qui vous paraîtra difficile en début d'année vous paraîtra (enfin, il faut l'espérer !) une évidence en fin d'année.

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

- ▷ essayer 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.

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

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

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

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

Ressources

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

- ▷ Comment mon ordinateur *réfléchit* ? <https://www.youtube.com/watch?v=TIkBcrbzYf0>

Deuxième partie

Les bases de l'algorithmique

4	Spécifier le problème	23
5	Premiers algorithmes	31
6	Une question de choix	51
7	Décomposer le problème	59
8	Un travail répétitif	69

Chapitre 4

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. Il ne s'agit pas d'un travail facile et c'est celui par lequel nous allons commencer.

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 attendu ¹.

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.

Les noms

Pour identifier clairement chaque **donnée** et pouvoir y faire référence dans le futur algorithme nous devons lui attribuer un **nom** ². 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.

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

2. Dans ce cours, on va choisir des noms en Français, mais vous pouvez très bien choisir des noms Anglais si vous vous sentez suffisamment à l'aise avec cette langue.

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

On peut envisager les noms suivants :

- ▷ longueur est probablement le plus approprié.
- ▷ longueurRectangle peut se justifier pour éviter toute ambiguïté avec une autre longueur.
- ▷ long peut être admis si le contexte permet de comprendre immédiatement l'abréviation.
- ▷ l et lg sont à proscrire car pas assez explicites.
- ▷ laLongueurDuRectangle est inutilement long.
- ▷ bidule ou temp ne sont pas de bons choix car ils n'ont aucun lien avec la donnée.

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.

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

On peut envisager le verbe `calculerSurfaceRectangle` ou le nom `surfaceRectangle` (notre préféré). On pourrait aussi simplifier en `surface` s'il est évident qu'on traite des rectangles.

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.

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.

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

entier	pour les nombres entiers
réel	pour les nombres réels
chaîne	pour les chaînes de caractères, les textes (par exemple : "Bonjour", "Bonjour le monde!", "a", "...")
booléen	quand la valeur ne peut être que vrai ou faux

Exemples.

- ▷ Pour la longueur, la largeur et la surface d'un rectangle, on prendra un réel.
- ▷ Pour le nom d'une personne, on choisira la chaîne.
- ▷ Pour l'âge d'une personne, un entier est indiqué.
- ▷ Pour décrire si un étudiant est doubleur ou pas, un booléen 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.

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.

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

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.

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*³. On pourra alors définir et utiliser un type `Date` et il ne sera plus nécessaire de décomposer une date en trois morceaux.

3. L'orienté objet que vous verrez en Java le permet également et même mieux.

Exercice

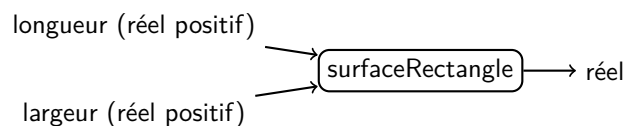
Quel(s) type(s) de données utiliseriez-vous pour représenter

- ▷ 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 ?

Résumé graphique

Toutes les informations déjà collectées sur le problème peuvent être représentées graphiquement.

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



Exemples numériques

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

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 : `surfaceRectangle(3, 2)` donne/vaut 6.

Exercices

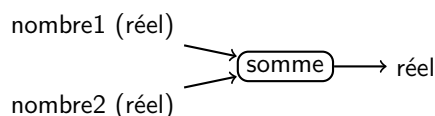
Pour les exercices suivants, nous vous demandons d'imiter la démarche décrite dans ce chapitre, à savoir :

- ▷ 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.⁴ 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.

4. Nous allons de temps en temps fournir des solutions. En algorithmique, il y a souvent **plus qu'une** solution possible. Ce n'est donc pas parce que vous avez trouvé autre chose que c'est mauvais. 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.

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.

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.

Chapitre 5

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, le *Langage de Description d'Algorithmes* (ou *LDA*).



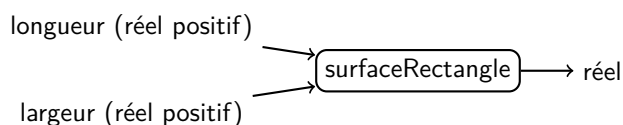
Un problème simple

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 :



Exemples.

▷ `surfaceRectangle(3,2)` donne 6 ;

▷ `surfaceRectangle(3.5,1)` donne 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 largeur¹.

1. 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 qui ne vous sont pas complètement étrangers.

En LDA, la solution s'écrit :

```

algorithme surfaceRectangle(longueur, largeur : réels) → réel
  retourner longueur * largeur
fin algorithme

```

La paire **algorithme-fin algorithme** permet 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 **retourner** 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, pour le moment, naturellement comme vous le feriez en mathématique. La seule différence notable est l'utilisation de ***** pour indiquer une multiplication. Nous donnerons plus de détails plus loin.

Pour indiquer qu'on veut **exécuter** un algorithme (on dit aussi *appeler*) il suffit d'indiquer son nom et les valeurs concrètes à donner aux paramètres. Ainsi, `surfaceRectangle(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.

Vérifier l'algorithme

Lorsque vous avez terminé un exercice, vous le montrez à votre professeur pour qu'il vous dise s'il est correct ou pas. Fort bien ! Mais vous pourriez trouver la réponse tout seul. Il vous suffit d'exécuter l'algorithme avec des exemples numériques et de vérifier que la réponse fournie est correcte. Votre professeur reste indispensable pour :

- ▷ vérifier qu'il fonctionne également dans certains cas particuliers auxquels il est difficile de penser quand on débute ;
- ▷ donner son avis sur la qualité de votre solution c-à-d essentiellement sur sa lisibilité. Nous y reviendrons.

Vous éprouvez souvent des difficultés à tester un algorithme car vous oubliez d'**éteindre votre cerveau**. Il faut agir comme une machine et exécuter **ce qui est écrit** pas ce que vous pensez avoir écrit, 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	✓



Attention : Dans tous les exercices qui suivront, chaque fois qu'on vous demandera d'écrire un algorithme, on attendra de vous : de spécifier le problème, de fournir des exemples, d'écrire l'algorithme et de le vérifier sur vos exemples. Ce n'est que si tous ces éléments sont présents que votre solution pourra être considérée comme complète.

Exercices

Les exercices suivants ont déjà été analysés dans un précédent chapitre et des exemples numériques ont été choisis. Il ne vous reste plus qu'à écrire l'algorithme et à le vérifier pour les exemples choisis.

Vous pouvez vous baser sur la fiche [1 page 144](#) 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.

1 Somme de 2 nombres

Calculer la somme de deux nombres donnés.



Solution. Rappelons ce que nous avons obtenu lors de la phase d'analyse du problème.



Sommer deux nombres est un problème trivial. L'algorithme s'écrit simplement :

```

algorithme somme(nombre1, nombre2 : réels) → réel
|   retourner nombre1 + nombre2
fin algorithme
  
```

Cet exercice est plutôt simple et il est 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	✓

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.

9 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.

10 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.

Décomposer les calculs

Dans les exercices de la section précédente, vous avez écrit quelques longues formules². Pour que cela reste lisible, il serait bon de pouvoir *décomposer* le calcul en étapes. Pour ce faire, nous devons introduire deux nouvelles notions : les *variables locales* et *l'assignation*.

Les variables



Une **variable locale** (ou simplement variable) est une zone mémoire à laquelle on a donné un nom et qui contiendra des valeurs d'un type donné. Elles vont servir à retenir des étapes intermédiaires de calculs.

- ▷ On parle de **variable** car son contenu *peut* changer pendant le déroulement de l'algorithme.
- ▷ On l'appelle **locale** car elle n'est connue et utilisable qu'au sein de l'algorithme où elle est déclarée³.

Pour être utilisable, une variable doit être *déclarée*⁴ 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.

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

longueur, largeur : réels

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

2. Et ce n'est encore rien comparé à ce qui nous attend ;)

3. Certains langages proposent également des variables *globales* qui sont connues dans tout un programme. Nous ne les utiliserons pas dans ce cours ; voilà pourquoi on se contentera de dire "variable".

4. Certains langages permettent d'utiliser des variables sans les déclarer. Ce ne sera pas le cas de Java.

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.

```
nomVariable ← expression
```

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 :

```
denRes ← den1 * den2
cpt ← cpt + 1
moyenne ← (nombre1 + nombre2) / 2
test ← a < b // pour une variable logique
maChaine ← "bonjour"
maChaine ← bonjour // comprenez-vous la différence ?
```

Et d'autres qui ne le sont pas :

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



Remarques

- ▷ Une assignation n'est pas une égalité, une définition.
Ainsi, l'assignation `cpt ← cpt + 1` ne veut pas dire que $cpt = cpt + 1$, 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 où on exécute 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.
- ▷ La valeur affectée à une variable doit être compatible avec son type. Pas question de mettre une chaîne dans une variable booléenne.

Tracer un algorithme

Pour vérifier qu'un algorithme est correct, on sera souvent amené à le **tracer**. Cela consiste à suivre l'évolution des variables et à vérifier qu'elles contiennent bien à tout moment la valeur attendue.

Exemple. Traçons des bouts d'algorithmes.

#	a	b	c
1: a, b, c : entiers	indéfini	indéfini	indéfini
2: a ← 12	12		
3: b ← 5		5	
4: c ← a - b			7
5: a ← a + c	19		
6: b ← a		19	

1: a, b, c : entiers	#	a	b	c
2: a ← 12	1	indéfini	indéfini	indéfini
3: c ← a - b	2	12		
4: d ← c - 2	3			???
	4			???

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

11 Tracer des bouts de code

Suivez l'évolution des variables pour les bouts d'algorithmes donnés.

1: a, b, c : entiers	#	a	b	c
2: a ← 42	1			
3: b ← 24	2			
4: c ← a + b	3			
5: c ← c - 1	4			
6: a ← 2 * b	5			
7: c ← c + 1	6			
	7			

1: a, b, c : entiers	#	a	b	c
2: a ← 2	1			
3: b ← a ³	2			
4: c ← b - a ²	3			
5: a ← \sqrt{c}	4			
6: a ← a / a	5			
	6			

12 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: algorithme <i>vitesseKMH</i> (distanceM, duréeS : réels) → réel
2: distanceKM, duréeH : réels
3: distanceKM ← 1000 * distanceM
4: duréeH ← 3600 * duréeS
5: retourner $\frac{\text{distanceKM}}{\text{duréeH}}$
6: fin algorithme

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.

Exercices de décomposition

Savoir, face à un cas concret, s'il est préférable de décomposer le calcul ou pas, n'est pas toujours évident. La section 5.5 page 46 sur la lisibilité vous apportera des arguments qui permettront de trancher.

Les exercices qui suivent sont suffisamment complexes que pour mériter une décomposition du calcul. Ils ont déjà été analysés dans un précédent chapitre. On vous demande à présent d'en rédiger une solution et de la tracer pour vérifier que le résultat est correct. Vous pouvez vous baser sur la fiche 2 page 145 qui présente un exemple complet.

13 Durée de 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.



Solution. L'analyse du problème aboutit à :



La formule qui lie les trois éléments est :

$$\text{vitesse} = \frac{\text{distance}}{\text{temps}} \quad \text{qu'on peut aussi exprimer} \quad \text{temps} = \frac{\text{distance}}{\text{vitesse}}$$

pour autant que les unités soient compatibles. Dans notre cas, il faut d'abord convertir la distance en mètres, selon la formule :

$$\text{vitesseM} = 1000 * \text{vitesseKM}$$

quelques exemples numériques :

▷ duréeTrajet(1, 1) donne 1000

▷ duréeTrajet(0.5, 0.2) donne 400

L'algorithme s'écrit :

```

1: algorithme duréeTrajet(vitesseMS, distanceKM : réels) → réel
2:   distanceM : réel
3:   distanceM ← 1000 * distanceKM
4:   retourner distanceM / vitesseMS
5: fin algorithme
  
```

Vérifions l'algorithme pour duréeTrajet(1, 1)

#	vitesseMS	distanceKM	distanceM	résultat
1	1	1		
2			indéfini	
3			1000	
4				1000

et pour duréeTrajet(0.5, 0.2)

#	vitesseMS	distanceKM	distanceM	résultat
1	0.5	0.2		
2			indéfini	
3			200	
4				400

14 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.

15 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.

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 pour vous.

Un peu de vocabulaire

Une première difficulté que vous rencontrez généralement est liée au vocabulaire utilisé. Qu'entend-on exactement par un opérateur ? un opérande ? 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.

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.

16 Analyse d'expression

Voici une série d'expressions. On vous demande d'identifier tous les opérateurs et leurs opérandes, d'indiquer si les opérateurs sont unaires ou binaires et d'identifier les littéraux. On vous demande 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).

- ▷ $a+1$
- ▷ $(a+b)*12-4*(-a-b)$
- ▷ $a+(b*12)-4*-a$

Les comparaisons et les assignations de variables booléennes

Si je vous dis que $3 + 1$ est un calcul dont le résultat est 4, un entier vous n'aurez aucun mal à me croire ; cela vous paraît évident. Ce qui l'est peut-être moins c'est que $1 < 3$ 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 (les variables à gauche du \leftarrow sont des variables booléennes) :

<code>positif \leftarrow nb > 0</code>	<code>// positif est mis à vrai si le nb est positif</code>
<code>adulte \leftarrow âge \geq 21</code>	<code>// adulte est vrai si l'âge est 21 ou plus</code>
<code>réussi \leftarrow cote \geq 10</code>	<code>// réussi est mis à vrai si la cote est supérieure ou égale à 10</code>
<code>parfait \leftarrow nbFautes = 0</code>	<code>// c'est parfait si le nombre de fautes est 0</code>

17 Écrire des expressions booléennes

Pour chacune des phrases suivantes, écrivez l'assignation qui lui correspond.

- ▷ La variable booléenne *néгатif* 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.

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
ET	conjonction logique	vrai si les 2 conditions sont vraies
OU	disjonction logique	vrai si au moins une des 2 conditions est vraie

Ce qu'on peut résumer par les tableaux suivants :

a	b	a ET b	a OU b
vrai	vrai	vrai	vrai
vrai	faux	faux	vrai
faux	vrai	faux	vrai
faux	faux	faux	faux

a	NON a
vrai	faux
faux	vrai

On peut les utiliser pour donner une valeur à une variable booléenne. Par exemple :

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

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 ambigus⁵.

Par exemple, ne pas écrire : $\text{tarifRéduit} \leftarrow \text{âge} < 18 \text{ OU } \geq 60$

Loi de De Morgan. Lorsqu'on a une expression complexe faisant intervenir des négations, on peut utiliser la *Loi de De Morgan* pour la simplifier. Cette loi stipule que :

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

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

Par exemple : $\text{tarifRéduit} \leftarrow \text{NON } (18 \leq \text{âge ET } \text{âge} < 60)$
 peut s'écrire aussi : $\text{tarifRéduit} \leftarrow (\text{NON } 18 \leq \text{âge}) \text{ OU } (\text{NON } \text{âge} < 60)$
 ce qui se simplifie en : $\text{tarifRéduit} \leftarrow \text{âge} < 18 \text{ OU } 60 \leq \text{âge}$

Priorités et parenthèses. Lorsqu'on écrit une expression mêlant les opérateurs logiques, on considère que NON est prioritaire sur ET qui l'est sur OU.

Ainsi l'expression : $\text{NON } a \text{ OU } b \text{ ET } c$ doit se comprendre : $(\text{NON } a) \text{ OU } (b \text{ ET } c)$. Vous pouvez toujours ajouter des parenthèses non nécessaires pour vous assurer d'être bien compris.

Évaluation court-circuitée. Les opérateurs ET et OU sont des opérateurs court-circuités. Cela signifie que le calcul s'arrête dès qu'on peut être sûr de la réponse finale. En particulier, si la première partie d'un ET est fausse, on est sûr que le résultat sera faux quelle que soit la suite. Et si la première partie d'un OU est vraie on est sûr que le résultat sera vrai.

