Développement I
(Algorithmique et Java)

DEV 1 - DEV

2019-2020

A.Hallal
C.Leignel
DP.Bischop
F.Servais
J.Beleho
M.Codutti
N.Richard
P.Bettens
S.Rexhep

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

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

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

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

Document produit avec LaTeX.
Version du 17 septembre 2019.

Crédits

 Icône « Book dictionary » de Thomas Elbig [1] de The noun project [2]

 Icône « Maze » de Gilbert Bages [3] de The noun project

 Icône « Attention » de Vineet Kumar Thakur [4] de The noun project

 Icône « Card » de Alexander Skowalsky [5] de The noun project

 Icône « Stop » de Gregor Cresnar [6] de The noun project

 Icône « Coffee » de Luis Prado [7] de The noun project

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

Table des matières

Première partie

Introduction générale

Chapitre 1

Résoudre des problèmes

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

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

Contenu

1.1 La notion de problème

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...).

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

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

1.1.2 Poser le problème

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

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

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

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

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

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

Exercice Un problème flou.
Soit le problème suivant : « Calculer la moyenne de nombres entiers. ».
Ce problème est-il bien posé ?
Expliquez pourquoi cet énoncé n'est pas bon.
Proposez un énoncé qui soit acceptable.

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

1.2 Procédure de résolution

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

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

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

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

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

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

1.2.1 Chronologie des opérations

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

1.2.2 Les opérations élémentaires

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

Ce concept est évidemment relatif à ce qu'un exécutant est capable de réaliser. Cette capacité, il la possède d'abord parce qu'il est **construit** d'une certaine façon (capacité innée). Ensuite parce que, par construction aussi, il est doté d'une faculté

d'**apprentissage** lui permettant d'assimiler, petit à petit, des procédures non élémentaires qu'il exécute souvent. Une opération non élémentaire pourra devenir une primitive un peu plus tard.

1.2.3 Les opérations bien définies

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

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

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

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

Expérience. Le dessin.

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

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

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

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 en la déposant dans une assiette le matin.

1.2.5 Opérations à répéter

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

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

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

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

1.2.6 À propos des données

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

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

1.3 Ressources

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

- ▷ Les Sépas 18 - Les algorithmes [10]
- ▷ Les Sépas 11 - Un bug [11]
- ▷ Le crépier psycho-rigide comme algorithme [12]
- ▷ Le baseball multicouleur comme algorithme [13]
- ▷ Le jeu de Nim comme algorithme [14]
- ▷ Code Studio [15], un site d'apprentissage et d'initiation à l'algorithmique et la notion de programme.
- ▷ Coding game [16], un site d'apprentissage de la programmation un peu différent du précédent... et un peu plus complexe.

10. https://www.youtube.com/watch?v=hG9Jty7P6Es
11. https://www.youtube.com/watch?v=deI0GV5sWTY
12. https://pixees.fr/?p=446
13. https://pixees.fr/?p=450
14. https://pixees.fr/?p=443
15. http://studio.code.org/
16. https://www.codingame.com

Chapitre 2

Les algorithmes et les programmes

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

Contenu

2.1 Algorithmes et programmes

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

2.1.1 Algorithme

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

Algorithme : Procédure de résolution d'un problème contenant des opérations bien définies portant sur des informations, s'exprimant dans une séquence définie sans 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, pseudocode...

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

Exemple simple

Calculer la circonférence d'un cercle.

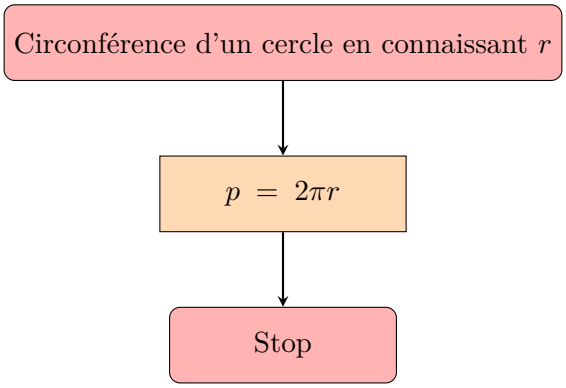
Un algorithme en **langage naturel** pourrait être :

