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 :

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

Document produit avec LaTeX.
Version du 6 mai 2015.

Table des matières

Première partie

Introduction aux algorithmes

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.*

1.1 La notion de problème

1.1.1 Préliminaires : utilité de l'ordinateur

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

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

1.1.2 Poser le problème

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

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 se pose le problème :** 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 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. ».
Qu'est-ce qui vous parait flou dans cet énoncé ?

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

1.2 Procédure de résolution

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

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

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

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

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

1.2.1 Chronologie des opérations

Pour ce qui concerne l'ordinateur, le travail d'exécution d'une marche à suivre est impérativement **séquentiel**. C'est-à-dire que les instructions d'une procédure de résolution sont

exécutées **une et une seule fois** dans l'ordre où elles apparaissent dans le code. Cependant certains artifices d'écriture permettent de **répéter** l'exécution d'opérations ou de la **conditionner** (c'est-à-dire de choisir si l'exécution aura lieu oui ou non en fonction de la réalisation d'une condition).

1.2.2 Les opérations élémentaires

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

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

1.2.3 Les opérations bien définies

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

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

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

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

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

3. Chacun cache à présent son propre dessin et échange sa feuille d'instructions avec celle de quelqu'un d'autre.
4. Chacun s'efforce ensuite de reproduire le dessin d'un autre en suivant **scrupuleusement** les instructions indiquées sur la feuille reçue en échange, **sans tenter d'initiative** (par exemple en croyant avoir compris ce qu'il faut dessiner).
5. Nous examinerons enfin les différences entre l'original et la reproduction et nous tenterons de comprendre pourquoi elles se sont produites (par imprécision des instructions ou par mauvaise interprétation de celles-ci par le dessinateur...)

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

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

1.2.4 Opérations soumises à une condition

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

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

1.2.5 Opérations à répéter

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

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

1.2.6 À propos des données

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

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

1.3 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 multicouleur 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.

3.1 Algorithmes et programmes

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

3.1.1 Algorithme

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

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

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

3.1.2 Programme

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

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

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

3.1.3 Les constituants principaux de l'ordinateur

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

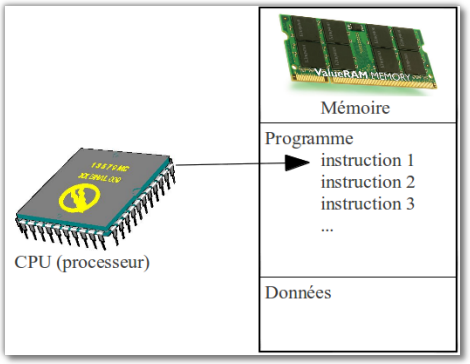
Le **hardware** est constitué de l'ordinateur proprement dit et regroupe les entités suivantes :

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

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

3.1.4 Exécution d'un programme

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



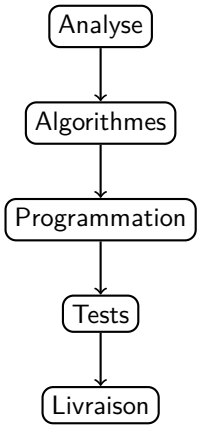
Comment fonctionne le processeur ?
De façon très simplifiée, on passe par les étapes suivantes :

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

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

3.2 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 paraitra difficile en début d'année vous paraitra (enfin, il faut l'espérer !) une évidence en fin d'année.

3.3 Conclusion

L'informatisation de problèmes est un processus essentiellement dynamique, contenant des allées et venues constantes entre les différentes étapes. Codifier un algorithme dans un langage de programmation quelconque n'est certainement pas la phase la plus difficile de ce

processus. Par contre, élaborer une démarche logique de résolution d'un problème est probablement plus complexe.

Le but 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 récursifs
- ▷ les algorithmes liés au traitement des structures de données particulières telles que listes, files d'attente, piles, arbres, graphes, tables de hachage, etc.

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

3.4 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

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.

4.1 Déterminer les données et le résultat

La toute première étape est de parvenir à extraire d'un énoncé de problème, quelles sont les données et quel est le résultat 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.

4.2 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 ambigüité 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.

4.3 Les types

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

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

entier	pour les nombres entiers
réel	pour les nombres réels
chaine	pour les chaines 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 chaine.
- ▷ 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 chaine (par ex : "mars") car les manipulations, les calculs seront plus simples.

4.3.1 Il n'y a pas d'unité

Un type numérique indique que les valeurs possibles seront des nombres. Il n'y a là aucune notion d'unité. Ainsi, la longueur d'un rectangle, un réel, peut valoir 2.5 mais certainement pas $2.5cm$. 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'expliciter en la nommant longueurCM.

4.3.2 Préciser les valeurs possibles

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.

4.3.3 Le type des données complexes

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

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

On pourrait la représenter dans une chaine (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.

4.3.4 Exercice

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

- ▷ le prix d'un produit en grande surface ?
- ▷ la taille de l'écran de votre ordinateur ?
- ▷ votre nom ?

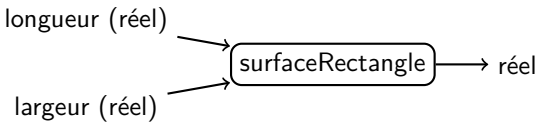
3. L'orienté objet que vous verrez en Java le permet également et même mieux.

- ▷ votre adresse ?
- ▷ le pourcentage de remise proposé pour un produit ?
- ▷ une date du calendrier ?
- ▷ un moment dans la journée ?

4.4 Résumé graphique

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

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

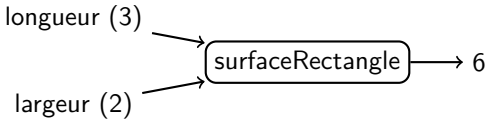


4.5 Exemples numériques

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 6.