Pourquoi un tel comportement ? Cela permet de gagner du temps bien sûr, mais cela permet également d'éviter des erreurs d'exécution.

Exemples.

ok $\leftarrow 1/b < 0.1$	// provoque une erreur et un arrêt de l'algorithme si $b=0$.
ok $\leftarrow b \neq 0 \text{ ET } 1/b < 0.1$	// donne la valeur faux à ok si $b=0$ (court-circuit).
ok $\leftarrow 1/b < 0.1 \text{ ET } b \neq 0$	// provoque une erreur et un arrêt de l'algorithme si $b=0$.

Cette propriété sera abondamment utilisée dans le parcours de tableaux.

18 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.

- ▷ $\text{ok} \leftarrow \text{adulte} = \text{vrai}$
- ▷ $\text{ok} \leftarrow \text{adulte} = \text{faux}$
- ▷ $\text{ok} \leftarrow \text{etudiant} = \text{vrai ET } \text{jeune} = \text{faux}$
- ▷ $\text{ok} \leftarrow \text{NON } (\text{adulte} = \text{vrai}) \text{ ET NON } (\text{adulte} = \text{faux})$
- ▷ $\text{nbA3chiffres} \leftarrow \text{NON } (\text{nb} < 100 \text{ OU } \text{nb} \geq 1000)$

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

19 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.
- ▷ Pour réussir GEN1, il faut au moins 10 dans chacune des AA qui le composent (math, anglais, compa).

La division entière et le reste

La **division entière** consiste à effectuer une division en ne gardant que la partie entière du résultat. Dans ce cours, nous la noterons **DIV**. Dit autrement, **a DIV b** indique combien de fois on peut *mettre* b dans a.



Exemples :

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

Le **reste** de la division entière de a par b est ce qui n'a pas été repris dans la division. On va le noter **a MOD b** et on dira *a modulo b*.



Un exemple vous aidera à comprendre. Imaginons qu'une classe comprend 14 étudiants qu'il faut réunir par 3 dans le cadre d'un travail de groupe. On peut former 4 groupes mais il restera 2 étudiants ne pouvant former un groupe complet. C'est le reste de la division de 14 par 3.

Exemples :

- ▷ 7 MOD 2 vaut 1
- ▷ 6 MOD 6 vaut 0
- ▷ 8 MOD 2 vaut 0
- ▷ 6 MOD 7 vaut 6

20 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 DIV 3 = 4 et 14 MOD 3 = 2.

- ▷ 11 DIV 3 = ____ reste ____
- ▷ 11 DIV ____ = 2 reste 3
- ▷ 3 DIV 11 = ____ reste ____
- ▷ ____ DIV 3 = 3 reste 1

21 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 ?

```

algorithme versPrixRond(prixCentimes : entier) → entier
|   retourner 100 - (prixCentimes MOD 100)
fin algorithme

```

test n°	prixCentimes	réponse correcte	valeur retournée	Correct ?
1	130	70		
2	40	60		
3	99	1		
4	100	0		

Tester la divisibilité

Les deux opérateurs MOD et DIV sont-ils vraiment utiles ? Oui ! Ils vont servir pour tester si un nombre est un multiple d'un autre et pour extraire des chiffres d'un nombre. Commençons par la divisibilité.

Imaginons qu'on veuille tester qu'un nombre est pair. Qu'est-ce qu'un nombre pair ? Un nombre qui est multiple de 2. C'est-à-dire dont le reste de la division par 2 est nul.

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

On peut donc écrire : `pair ← nb MOD 2 = 0`.

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

On voit que les DIV et MOD avec des puissances de 10 permettent de garder les chiffres de droite (MOD) ou d'enlever les chiffres de droite (DIV). Combinés, ils permettent d'extraire n'importe quel chiffre d'un nombre.

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

Le hasard

Il existe de nombreuses applications qui font intervenir le hasard. Par exemple dans les jeux où on doit mélanger des cartes, lancer des dés, faire apparaître des ennemis de façon aléatoire...

Le vrai hasard n'existe pas en algorithmique puisqu'il s'agit de suivre des étapes précises dans un ordre fixé. Pourtant, on peut concevoir des algorithmes qui *simulent* le hasard⁶. À partir d'un nombre donné⁷ (qu'on appelle *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. Nous allons juste supposer qu'on dispose, sans devoir l'écrire, d'un algorithme qui fournit un entier aléatoire.

$$n \text{ (entier)} \longrightarrow \boxed{\text{hasard}} \longrightarrow \text{entier (entre 1 et } n \text{)}$$

Dans nos algorithmes, nous pourrions donc écrire `hasard(n)` pour obtenir un nombre entier aléatoire entre 1 et n inclus.

Exemple.

```
// Simule le lancer d'un dé
algorithme dé() → entier
|   retourner hasard(6)
fin algorithme
```

Cet algorithme `hasard()` peut être utilisé pour générer des nombres appartenant à d'autres intervalles.

6. On parle de pseudo-hasard ou d'algorithmes pseudo-aléatoires.

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

22 Chiffre aléatoire

Écrivez un algorithme qui donne un chiffre (de 0 à 9 donc) aléatoire.

23 Nombre aléatoire entre min et max

Écrivez un algorithme qui donne un nombre entier aléatoire compris entre min et max inclus.

```
algorithme hasard(min, max : entiers) → entier
```

**Exercices récapitulatifs sur les difficultés de calcul**

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.

24 Nombre multiple de 5

Calculer si un nombre entier positif 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) → **multiple5** → 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: algorithme multiple5(nombre : entier) → booléen
2:   retourner nombre MOD 5 = 0
3: fin algorithme
```

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	✓

25 Nombre entier positif se terminant par un 0

Calculer si un nombre donné se termine par un 0.

26 Les centaines

Calculer la partie *centaine* d'un nombre entier positif quelconque.

27 Somme des chiffres

Calculer la somme des chiffres d'un nombre entier positif inférieur à 1000.

28 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 !

29 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.

30 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.

31 Un double au 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).

32 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.

Des algorithmes 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 :

Puisqu'il existe plusieurs algorithmes qui résolvent un problème, lequel préférer ?

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.

L'efficacité

L'**efficacité** désigne le fait que l'algorithme (le programme) résout ⁸ bien le problème donné. C'est un minimum !

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

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 (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 rencontrée⁹ ;
- ▷ 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 d'un algorithme, vous-même ne le comprendrez plus quand vous le relirez quelque temps plus tard !

Comparer la lisibilité de deux algorithmes n'est pas une tâche évidente car c'est une notion subjective. Il faut se demander quelle version va être la plus facilement comprise par la majorité des lecteurs. La section [5.5 page suivante](#) explique ce qui peut être fait pour rendre ses algorithmes plus lisibles.

La rapidité

La **rapidité** indique si l'algorithme (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'*efficacité* qui traite de la gestion économe des ressources. Nous reparlerons de rapidité dans le chapitre consacré à la *complexité* des algorithmes.

La taille

Nous voyons parfois des étudiants contents 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.

Conclusion

Tout ces critères n'ont pas le même poids. Le point le plus important est bien sûr d'écrire un algorithme correct mais ne vous arrêtez pas là ! Demandez-vous s'il n'est pas possible de le retravailler pour améliorer sa lisibilité¹⁰.

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

10. 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.

Améliorer la lisibilité d'un algorithme

On vient de le voir, la lisibilité est une qualité essentielle que doivent avoir nos algorithmes. Qu'est ce qui permet d'améliorer la lisibilité d'un algorithme ?

1. 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 (on parle de *mots-clés*) sont mis en évidence (en gras¹¹).
 - ▷ On écrit une seule instruction 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.

```

algorithme duréeTrajet(vitesseMS, distanceKM : réels) → réel
distanceM : réel
distanceM ← 1000 * distanceKM
retourner distanceM / vitesseMS
Fin algorithme

```



```

algorithme duréeTrajet(vitesseMS, distanceKM : réels) → réel
  distanceM : réel
  distanceM ← 1000 * distanceKM    retourner distanceM / vitesseMS
fin algorithme

```



Il faudra préférer

```

algorithme duréeTrajet(vitesseMS, distanceKM : réels) → réel
  distanceM : réel
  distanceM ← 1000 * distanceKM
  retourner distanceM / vitesseMS
fin algorithme

```



2. 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 qu'on jugera le plus lisible.
 - ▷ On peut introduire des commentaires et/ou des constantes. Deux concepts que nous allons développer maintenant.

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 :

- ▷ Ceux placés **au-dessus** de l'algorithme qui expliquent **ce que fait** l'algorithme et dans quelles **conditions** il fonctionne (les contraintes sur les paramètres). On parle aussi de **documentation**.

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

- ▷ Ceux placés **dans** l'algorithme 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. Voici quelques mauvais commentaires

```
// Exemples de mauvais commentaires
longueur : réel          // La longueur est un réel
somme ← 0               // On initialise la somme à 0
```

Notez qu'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

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

dépourvue de commentaires est bien préférable aux lignes suivantes :

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

Pour résumer :

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

Exemple. 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.
algorithme surfaceRectangle(longueur, largeur : réels) → réel
|   retourner longueur * largeur
fin algorithme
```

33 Commenter la durée du trajet

Commentez l'algorithme qui calcule la durée d'un trajet (exercice 13 page 37).

Constantes

Une **constante** est une information pour laquelle nom, type et valeur sont figés. La liste des constantes utilisées dans un algorithme apparaîtra dans la section déclaration des variables ¹² sous la forme suivante ¹³ :

```
constante PI = 3,1415
constante SEUIL_RÉUSSITE = 12
constante ESI = "École Supérieure d'Informatique"
```

Il est inutile de spécifier leur type, 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.

12. Ou en dehors des algorithmes s'il s'agit d'une constante universelle partagée par plusieurs algorithmes.

13. L'usage est d'utiliser des noms en majuscule.

- ▷ Une plus grande facilité pour modifier le code si la constante vient à changer (modification légale du seuil de réussite par exemple).

34 Utiliser une constante

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

Interagir avec l'utilisateur

Reprenons l'algorithme `surfaceRectangle` 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 ces données ? Et que faire du résultat ?

Tout d'abord, un algorithme peut utiliser (on dit **appeler**) un autre algorithme¹⁴. Pour ce faire, il doit spécifier les valeurs des paramètres ; il peut alors utiliser le résultat. L'appel s'écrit ainsi :

```
surface ← surfaceRectangle(122,3.78)           // On appelle l'algorithme surfaceRectangle
```

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. . .

Afficher un résultat

Si on veut écrire un programme concret (en Java par exemple) qui permet de calculer des surfaces de rectangles, il faudra bien que ce programme communique le résultat à l'utilisateur du programme. On va l'indiquer avec la commande **afficher**. Ce qui donne :

```
surface ← surfaceRectangle(122,3.78)
afficher surface
```

ou, plus simplement :

```
afficher surfaceRectangle(122,3.78)
```

L'instruction **afficher** 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 ici¹⁵.

Demander des valeurs

Le bout d'algorithme qu'on vient d'écrire n'est pas encore très utile puisqu'il calcule toujours la surface du même rectangle. Il serait intéressant de demander à l'utilisateur ce que valent la longueur et la largeur. C'est le but de la commande **demander**.

```
demander longueur
demander largeur
afficher surfaceRectangle(longueur, largeur)
```

14. Cet autre algorithme doit exister *quelque part* : sur la même page, une autre page, un autre document, peu importe. Quand on codera cet algorithme, les contraintes seront plus fortes car il faudra que l'ordinateur trouve cet autre bout de code pour pouvoir l'exécuter.

15. Ce sera bien sûr une question importante quand il s'agira de traduire l'algorithme en un programme.

L'instruction **demander** 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.

On peut combiner les demandes :

```
demander longueur, largeur  
afficher surfaceRectangle(longueur, largeur)
```

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.

Rien n'empêche d'écrire, à partir de là, un algorithme qui interagit avec l'utilisateur et fait appel à l'algorithme qui réalise le calcul. Par exemple :

```
algorithme TestSurface()  
    longueur, largeur : réels  
    demander longueur, largeur  
    afficher surfaceRectangle(longueur, largeur)  
fin algorithme
```

35 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 correspondant en heures-minutes-secondes. Par exemple, si on est 3726 secondes après minuit alors il est 1h2'6".



Chapitre 6

Une question de choix

Vous avez déjà eu l'occasion d'aborder les alternatives lors de votre initiation aux algorithmes sur le site code.org. Par exemple, vous avez indiqué au zombie quelque chose comme : « S'il existe un chemin à gauche alors tourner à gauche ».

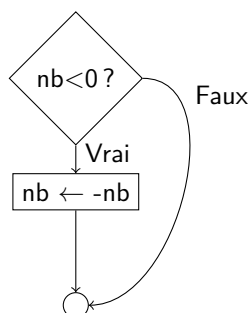
Les **alternatives** permettent de n'exécuter des instructions que si une certaine *condition* est vérifiée. Avec le zombie, vous testiez son environnement ; dans nos algorithmes, vous allez tester les données.

Les algorithmes 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. Procédons par étapes.

Le si

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, on ne sait pas. Et supposons qu'on veuille le rendre positif. On peut tester son signe. S'il est négatif on peut l'inverser. Par contre, s'il est positif, il n'y a rien à faire. Voici comment on peut l'écrire, graphiquement¹ et en LDA :



```
1: si nb < 0 alors
2:   | nb ← -nb
3: fin si
```

Traçons-le dans deux cas différents pour bien illustrer son déroulement.

1. Ce graphique, appelé *organigramme* ou encore *ordinogramme* permet de représenter un algorithme de façon plus visuelle. cf. http://fr.wikipedia.org/wiki/Organigramme_de_programmation.

#	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).



Attention! Vous faites parfois la confusion. 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.

1 Compréhension

Tracez cet algorithme et donnez la valeur de retour.

```

algorithme exerciceA(a, b : entiers) → entier
    c : entier
    c ← 2 * a
    si c > b alors
        c ← c - b
    fin si
    retourner c
fin algorithme

```

▷ exerciceA(2, 5) = ____

▷ exerciceA(4, 1) = ____

2 Simplification d'algorithmes

Voici quelques extraits d'algorithmes corrects du point de vue de la syntaxe mais contenant des lourdeurs d'écriture. Simplifiez-les.

```

si ok = vrai alors
    afficher nombre
fin si

```

```

si ok = faux alors
    afficher nombre
fin si

```

```

si ok = vrai OU ok = faux alors
    afficher nombre
fin si

```

```

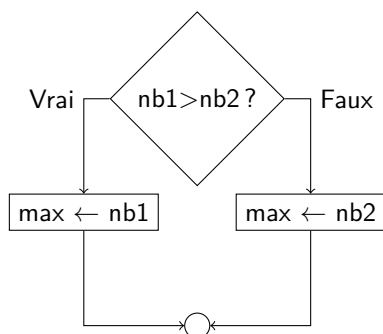
si ok = vrai ET ok = faux alors
    afficher nombre
fin si

```

Le si-sinon

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. Supposons qu'on veuille déterminer le maximum de deux nombres, c'est-à-dire la plus grande des deux valeurs. Dans la solution, il y a 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: si nb1 > nb2 alors
2:   max ← nb1
3: sinon
4:   max ← nb2
5: fin si
  
```

Traçons-le dans différentes situations.

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

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

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

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

3 Compréhension

Tracez ces algorithmes et donnez la valeur de retour.