En connaissant le rayon, le calcul de la circonférence se fait grâce à

$$p = 2\pi r$$

langage naturel

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



ORGANIGRAMME 1 – Illustration d'un organigramme

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

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

pseudocode

2.1.2 Programme

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

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

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

Un exemple simple (suite)

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

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

java

2.1.3 Les constituants principaux de l'ordinateur

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

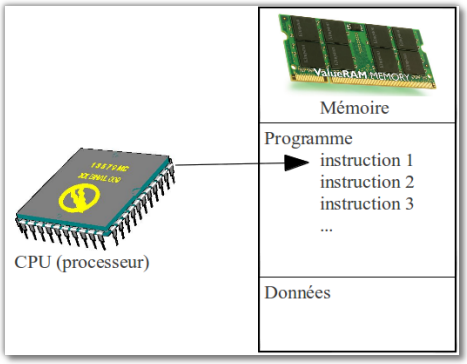
Le **hardware** est constitué de l'ordinateur proprement dit et regroupe les entités suivantes :

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

2.1.4 Exécution d'un programme

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



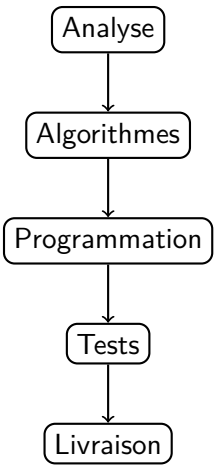
Comment fonctionne le processeur ?
De façon très simplifiée, les étapes suivantes ont lieu :

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

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

2.2 Les phases d'élaboration d'un programme

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



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

Notons que ce processus n'est pas linéaire. À chaque phase, on pourra détecter

des erreurs, imprécisions ou oublis des phases précédentes et revenir en arrière. Un problème peut être amélioré, complexifié, modifié ou étendu.

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

- ▷ je modifie ma recette et remplace le lait par du lait de soja ;
- ▷ je change de plat ;
- ▷ je notifie qu'il ou elle n'est plus bienvenue ou qu'elle recevra une tartine [17].

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 pour écrire nos algorithmes est plus souple et plus général que le langage Java par exemple (où il faut être précis au « ; » près). Nous commencerons simplement par écrire en langage naturel.
- ▷ De plus, un algorithme écrit facilite le dialogue dans une équipe de développement. « J'ai écrit un algorithme pour résoudre le problème qui nous occupe. Qu'en pensez-vous ? Pensez-vous qu'il est correct ? Avez-vous une meilleure idée ? ». L'algorithme est plus adapté à la communication car plus lisible.
- ▷ Enfin, si l'algorithme est écrit, il pourra facilement être traduit dans n'importe quel langage de programmation. La traduction d'un langage de programmation à un autre est un peu moins facile à cause des particularités propres à chaque langage.

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

2.3 Conclusion

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

17. Je m'autorise un accord de proximité.

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

Nous tenterons :

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

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

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

2.4 Ressources

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

- ▷ C'est pas sorcier ! Ordinateur, tout un programme [18]

18. `https://www.youtube.com/watch?v=c96KP5jZVYk`

Deuxième partie

Les bases de l'algorithmique et de la programmation

Chapitre 3

Spécifier le problème

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

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

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

Contenu

3.1 Déterminer les données et le résultat

La toute première étape est de parvenir à extraire d'un énoncé de problème, quelles sont les données et quel est le résultat attendu [19]. Dans la suite, nous utiliserons un exemple très simple afin d'illustrer les concepts qui nous intéressent.

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

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

Quelles sont les données ? Il y en a deux :

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

Quel est le résultat attendu ? la surface du rectangle.

3.2 Les noms

Pour identifier clairement chaque **donnée** et pouvoir y faire référence dans le futur algorithme et dans le programme nous devons lui attribuer un **nom**[20]. Il est important de bien choisir les noms. Le but est de trouver un nom qui soit suffisamment court, tout en restant explicite et ne prêtant pas à confusion.