4.6 Exercices

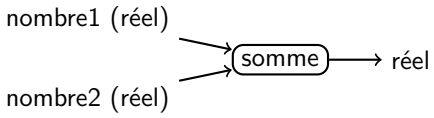
Pour les exercices suivants, nous vous demandons d'imiter la démarche décrite dans ce chapitre, à savoir :

- ▷ 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.

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.

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.

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 14 km/h.

12 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.

13 Somme des chiffres

Calculer la somme des chiffres d'un nombre entier de 3 chiffres.

14 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.

15 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.

16 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.

17 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.

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*).

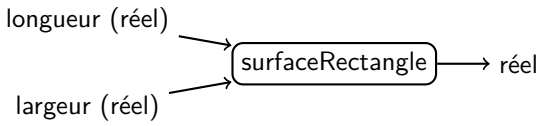
5.1 Un problème simple

5.1.1 Trouver l'algorithme

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

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

Analyse. Nous sommes arrivés à la spécification suivante :



Exemples.

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

Comment résoudre ce problème ? La toute première étape est de comprendre comment passer des données au résultat. Ici, on va se baser sur la formule de la surface :

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

La surface s'obtient donc en multipliant la longueur par la largeur [1].

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 : réel) → 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 (nous verrons comment) 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.

5.1.2 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 n'est indispensable que 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.

5.1.3 Exercices

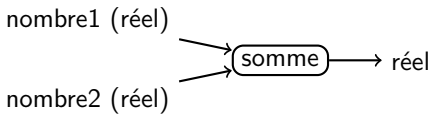
Les exercices suivants ont déjà été analysés dans un précédent chapitre et des exemples numériques ont été choisis. Il ne vous reste plus qu'à écrire l'algorithme et à le vérifier pour les exemples choisis.

Vous pouvez vous baser sur la fiche 1 page 99 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 obtenus lors de la phase d'analyse du problème.



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

```
algorithme somme(nombre1, nombre2 : réel) → 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	✓
1	3	2.5	5.5	5.5	✓
2	-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.

5.2 Décomposer les calculs

Dans les exercices de la section précédente, vous avez écrit quelques longues formules [2]. 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*.

5.2.1 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 déclarée [3].

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 abrégerons par :

```
longueur, largeur : réels
```

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

5.2.2 L'assignation

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

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

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.

```
nomVariable ← expression
```

Rappelons qu'une **expression** est un calcul, une combinaison de variables et d'opérateurs. Une expression a une valeur.

Exemples d'assignations correctes.

```
somme ← nombre1 + nombre2
denRes ← den1 * den2
cpt ← cpt + 1
test ← a < b                                          // pour une variable logique
maChaine ← "Bon"
```

Exemples d'assignations incorrectes

```
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 chaine dans une variable booléenne.

5.2.3 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 de verifier qu'elles contiennent bien à tout moment la valeur attendue.

Exemple. Traçons des bouts d'algorithmes.

```
1: a, b, c : entiers
2: a ← 12
3: b ← 5
4: c ← a - b
5: a ← a + c
6: b ← a
```

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

```
1: a, b, c : entiers
2: a ← 12
3: c ← a - b
4: d ← c - 2
```

#	a	b	c
1	indéfini	indéfini	indéfini
2	12	|	|
3	|	|	? ? ?
4	|	|	? ? ?

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

11 Tracer des bouts de code

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

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

#	a	b	c
1			
2			
3			
4			
5			
6			
7			

```
1: a, b, c : entiers
2: a ← 2
3: b ← a^3
4: c ← b - a^2
5: a ← √c
6: a ← a / a
```

#	a	b	c
1			
2			
3			
4			
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 vitesseKMH(distanceM, duréeS : réel) → réel
2:     distanceKM, duréeH : réel
3:     distanceKM ← 1000 * distanceM
4:     duréeH ← 3600 * duréeS
5:     retourner distanceKM / 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.

5.2.4 Exercices de décomposition

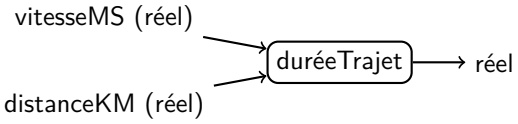
Savoir, face à un cas concret, s'il est préférable de décomposer le calcul ou pas, n'est pas toujours évident. La section 5.5 page 40 sur la lisibilité vous apportera des arguments qui permettront de trancher.

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 101 qui présente un exemple complet.

13 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.

Solution. L'analyse du problème aboutit à :



La formule qui lie les trois éléments est :

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

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 exemple numériques :

▷ duréeeTrajet(1, 1) donne 1000 ▷ duréeeTrajet(0.5, 0.2) donne 400

L'algorithme s'écrit :

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

Vérifions l'algorithme pour duréeeTrajet(1, 1)

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

et pour duréeeTrajet(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 14 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.

5.3 Quelques difficultés liées au calcul

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

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

Examinons ces situations une à une en fournissant des exemples et des exercices pour que cela devienne naturel pour vous.

5.3.1 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 parait 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 ← sont des variables booléennes) :

- ▷ positif ← nb>0 // positif est mis à vrai si le nb est positif
- ▷ adulte ← âge≥21 // adulte est vrai si l'âge est 21 ou plus
- ▷ réussi ← cote≥10 // réussi est mis à vrai si la cote est supérieure ou égale à 10
- ▷ parfait ← nbFautes=0 // c'est parfait si le nombre de fautes est 0

16 **Écrire des expressions booléennes**

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

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

5.3.2 Les opérations logiques

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

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 :

- ▷ tarifPlein ← 18≤âge ET âge<60
- ▷ distinction ← 16≤cote ET cote<18
- ▷ nbA3chiffres ← 100≤nb ET nb≤999
- ▷ tarifRéduit ← NON tarifPlein
- ▷ tarifRéduit ← NON (18≤âge ET âge<60)
- ▷ tarifRéduit ← âge<18 OU 60≤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 ambigus [5].
Par exemple, ne pas écrire : tarifRéduit ← âge<18 OU ≥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 : tarifRéduit ← NON (18≤âge ET âge<60)
peut s'écrire aussi : tarifRéduit ← (NON 18≤âge) OU (NON âge<60)
ce qui se simplifie en : tarifRéduit ← âge<18 OU 60≤âge

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

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

17 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.