```

algorithme exerciceB(b, a : entiers) → entier
  c : entier
  si a > b alors
    c ← a DIV b
  sinon
    c ← b MOD a
  fin si
  retourner c
fin algorithme
  
```

▷ exerciceB(2, 3) = ____

▷ exerciceB(4, 1) = ____

```

algorithme exerciceC(x1, x2 : entiers) → entier
    ok : booléen
    ok ← x1 > x2
    si ok alors
        ok ← ok ET x1 = 4
    sinon
        ok ← ok OU x2 = 3
    fin si
    si ok alors
        x1 ← x1 * 1000
    fin si
    retourner x1 + x2
fin algorithme

```

▷ *exerciceC*(2, 3) = ____

▷ *exerciceC*(4, 1) = ____

4 Simplification d'algorithmes

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

```

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

```

```

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

```

5 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 147](#).

6 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 ?

7 Fonction de Syracuse



Écrire un algorithme qui, étant donné un entier n quelconque, retourne le résultat de la fonction $f(n) = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n + 1 & \text{si } n \text{ est impair} \end{cases}$

8 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.

Le si-sinon-si

Avec cette construction, on peut indiquer à un endroit de l'algorithme plus que 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.

Ici, lorsqu'on va examiner le nombre, trois chemins vont s'offrir à nous.



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
	0	indéfini	
1			faux
3			faux
6		"négatif"	

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.

```

si nb>0 alors
  | signe ← "positif"
sinon
  | si nb=0 alors
  |   | signe ← "nul"
  | sinon
  |   | signe ← "négatif"
  | fin si
fin si
  
```

- ▷ 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.

```

si quantité<10 alors
  | prixUnitaire ← 10
sinon si quantité<100 alors // On sait que ce n'est pas <10 ; inutile de le tester
  | prixUnitaire ← 8
sinon
  | prixUnitaire ← 6
fin si
  
```

9 Maximum de 3 nombres

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

10 Le signe

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

11 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.

12 Dés identiques

Écrire un algorithme qui lance trois dés et indique si on a obtenu 3 dés de valeur identique, 2 ou aucun.

13 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.

Le selon-que

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

On peut écrire une solution avec un **si-sinon-si** mais le **selon-que** est plus lisible.

```
selon que numéroJour vaut
1: nomJour ← "lundi"
2: nomJour ← "mardi"
3: nomJour ← "mercredi"
4: nomJour ← "jeudi"
5: nomJour ← "vendredi"
6: nomJour ← "samedi"
7: nomJour ← "dimanche"
fin selon que
```


remplace avantageusement

```

si numéroJour=1 alors
|   nomJour ← "lundi"
sinon si numéroJour=2 alors
|   nomJour ← "mardi"
sinon si numéroJour=3 alors
|   nomJour ← "mercredi"
sinon si numéroJour=4 alors
|   nomJour ← "jeudi"
sinon si numéroJour=5 alors
|   nomJour ← "vendredi"
sinon si numéroJour=6 alors
|   nomJour ← "samedi"
sinon
|   nomJour ← "dimanche"
fin si

```



Remarques.

- ▷ On peut spécifier plusieurs valeurs pour un cas donné.
- ▷ On peut mettre un cas **défaut** qui sera exécuté si la valeur n'est pas reprise par ailleurs.

La syntaxe générale est :

```

selon que expression vaut
|   liste1 de valeurs séparées par des virgules :
|   Instructions
|   liste2 de valeurs séparées par des virgules :
|   Instructions
|   ...
|   listek de valeurs séparées par des virgules :
|   Instructions
autres :
|   Instructions
fin selon que

```

14 Numéro du jour

Écrire un algorithme qui retourne le numéro du jour de la semaine reçu en paramètre (1 pour "lundi", 2 pour "mardi"...).



15 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.

16 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.

Exercices de synthèse

Dans les exercices qui suivent, à vous de déterminer si une instruction de choix est nécessaire et laquelle est la plus adaptée.

17 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.

18 Réussir GEN1



l'UE (Unité d'enseignement) GEN1 est composée de trois AA (activité d'apprentissage) : Mathématique, Communication anglophone et Comptabilité². Pour réussir cette unité d'enseignement, il faut que la cote attribuée à chaque AA soit supérieure ou égale à 50%. Si c'est le cas, la cote attribuée à l'UE est une moyenne **pondérée** des trois cotes d'AA (avec la pondération 6 pour Mathématique et 2 pour les autres AA).

Écrire un algorithme qui reçoit les 3 cotes (sur 20) d'AA d'un étudiant pour l'UE GEN1 et qui **affiche** un message indiquant si l'étudiant a réussi ou pas cette UE. S'il a réussi, l'algorithme affiche également la cote d'UE (sur 20).

19 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 ?

20 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.

21 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.

2. Sans parler de Méthodologie qui ne donne pas lieu à une évaluation.

Chapitre 7

Décomposer le problème

Motivation

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 gros 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, on 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, dans un algorithme, on appelle un autre algorithme pour résoudre un sous-problème, le lecteur 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.

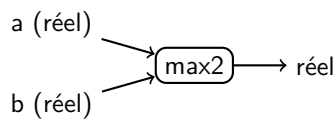
Un algorithme qui résout une partie de problème est parfois appelé **fonction**, **procédure**, **méthode** ou encore **module** en fonction du langage et du contexte. Il y a quelques nuances mais elles importent peu ici.

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.



```

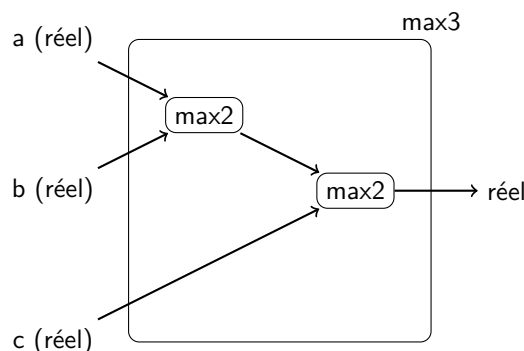
algorithme max2(a : réel, b : réel) → réel
    max : réel
    si a > b alors
        max ← a
    sinon
        max ← b
    fin si
    retourner max
fin algorithme

```

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.

qu'on peut illustrer ainsi :



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*¹ 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 :

```

algorithme max3(a : réel, b : réel, c : réel) → réel
    maxab, max : réels
    maxab ← max2(a,b)
    max ← max2(maxab,c)
    retourner max
fin algorithme

```

qu'on peut encore simplifier en :

```

algorithme max3(a,b,c : réels) → réel
    retourner max2( max2(a,b) ,c)
fin algorithme

```

Les paramètres

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 pratique, on peut rencontrer trois sortes de paramètres.

1. Cette approche serait fastidieuse, engendrerait de nombreuses erreurs lors du recopiage et serait difficile à lire.

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.

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

```
// Code appelant
monAlgo(expr)
```

```
// Code appelé
algorithme monAlgo(par $\downarrow$  : entier)
...
```

C'est comme si l'algorithme `monAlgo` commençait par l'affectation $\text{par} \leftarrow \text{expr}$.

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

```
1: algorithme test()                                     // Code appelant
2:   max : réel
3:   max  $\leftarrow$  max3(3, 2, 5)
4:   afficher max
5: fin algorithme
6:
7: algorithme max3(a $\downarrow$ , b $\downarrow$ , c $\downarrow$  : réels)  $\rightarrow$  réel // Code appelé
8:   maxab, max : réels
9:   maxab  $\leftarrow$  max2(a,b)
10:  max  $\leftarrow$  max2(maxab,c)
11:  retourner max
12: fin algorithme
```

Traçons son exécution.

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

Note : 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`.

Le paramètre en sortie

Le paramètre en **sortie** correspond à un résultat de l'algorithme. Avec la notation que nous utilisons, un algorithme ne peut retourner qu'une seule valeur ce qui est parfois une contrainte trop forte. Les paramètres en sortie vont permettre à l'algorithme de fournir plusieurs réponses. On fera suivre le nom du paramètre d'une flèche vers le haut (\uparrow) pour rappeler son rôle. Un tel paramètre n'aura pas de valeur au début de l'algorithme mais s'en verra attribuer une par l'algorithme.

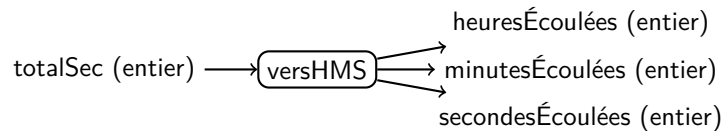
Lors de l'appel, on fournit une **variable** qui recevra la valeur finale du paramètre. Voici un cas général de paramètre en sortie.

```
// Code appelant
monAlgo(variable)
```

```
// Code appelé
algorithme monAlgo(par $\uparrow$  : entier)
...
```

Il n'y a **pas de retourner** puisque les résultats sont en paramètres de sortie et pas comme valeur *retournée*. C'est comme si, à la fin de l'algorithme appelé, on avait l'assignation : $\text{variable} \leftarrow \text{par}$.

Exemple. On peut envisager un algorithme qui reçoit une durée exprimée en seconde et fournisse trois paramètres en sortie correspondant à cette même durée exprimée en heures, minutes et secondes. En voici le schéma et la solution :



Voici une solution et un appel possible.

```

1: algorithme versHMS(totalSec↓, heuresÉcoulées↑, minutesÉcoulées↑, secondesÉcoulées↑ :
2:                                     entiers)
3:   heuresÉcoulées ← totalSec DIV (60*60)
4:   minutesÉcoulées ← totalSec MOD (60*60) DIV 60
5:   secondesÉcoulées ← totalSec MOD 60
6: fin algorithme
7:
8: algorithme test()
9:   heure, minute, seconde : entiers
10:  versHMS(65536, heure, minute, seconde)
11:  afficher heure, minute, seconde
12: fin algorithme
  
```

Traçons-le.

#	test			versHMS			
	heure	minute	seconde	totalSec	heuresEcoulees	minutesEcoulees	secondesEcoulees
9	indéfini	indéfini	indéfini				
10, 1				65536	indéfini	indéfini	indéfini
3					18		
4						12	
5							16
6, 10	18	12	16				

Le paramètre en entrée-sortie

Le paramètre en **entrée-sortie** correspond à une situation mixte. Il est à la fois une donnée et un résultat de l'algorithme. Cela signifie que l'algorithme a pour but de le modifier. Un tel paramètre sera suivi d'une double flèche ($\downarrow\uparrow$).

Lors de l'appel, on fournit **une variable**. 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.

```
// Code appelant
monAlgo(variable)
```

```
// Code appelé
algorithme monAlgo(par↓↑ : entier)
...
```

C'est comme si, dans le code appelé, on avait une première ligne pour donner sa valeur au paramètre ($\text{par} \leftarrow \text{variable}$) et une dernière ligne pour effectuer l'assignation opposée ($\text{variable} \leftarrow \text{par}$). Il n'y a pas de **retourner**.

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

nb (réel) \longleftrightarrow valAbsolue

```

1: algorithme valAbsolue(nb,  $\downarrow \uparrow$  : réel)
2:   si nb < 0 alors
3:     |   nb  $\leftarrow$  -nb
4:   fin si
5: fin algorithme
6:
7: algorithme test()
8:   température : réel
9:   température  $\leftarrow$  -12.5
10:  valAbsolue(température)
11:  afficher température
12: fin algorithme

```

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		

La valeur de retour

Une valeur de retour est toujours possible, mais jamais obligatoire, quelles que soient les sortes de paramètres. Ainsi, on peut imaginer un algorithme qui possède un paramètre en sortie **et** qui retourne également une valeur.

Attention ! 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. Ainsi, les utilisations suivantes de l'algorithme `valAbsolue`, décrit plus haut, sont incorrectes.

```

afficher valAbsolue(température) + 1
tempAbsolue  $\leftarrow$  valAbsolue(température)

```



Si un algorithme possède une seule valeur en sortie, on préférera toujours une valeur en retour à un paramètre en sortie ; le code appelant sera plus lisible.

Résumons

Reprenons tout ce qu'on vient de voir avec un exemple d'algorithme qui possède tous les types de paramètres.

```

1: algorithme testDivision()
2:   cptAppels, nb1, nb2, quotient, resteDiv : entiers
3:   cptAppels ← 0
4:   nb1 ← 5
5:   nb2 ← 3
6:   quotient ← division(cptAppels, nb1, nb2, resteDiv)
7:   afficher quotient
8:   afficher resteDiv
9:   afficher cptAppels
10:  nb1 ← 7
11:  nb2 ← 9
12:  quotient ← division(cptAppels, nb1, nb2, resteDiv)
13:  afficher quotient
14:  afficher resteDiv
15:  afficher cptAppels
16: fin algorithme
17:
18: algorithme division(compteur↓↑ : entier, dividende↓ : entier, diviseur↓ : entier, reste↑ : entier)
   → entier
19:   compteur ← cptAppels
20:   dividende ← nb1
21:   diviseur ← nb2
22:
23:   // Le code proprement dit de l'algorithme
24:   quotient : entier
25:   compteur ← compteur + 1
26:   quotient ← dividende DIV diviseur
27:   reste ← dividende MOD diviseur
28:
29:   cptAppels ← compteur
30:   resteDiv ← reste
31:   retourner quotient
32: fin algorithme

```

Traçons-le.

#	testDivision					division				
	cptAppels	nb1	nb2	quotient	resteDiv	compteur	dividende	diviseur	reste	quotient
3	0									
4		5								
5			3							
6, 18						0	5	3		
25						1				
26										1
27									2	
32, 6	1			1	2					
10		7								
11			9							
12, 18						1	7	9		
25						2				
26										0
27									7	
32, 6	2			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 ↓ et ↓↑ en début d'algorithme et pour donner

des valeurs aux paramètres effectifs \uparrow et \downarrow 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.

Les figures 7.1, 7.2 et 7.3 résument de manière graphique les trois types de passage de paramètres.

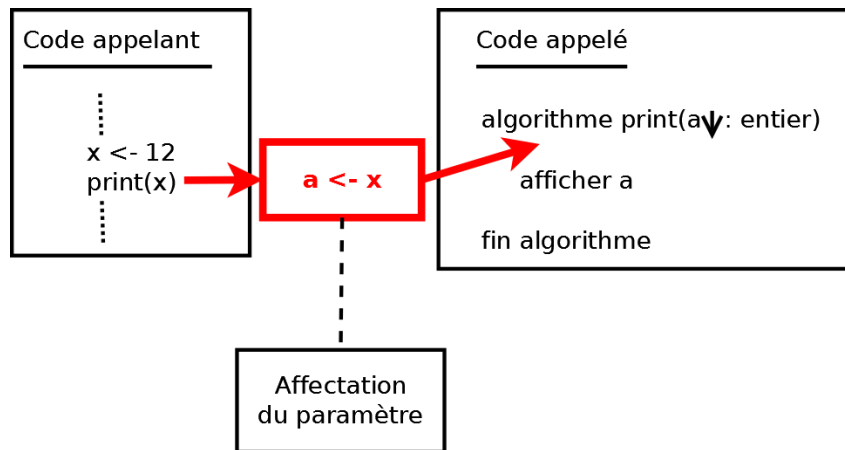


FIGURE 7.1 – Paramètre entrant



FIGURE 7.2 – Paramètre sortant

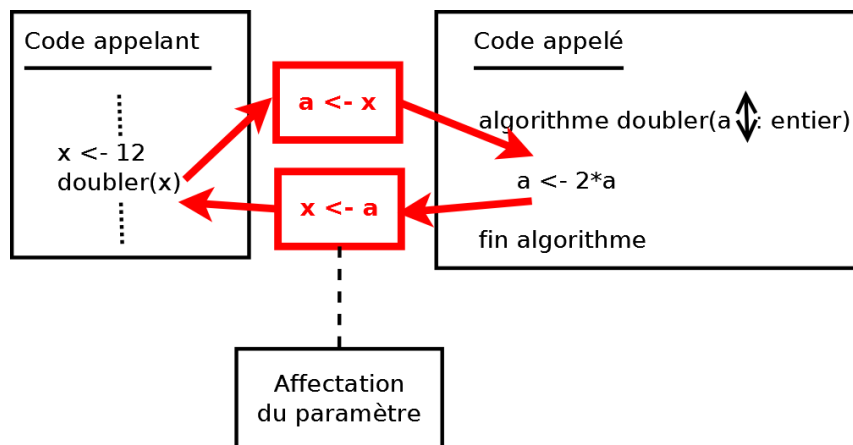


FIGURE 7.3 – Paramètre entrant-sortant

Exercices

1 Tracer des algorithmes

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

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

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

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

algorithme soustraction(a↓, b↓, c↑ : entiers)
  c ← a - b
fin algorithme
```

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

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

```

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

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

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

```

2 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 ?

```

algorithme PGCD(a↓, b↓ : entiers) → entier

```

```

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

```

3 Maximum de 4 nombres

Écrivez un algorithme qui calcule le maximum de 4 nombres.