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

On peut envisager les noms suivants :

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

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

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

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

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

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

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

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

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

Notons que les langages de programmation imposent certaines limitations (parfois différentes d'un langage à l'autre) ce qui peut nécessiter une modification du nom lors de la traduction de l'algorithme en un programme. À chaque langage de programmation sont également associées des **conventions d'écriture** que les développeurs respectent. Bien que nous y reviendrons plus en détail dans la section 4.6, notons déjà que la simple convention de nom de méthode change d'un langage à l'autre.

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

3.3 Les types

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

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

Dans un premier temps, nous utiliserons ces **types**[21] :

Type	
int	pour les nombres entiers
double	pour les nombres réels
String[22]	pour les chaines de caractères, les textes par exemple : "Bonjour", "Bonjour le monde !", "a", ""...
boolean	quand la valeur ne peut être que true (vrai) ou false (faux)

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

22. Notez la majuscule, indispensable en Java

Exemples.

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

3.3.1 Il n'y a pas d'unité

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

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

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

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

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

3.3.2 Préciser les valeurs possibles

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

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

Exemples.

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

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

3.3.3 Le type des données complexes

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

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

On pourrait la représenter dans une 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 tard, vous verrez qu'il est possible de définir de nouveaux types de données grâce aux *structures* pour les algorithmes et aux *classes* pour les langages orientés objets tels que Java.

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

3.3.4 Exercice

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

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

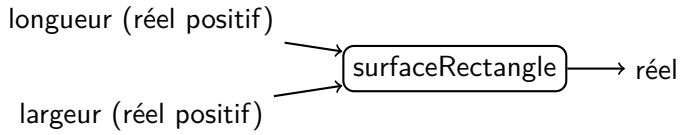
3.4 Résumés

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

3.4.1 Résumé graphique

Représentation graphique du problème.

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



3.4.2 Résumé textuel

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

Exemple.

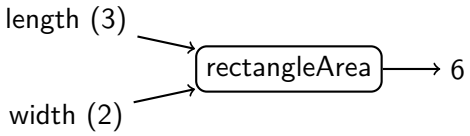
Données	longueur (un réel positif) largeur (un réel positif)
Résultat	la surface du rectangle

3.5 Exemples numériques

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

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

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



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

Chapitre 4

Premiers algorithmes

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

En fonction de la difficulté du problème et de notre sensibilité, nous pouvons représenter un algorithme de plusieurs manières ; en langage naturel (en français ou en anglais), en pseudocode ou avec un organigramme... Dans ce chapitre, nous présenterons les algorithmes en langage naturel, en pseudocode, avec un organigramme et en langage Java.

Dans la suite du cours, nous nous contenterons de les écrire en langage Java.

Contenu

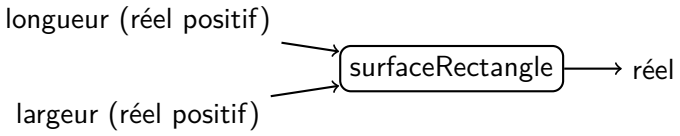
4.1 Exercice résolu : un problème simple

4.1.1 Trouver l'algorithme

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

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

Analyse. Nous sommes arrivés à la spécification suivante :



ou encore :

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

Exemples.

La surface du rectangle 3,2 vaut 6. La surface du rectangle 3.5, 1 vaut 3.5.

ou

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

ou

Données		
L	3	3.5
l	2	1
Résultat	6	3.5

Les exemples sont essentiels. Ce sont eux qui vont nous permettre de tester notre algorithme. Notre programme.

Comment résoudre ce problème ? La toute première étape est de comprendre le lien entre les données et le résultat. Ici, le lien est la formule du calcul d'une surface.

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

La surface s'obtient donc en multipliant la longueur par la largeur [23].

4.1.2 Vérifier l'algorithme

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

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

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

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

4.1.3 Écrire l'algorithme

4.1.3.1 Langage naturel

En **langage naturel**, une solution aurait simplement cette allure :

La surface s'obtient grâce à la formule :

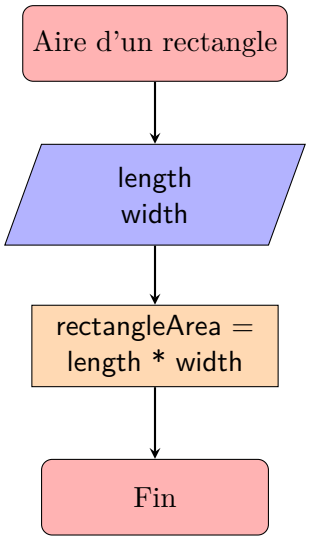
$$surface = longueur * largeur$$

langage naturel

4.1.3.2 Organigramme

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

23. 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 nous concentrons sur des problèmes simples.