- ▷ ok ← adulte = vrai
- ▷ ok ← adulte = faux
- ▷ ok ← etudiant = vrai ET jeune = faux
- ▷ ok ← NON (adulte = vrai) ET NON (adulte = faux)
- ▷ nbA3chiffres ← NON (nb<100 OU nb≥1000)

18 Expressions logiques

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

- ▷ J'irai au cinéma si le film me plait et que j'ai 20€ en poche.
- ▷ Je n'irai pas au cinéma si je n'ai pas 20€ en poche.
- ▷ Je brosserai 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, compta).

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

5.3.3 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
- ▷ 8 DIV 2 vaut 4
- ▷ 6 DIV 6 vaut 1
- ▷ 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
- ▷ 8 MOD 2 vaut 0
- ▷ 6 MOD 6 vaut 0
- ▷ 6 MOD 7 vaut 6

Pour indiquer le lien entre la division et le reste, on écrira : $9/2 = 4$ reste 1.

19 Calculs

Voici quelques petits calculs. On vous demande de remplir les trous.

- ▷ 11/3 = ___ reste ___
- ▷ 3/11 = ___ reste ___
- ▷ 11/___ = 2 reste 3
- ▷ ___/3 = 3 reste 2

20 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.

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

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

Extraire les chiffres d'un nombre

Faisons une petite expérience numérique.

calcul	résultat
65536 MOD 10	6
65536 MOD 100	36
65536 MOD 1000	536
65536 MOD 10000	5536

calcul	résultat
65536 DIV 10	6553
65536 DIV 100	655
65536 DIV 1000	65
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 droites (DIV). Combinés, ils permettent d'extraire n'importe quel chiffre d'un nombre.

Exemple : 65536 DIV 100 MOD 10 = 3.

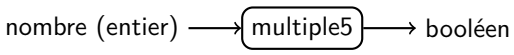
5.3.4 Exercices sur les difficultés de calcul

Les exercices qui suivent n'ont pas tous étés 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.

21 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.



Exemples.

- ▷ multiple5(4) donne faux
- ▷ multiple5(15) donne vrai
- ▷ multiple5(0) donne vrai
- ▷ multiple5(-10) 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	✓

22 Nombre entier positif se terminant par un 0

Calculer si un nombre donné se termine par un 0.

23 Les centaines

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

24 Somme des chiffres

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

25 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.

26 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.

27 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.

28 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.

5.4 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.

5.4.1 L'efficacité

L'**efficacité** désigne le fait que l'algorithme (le programme) résout [6] bien le problème donné. C'est un minimum !

5.4.2 La lisibilité

La **lisibilité** indique si une personne qui lit l'algorithme (ou le programme) peut facilement percevoir comment il fonctionne. C'est crucial car un algorithme (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 rencontrée [7] ;
- ▷ 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 quelques 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 le 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.

5.4.3 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'*efficience* qui traite de la gestion économe des ressources. Nous reparlerons de rapidité dans le chapitre consacré à la *complexité* des algorithmes.

5.4.4 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.

6. À ne pas confondre avec *l'efficience* qui indique qu'il est économe en ressources.
7. On parle du processus de *déverminage* (ou *debugging* en Anglais).

5.4.5 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 re-travailler pour améliorer sa lisibilité [8].

5.5 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 ?

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 gras [9]).
- ▷ 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.

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.

5.5.1 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 ; 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).
- ▷ 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'apparait 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)
```

8. 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.

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

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 mettre des commentaires au-dessus du programme pour expliquer ce qu'il fait et re-travaillez votre algorithme pour que tout commentaire à l'intérieur de l'algorithme devienne superflu.**

Exemple. Voici comment on pourrait commenter 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 : réel) → réel
    retourner longueur * largeur
fin algorithme
```

29 Commenter la durée du trajet

Commentez l'algorithme qui calcule la durée d'un trajet (exercice 13 page 33).

5.5.2 Constantes

Une **constante** est une information pour laquelle nom, type et valeur sont figés. La liste des constantes utilisées dans un algorithme apparaitra dans la section déclaration des variables [10] sous la forme suivante [11] :

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

30 Utiliser une constante

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

5.6 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 connait la longueur et la largeur. Mais d'où viennent ces données ? Et que faire du résultat ?

10. Ou en dehors des algorithmes s'il s'agit d'une constante universelle partagée par plusieurs algorithmes.
11. L'usage est d'utiliser des noms en majuscule.

Tout d'abord, un algorithme peut utiliser (on dit **appeler**) un autre algorithme [12]. 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...

5.6.1 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 ici [13].