4 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.

5 Réussir GEN1

Reprenons l'exercice 18 page 58. Cette fois-ci on ne veut rien afficher mais fournir deux résultats : un booléen indiquant si l'étudiant a réussi ou pas et un entier indiquant sa cote (qui n'a de sens que s'il a réussi).

6 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".

7 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 32 page 44](#)).

8 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.

9 Généraliser un algorithme

Dans l'exercice [24 page 43](#), 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 ?

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.



Attention ! D'expérience, nous savons que ce chapitre est difficile à appréhender. Beaucoup d'entre vous perdent pied ici. Accrochez-vous, faites bien tous les exercices proposés, montrez vos solutions à votre professeur et demandez de l'aide dès que vous vous sentez perdu !



La notion de travail répétitif

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

- ▷ le travail à répéter ;
- ▷ une indication qui permet de savoir quand s'arrêter.

Examinons quelques exemples pour fixer notre propos.

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

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

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

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

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

- ▷ La tâche à répéter est : « afficher un nombre ».
- ▷ On indique qu'on doit le faire pour tous les nombres de 1 à 100. On doit donc commencer avec 1, passer à chaque fois au nombre suivant et s'arrêter après avoir affiché le nombre 100.

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, on traite un dossier mais à chaque fois un différent ; on affiche un nombre 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.

Exemple - Afficher les nombres de 1 à 5

Si on veut un algorithme qui affiche les nombres de 1 à 5 sans utiliser de boucle, on pourrait écrire :

```
afficher 1
afficher 2
afficher 3
afficher 4
afficher 5
```

Ces cinq instructions sont proches mais pas tout-à-fait identiques. En l'état, on ne peut pas encore en faire une boucle¹ ; il va falloir ruser. On peut obtenir le même résultat avec l'algorithme suivant :

```
nb ← 1
afficher nb
nb ← 2
afficher nb
nb ← 3
afficher nb
nb ← 4
afficher nb
nb ← 5
afficher nb
```

ou encore

```
1: nb ← 1
2: afficher nb
3: nb ← nb + 1
4: afficher nb
5: nb ← nb + 1
6: afficher nb
7: nb ← nb + 1
8: afficher nb
9: nb ← nb + 1
10: afficher nb
11: nb ← nb + 1
```

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 :

```
afficher nb
nb ← nb + 1
```

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 une².

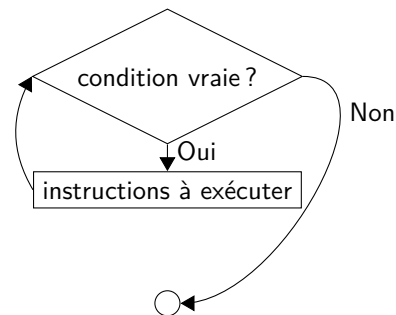
1. Vous vous dites peut-être que ce code est simple ; inutile d'en faire une boucle. Ce n'est qu'un exemple. Que feriez-vous s'il fallait afficher les nombres de 1 à 1000 ?

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

« tant que »

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

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



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

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 :

```
afficher nb
nb ← nb + 1
```

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: algorithme compteur5()
2:   nb : entier
3:   nb ← 1
4:   tant que nb ≤ 5 faire
5:     afficher nb
6:     nb ← nb + 1
7:   fin tant que
8:   afficher "nb vaut ", nb
9: fin algorithme
```

#	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



Exemple - Généralisation à n nombres

On peut généraliser l'exemple précédent en affichant tous les nombres de 1 à n où n est une donnée de l'algorithme.

```

algorithme compteur(n : entier)
  nb : entier
  nb ← 1
  tant que nb ≤ n faire
    afficher nb
    nb ← nb + 1
  fin tant que
fin algorithme

```

Exercices

1 Compréhension d'algorithmes

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

```

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

```

```

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

```

```

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

```

2 Afficher des nombres



En utilisant un **tant que**, écrire un algorithme qui reçoit un entier n positif et affiche

- les nombres de 1 à n ;
- les nombres de 1 à n en ordre décroissant ;
- les nombres impairs de 1 à n ;
- les nombres de $-n$ à n ;
- les multiples de 5 de 1 à n ;
- les multiples de n de 1 à 100.

« pour »

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

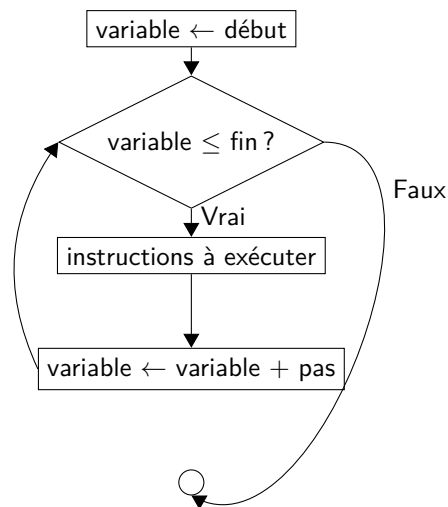
```
pour variable de début à fin par pas faire
  séquence d'instructions à exécuter
fin pour
```

Dans ce type de structure, **début**, **fin** et **pas** peuvent être des constantes, des variables ou des expressions 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, on va considérer qu'il n'est pas nécessaire de la déclarer et qu'elle n'est pas utilisable en dehors de la boucle³.



Exemples

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

```
1: algorithme compterJusque5()
2:   // par défaut le pas est de 1
3:   pour nb de 1 à 5 faire
4:     afficher nb
5:   fin pour
6:   afficher "nb n'existe plus"
7: fin algorithme
```

#	nb	condition	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



Si on veut généraliser l'affichage à n nombres, on a :

```
algorithme compterJusqueN(n : entier)
  pour nb de 1 à n faire
    afficher nb
  fin pour
fin algorithme
```

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

Un pas négatif

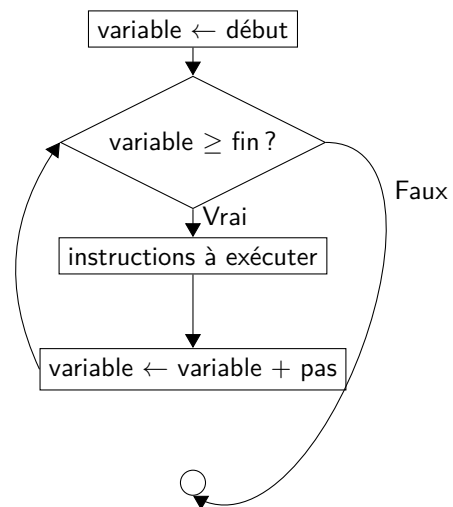
Le pas est parfois négatif, dans le cas d'un compte à rebours, par exemple. Dans ce cas, la boucle s'arrête lorsque la variable prend une valeur plus petite que la valeur de fin (cf. le test dans l'organigramme ci-contre).

Exemple : Compte à rebours à partir de n .

```

algorithme compterÀReboursDécroissant( $n$  : entier)
  pour nb de  $n$  à 1 par -1 faire
    afficher nb
  fin pour
  afficher "Partez !"
fin algorithme

```



Cohérence

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

Exemples :

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

Modification des variables de contrôle



Il est important de ne pas modifier dans la séquence d'instructions une des variables de contrôle **début**, **fin** ou **pas** ! Il est aussi fortement déconseillé de modifier « manuellement » la variable de contrôle au sein de la boucle **pour**. Il ne faut pas l'initialiser en début de boucle, et ne pas s'occuper de sa modification, l'instruction $i \leftarrow i + \text{pas}$ étant automatique et implicite à chaque étape de la boucle.

Exemple – Afficher uniquement les nombres pairs

Cette fois-ci on affiche uniquement les nombres **pairs** jusqu'à la limite n .

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

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



```

algorithme afficherPair( $n$  : entier)
  pour nb de 2 à  $n$  par 2 faire
    afficher nb
  fin pour
fin algorithme

```

La section sur les suites proposera d'autres solutions pour ce problème.

Exercices

3 Compréhension d'algorithmes

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

```

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

```

```

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

```

4 Afficher des nombres

Reprenons un exercice déjà donné avec le **tant que**. En utilisant un **pour**, 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.

« faire – tant que »

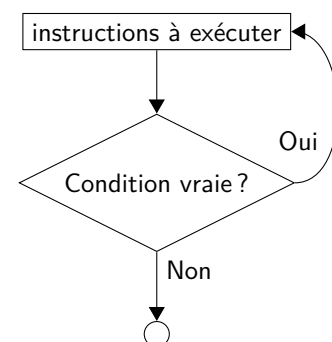
Cette structure est très proche du «faire - tant que » à ceci près que le test est fait à la fin et pas au début⁴. La tâche est donc toujours exécutée au moins une fois.

```

faire
  séquence d'instructions à exécuter
tant que condition

```

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.



4. 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.

Exemple

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



```

1: algorithme compteur5()
2:   nb : entier
3:   nb ← 1
4:   faire
5:     afficher nb
6:     nb ← nb + 1
7:   tant que nb ≤ 5
8:     afficher "nb vaut ", nb
9:   fin algorithme

```

#	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

Exercices

5 Compréhension d’algorithmes

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

```

algorithme boucle4()
  pair, grand : booléens
  p, x : entiers
  x ← 1
  p ← 1
  faire
    p ← 2 * p
    x ← x + p
    pair ← x MOD 2 = 0
    grand ← x > 15
  tant que NON pair ET NON grand
    afficher x
fin algorithme

```

6 Afficher des nombres



Reprenons un exercice déjà fait avec le **tant que** et le **pour** en utilisant cette fois un **faire tant que**. Écrire un algorithme qui reçoit un entier n positif et affiche

- les nombres de 1 à n ;
- les nombres de 1 à n en ordre décroissant ;
- les nombres de $-n$ à n ;
- les multiples de 5 de 1 à n ;
- les multiples de n de 1 à 100.

Quel type de boucle choisir ?

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



Acquisition de données multiples

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, on va devoir stocker chaque valeur donnée par l'utilisateur dans une seule et même variable et la traiter avant de passer à la suivante. Prenons un exemple concret pour mieux comprendre.

On veut pouvoir calculer (et retourner) la somme d'une série de nombres donnés par l'utilisateur.

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

Variante 1 : nombre de valeurs 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.
algorithme sommeNombres() → entier                                // Variante 1
    nbValeurs : entier                                              // nb de valeurs à additionner
    valeur : entier                                                // un des termes de l'addition
    somme : entier                                                  // la somme
    somme ← 0                                                       // la somme se construit petit à petit. 0 au départ
    demander nbValeurs
    pour i de 1 à nbValeurs faire
        demander valeur
        somme ← somme + valeur
    fin pour
    retourner somme
fin algorithme
```

7 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.

Variante 2 : stop ou encore



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

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

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

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

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

8 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.

Variante 3 : valeur sentinelle

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



Ici, on se base sur la valeur entrée pour décider si on continue ou pas. Il faut donc **toujours** effectuer un test après une lecture de valeur. C'est pour cela qu'il faut effectuer une lecture avant 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.
algorithme sommeNombresPositifs() → entier // Variante 3
    valeur : entier // un des termes de l'addition
    somme : entier // la somme
    somme ← 0
    demander valeur
    tant que valeur ≠ -1 faire
        somme ← somme + valeur
        demander valeur // remarquer l'endroit où on lit une valeur.
    fin tant que
    retourner somme
fin algorithme
```

9 Choix de la valeur sentinelle

Quelle valeur sentinelle prendrait-on pour additionner une série de cotes d'interrogations ? Une série de températures ?

10 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.



11 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.

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 **pour** 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 vont avec. Elles pourront se généraliser à beaucoup d'autres exemples.

Exemple - Afficher les carrés



On veut afficher les n premiers nombres carrés parfaits : 1, 4, 9, 16, 25...

Si on vous demande : "Quel est le 7^e nombre à afficher?". Vous répondrez : "Facile! C'est 7², 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 :

```

algorithme suiteCarrés(n : entier)
  pour i de 1 à n faire
    afficher  $i^2$ 
  fin pour
fin algorithme

```

Dans cette solution, la variable de contrôle compte simplement le nombre d'itérations. On calcule le nombre à afficher en fonction cette variable de contrôle (ici le carré convient). Par une vieille habitude des programmeurs⁵, une variable de contrôle qui se contente de compter les passages dans la boucle est souvent nommée i . On l'appelle aussi « itérateur ».

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

Exemple - 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 on peut 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. Une solution élégante et facilement adaptable à d'autres situations est :

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


```

algorithme suite(n : entier)
    val : entier
    val ← 1re valeur à afficher
    pour i de 1 à n faire
        afficher val
        val ← la valeur suivante calculée à partir de la valeur courante
    fin pour
fin algorithme

```

qui, dans notre exemple précis, devient :

```

// affiche la suite 1 2 4 7 11 16 ...
algorithme suite(n : entier)
    val : entier
    val ← 1
    pour i de 1 à n faire
        afficher val
        val ← val + i
    fin pour
fin algorithme

```



12 Suites

Écrire les algorithmes qui affichent les n premiers termes des suites suivantes. À vous de voir quel est le canevas de solution le plus adapté.



- 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...

Exercices récapitulatifs

Pour tous ces exercices, nous vous donnons peu d'indications sur la solution à mettre en œuvre. À vous de jouer...

13 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.



```

algorithme lireNb(n : entier) → entier

```

Solution. Si la valeur donnée par l'utilisateur était toujours correcte, il suffirait d'écrire :

```

algorithme lireNb(n : entier) → entier
    val : entier
    demander val
    retourner val
fin algorithme

```

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 :

```

algorithme lireNb(n : entier) → entier
    val : entier
    demander val
    tant que val < 1 OU val > n faire
        afficher "valeur incorrecte"
        demander val
    fin tant que
    retourner val
fin algorithme

```

La solution qu'on obtient est proche d'une lecture répétée avec valeur sentinelle. Dans ce cas, la valeur sentinelle est toute valeur correcte.

14 Lancé de dés



Écrire un algorithme qui lance n fois un dé et compte le nombre de fois qu'une certaine valeur est obtenue.

```

algorithme lancerDé(n : entier, val : entier) → entier

```

15 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, $0! = 1$.

16 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.

17 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

```

18 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.

19 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.

20 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.

**21 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 $1 + 3 + 3 + 0 + 4 + 5 = 16$.

**22 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 $28 = 1 + 2 + 4 + 7 + 14$.

**23 Décomposition en facteurs premiers**

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

24 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.

**25 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.

**26 Jeu de la fourchette**

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



Troisième partie

Les tableaux

9 Les tableaux	87
10 Gérer les données dans un tableau	99
11 Le tri	109
12 Exercices sur les tableaux	117

Chapitre 9

Les tableaux

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



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.
algorithme statistiquesVentesSansTableau()

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

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

    afficher "Introduisez le numéro du produit : "
    demander numéroProduit

    tant que numéroProduit ≠ -1 faire

        afficher "Introduisez la quantité vendue : "
        demander quantité

        selon que numéroProduit vaut 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
        fin selon que

        afficher "Introduisez le numéro du produit : "
        demander numéroProduit

    fin tant que

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

fin algorithme

```

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

Au lieu d'avoir à manier dix compteurs distincts (`cpt0`, `cpt1`, etc.), nous allons envisager une seule « grande » variable `cpt` compartimentée en dix « cases » ou « sous-variables » (qu'on appelle aussi les « éléments » du tableau). Elles se distingueront les unes des autres par un numéro (on dit 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 avec tableau.

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

    cpt : tableau de 10 entiers
    i, numéroProduit, quantité : entiers

    pour i de 0 à 9 faire
        cpt[i] ← 0
    fin pour

    afficher "Introduisez le numéro du produit : "
    demander numéroProduit

    tant que numéroProduit ≠ -1 faire

        afficher "Introduisez la quantité vendue : "
        demander quantité

        cpt[numéroProduit] ← cpt[numéroProduit] + quantité

        afficher "Introduisez le numéro du produit : "
        demander numéroProduit

    fin tant que

    pour i de 0 à 9 faire
        afficher "quantité vendue de produit ", i, " : ", cpt[i]
    fin pour

fin algorithme
```

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.