ORGANIGRAMME 2 – Solution de rectangleArea

4.1.3.3 Pseudocode

Un **pseudocode** d'une solution s'écrit :

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

pseudocode

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

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

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

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

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

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

Dans ces notes, nous utiliserons « → » pour montrer ce que retourne l'algorithme mais « : » ou un simple « ␣ » [24] font l'affaire.

4.1.3.4 Java

Une solution en Java s'écrit :

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

```

java

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

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

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

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

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

java

4.1.4 Remarques

Attention :

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

- ▷ spécifier le problème ;
- ▷ fournir des exemples, les tests ;
- ▷ construire son algorithme ;
- ▷ tester le programme et constater qu'il fournit bien les résultats attendus

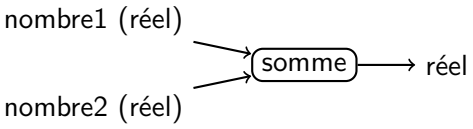
4.2 Résolution d'un second problème simple

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

Vous pouvez vous baser sur la fiche 1 page 206 qui résume la résolution du calcul de la surface d'un rectangle, depuis l'analyse de l'énoncé jusqu'à l'algorithme et à sa vérification.

Problème. Calculer la somme de deux nombres donnés.

Analyse.



ou :

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

Exemples Les exemples qui deviendront les tests :

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

Algorithme L'algorithme s'écrit simplement en langage naturel :

```
somme = n1 + n2
```

langage naturel

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

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

java

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

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

4.3 Décomposer les calculs ; variables et assignation

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

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

4.3.1 Les variables

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

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

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

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

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

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

langage naturel

Nous pouvons abréger – et passer à l'anglais – en :

```
réels length, width
```

langage naturel

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

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

En langage Java, cela donne :

```java
double length, width;
```

java

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

```java
double length;
double width;
```

java

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

4.3.2 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.

Pour cette assignation, nous pourrions utiliser une flèche ← ou le symbole d'égalité =. Dans ces notes nous utiliserons le symbole d'égalité =.

```
variableName = expression
```

langage naturel

En Java, nous utiliserons exactement le même symbole = :

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

java

Bien que nous utilisions le symbole de l'égalité mathématique, l'assignation n'a pas le même sens.

- ▷ l'égalité mathématique définit que deux choses sont égales et le resteront ;
- ▷ l'assignation assigne une valeur – le membre de droite – à la variable se trouvant à gauche à un moment t.

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

Exemples. Quelques assignations correctes en algorithmique :

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

langage naturel

Et ces même exemples — corrects — en Java :

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

java

Remarques

- ▷ **Une assignation n'est ni une égalité, ni une définition**.
  Ainsi, l'assignation $\mathsf{cpt} = \mathsf{cpt} + 1$ ne veut pas dire que cpt et cpt + 1 sont égaux, ce qui est mathématiquement faux mais que la *nouvelle* valeur de cpt doit être calculée en ajoutant 1 à sa valeur actuelle.
  Ce calcul doit être effectué au moment de l'exécution de cette instruction.
- ▷ Seules les variables déclarées peuvent être affectées.
- ▷ Toutes les variables apparaissant dans une expression doivent avoir été affectées préalablement. Le contraire provoquerait une erreur, un arrêt de l'algorithme ou du programme.
- ▷ La valeur affectée à une variable doit être compatible avec son type. Pas question de mettre une chaine dans une variable booléenne.
- ▷ Certaines personnes utilisent ← comme symbole d'assignation pour les algorithmes. C'est très bien aussi.
  Nous utiliserons dans ce cours le symbole = afin d'être plus proche du langage Java.

4.3.3 Tracer un algorithme

Pour vérifier qu'un algorithme est correct, le développeur ou la développeuse sera souvent amenée[25] à le tracer.

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

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

Exemple. Traçons des extraits d'algorithmes.