5.6.2 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)
```

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 et écrire :

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

5.6.3 Algorithme sans paramètre

Les paramètres d'un algorithme sont destinés aux autres algorithmes qui vont l'utiliser. Ils ne sont pas présents lorsque les données sont précisées par l'utilisateur. Idem pour la valeur

12. 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.
13. Ce sera bien sûr une question importante quand il s'agira de traduire l'algorithme en un programme.

de retour. Au final, on pourrait écrire :

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

Cet algorithme n'a ni paramètre ni valeur de retour mais il fait appel (utilise) un autre algorithme.

5.6.4 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.

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 ».

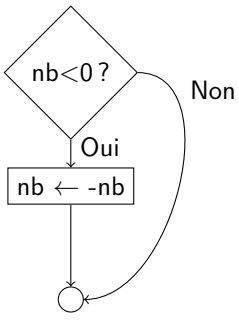
Les **alternatives** permettent de n'exécuter des instructions que si une certaine *condition* est vérifiée. Avec le zombie, vous testiez son environnement ; dans nos algorithmes, vous allez tester les données.

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 ordres. Les alternatives permettent de créer des histoires différentes, d'adapter les instructions aux valeurs concrètes des données. Procédons par étapes.

6.1 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, graphiquement [1] et en LDA :




```
si nb<0 alors
    nb ← -nb
fin si
```

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

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.

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 : entier) → 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.

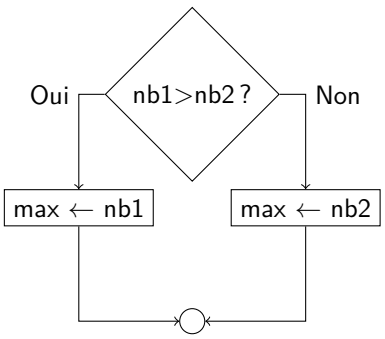
```
si ok = vrai alors
    afficher nombre
fin si
```

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

6.2 Le si-sinon

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 pendre la valeur du premier nombre ou du second selon que le premier est plus grand que le second ou pas.




```
si nb1<nb2 alors
    max ← nb1
sinon
    max ← nb2
fin si
```

3 Compréhension

Tracez ces algorithmes et donnez la valeur de retour.

```
algorithme exerciceB(b, a : entier) → 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 : entier) → 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 105.

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 $f(n) = \begin{cases} n/2 & si\ n\ est\ pair \\ 3n+1 & si\ n\ est\ impair \end{cases}$

8 Tarif réduit ou pas

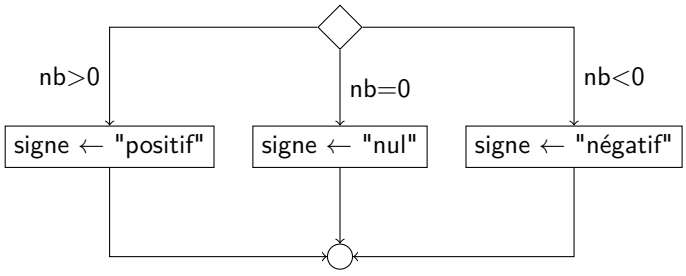
Dans une salle de cinéma, le tarif plein pour une place est de 8€. Les personnes ayant droit au tarif réduit payent 7€. Écrivez un algorithme qui reçoit un booléen indiquant si la personne peut bénéficier du tarif réduit et qui retourne le prix à payer.

6.3 Le si-sinon-si

Partons à nouveau d'un exemple pour illustrer cette instruction.

Exemple. On voudrait mettre dans la chaine 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.




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

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 donnés 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 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.

6.4 Le selon-que

Cette nouvelle instruction permet d'écrire plus lisiblement *certains* **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.

```
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
```

```
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
```

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
 liste_1 de valeurs séparées par des virgules :
    Instructions
 liste_2 de valeurs séparées par des virgules :
    Instructions
    ...
 liste_k de valeurs séparées par des virgules :
    Instructions
 autres :
    Instructions
fin selon que
```

13 Numéro du jour

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

14 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.

6.5 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.

15 Réussir DEV1

Pour réussir l'UE (unité d'enseignement) GEN1, 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 à réussi cette UE.

16 Réussir GEN1

Pour réussir l'unité d'enseignement GEN1, il faut que la cote attribuée à chaque AA (activité d'apprentissage) soit supérieure ou égale à 50%. 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. S'il a réussi, l'algorithme affiche également sa moyenne (cherchez quelle est la pondération entre ces AA).

17 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 ?

18 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.

19 Valider une date

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

20 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.

Chapitre 7

Décomposer le problème

7.1 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.

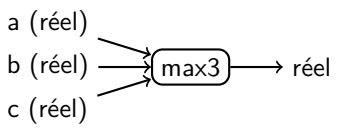
Dans la réalité, les problèmes sont plus gros et il devient nécessaire de les décomposer en sous-problèmes. On parle d'une *approche modulaire*. Les avantages d'une telle décomposition sont multiples.

- ▷ **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 apparait 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 accroit 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.

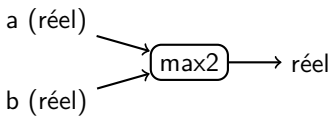
Parmis les autres avantages, que vous pourrez moins percevoir en début d'apprentissage, citons la possibilité de répartir le travail dans une équipe.

7.2 Exemple

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



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

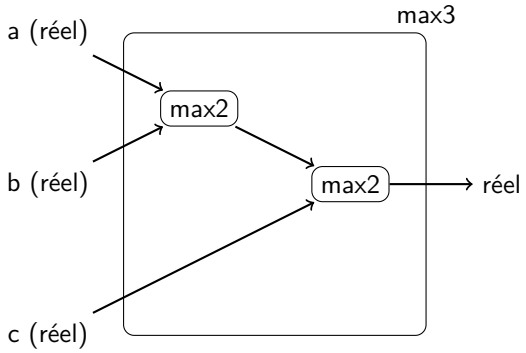



```
algorithme max2(a : réel, b : réel) → réel
    max : réel
    si a > b alors
        max ← a
    sinon
        max ← b
    fin si
    retourner max
fin algorithme
```

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, comment faire concrètement pour introduire le calcul du maximum de 2 nombres dans l'algorithme calculant le maximum de 3 nombres ? Une solution consiste à « copier-coller » les lignes de max2 dans max3, en adaptant son contenu au contexte : maxab est calculé et ré-utilisé dans un calcul ultérieur. Ceci donnerait :