Déclaration



Pour déclarer un tableau, on écrit :

```
nomTableau : tableau de taille TypeElément
```

où *taille* est le nombre d'éléments et *TypeElément* est le type des éléments que l'on trouvera dans le tableau. Tous les types sont permis mais tous les éléments sont du même type.

Utilisation

Un élément (une case) du tableau peut être accédé via la notation

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

La première case du tableau porte l'indice 0¹, la dernière l'indice *taille*-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 : `tabEntiers : tableau de 100 entiers` il est interdit d'utiliser `tabEntiers[-1]` ou `tabEntiers[100]`.

1 Déclarer et initialiser

Écrire un algorithme qui déclare un tableau de 100 chaînes et met votre nom dans la 3^e case du tableau.

2 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*.

3 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.

4 Compréhension

Expliquez la différence entre `tab[i] ← tab[i+1]` et `tab[i] ← tab[i]+1`.

Initialisation compacte

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. L'initialisation d'un tableau peut se faire via une série d'attributions, case par case mais on va aussi se permettre une notation compacte² :

```
tab ← {2, 3, 5, 7}           // équivalent à tab[0] ← 2, tab[1] ← 3, tab[2] ← 5, tab[3] ← 7
tab ← {0, ..., 0}           // tous les éléments du tableau sont initialisés à 0
```

1. 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.

2. Peu de langages de programmation ont une notation compacte de ce genre. Java en possède une version moins puissante qui ne permet de coder que le premier des deux exemples. C'est pourquoi, en début d'apprentissage, on vous demandera souvent d'initialiser des tableaux sans utiliser cette notation compacte.

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. Voyons cela en détail.

Passer un tableau en paramètre

Les trois sortes de passages de paramètres sont possibles. Elles servent à spécifier ce qui sera fait avec les cases du tableau. Plus précisément :

- ▷ \downarrow : 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
algorithme afficher(tab $\downarrow$  : tableau de 10 entiers)
    pour i de 0 à 9 faire
        | afficher tab[i]
    fin pour
fin algorithme

// Utilisation possible
monTab : tableau de 10 entiers
monTab  $\leftarrow$  {2,3,5,7,11,13,17,19,23,29}
afficher(monTab)
```

Rappelons qu'il s'agit du passage par défaut si aucune flèche n'est indiquée.

- ▷ $\downarrow\uparrow$: indique que l'algorithme va consulter/modifier les valeurs du tableau reçu en paramètre. Exemple :

```
// Inverse le signe des éléments du tableau
algorithme inverserSigne(tab $\downarrow\uparrow$  : tableau de 10 entiers)
    pour i de 0 à 9 faire
        | tab[i]  $\leftarrow$  -tab[i]
    fin pour
fin algorithme

// Utilisation possible
monTab : tableau de 10 entiers
monTab  $\leftarrow$  {2,-3,5,-7,11,13,17,-19,23,29}
inverserSigne(monTab)
```

- ▷ \uparrow : indique que l'algorithme va assigner des valeurs au tableau reçu en paramètre. Les éléments de ce tableau n'ont donc pas à être initialisés avant d'appeler cet algorithme. Exemple :

```
// Initialise un tableau reçu en paramètre
algorithme initialiser(tab $\uparrow$  : tableau de 10 entiers)
    pour i de 0 à 9 faire
        | tab[i]  $\leftarrow$  i
    fin pour
fin algorithme

// Utilisation possible
monTab : tableau de 10 entiers
initialiser(monTab)
```

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 :

```
// Retourne un tableau initialisé de 10 entiers
algorithme créer() → tableau de 10 entiers
|   tab : tableau de 10 entiers
|   pour i de 0 à 9 faire
|   |   tab[i] ← i
|   fin pour
|   retourner tab
fin algorithme

// Utilisation possible
monTab : tableau de 10 entiers
monTab ← créer()
```

Paramétrer la taille

Un tableau peut être passé en paramètre à un algorithme mais qu'en est-il de sa taille ? Plus haut, nous avons écrit un algorithme qui affiche un tableau de 10 entiers. Il serait utile de pouvoir appeler le même algorithme avec des tableaux de tailles différentes. Pour permettre cela, la taille du tableau reçu en paramètre peut être déclarée avec un nom (habituellement n) qui peut être considéré comme un paramètre entrant et peut être utilisé dans l'algorithme. Idem pour un tableau en retour.

Exemple :

```
// Affiche un tableau de taille quelconque
algorithme afficher(tab : tableau de n entiers)
|   afficher "Tableau de ", n, " éléments".
|   pour i de 0 à n-1 faire           // Attention ! le dernier élément est à l'indice n-1
|   |   afficher tab[i]
|   fin pour
fin algorithme

// Crée un tableau d'entiers de taille n, l'initialise à 0 et le retourne.
algorithme créer(n : entier) → tableau de n entiers
|   tab : tableau de n entiers
|   pour i de 0 à n-1 faire
|   |   tab[i] ← 0
|   fin pour
|   retourner tab
fin algorithme

algorithme test()
|   entiers : tableau de 20 entiers
|   entiers ← créer(20)
|   afficher(entiers)
fin algorithme
```

La fiche 5 page 148 récapitule tout ça.

5 Trouver les entêtes

Écrire les entêtes (et uniquement les entêtes) des algorithmes qui résolvent les problèmes suivants :

- a) Écrire un algorithme qui inverse le signe de tous les éléments négatifs dans un tableau d'entiers.
- b) Écrire un algorithme qui donne le nombre d'éléments négatifs dans un tableau d'entiers.
- c) Écrire un algorithme qui détermine si un tableau d'entiers contient au moins un nombre négatif.
- d) 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.

Parcours d'un tableau

Dans la plupart des problèmes que vous rencontrerez vous serez amené à parcourir un tableau. Il est important de maîtriser ce parcours. Examinons les situations courantes et voyons quelles solutions conviennent.

Parcours complet

Vous avez déjà vu dans les exemples comment parcourir tous les éléments d'un tableau. On s'en sort facilement avec une boucle **pour**. La fiche 6 page 150 décrit comment afficher tous les éléments d'un tableau. On pourrait faire autre chose avec ces éléments : les sommer, les comparer...

6 Inverser le signe des éléments

Écrire un algorithme qui inverse le signe de tous les éléments négatifs dans un tableau d'entiers.

7 Somme

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



8 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.

9 Compter les éléments négatifs

Écrire un algorithme qui donne le nombre d'éléments négatifs dans un tableau d'entiers.

Parcours partiel

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

- ▷ on cherche la présence d'un élément et on vient de le trouver ;
- ▷ on vérifie qu'il n'y a pas de 0 et on vient d'en trouver un ;
- ▷ on vérifie que tous les éléments sont positifs et on vient d'en trouver un qui est négatif ou nul.

Pour résoudre ce problème, on peut partir du parcours complet et :

- ▷ Transformer le **pour** en **tant que**.
- ▷ Introduire un test d'arrêt.

La fiche 7 page 151 détaille un parcours partiel.

10 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.

11 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.

Taille logique et taille physique

Parfois, on ne sait pas exactement quelle taille doit avoir un tableau. 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 cas³ 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. On s'en sort en stockant les valeurs dans la partie basse du tableau (à gauche) et en retenant, 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 on ne peut pas tester si une case est non initialisée) ou bien contenir une valeur qui n'est pas pertinente.

3. En tout cas, suffisamment grand pour tous les cas qu'on accepte de prendre en compte ; il faudra bien fixer une limite.

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.

```

algorithme stockerValeurs()
  tab : tableau de 1000 entiers
  nbÉléments : entier // C'est la taille logique
  valeur : entier
  nbÉléments ← 0
  demander valeur
  tant que valeur > 0 ET nbÉléments < 1000 faire
    tab[nbÉléments] ← valeur
    nbÉléments ← nbÉléments + 1
    demander valeur
  fin tant que
  si valeur > 0 alors // Pourquoi tester valeur et pas nbÉléments ?
    afficher "La limite physique a été atteinte"
  fin si
fin algorithme

```

12 Modifier l'exemple

Tel quel, l'exemple ci-dessus n'est pas très intéressant. Ce serait mieux s'il retournait le tableau ce qui permettrait effectivement un traitement ultérieur des valeurs. Modifiez-le en ce sens.

Des tableaux qui ne commencent pas à 0

Les tableaux que nous avons vus jusqu'à présent *commencent à 0* ce qui signifie que la première case est d'indice 0. Il peut être intéressant d'utiliser des tableaux qui commencent à un autre indice.

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 :

```

algorithme intituléJour(numéroJour : entier) → chaîne
  selon que numéroJour vaut
    1: retourner "lundi"
    2: retourner "mardi"
    3: retourner "mercredi"
    4: retourner "jeudi"
    5: retourner "vendredi"
    6: retourner "samedi"
    7: retourner "dimanche"
  fin selon que
fin algorithme

```

Mais une solution plus élégante passe par l'utilisation d'un tableau. Appelons-le `nomsJours` 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"

Pour obtenir le nom d'un jour, il suffit d'écrire : `nomsJours[numéroJour]`.

On voit dans cet exemple qu'il est plus naturel que le tableau commence à 1. Pour déclarer un tel tableau, nous utiliserons la notation suivante :

```
nomsJours : tableau [1 à 7] de chaînes
```

L'algorithme devient :

```
algorithme intituléJour(numéroJour : entier) → chaine
    nomsJours : tableau [1 à 7] de chaînes
    nomsJours ← {"lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"}
    retourner nomsJours[numéroJour]
fin algorithme
```

Comment traduire ça dans un langage de programmation ? Certains langages comme Pascal permettent de choisir l'indice de début d'un tableau. D'autres l'imposent, parfois à 1 comme Cobol mais le plus souvent à 0 comme Java. Que faire dans ce cas ? Il existe deux stratégies.

1. La première passe par une conversion d'indice. On stocke les données dans le tableau comme nous l'impose le langage

0	1	2	3	4	5	6
"lundi"	"mardi"	"mercredi"	"jeudi"	"vendredi"	"samedi"	"dimanche"

et on accède à l'intitulé du jour i dans la case $i - 1$. Ce qui donne

```
algorithme intituléJour(numéroJour : entier) → chaine
    nomsJours : tableau de 7 chaînes
    nomsJours ← {"lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"}
    retourner nomsJours[numéroJour-1]
fin algorithme
```

2. La seconde approche crée un tableau avec une case en plus et n'utilise pas la première.

0	1	2	3	4	5	6	7
	"lundi"	"mardi"	"mercredi"	"jeudi"	"vendredi"	"samedi"	"dimanche"

L'algorithme devient

```
algorithme intituléJour(numéroJour : entier) → chaine
    nomsJours : tableau de 8 chaînes
    nomsJours ← {"", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"}
    retourner nomsJours[numéroJour]
fin algorithme
```

Cette seconde approche est à préférer car elle modifie le moins l'algorithme de départ et entraînera moins d'erreurs de conversion (au prix d'un faible gaspillage de la mémoire). Mais elle n'est pas toujours possible. Par exemple, imaginons qu'on veuille prendre des relevés de températures (arrondies à l'entier) à Bruxelles et retenir le nombre de fois que chaque température a été relevée. Le plus évident serait d'utiliser un tableau allant de -30 à 40 où la case i donne le nombre de fois que la température i a été relevée. Pour cet exemple, impossible d'utiliser la seconde approche mais la première est toujours possible en stockant le nombre de relevés de la température i dans la case $i + 30$.

13 Lancers de deux dés



Écrire un algorithme qui lance n fois deux dés et compte le nombre de fois que chaque somme apparaît.

```
algorithme lancers2Dés(n : entier) → tableau [2 à 12] de entiers
```


14 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.



Chapitre 10

Gérer les données dans un tableau

Une utilisation courante des tableaux est le stockage des données changeantes. Lors de l'évolution de l'algorithme des données sont ajoutées, recherchées, supprimées, modifiées.

Par exemple, imaginons que l'école organise un événement libre et gratuit pour les étudiants. Mais pour y assister, ils doivent s'inscrire. On veut fournir ce qu'il faut pour :

- ▷ inscrire un étudiant ;
- ▷ vérifier si un étudiant est inscrit ;
- ▷ désinscrire un étudiant (s'il change d'avis par exemple) ;
- ▷ afficher la liste des inscrits.

Pour ce faire, on pourra stocker les numéros des étudiants inscrits dans un tableau ¹. Comme le tableau ne sera généralement pas rempli, on va gérer sa taille logique. On a donc deux variables :

- ▷ `inscrits` : le tableau des numéros d'étudiants
- ▷ `nblinscrits` : le nombre d'étudiants actuellement inscrits (la taille logique du tableau)

On va envisager deux cas :

1. Les numéros d'étudiants seront placés dans le tableau sans ordre imposé.
2. Les numéros d'étudiants seront placés dans l'ordre.

Données non triées

Dans cette approche, les valeurs sont stockées dans le tableau dans un ordre quelconque. Par exemple, on pourrait avoir la situation suivante :

inscrits =								
42010	41390	43342	42424	?	?	?	?	...
nblinscrits = 4								

indiquant qu'il y a pour le moment quatre étudiants inscrits dont les numéros sont ceux repris dans les quatre premières cases du tableau.

1. Une vraie solution de ce problème utiliserait probablement une base de données mais ça c'est une autre histoire et un autre cours ;)

Inscription

Commençons par l'inscription. Pour inscrire un étudiant, 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 :

inscrits =								
42010	41390	43342	42424	42123	?	?	?	...
nbInscrits = 5								

L'algorithme est assez simple :

```
// Inscrit un étudiant de numéro donné.
algorithme inscrire(inscrits↓↑ : tableau de n entiers, nbInscrits↓↑ : entier, numéro↓ : entier)
    // 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)
    inscrits[nbInscrits] ← numéro
    nbInscrits ← nbInscrits + 1
fin algorithme
```

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. On pourrait se contenter de retourner un booléen indiquant si oui ou non on l'a trouvé mais retournons plutôt un entier indiquant l'indice où il est (-1 si on ne l'a pas trouvé), ce sera plus utile.

```
// Vérifie si un étudiant est inscrit et donne sa position (-1 si non inscrit)
algorithme vérifier(inscrits↓ : tableau de n entiers, nbInscrits↓ : entier, numéro↓ : entier) → entier
    i : entier
    i ← 0
    tant que i < nbInscrits ET inscrits[i] ≠ numéro faire
        i ← i + 1
    fin tant que
    si i < nbInscrits alors
        retourner i
    sinon
        retourner -1
    fin si
fin algorithme
```

La fiche 9 page 154 reprend cet algorithme dans un cadre plus général.

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. La désinscription de l'étudiant numéro 41390 donnerait ²

inscrits =								
42010	42123	43342	42424	42123	?	?	?	...
nbInscrits = 4								

L'algorithme est assez simple à écrire si on utilise la recherche écrite juste avant :

2. 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 nbInscrits).