```
algorithme max3(a : réel, b : réel, c : réel) → réel
    maxab, max : réels
    si a > b alors
        maxab ← a
    sinon
        maxab ← b
    fin si
    si maxab > c alors
        max ← maxab
    sinon
        max ← c
    fin si
    retourner max
fin algorithme
```

Bien que correcte, cette démarche est cependant déconseillée et peut s'avérer fastidieuse, dangereuse et contre-productive.

- ▷ Imaginons qu'il eût fallu de cette façon « mixer » deux algorithmes d'une cinquantaine de lignes chacun. Le résultat serait un algorithme d'une centaine de lignes qui perdrait en lisibilité et clarté.
- ▷ De plus, l'opération effectuée n'est pas sans risques : que se passerait-il si les deux algorithmes contiennent chacun une variable nommée de manière identique ? Cette « transplantation » demande donc la vérification de toutes les variables utilisées, la réécriture des lignes de déclarations de variables de façon à y inclure celles du module « greffé », etc.

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

Le mieux, est d'utiliser (d'appeler) l'algorithme max2 dans max3.

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

7.3 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.

7.3.1 Le paramètre en entrée

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

Lors de l'appel, on fournit la **valeur** ou, plus généralement une expression dont la valeur sera donnée au paramète.

Illustration. Voici un cas général de paramètre en entrée.

```
// Code appelant
algo(expr)
```

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

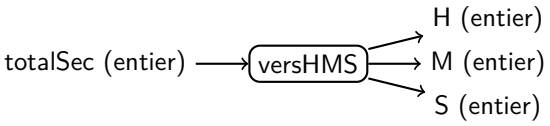
C'est comme si l'algorithme monAlgo commençait pas l'affectation par ← expr.

7.3.2 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 (↑) pour rappeler son rôle. Un tel paramètre n'aura pas de valeur au début de l'algorithme mais s'en verra attribuée une par l'algorithme.

Lors de l'appel, on fournit une **variable** qui recevra la valeur finale du paramètre.

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 :




```
algorithme versHMS(totalSec↓ : entier, H↑ : entier, M↑ : entier, S↑ : entier)
    H ← totalSec DIV (60*60)
    M ← totalSec MOD (60*60) DIV 60
    S ← totalSec MOD 60
fin algorithme
```

Il n'y a donc pas de **retourner** puisque les résultats sont en paramètres de sortie et pas comme valeur *retournée*. Un appel possible pourrait être :

```
versHMS(65536, heure, minute, seconde)
```

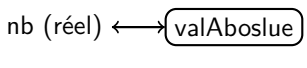
C'est comme si, à la fin de l'algorithme versHMS, on avait les assignations suivantes : heure ← H, minute ← M et seconde ← S.

7.3.3 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 (↓↑).

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.

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 :




```
algorithme valAbsolue(nb↓↑ : réel)
    si nb<0 alors
        nb ← -nb
    fin si
fin algorithme
```

Un appel possible pourrait être :

```
valAbsolue(température)
```

C'est comme si, dans valAbsolue, on avait une première ligne pour donner sa valeur au paramètre (nb ← température) et une dernière ligne pour effectuer l'assignation opposée (nb ← température).

7.4 La valeur de retour

Une valeur de retour est toujours possible, mais jamais obligatoire, quels que soient les sortes des 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 →) n'a pas de valeur ; il ne peut pas apparaître dans une expression ou être assigné à une variable.

7.5 Résumons

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

```
algorithme test()
    afficher f(pE1, pE2, pE3)
fin algorithme

algorithme f(pF1↓ : entier, pF2↑ : entier, pF3↓↑ : entier) → entier
    pF1 ← pE1
    pF3 ← pE3

    // Le code proprement dit de l'algorithme

    pE2 ← pF2
    pE3 ← pF3
    retourner réponse
fin algorithme
```

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'algorihtme 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 ↑ et ↓↑ en fin d'algorithme.

On comprend pourquoi pE1 peut être une expression quelconque mais que pE2 et pE3 doivent être des variables, puisqu'elle vont recevoir une valeur.

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.

TODO : Si quelqu'un se sent de faire un schéma plus parlant que ce bla-bla...

7.6 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
    addition(b, a, b)
    afficher 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] lire 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 Validité d'une date

Vous avez déjà écrit un algorithme qui vérifie la validité d'une date. Revenez-y et voyez comment vous pouvez le décomposer pour accroitre sa lisibilité et créer des modules utiles et réutilisables.

6 Réussir GEN1

Reprenons l'exercice 16 page 51. 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).

7 Généraliser un algorithme

Dans l'exercice 21 page 37, 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. Nous voyons ici comment incorporer des boucles dans nos codes et comment les utiliser à bon escient.

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

8.1 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.

8.2 Une même instruction, des effets différents

Comprenez bien que c'est toujours la même tâche qui est exécutée mais pas avec le même effet à chaque fois. Ainsi, 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.

8.2.1 Exemple - Afficher les nombres de 1 à 5

Sans 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 boucle [1] ; il va falloir ruser. On peut obtenir le même résultat avec l'algorithme suivant :

```
nb ← 1
afficher nb
nb ← nb + 1
afficher nb
nb ← nb + 1
afficher nb
nb ← nb + 1
afficher nb
nb ← nb + 1
afficher nb
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 exactement cinq copies. Le travail à répéter est donc :

```
afficher nb
nb ← nb + 1
```

Il existe plusieurs structures répétitives qui vont se distinguer par la façon dont on va contrôler le nombre de répétitions. Voyons-les une à une.

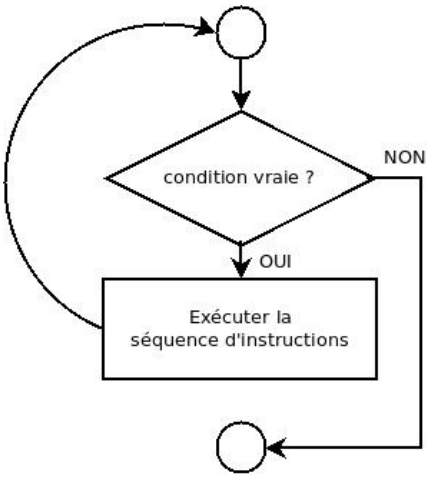
8.3 « tant que »

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
```

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 faillait afficher les nombres de 1 à 1000 ?

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



8.3.1 Exemple - Afficher les nombres de 1 à 5

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

```
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) :

```
nb ← 1
tant que nb ≤ 5 faire
    afficher nb
    nb ← nb + 1
fin tant que
```

8.3.2 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
```

8.3.3 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
    c : pt, 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 les n premiers

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

a) les nombres de 1 à n ;

b) les nombres de 1 à n en ordre décroissant ;

c) les nombres impairs de 1 à n ;

d) les n premiers nombres impairs (attention, c'est un peu plus difficile).

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

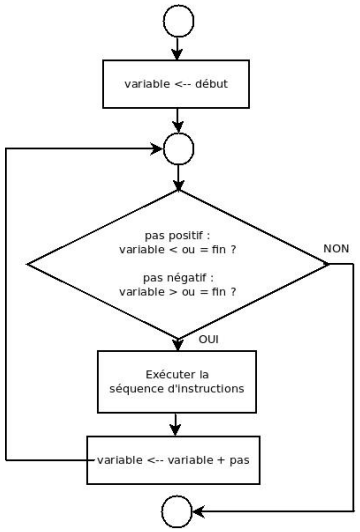
8.4 « pour »

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). Ce pas est parfois négatif, dans le cas d'un compte à rebours, par exemple pour n de 10 à 1 par -1.

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



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.
```

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

La variable de contrôle ne servant que pour la boucle et étant forcement entière, on va considérer qu'il n'est pas nécessaire de la déclarer et qu'elle n'est pas utilisable en dehors de la boucle [2].

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

8.4.1 Exemples – Afficher des nombres

Restons avec notre exemple d'affichage des nombres de 1 à 5.

```
algorithme compterJusque5()
    pour nb de 1 à 5 faire                                  // par défaut le pas est de 1
        afficher nb
    fin pour
fin algorithme
```

Si on veut généraliser à n nombres, on a :

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

Et si on veut afficher un compte à rebours :

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

8.4.2 Exemple – Afficher uniquement les nombres pairs

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

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

Notez que n peut être impair. Si n vaut 11, l'affichage est le même que pour 10. On peut utiliser un « pour ». De 1 à n, il y a exactement « n DIV 2 » nombres à afficher. La difficulté vient du lien à faire entre la variable de contrôle et le nombre à afficher.

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

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

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

```
algorithme afficherPair(n : entier)
    pour i de 1 à n DIV 2 faire
        afficher 2 * i
    fin pour
fin algorithme
```

Par une vieille habitude des programmeurs[3], une variable de contrôle qui se contente de compter les passages dans la boucle est souvent nommée i. On l'appelle aussi « itérateur ».

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

8.4.3 Exemple – Afficher les premiers nombres pairs

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

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

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

```
algorithme afficherPair(n : entier)
    pour i de 1 à n faire
        afficher 2 * i
    fin pour
fin algorithme
```

8.4.4 Exercices

4 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
```

5 Afficher les n premiers

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 n premiers carrés parfaits (1, 4, 9...) ;

d) les nombres impairs de 1 à n ;

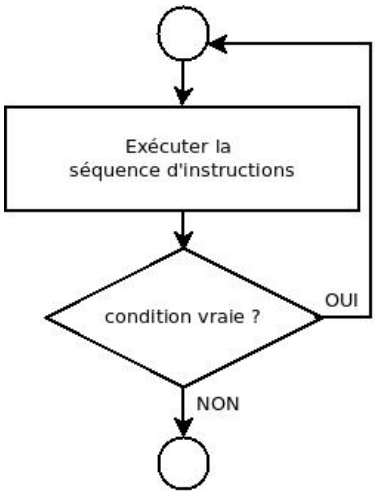
e) les n premiers nombres impairs.

8.5 « faire – tant que »

Cette structure est très proche du «faire - tant que » à ceci près que le test est fait à la fin et pas au début. 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-dessous décrit le déroulement de cette boucle.



6 Compréhension d'algorithmes

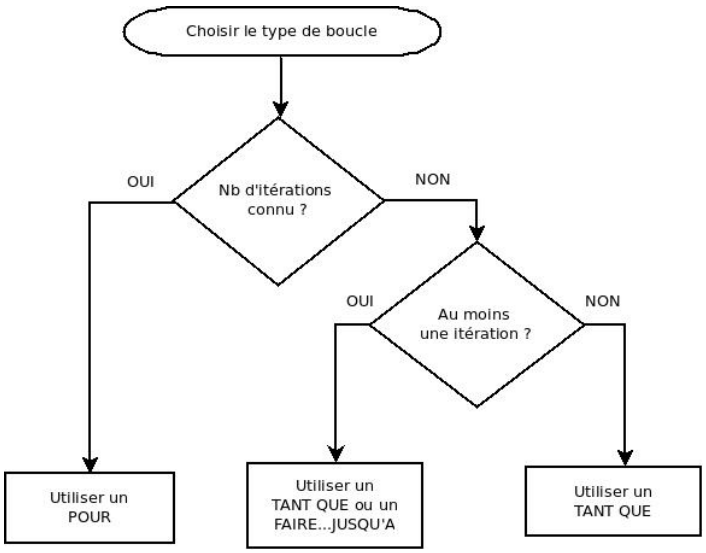
Quels sont les affichages réalisés lors de l'exécution des algorithmes suivants ?