```
// désinscrit l'étudiant donné
algorithme désinscrire( inscrits↓↑ : tableau de n entiers, nbInscrits↓↑ : entier, numéro↓ : entier )
    pos : entier
    pos ← vérifier(inscrits, nbInscrits, numéro)
    // On pourrait vérifier et générer une erreur si l'étudiant n'est pas inscrit
    inscrits[pos] ← inscrits[nbInscrits-1]
    nbInscrits ← nbInscrits - 1
fin algorithme
```

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 Limite au 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 ?

Données triées

Imaginons à présent que nous maintenons un ordre dans les données du tableau.

Si on reprend l'exemple utilisé pour les données non triées, on a :

inscrits =								
41390	42010	42424	43342	?	?	?	?	...
nbInscrits = 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.

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 ça.

L'algorithme est assez proche de celui de la vérification d'une inscription vu précédemment si ce n'est qu'on peut s'arrêter dès qu'on trouve un numéro plus grand. On va aussi ajouter un booléen indiquant si la valeur a été trouvée.

```
// Recherche un étudiant.
// - trouvé : indique si oui ou non il a été trouvé
// - pos : indique la position où a été trouvé l'étudiant ou la position où il aurait dû être
algorithme rechercher( inscrits↓ : tableau de n entiers, nbInscrits↓ : entier,
                                numéro↓ : entier, trouvé↑ : booléen, pos↑ : entier )
    pos ← 0
    tant que pos < nbInscrits ET inscrits[pos] < numéro faire
        pos ← pos + 1
    fin tant que
    trouvé ← pos < nbInscrits ET inscrits[pos] = numéro
fin algorithme
```

Comprenez-vous pourquoi :

- ▷ on trouve un < dans la condition du tant que et pas un ≠ ?
- ▷ l'expression assignée à trouvé est composée de deux parties et utilise un = ?

Inscrire un étudiant

L'inscription d'un étudiant se fait en trois étapes :

1. On recherche l'étudiant via l'algorithme qu'on vient de voir, ce qui nous donne la place où le placer.
2. On libère la place pour l'y mettre. Comprenez-bien que cette opération n'est pas triviale. Si vous tenez des cartes en main, il est tout aussi facile d'y 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. On peut placer l'élément à la position libérée.

Ce qui nous donne :

```
// Inscrit un étudiant de numéro donné.
algorithme inscrire( inscrits↓↑ : tableau de n entiers, nbInscrits↓↑ : entier, numéro↓ : entier )
    pos : entier
    trouvé : booléen
    rechercher( inscrits, nbInscrits, numéro, trouvé, pos )
    décalerDroite( inscrits, pos, nbInscrits-1)
    inscrits[pos] ← numéro
    nbInscrits ← nbInscrits + 1
fin algorithme

// Décale d'une position à droite les éléments entre la position début et fin
algorithme décalerDroite( tab↓↑ : tableau de n entiers, début↓ : entier, fin↓ : entier )
    pour i de fin à début par -1 faire
        tab[i+1] ← tab[i]
    fin pour
fin algorithme
```

Vérifier une inscription

Cette opération est triviale.

```
// Vérifie si un étudiant est inscrit et donne sa position (-1 si non inscrit)
algorithme vérifier( inscrits↓ : tableau de n entiers, nbInscrits↓ : entier,
                                numéro↓ : entier ) → entier
    pos : entier
    trouvé : booléen
    rechercher( inscrits, nbInscrits, numéro, trouvé, pos )
    si trouvé alors
        retourner pos
    sinon
        retourner -1
    fin si
fin algorithme
```

Désinscription

Ici, il va falloir décaler vers la gauche les éléments qui suivent celui à supprimer.

```
// désinscrit l'étudiant donné
algorithme désinscrire( inscrits↓↑ : tableau de n entiers, nbInscrits↓↑ : entier, numéro↓ : entier )
    pos : entier
    trouvé : booléen
    rechercher( inscrits, nbInscrits, numéro, trouvé, pos )
    décalerGauche( inscrits, pos+1, nbInscrits-1)
    nbInscrits ← nbInscrits - 1
fin algorithme

// Décale d'une position à gauche les éléments entre la position début et fin
algorithme décalerGauche( tab↓↑ : tableau de n entiers, début↓ : entier, fin↓ : entier )
    pour i de début à fin faire
        tab[i-1] ← tab[i]
    fin pour
fin algorithme
```

La fiche [10 page 155](#) reprend cet algorithme dans un cadre plus général.

Exercices

5 Éviter la double inscription

Comment modifier l'algorithme d'inscription pour s'assurer qu'un étudiant ne s'inscrive pas deux fois ?

6 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 ?

7 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 ?

Intérêt de trier les valeurs ?

Est-ce que trier les valeurs est vraiment intéressant ? On voit que la recherche est un peu plus rapide (en moyenne 2x plus rapide si l'élément ne s'y trouve pas). Par contre, l'ajout

et la suppression d'un élément sont plus lents. En plus, 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.

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, on va prendre 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 *val* la valeur recherchée dans une zone délimitée par les indices *indiceGauche* et *indiceDroit*. On commence par déterminer l'élément médian, c'est-à-dire celui qui se trouve « au milieu » de la zone de recherche³ ; son indice sera déterminé par la formule

$$\text{indiceMédian} \leftarrow (\text{indiceGauche} + \text{indiceDroit}) \text{ DIV } 2$$

On compare alors *val* avec la valeur de cet élément médian ; il est possible qu'on ait trouvé la valeur cherchée ; sinon, on partage la zone de recherche en deux parties : une qui **ne contient certainement pas** la valeur cherchée et une qui **pourrait la contenir**. C'est cette deuxième partie qui devient la nouvelle zone de recherche. On adapte *indiceGauche* ou *indiceDroit* en conséquence. On réitère le processus jusqu'à ce que la valeur cherchée soit trouvée ou que la zone de recherche soit réduite à sa plus simple expression, c'est-à-dire un seul élément.

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, *indiceGauche* vaut 0 et *indiceDroit* 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 : $\text{indiceMédian} \leftarrow (0 + 19) \text{ DIV } 2$, 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 *indiceGauche* qui vaut 10 et *indiceDroit* 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 : $\text{indiceMédian} \leftarrow (10 + 19) \text{ DIV } 2$, 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

3. Cet élément médian n'est pas tout à fait au milieu dans le cas d'une zone contenant un nombre pair d'éléments.

Troisième étape : $\text{indiceMédian} \leftarrow (10 + 13) \text{ DIV } 2$, c'est-à-dire 11 où se trouve l'élément 20. La zone de recherche devient indiceGauche vaut 12 et indiceDroit vaut 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

Quatrième étape : $\text{indiceMédian} \leftarrow (12 + 13) \text{ DIV } 2$, 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

```

algorithme rechercheDichotomique(
    tab↓ : tableau de n entiers, valeur↓ : entier, pos↑ : entier ) → booléen
    indiceDroit, indiceGauche, indiceMédian : entiers
    candidat : entier
    trouvé : booléen

    indiceGauche ← 0
    indiceDroit ← n-1
    trouvé ← faux

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

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

    retourner trouvé
fin algorithme

```

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.

Une approche pratique : la simulation numérique

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

Une approche théorique : la complexité

Avec cette approche, on va 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 on recherche un élément. On peut considérer que l'opération de base est la comparaison avec un élément du tableau. La question est alors la suivante : pour un tableau de taille n , à combien de comparaisons faut-il procéder pour trouver l'élément (ou se rendre compte de son absence) ?

Pour la recherche linéaire

Cela dépend évidemment de la position de la valeur à trouver. Dans le meilleur des cas c'est 1, dans le pire c'est n mais on peut dire, qu'en moyenne, cela 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

On voit que c'est surtout pour des grands problèmes que la recherche dichotomique montre toute son efficacité. On voit aussi que ce qui est important pour mesurer la complexité c'est l'ordre de grandeur du nombre de comparaisons.

On dira que la recherche simple est un algorithme de complexité linéaire (c'est-à-dire que le nombre d'opérations est de l'ordre de n ou proportionnel à n ou encore que si on double la taille du problème, le temps va aussi être doublé) ce qu'on note en langage plus mathématique $O(n)$ (prononcé « grand O de n »).

Pour la recherche dichotomique, la complexité est logarithmique, et on le note $O(\log_2(n))$ ce qui veut dire que si on double la taille du problème, on a juste une itération en plus dans la boucle.

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}	$10^{301029995}$

On voit ainsi que si on trouve un algorithme correct pour résoudre un problème mais que cet algorithme est de complexité exponentielle alors cet algorithme ne sert à rien en pratique. Par exemple un algorithme de recherche de complexité exponentielle, exécuté sur une machine pouvant effectuer un milliard de comparaisons par secondes, prendrait plus de dix mille milliards d'années pour trouver une valeur dans un tableau de 100 éléments.

8 Calcul de complexités

Quelle est la complexité d'un algorithme qui :

- recherche le maximum d'un tableau de n éléments ?
- remplace par 0 toutes les occurrences du maximum d'un tableau de n éléments ?
- vérifie si un tableau contient deux éléments égaux ?
- vérifie si les éléments d'un tableau forment un palindrome ?
- cherche un élément dans un tableau en essayant des cases au hasard jusqu'à le trouver ?

9 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.

10 Réflexion

L'algorithme de recherche dichotomique est-il toujours à préférer à l'algorithme de recherche linéaire ?

Chapitre 1

Le tri

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



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, on est amené à devoir trier tout 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 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 qu'on veut 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 l'étudiant. On dira de cet ensemble qu'il est classé en **majeur** sur le numéro de groupe et en **mineur** sur le nom d'étudiant.

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

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

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

```
// Trie le tableau reçu en paramètre (via un tri par insertion).
algorithme triInsertion(tab↓↑ : tableau de n entiers)
    i, j, valÀInsérer : entiers
    pour i de 1 à n-1 faire
        valÀInsérer ← tab[i]
        // recherche de l'endroit où insérer valÀInsérer dans le
        // sous-tableau trié et décalage simultané des éléments
        j ← i - 1
        tant que j ≥ 0 ET valÀInsérer < tab[j] faire
            tab[j+1] ← tab[j]
            j ← j - 1
        fin tant que
        tab[j+1] ← valÀInsérer
    fin pour
fin algorithme
```

Tri par sélection des minima successifs

Dans ce tri, on recherche à chaque étape la plus petite valeur de l'ensemble non encore trié et on 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).
algorithme triSélectionMinimaSuccessifs(tab↓↑ : tableau de n entiers)
    i, indiceMin : entier
    pour i de 0 à n - 2 faire                                // i correspond à l'étape de l'algorithme
        indiceMin ← positionMin( tab, i, n-1 )
        swap( tab[i], tab[indiceMin] )
    fin pour
fin algorithme
```

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



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

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 :

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 `indiceBulle` est à la fois le compteur d'étapes et aussi l'indice de l'élément récepteur de la bulle à la fin d'une étape donnée.

```
// Trie le tableau reçu en paramètre (via un tri bulle).
algorithme triBulle(tab↓↑ : tableau de n entiers)
  indiceBulle, i : entiers
  pour indiceBulle de 0 à n-1 faire
    pour i de n - 2 à indiceBulle par -1 faire
      si tab[i] > tab[i + 1] alors
        swap( tab[i], tab[i + 1] )           // voir algorithme précédent
      fin si
    fin pour
  fin pour
fin algorithme
```

Cas particuliers

Tri partiel

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

Sélection des meilleurs

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

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

```
algorithme meilleurs(plusPetits↓↑ : tableau de k+1 entiers, nbValeurs↑ : entier)
  val : entier
  nbValeurs ← 0
  demander val
  tant que val ≥ 0 faire
    insérer( val, plusPetits, nbValeurs )
    demander val
  fin tant que
fin algorithme
```

```
algorithme insérer(val↓ : entier, plusPetits↓↑ : tableau de k+1 entiers, nbValeurs↓↑ : entier)
  i : entier
  i ← nbValeurs - 1
  tant que i ≥ 0 ET val < plusPetits[i] faire
    plusPetits[i + 1] ← plusPetits[i]
    i ← i - 1
  fin tant que
  plusPetits[i + 1] ← val
  si nbValeurs < k alors
    nbValeurs ← nbValeurs + 1
  fin si
fin algorithme
```

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.

Chapitre 12

Exercices sur les tableaux

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.

1 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.



2 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.

3 Maximum

Écrire un algorithme qui reçoit en paramètre le tableau `tabEnt` de n entiers et qui retourne la plus **grande** valeur de ce tableau.



4 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.



5 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.

6 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.

7 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.

8 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.

9 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}.

10 Les doublons



Écrire un algorithme qui vérifie si un tableau de chaînes contient au moins 2 éléments égaux.

11 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 raze 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 .

12 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.

algorithme *testerProposition*(proposition↓, solution↓ : tableau de n chaînes,
bienPlacés↑, malPlacés↑ : entiers)

Quatrième partie

Compléments

13 Les chaînes	123
14 Les variables structurées	129

Chapitre 13

Les chaines

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 ».



Introduction

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 manipuler le contenu de chaînes. Mais procédons par étapes.

Longueur

La **longueur** d'une chaîne est le nombre de caractères qu'elle contient. Pour la connaître on utilisera la notation `long(uneChaîne)`. Par exemple :



- ▷ `long("Bonjour")` vaut 7.
- ▷ `long("Une chaîne")` vaut 10 (l'espace compte pour 1).
- ▷ `long("A")` est 1.
- ▷ `long("")` est égal à 0.

Chaîne et caractère

Certains langages, comme Java, distinguent clairement le type chaîne et le type caractère. Ainsi, en Java, le littéral `'A'` représente un caractère qui est différent du littéral `"A"` qui représente une chaîne (composée d'un seul caractère). Cette distinction est essentiellement dictée par des détails d'implémentation ; un caractère pouvant se représenter de façon plus économe qu'une chaîne.

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.

Ici, nous introduirons un type `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, `caractère` n'est qu'un raccourci pour dire : *une chaîne de longueur 1*.

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 :

```
texte : chaîne
texte ← "Hello"
afficher texte[5]                                // Affiche "o"
texte[2] ← "a"    // texte vaut "Hallo" ; un caractère doit être remplacé par un seul caractère.
```

Exemple. Écrivons un algorithme qui vérifie si un mot donné contient une lettre donnée.

```
algorithme contient(mot : chaîne, lettre : caractère) → booléen
    i : entier
    i ← 1
    tant que i ≤ long(mot) ET mot[i] ≠ lettre faire
        i ← i + 1
    fin tant que
    retourner i ≤ long(mot)
fin algorithme
```

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 :

```
texte ← "al" + "go" + "rithmique"    // assigne la chaîne "algorithmique" à la variable texte.
```

Exemple. Écrivons un algorithme qui inverse toutes les lettres d'un mot. Ainsi, "algo" deviendra "ogla".

```
algorithme miroir(mot : chaîne) → chaîne
    miroir : chaîne
    miroir ← ""
    pour i de 1 à long(mot) faire
        miroir ← mot[i] + miroir
    fin pour
    retourner miroir
fin algorithme
```

Manipuler les caractères

Les algorithmes qui suivent pourraient être écrits avec ce que nous avons déjà introduit mais ce serait long et fastidieux. Nous allons supposer qu'ils existent et on pourra les utiliser si

nécessaire.

estLettre(car : caractère) → booléen

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

estMinuscule(car : caractère) → booléen

Permet de savoir si le caractère est une lettre minuscule.

estMajuscule(car : caractère) → booléen

Permet de savoir si le caractère est une lettre majuscule.

estChiffre(car : caractère) → booléen

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

majuscule(texte : chaîne) → chaîne

Retourne une chaîne où toutes les lettres du texte ont été converties en majuscules.

minuscule(texte : chaîne) → chaîne

Retourne une chaîne où toutes les lettres du texte ont été converties en minuscules.

L'alphabet

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

numLettre(car : caractère) → entier

Retourne toujours un entier entre 1 et 26. Par exemple `numLettre("E")` donnera 5, ainsi que `numLettre("e")`. Cette fonction traite donc de la même manière les majuscules et les minuscules. `numLettre` retournera aussi 5 pour les caractères "é", "è", "ê", "ë"...). Attention, il est interdit d'utiliser cette fonction si le caractère n'est pas une lettre !