```
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
```

8.6 Quel type de boucle choisir ?

En pratique, il est possible d'utiliser systématiquement la boucle tant que qui peut s'adapter à toutes les situations. Cependant, il est plus clair d'utiliser la boucle pour dans les cas où le nombre d'itérations est fixé et connu à l'avance (par là, 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.



8.7 Exercices récapitulatifs

7 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, $0\,! = 1$.

8 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.

9 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.

10 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.

11 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 $28 = 1 + 2 + 4 + 7 + 14$.

12 Décomposition en facteurs premiers

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

13 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.

14 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.

15 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] ».

Chapitre 9

Les tableaux

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

9.1 Utilité des tableaux

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

Exemple. Statistisques de ventes.

> Un gérant d'une entreprise commerciale souhaite connaitre l'impact d'une journée de promotion publicitaire sur la vente de dix de ses produits. Pour ce faire, les numéros de ces produits (numérotés de 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 :

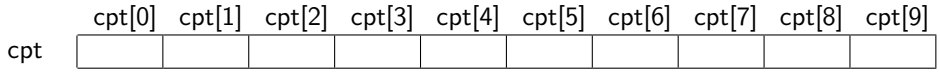
- ▷ le traitement de début de journée, qui consiste essentiellement à mettre les compteurs des quantités vendues pour chaque produit à 0
- ▷ le traitement itératif durant toute la journée : au fur et à mesure des ventes, il convient de les enregistrer, c'est-à-dire d'ajouter au compteur des ventes d'un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

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

```
// Calcule et affiche la quantité vendue de 10 produits.
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
            0 : cpt1 ← cpt10 + 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, cpt21, etc.), nous allons envisager une seule « grande » variable cpt compartimentée en dix « sous-variables » qui se distingueront les unes des autres par un indice : cpt0 deviendrait ainsi cpt[1], cpt1 deviendrait cpt[1], et ainsi de suite jusqu'à cpt9 qui deviendrait cpt[9].



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

Voici la version avec tableau.

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

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

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

fin algorithme
```

9.2 Définitions

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

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

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

9.3 Notations

Pour déclarer un tableau, on écrit :

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

où TypeElément est le type des éléments que l'on trouvera dans le tableau. Tous les types sont permis.

Une fois un tableau déclaré, seuls les éléments d'indice compris entre 0 et taille-1 peuvent être utilisés. Par exemple, si on déclare :

```
tabEntiers : tableau de 100 entiers
```

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

N.B. : Nous considérons que la première case du tableau porte le numéro 0 comme c'est le cas dans beaucoup de langage 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 ce qui sera plus naturel pour certains problèmes.

9.4 Parcours d'un tableau à une dimension

Dans la plupart des problèmes que vous rencontrerez vous serez amener à parcourir le tableau dans sa totalité ou en partie. Il est important de maitriser ce parcours. Examinons les situations courantes et voyons quelles solutions conviennent.

Soit le tableau tab déclaré ainsi

```
tab : tableau de n entiers                                    // où un autre type
```

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

9.4.1 Parcours complet

Pour parcourir complètement un tableau, on peut utiliser la boucle **pour**. Dans l'algorithme suivant, on affiche tous les élements du tableau mais on pourrait faire autre chose avec ces éléments : les sommer, les comparer...

```
// Parcours complet d'un tableau via une boucle pour
pour i de 0 à n-1 faire
    afficher tab[i]
fin pour
```

9.4.2 Parcours avec sortie prématurée

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

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

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

```
// Parcours complet d'un tableau via une boucle tant-que
i : entier
i ← 0
tant que i < n faire
    afficher tab[i]
    i ← i + 1
fin tant que
```

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

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

[A] Sans variable booléenne

```
// Parcours partiel d'un tableau sans variable booléenne
i : entier
i ← 0
tant que i < n ET test sur tab[i] dit que on continue faire
    i ← i + 1
fin tant que
si i = n alors
    // on est arrivé au bout
sinon
    // arrêt prématuré à l'indice i.
fin si
```

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.

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

Dans certains cas, le **si-sinon** peut se simplifier en un simple **return** d'une condition.

Exemple : Recherche d'un zéro dans un tableau.

```
// Indique si un zéro est présent dans le tableau
algorithme contientZéro(tab : tableau de n entiers) → booléen
    i : entier
    i ← 0
    tant que i < n ET tab[i] ≠ 0 faire
        i ← i + 1
    fin tant que
    retourner i < n                          // Si le test est vrai c'est qu'on a trouvé un 0
fin algorithme
```

[B] Avec variable booléenne

```
// Parcours partiel d'un tableau avec variable booléenne
i : entier
trouvé : booléen
i ← 0
trouvé ← faux
tant que i < n ET NON trouvé faire
    si test sur tab[i] dit que on a trouvé alors
        trouvé ← vrai
    sinon
        i ← i + 1
    fin si
fin tant que
// tester le booléen pour savoir si arrêt prématuré.
```

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

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

Troisième partie

Les algorithmes fondamentaux

Chapitre 10

Agrégation des données

On va retrouver ici tout ce qui concerne l'agrégation de données, d'un tableau essentiellement

- ▷ somme, moyenne...
- ▷ recherche de maximum/minimum
- ▷ indices du maximum/minimum
- ▷ ...

Chapitre 11

Recherche de valeurs

> On va retrouver ici tout ce qui concerne la recherche de valeur dans un tableau, trié ou pas.

Chapitre 12

Tris

> On va retrouver ici, peu ou prou le chapitre actuel sur les tris.

Chapitre 13

Acquisition des données

> On va retrouver ici tout ce qui concerne la lecture de données multiples (avec valeur sentinelle).

Chapitre 14

Les suites

> On va retrouver ici tout ce qui concerne la génération de suites.

Quatrième partie

Compléments

Chapitre 15

Les chaines

> Traiter ici de la manipulation des chaines. Cf. le syllabus de cette année qu'il faudra peut-être adapter.

Chapitre 16

Les structures

Les structures sont une préparation aux cours du second quadrimestre

- ▷ l'OO en java
- ▷ les fichiers structurés en persistance des données
- ▷ les données structurées en Cobol

Cinquième partie

Les annexes

Annexe A

Les fiches

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

Fiche n° 1 – Un calcul simple

Le problème

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

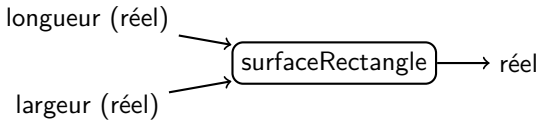
Spécification

Données

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

Toutes les données sont des réels positifs ou nuls.

Résultat. Un réel représentant la surface du rectangle



Exemples

- ▷ surfaceRectangle(4, 3) donne 12
- ▷ surfaceRectangle(2.5, 2) donne 5

Analyse de la solution

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

$$\text{surface} = \text{longueur} * \text{largeur} \tag{A.1}$$

Solution

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

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 accroitre la lisibilité (cf. fiche 2 page 101)

Fiche n° 2 – Un calcul complexe

Le problème

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

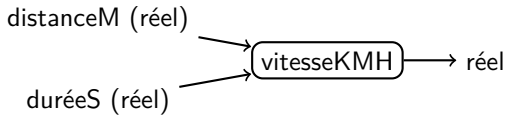
Spécification

Données

- ▷ la distance parcourue par le véhicule (en m) ;
- ▷ la durée du parcours (en s).

Toutes les données sont des réels

Résultat. Un réel représentant la vitesse du véhicule (en km/h).



Exemples

- ▷ vitesseKMH(100,10) donne 36
- ▷ vitesseKMH(10000,3600) donne 10

Analyse de la solution

La vitesse est liée à la distance et à la durée par la formule :

$$\text{vitesse} = \frac{\text{distance}}{\text{durée}} \tag{A.2}$$

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.

Solution

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

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 assignation (cf. fiche 1 page 99).

Fiche n° 3 – Un nombre (im)pair

Le problème

Un nombre reçu en paramètre est-il pair ?

Analyse

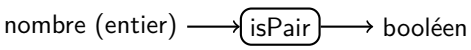
Un nombre est pair si il est multiple de 2. C'est-à-dire si le reste de sa division par 2 vaut 0.

```
estPair est vrai si nombre MOD 2 = 0
```

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.



Exemples

▷ isPair(2016) donne *vrai*

▷ isPair(2015) donne *faux*

Solution

```
algorithme isPair(nombre : entier) → booléen
    retourner nombre MOD 2 = 0
fin algorithme
```

Alternatives

```
algorithme isPair(nombre : entier) → booléen
    si nombre MOD 2 = 0 alors
        retourner vrai
    sinon
        retourner faux
    fin si
fin algorithme
```

Certains étudiants se sentent plus à l'aise avec la solution ci-contre en début d'année. C'est probablement parce qu'elle colle plus à la façon de l'exprimer en français. On les encourage toutefois à rapidement passer à la version plus compacte et, une fois habitué, plus lisible.

```
algorithme isPair(nombre : entier) → booléen
    si nombre MOD 2 = 0 alors
        retourner vrai
    fin si
    retourner faux
fin algorithme
```

On rencontre également ce genre de solution qui, pour certains, parait mieux que la précédente parce qu'elle ne contient pas de "sinon" et est donc plus courte. Il n'en n'est rien. Rappelons que la longueur de l'algorithme n'est pas, en soi, un critère de qualité.

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 101.

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

Le problème

Quel est le maximum de deux nombres ?

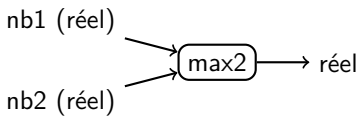
Analyse

Voilà un classique de l'algorithmique. Attention ! On ne veut pas savoir *lequel* est le plus grand mais juste la valeur. Il n'y a donc pas d'ambiguïté si les deux nombres sont égaux.

Données

- ▷ deux nombres réels

Résultat. Un réel contenant la plus grande des deux valeurs données.



Exemples

- ▷ max2(-3, 4) donne 4
- ▷ max2(7, 4) donne 7
- ▷ 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
```

Alternatives

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

En algorithmique, comme ailleurs, il existe des modes. Certaines personnes insistent pour qu'il n'y ait qu'un seul retour en fin d'algorithme ; d'autres admettent un retour à la fin de chaque branche de l'alternative. En début d'apprentissage, on vous demande de n'utiliser qu'un seul retour pour éviter tout abus.

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

Certains écrivent parfois une solution de ce genre mais ne la défendent pas avec les bons arguments. Le fait de ne pas avoir de "sinon" n'est absolument pas pertinent. Son avantage est qu'elle se généralise plus facilement au cas où il y a plusieurs nombres dont on veut le maximum. Pour deux nombres, on lui préférera la solution proposée plus haut.

Quand l'utiliser ?

Cet algorithme peut bien sûr être facilement adapté à la recherche du minimum..

Annexe B

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 *switch*.

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 ambigüité 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 chaine.

```
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

```
si condition alors
    Instructions
fin si
```

```
si condition alors
    Instructions
sinon
    Instructions
fin si
```

```
si condition alors
    Instructions
sinon si condition alors
    Instructions
sinon si condition alors
    ...
sinon
    Instructions
fin si
```

```
selon que expression vaut
 liste_1 de valeurs séparées par des virgules :
    Instructions
 liste_2 de valeurs séparées par des virgules :
    Instructions
    ...
 liste_k de valeurs séparées par des virgules :
    Instructions
 autres :
    Instructions
fin selon que
```

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

Les instructions de répétition

```
tant que condition faire
    Instructions
fin tant que
```

```
pour indice de début à fin [par pas] de
    à l par n faires tructions
fin pour
```

```
faire
    Instructions
tant que condition
```

La boucle pour ne peut être utilisée que pour des entiers.

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.

Index