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

lettreMaj(n : entier) → caractère

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".

lettreMin(n : entier) → caractère

Idem pour les minuscules.

Chaîne et nombre

Si une chaîne contient un nombre, on doit pouvoir le convertir, et inversement.

chaîne(n : réel) → chaîne

Transforme un nombre en chaîne. Ex : `chaîne(42)` retourne la chaîne "42" et `chaîne(3,14)` donnera "3,14".

nombre(ch : chaîne) → réel

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.

Extraction de sous-chaines

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

Permet d'extraire une portion d'une certaine longueur d'une chaine donnée, et ceci à partir d'une position donnée.

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

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

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

Recherche de sous-chaine

position(ch : chaine, sous-chaine : chaine) → entier

Permet de savoir si une sous-chaine donnée est présente dans une chaine 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-chaine recherchée. Par exemple, `position("algorithmique", "mi")` retournera 9. Si la sous-chaine ne s'y trouve pas, la fonction retourne 0. On peut admettre ici d'écrire un caractère à la place de la sous-chaine. Par exemple, `position("algorithmique", "m")` retournera également 9.

Exercices

1 Calcul de fraction

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

Solution. Voici une solution possible.

```

algorithme calculerFraction(fraction : chaine) → réel
    posFraction, numérateur, dénominateur : entiers
    posFraction ← position(fraction, "/")
    numérateur ← nombre(souschaine(fraction, 1, posFraction-1))
    dénominateur ← nombre(souschaine(fraction, posFraction+1, long(fraction)-posFraction))
    retourner numérateur / dénominateur
fin algorithme

```

2 Conversion de nom

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

3 Gauche et droite

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

4 Grammaire

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

5 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".

6 Les palindromes

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

- a) 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))
- b) 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.

7 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**.

8 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.

Chapitre 14

Les variables structurées

Le type structuré

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...

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** de la structure, possède un nom unique.

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

Définition d'une structure

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

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



`nomChamp1`, ..., `nomChampN` sont les noms des différents champs de la structure, et `type1`, ..., `typeN` les types de ces champs. Tous les types peuvent être envisagés : les types « simples » (entier, réel, booléen, chaîne), les tableaux et même d'autres types structurés dont la structure aura été préalablement définie.

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

structure Date jour : entier mois : entier année : entier fin structure	structure Moment heure : entier minute : entier seconde : entier fin structure	structure Point x : réel y : réel fin structure	structure Etudiant nom : chaîne matricule : entier section : chaîne fin structure
---	--	--	---

Déclaration d'une variable de type structuré

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

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

Utilisation des variables de type structuré

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

```
nomVariable.nomChamp
```

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

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

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

```
arrivée ← départ
```

qui résume les trois instructions suivantes :

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

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

```
anniversaire ← {1, 9, 1989}
```

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 :

```
algorithme nbSecondesEcoulees(départ↓, arrivée↓ : Moment) → entier
```

On pourra aussi lire ou afficher une variable structurée ¹.

```
demander unMoment
afficher unMoment
```

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

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

Exemple d'algorithme

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

```
// Reçoit 2 dates en paramètres et retourne une valeur
// négative si la première date est antérieure à la deuxième
// nulle si les deux dates sont identiques
// positive si la première date est postérieure à la deuxième.
algorithme comparerDates(date1↓, date2↓ : Date) → entier
    si date1.année ≠ date2.année alors
        retourner date1.année - date2.année
    sinon si date1.mois ≠ date2.mois alors
        retourner date1.mois - date2.mois
    sinon
        retourner date1.jour - date2.jour
    fin si
fin algorithme
```

Remarque sur les champs

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 trois numéros de téléphone (un tableau de trois chaînes).

```
structure Personne
    nom : chaîne
    naissance : Date
    téléphones : tableau de 3 chaînes
fin structure
```

Et voici un exemple d'utilisation.

```
moi : Personne
moi.naissance.jour ← 23
moi.téléphones[0] ← "0444/42.42.42"
```

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

Exercices sur les structures

1 Conversion moment-secondes

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

2 Conversion secondes-moment

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

3 Temps écoulé entre 2 moments

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

4 Milieu de 2 points

Écrire un module recevant deux points du plan et qui retourne le point situé au milieu des deux.

5 Distance entre 2 points

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

6 Un cercle

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

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

7 Un rectangle



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

- a) un module calculant le périmètre d'un rectangle reçu en paramètre ;
- b) un module calculant la surface d'un rectangle reçu en paramètre ;
- c) un module recevant en paramètre un rectangle R et les coordonnées d'un point P, et renvoyant **vrai** si et seulement si le point P est à l'intérieur du rectangle R ;

8 Validation de date

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

Cinquième partie

Conclusion

15 Exercices récapitulatifs

135

Chapitre 15

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.

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 lettres¹.

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.

1. dans le vrai jeu c'est un peu plus compliqué.

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 "".

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

algorithme *créerChemin*($n \downarrow$: entier) \rightarrow tableau de n entier

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

algorithme *enPremièrePosition*(positionsJoueurs↓ : tableau de m entiers) → entier

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.

algorithme *jouer*(chemin↓ : tableau de n entiers, positionsJoueurs↓↑ : tableau de m entiers, joueurCourant↓ : entier, valeurDé↓ : entier)


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.

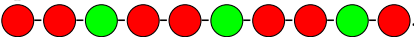
Cet algorithme met donc à jour le tableau *positionsJoueurs*.

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.

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 collier² dessiné sur une feuille de papier et ils doivent le colorier en répétant un *motif* donné³, 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 :

'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.

2. Par exemple, mais ça peut aussi être une chenille, un escargot, une simple suite de cases...

3. Ce qu'ils appellent un *algorithme*.

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.

Sixième partie

Les annexes

A Les fiches	143
B Bonnes (et mauvaises) pratiques	161
C Le LDA	165
D Aide mémoire	169

Annexe A

Les fiches

Vous trouverez ici toutes les fiches des algorithmes analysés dans ce cours.

1	Un calcul simple	144
2	Un calcul complexe	145
3	Un nombre pair	146
4	Maximum de deux nombres	147
5	Passage d'un tableau en paramètre	148
6	Parcours complet d'un tableau	150
7	Parcours partiel d'un tableau	151
8	Maximum dans un tableau	153
9	Tableau non trié	154
10	Tableau trié	155
11	Recherche dichotomique	157
12	Tri d'un tableau	158

Fiche n° 1 – Un calcul simple

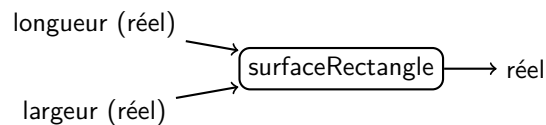
Calculer la surface d'un rectangle à partir de sa longueur et sa largeur.

Spécification

Données (toutes réelles et non négatives) :

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

Résultat : un réel représentant la surface du rectangle.



Exemples

- ▷ `surfaceRectangle(3, 2)` donne 6
- ▷ `surfaceRectangle(3.5, 1)` donne 3.5

Solution

La surface d'un rectangle est obtenue en multipliant la largeur par la longueur.

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

Ce qui donne :

```

algorithme surfaceRectangle(longueur, largeur : réel) → réel
|   retourner longueur * largeur
fin algorithme
  
```

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

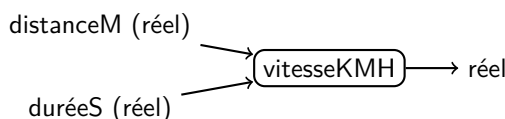
Calculer la vitesse moyenne (en km/h) d'un véhicule dont on donne la distance parcourue (en mètres) et la durée du parcours (en secondes).

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

- ▷ `vitesseKMH(100,10)` donne 36
- ▷ `vitesseKMH(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 :

```

algorithme vitesseKMH(distanceM, duréeS : réel) → réel
  distanceKM, duréeH : réel
  distanceKM ←  $\frac{\text{distanceM}}{1000}$ 
  duréeH ←  $\frac{\text{duréeS}}{3600}$ 
  retourner  $\frac{\text{distanceKM}}{\text{duréeH}}$ 
fin algorithme
  
```

Vérification

test n°	distance en mètres	durée en secondes	réponse attendue	réponse fournie	
1	100	10	36	36	✓
2	10000	3600	10	10	✓

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 précédente](#)).

Fiche n° 3 – Un nombre pair

Un nombre reçu en paramètre est-il pair ?

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.

nombre (entier) \longrightarrow **estPair** \longrightarrow booléen

Exemples

- ▷ `estPair(2016)` donne *vrai*
- ▷ `estPair(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.

```
algorithme estPair(nombre : entier)  $\rightarrow$  booléen  
|   retourner nombre MOD 2 = 0  
fin algorithme
```

Attention à éviter les mauvaises écritures expliquées à l'annexe [B.2 page 161](#) : 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 précédente](#).

On peut également s'inspirer de cette solution quand il faut donner sa valeur à une variable booléenne.

Fiche n° 4 – Maximum de deux nombres

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

```

algorithme max2(nb1 : réel, nb2 : réel) → réel
  max : réel
  si nb1 > nb2 alors
    max ← nb1
  sinon
    max ← nb2
  fin si
  retourner max
fin algorithme
  
```

Attention à éviter les mauvaises écritures expliquées à l'annexe [B.3 page 162](#).

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

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
algorithme afficher(tab↓ : tableau de n entiers)
  pour i de 0 à n-1 faire
    afficher tab[i]
  fin pour
fin algorithme

// Utilisation possible
monTab : tableau de 10 entiers
monTab ← {2,3,5,7,11,13,17,19,23,29}
afficher(monTab)
```

Rappelons qu'il s'agit du passage 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
algorithme inverserSigne(tab↓↑ : tableau de n entiers)
  pour i de 0 à n-1 faire
    tab[i] ← -tab[i]
  fin pour
fin algorithme

// Utilisation possible
monTab : tableau de 5 entiers
monTab ← {2,-3,5,-7,11}
inverserSigne(monTab)
```

- ▷ ↑ : indique que l'algorithme va assigner des valeurs au tableau reçu en paramètre. Les éléments de ce tableau n'ont donc pas à être initialisés avant d'appeler cet algorithme. Exemple :

```
// Initialise un tableau de n entiers reçu en paramètre
algorithme initialiser(tab↑ : tableau de n entiers)
  pour i de 0 à n-1 faire
    tab[i] ← i
  fin pour
fin algorithme

// Utilisation possible
monTab : tableau de n entiers
initialiser(monTab)
```

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.  
algorithme créer(n : entier) → tableau de n entiers  
| tab : tableau de n entiers  
| pour i de 0 à n-1 faire  
| | tab[i] ← 0  
| fin pour  
| retourner tab  
fin algorithme  
  
// Utilisation possible  
algorithme test()  
| entiers : tableau de 20 entiers  
| entiers ← créerTableau(20)  
| afficher(entiers)  
fin algorithme
```

Fiche n° 6 – Parcours complet d'un tableau

Afficher tous les éléments d'un tableau d'entiers.

Spécification

Données : le tableau à afficher

Résultat : aucun. Se contente d'afficher les éléments.

Solution

Puisqu'on parcourt tout le tableau, on peut utiliser une boucle *pour*.

```
algorithme afficherTab(tab : tableau de n entiers)
  pour i de 0 à n-1 faire
    afficher tab[i]
  fin pour
fin algorithme
```

Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois qu'on doit examiner **tous** les éléments d'un tableau, quel que soit le traitement qu'on en fait : les afficher, les sommer, les comparer...

Fiche n° 7 – Parcours partiel d'un tableau

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.

tab (tableau d'entiers) \longrightarrow contientZéro \longrightarrow booléen

Solution

Contrairement au parcours complet (cf. [fiche 6 page précédente](#)) on va utiliser un *tant que* car on peut s'arrêter dès qu'on trouve ce qu'on cherche.

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

```

algorithme contientZéro(tab : tableau de n entiers)  $\rightarrow$  booléen
    i : entier
    i  $\leftarrow$  0
    tant que i < n ET tab[i]  $\neq$  0 faire
        i  $\leftarrow$  i + 1
    fin tant que
    retourner i < n                                // Si i<n -> arrêt prématuré -> on a trouvé un 0.
fin algorithme

```

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 ([5.3.3 page 40](#)).

[B] Avec variable booléenne

```

algorithme contientZéro(tab : tableau de n entiers)  $\rightarrow$  booléen
    i : entier
    zéroPrésent : booléen
    zéroPrésent  $\leftarrow$  faux
    i  $\leftarrow$  0
    tant que i < n ET NON zéroPrésent faire
        zéroPrésent  $\leftarrow$  tab[i] = 0
        i  $\leftarrow$  i + 1
    fin tant que
    retourner zéroPrésent
fin algorithme

```

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 :

```
si tab[i] = 0 alors  
|   zéroPrésent ← vrai  
sinon  
|   i ← i + 1  
fin si
```

Dans notre exemple, on cherche un élément particulier (un 0). Dans le cas où on vérifie si tous les éléments possèdent une certaine propriété (être positifs par exemple), on veillera à adapter le nom du booléen et son utilisation (par exemple un booléen appelé `tousPositifs`, initialisé à vrai avec un `...ET tousPositifs` dans le test.

Dans tous les cas, faites attention à ne pas utiliser `tab[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 [B.4 page 162](#).

Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois qu'on parcourt un tableau mais qu'un arrêt avant la fin 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

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.

```
algorithme max(tab : tableau de n entiers) → entier
    max : entier
    max ← tab[0]
    pour i de 1 à n-1 faire
        si tab[i] > max alors
            max ← tab[i]
        fin si
    fin pour
    retourner max
fin algorithme
```

Variante

On peut être intéressé par la position où se trouve le (en tout cas un) maximum.

```
algorithme max(tab : tableau de n entiers) → entier
    posMax : entier
    posMax ← 0
    pour i de 1 à n-1 faire
        si tab[i] > tab[posMax] alors
            posMax ← i
        fin si
    fin pour
    retourner posMax
fin algorithme
```

Fiche n° 9 – Tableau non trié

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

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)
algorithme vérifier(tab↓ : tableau de n entiers, nbElem↓ : entier, nbRecherché↓ : entier) → entier
| i : entier
| i ← 0
| tant que i < nbElem ET tab[i] ≠ nbRecherché faire
| | i ← i + 1
| fin tant que
| si i < nbElem alors
| | retourner i
| sinon
| | retourner -1
| fin si
fin algorithme
```

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 tableau.
algorithme ajouter(tab↓↑ : tableau de n entiers, nbElem↓↑ : entier, nbAjouter↓ : entier)
| si nbElem < n ET vérifier(tab, nbElem, nbAjouter) ≠ -1 alors
| | tab[nbElem] ← nbAjouter
| | nbElem ← nbElem + 1
| fin si
fin algorithme
```

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.
algorithme supprimer( tab↓↑ : tableau de n entiers, nbElem↓↑ : entier, nbSupprimer↓ : entier )
| pos : entier
| pos ← vérifier(tab, nbElem, nbSupprimer)
| si pos ≠ -1 alors
| | tab[pos] ← tab[nbElem-1]
| | nbElem ← nbElem - 1
| fin si
fin algorithme
```

Fiche n° 10 – Tableau trié

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 trouvé la valeur ou la position où elle aurait dû être

```
// Recherche un étudiant.
// - trouvé : indique si oui ou non il a été trouvé
// - pos : indique la position où a été trouvé la valeur ou la position où elle aurait dû être
algorithme rechercher( tab↓ : tableau de n entiers, nbElem↓ : entier,
                                nbRecherché↓ : entier, trouvé↑ : booléen, pos↑ : entier )
|
|   pos ← 0
|   tant que pos < nbElem ET tab[pos] < nbRecherché faire
|       pos ← pos + 1
|   fin tant que
|   trouvé ← pos < nbElem ET tab[pos] = nbRecherché
fin algorithme
```

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 inscrit)
algorithme vérifier( tab↓ : tableau de n entiers, nbElem↓ : entier,
                                nbRecherché↓ : entier ) → entier
|
|   pos : entier
|   trouvé : booléen
|   rechercher( tab, nbElem, nbRecherché, trouvé, pos )
|   si trouvé alors
|       retourner pos
|   sinon
|       retourner -1
|   fin si
fin algorithme
```

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

```
// Ajouter un nombre donné.
algorithme ajouter( tab↓↑ : tableau de n entiers, nbElem↓↑ : entier, nbAjouter↓ : entier )
    pos : entier
    trouvé : booléen
    rechercher( tab, nbElem, nbAjouter, trouvé, pos )
    décalerDroite( tab, pos, nbElem )
    tab[pos] ← nbAjouter
    nbElem ← nbElem + 1
fin algorithme

// Décale d'une position à droite les éléments entre la position début et fin
algorithme décalerDroite( tab↓↑ : tableau de n entiers, début↓ : entier, fin↓ : entier )
    pour i de fin à début par -1 faire
        tab[i+1] ← tab[i]
    fin pour
fin algorithme
```

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

```
// supprimer l'élément donné
algorithme supprimer( tab↓↑ : tableau de n entiers, nbElem↓↑ : entier, nbSupprimer↓ : entier )
    pos : entier
    trouvé : booléen
    rechercher( tab, nbElem, nbSupprimer, trouvé, pos )
    décalerGauche( tab, pos+1, nbElem )
    nbElem ← nbElem - 1
fin algorithme

// Décale d'une position à gauche les éléments entre la position début et fin
algorithme décalerGauche( tab↓↑ : tableau de n entiers, début↓ : entier, fin↓ : entier )
    pour i de début à fin faire
        tab[i-1] ← tab[i]
    fin pour
fin algorithme
```

Fiche n° 11 – Recherche dichotomique

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
- ▷ la valeur recherchée

Résultat :

- ▷ un booléen indiquant si la valeur a été trouvée
- ▷ 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.

```

algorithme rechercheDichotomique(
    tab↓ : tableau de n entiers, valeur↓ : entier, pos↑ : entier ) → booléen
    indiceDroit, indiceGauche, indiceMédian : entiers
    candidat : entier
    trouvé : booléen

    indiceGauche ← 0
    indiceDroit ← n-1
    trouvé ← faux

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

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

    retourner trouvé
fin algorithme

```

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

Nous avons vu trois algorithmes

Le tri par insertion

```
// Trie le tableau reçu en paramètre (via un tri par insertion).
algorithme triInsertion(tab↓↑ : tableau de n entiers)
  i, j, valÀInsérer : entiers
  pour i de 1 à n-1 faire
    valÀInsérer ← tab[i]
    // recherche de l'endroit où insérer valÀInsérer dans le
    // sous-tableau trié et décalage simultané des éléments
    j ← i - 1
    tant que j ≥ 0 ET valÀInsérer < tab[j] faire
      tab[j+1] ← tab[j]
      j ← j - 1
    fin tant que
    tab[j+1] ← valÀInsérer
  fin pour
fin algorithme
```

Le tri par sélection des minima successifs

```
// Trie le tableau reçu en paramètre (via un tri par sélection des minima successifs).
algorithme triSélectionMinimaSuccessifs(tab↓↑ : tableau de n entiers)
  i, indiceMin : entier
  pour i de 0 à n - 2 faire // i correspond à l'étape de l'algorithme
    indiceMin ← positionMin( tab, i, n-1 )
    swap( tab[i], tab[indiceMin] )
  fin pour
fin algorithme
```

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

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

Le tri bulle

```
// Trie le tableau reçu en paramètre (via un tri bulle).
algorithme triBulle(tab↓↑ : tableau de n entiers)
  indiceBulle, i : entiers
  pour indiceBulle de 0 à n-1 faire
    pour i de n - 2 à indiceBulle par -1 faire
      si tab[i] > tab[i + 1] alors
        swap( tab[i], tab[i + 1] )           // voir algorithme précédent
      fin si
    fin pour
  fin pour
fin algorithme
```


Annexe B

Bonnes (et mauvaises) pratiques

Au fil des années, nous voyons souvent les étudiants écrire les mêmes bouts de code qui sont discutables, voire à proscrire. Discutons ici des plus courants.

Tester un booléen

Lorsqu'il s'agit de comparer deux nombres dans la condition d'un **si**, vous écrivez tous assez naturellement : **si** ($\text{nb1} < \text{nb2}$) ...

Il ne vous viendrait pas à l'idée d'écrire : **si** ($\text{nb1} < \text{nb2} = \text{vrai}$) ...

Avec un booléen, les choses sont moins simples. Par exemple, si on doit faire quelque chose si la variable **adulte** est vraie, vous êtes nombreux à écrire le bout de code ci-contre, en tout cas en début d'année.

```
si adulte = vrai alors
| ...
fin si
```



Cette écriture est correcte mais inutilement lourde. La variable étant déjà un booléen qui vaut vrai ou faux, il suffit d'écrire :

```
si adulte alors
| ...
fin si
```



De même, si le test doit être vrai quand la variable booléenne est fausse, il suffit d'écrire :

```
si NON adulte alors
| ...
fin si
```



On comprend que la première écriture vous vienne d'abord et on la tolérera les premières séances mais vous devrez vite vous en débarrasser ; elle ne sera pas tolérée lors des évaluations.

Assigner un booléen

Lorsqu'il s'agit de donner une valeur à un booléen, on vous voit souvent écrire un si-sinon. Ce n'est pas la bonne idée.

Par exemple, si la variable booléenne `nul` doit valoir `vrai` si un nombre est égal à 0 et `faux` sinon, vous êtes nombreux à écrire le bout de code ci-contre.

```
si nb = 0 alors
  nul ← vrai
sinon
  nul ← faux
fin si
```



C'est correct mais inutilement compliqué. Puisque on assigne `vrai` si la condition est vraie et `faux` si la condition est fausse, il suffit d'assigner la condition. Ce qui donne :

```
nul ← nb = 0
```



À nouveau, on tolérera le si-sinon les premières séances mais on attend de vous que vous passiez rapidement à une simple assignation. Ce sera sanctionné lors des évaluations.

Assigner une valeur en fonction d'une condition

Examinons l'exemple ci-contre qui assigne un `prix` qui doit être différent en fonction du droit ou non à un tarif réduit.

```
si tarifRéduit alors
  prix ← 8
sinon
  prix ← 12
fin si
```



Nous voyons régulièrement des étudiants écrire plutôt fièrement la version ci-contre en arguant qu'elle est plus courte, donc mieux.

Elle 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.

```
prix ← 12
si tarifRéduit alors
  prix ← 8
fin si
```



On vous demande de vous en tenir à la première version. La seconde sera sanctionnée.

Interrompre une boucle pour

On voit trop souvent des étudiants écrire une boucle pour et l'interrompre anticipativement en modifiant l'indice ; c'est une très mauvaise idée.

Dans l'exemple ci-contre, si une certaine condition est vraie, on donne une grande valeur à `i` pour sortir de la boucle.

```
pour i de 1 à n faire
  ...
  si uneCondition alors
    i ← n+1 // Pour interrompre la boucle
  fin si
  ...
fin pour
```



Il est totalement interdit de modifier explicitement l'indice d'une boucle, quelle qu'en soit la raison. Ce n'est pas une bonne idée en terme de lisibilité et dans certains langages c'est interdit ou cela ne fait pas ce qui est attendu.

Nous tenons absolument à ce que vous passiez par une boucle tant-que et l'utilisation d'un booléen pour contrôler la fin. Par exemple :

```
i ← 1
fini ← faux
tant que i ≤ n ET NON fini faire
  ...
  si uneCondition alors
    | fini ← vrai
  fin si
  ...
  i ← i+1
fin tant que
```



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
tant que i ≤ n ET NON uneCondition faire
  ...
  i ← i+1
fin tant que
```



Plusieurs retourner

Dans le cours, nous vous avons donné comme règle : « Un seul **retourner** par algorithme ».

Cette règle n'est pas partagée par tous les programmeurs. Il existe un courant qui tolère plusieurs **retourner** dans certains cas précis.

Un cas concret est pour interrompre anticipativement une boucle. Certains écrivent :

```
pour i de 1 à n faire
  si ... alors
    | retourner quelquechose
  fin si
fin pour
retourner autrechose
```



Une autre situation est lorsque la valeur à retourner dépend d'un test. On voit alors des bouts de code qui ressemblent à :

```
si ... alors
  | retourner quelquechose
sinon
  | retourner autrechose
fin si
```



Ces écritures seront tolérées dans ce cours (et donc non sanctionnées) mais nous ne les encourageons pas par crainte d'abus : elles ne peuvent se justifier que pour des algorithmes très courts.

Annexe C

Le LDA

Dans cette annexe nous définissons le LDA (*le Langage de Description d'Algorithmes*) que nous allons utiliser. Nous ne nous attarderons pas sur les concepts ni sur certaines bonnes pratiques ; tout cela est vu dans les chapitres associés.

Nous utilisons un pseudo-code pour nous libérer des contraintes des langages de programmation.

- ▷ Un programme est une suite de lignes ne permettant pas d'utiliser pleinement les deux dimensions de la page (pensons à la mise en page des formules).
- ▷ Certaines constructions et règles n'existent que pour simplifier le travail du compilateur et/ou accélérer le code. C'est le cas, par exemple, de la syntaxe du *selon que*.

Dans vos réflexions, brouillons, premiers jets, nous vous encourageons à utiliser des notations qui vous sont propres et qui vous permettent de poser votre réflexion sur un papier et d'avancer vers une solution.

La version finale, toutefois, doit être lue par d'autres personnes. Il est **essentiel** qu'il n'y ait aucune ambiguïté sur le sens de votre écrit. C'est pourquoi, nous devons définir une notation à la fois souple et précise.

Cette notation doit aussi être adaptée à des étudiants de première année. Ce qui nous amène à ne pas introduire des nuances qui leur échappent encore et, parfois, à imposer des contraintes qui seront relâchées plus tard mais qui permettent de cadrer l'apprentissage d'un débutant.

Remarque : Ce guide n'est pas universel. En dehors de l'école, d'autres notations sont utilisées, parfois proches, parfois plus lointaines. Votre professeur pourra également introduire quelques notations qui ne sont pas reprises ici ou relâcher quelques contraintes définies ici. Lorsque vous changerez de professeur, soyez conscient que ces ajouts ne seront peut-être plus valables.

L'important est que le groupe qui doit communiquer au moyen d'algorithmes se soit préalablement mis d'accord sur des notations.

Un algorithme

```
algorithme nom(paramètres) → Type
|
Instructions
|
retourner expression
fin algorithme
```

```
algorithme nom(paramètres)
|
Instructions
fin algorithme
```

On permet l'utilisation du raccourci **algo**.

Types, variables et constantes

Les types permis sont : entier, réel, booléen et chaîne.

```
constante nom = valeur
var1, ... : Type
```

Les instructions de base

```
var ← expression
demander var1, var2...
afficher expression1, expression2...
erreur "raison" // Provoque l'arrêt de l'algorithme.
```

Les instructions de choix

<pre>si condition alors Instructions fin si</pre>	<pre>si condition alors Instructions sinon Instructions fin si</pre>	<pre>si condition alors Instructions sinon si condition alors Instructions sinon si condition alors ... sinon Instructions fin si</pre>
<pre>selon que expression vaut liste₁ de valeurs séparées par des virgules : Instructions liste₂ de valeurs séparées par des virgules : Instructions ... liste_k de valeurs séparées par des virgules : Instructions autres : Instructions fin selon que</pre>		

où l'expression peut être de type entier ou chaîne (pas de réel) et les valeurs sont des constantes.

Les instructions de répétition

<pre>tant que condition faire Instructions fin tant que</pre>	<pre>pour indice de début à fin par pas faire Instructions fin pour</pre>	<pre>faire Instructions tant que condition</pre>
--	---	--

- ▷ La boucle **pour** ne peut être utilisée que pour des entiers.
- ▷ Le **par** pas peut être omis si le pas vaut 1.
- ▷ Il n'est **pas nécessaire** de déclarer l'indice. Il ne peut être utilisé en dehors de la boucle et ne peut pas être modifié à l'intérieur de la boucle. De même, le **début**, la **fin** et le **pas** ne peuvent pas être modifiés dans la boucle.

Les tableaux

Le type tableau se note `tableau de n entiers`. Ainsi une déclaration ressemble à :

```
tab : tableau de n entiers
```

Les éléments se notent `tab[0]` à `tab[n-1]`. On remarque que nous avons choisi de faire commencer les tableaux à 0 par défaut mais on peut choisir un autre début en écrivant :

```
tab : tableau [1 à n] de entiers
```

On peut assigner toutes les valeurs d'un tableau via une notation compacte.

```
tab1 ← {1,1,2,3,5,8,13}
tab2 ← {1,...,1}
```

Les structures

Une structure se définit ainsi :

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

On peut alors déclarer naturellement une variable structurée.

```
var : NomDeLaStructure
```

Un champ s'accède grâce à la notation pointée : `var.nomChamp`.

Pour assigner une valeur à tous les champs d'une variable structurée, on peut assigner chaque champ séparément avec la notation pointée ou utiliser une notation compacte.

```
var ← {valeurChamp1, valeurChamp2, ..., valeurChampN}
```


Annexe D

Aide mémoire

Cet aide-mémoire peut vous accompagner lors d'une interrogation ou d'un examen. Il vous est permis d'utiliser ces méthodes sans les développer. Par contre, si vous sentez le besoin d'utiliser une méthode qui n'apparaît pas ici, il faudra en écrire explicitement le contenu.

Manipuler les nombres

hasard(*n* : entier) → entier

Donne un entier entre 1 et *n*.

Manipuler les chaînes

Remarque : lorsqu'on indique **caractère**, on signifie une chaîne de longueur 1.

chaîne[*i*]

Désigne le *i*^e caractère de la chaîne (en commençant à 1).

Ex : `texte[2] ← "a"` ou **afficher** `texte[1]`

chaîne1 + chaîne2

Produit une chaîne qui est la concaténation des deux chaînes.

long(chaîne : chaîne) → entier

Donne la longueur de la chaîne (nb de caractères).

estLettre(car : caractère) → booléen

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

estMinuscule(car : caractère) → booléen

Permet de savoir si le caractère est une lettre minuscule.

estMajuscule(car : caractère) → booléen

Permet de savoir si le caractère est une lettre majuscule.

estChiffre(car : caractère) → booléen

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

majuscule(texte : chaîne) → chaîne

Retourne une chaîne où toutes les lettres du texte ont été converties en majuscules.

minuscule(texte : chaîne) → chaîne

Retourne une chaîne où toutes les lettres du texte ont été converties en minuscules.

numLettre(car : caractère) → entier

Retourne toujours un entier entre 1 et 26. Par exemple `numLettre("E")` donnera 5, ainsi que `numLettre("e")`. Cette fonction traite donc de la même manière les majuscules et les minuscules. `numLettre` retournera aussi 5 pour les caractères "é", "è", "ê", "ë...". Attention, il est interdit d'utiliser cette fonction si le caractère n'est pas une lettre!

lettreMaj(n : entier) → caractère

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".

lettreMin(n : entier) → caractère

Idem pour les minuscules.

chaîne(n : réel) → chaîne

Transforme un nombre en chaîne. Ex : `chaîne(42)` retourne la chaîne "42" et `chaîne(3,14)` donnera "3,14".

nombre(ch : chaîne) → réel

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.

sousChaîne(ch : chaîne, pos : entier, long : entier) → chaîne

Permet d'extraire une portion d'une certaine longueur d'une chaîne donnée, et ceci à partir d'une position donnée.

position(ch : chaîne, sous-chaîne : chaîne) → entier

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. Par exemple, `position("algorithmique", "mi")` retournera 9. Si la sous-chaîne ne s'y trouve pas, la fonction retourne 0.