```java
int a, b et c;
a = 12;
b = 5;
c = a - b;
a = a + c;
b = a;
```

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

25. Je m'autorise un accord de proximité.

```java
int a, b, c;
a = 12;
c = a - b;
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 !

4.3.4 Exercice résolu : durée du trajet

Savoir, face à un cas concret, s'il est préférable de décomposer le calcul ou pas, n'est pas toujours évident. La section 4.6 page 49 sur la lisibilité vous apportera des arguments qui permettront de trancher.

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

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

Nous vous proposons de rédiger une solution complète.

Vous pouvez vérifier ensuite sur la fiche 2 page 208 qui présente une solution.

4.4 Quelques difficultés liées au calcul

Vous êtes habitués à effectuer des calculs. L'expérience nous montre toutefois que certains calculs posent des difficultés. Soit parce que ce sont des opérations peu utilisées, soit parce qu'elles ne sont habituellement pas sous la forme de calculs. Citons :

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

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

4.4.1 Un peu de vocabulaire

Quelques notions : Qu'est-ce qu'un opérateur ? Un opérande ? Rappelons et fixons ces notions.

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

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

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

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

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

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

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

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

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

4.4.2 Les comparaisons et les assignations de variables booléennes

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

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

Exemples. Voici quelques assignations correctes :

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

java

Remarque Pour tester l'égalité, nous utilisons le symbole == afin de le différencier du symbole = qui est le symbole de d'assignation.

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

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

4.4.3 Les opérations logiques

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

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

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

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

a	b	a AND b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a OR b
true	true	true
true	false	true
false	true	true
false	false	false

a	NON a
true	false
false	true

Ces opérateurs peuvent intervenir dans des expressions booléennes.

Exemples

- ▷ tarifPlein = 18≤âge ET âge<60
- ▷ distinction = 14≤cote ET cote<16
- ▷ nbA3chiffres = 100≤nb ET nb≤999
- ▷ tarifRéduit = NON tarifPlein
- ▷ tarifRéduit = NON (18≤âge ET âge<60)
- ▷ tarifRéduit = âge<18 OU 60≤âge

En Java, le AND (ET) s'écrit && [26] et le OR (OU), || [27].

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 [28].

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

Loi de De Morgan. Cette loi s'énonce :

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

... et s'écrit plus simplement :

$$\text{NON } (a \text{ AND } b) \Leftrightarrow \text{NON } a \text{ OR NON } b$$

$$\text{NON } (a \text{ OR } b) \Leftrightarrow \text{NON } a \text{ AND NON } b$$

Par exemple, ces trois propositions sont équivalentes :

tarifRéduit = NON (18≤âge AND âge<60)

tarifRéduit = (NON 18≤âge) OR (NON âge<60)

tarifRéduit = âge<18 OR 60≤âge

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

L'opérateur AND est prioritaire sur le OR.

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

26. & est le caractère *ampersand* (esperluette en français)
27. | est le caractère *pipe* (barre verticale en français)
28. Vous noterez que le nombre de "et" et de "ou" dans cette phrase ne facilite pas sa compréhension ;)

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

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

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

Exemples.

```java
boolean ok;

// Génère une erreur si b est nul
ok = 1/b < 0.1;

// Si b est nul, provoque une erreur et un arrêt
ok = 1/b < 0.1 && b != 0;

// YES ! C'est dans le bon ordre si b est nul,
// la condition est fausse et
// ok reçoit false
ok = b != 0 && 1/b < 0.1;
```

java

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

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

- ▷ beautifulBoolean = adulte == vrai
- ▷ beautifulBoolean = adulte == faux
- ▷ beautifulBoolean = etudiant == vrai AND jeune == faux
- ▷ beautifulBoolean = NON (adulte == vrai) AND NON (adulte == faux)
- ▷ nbA3chiffres = NON (nb<100 OR nb≥1000)

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

- ▷ J'irai au cinéma si le film me 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.

4.4.4 La division entière et le reste

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

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

$$D = d * q + r$$
$$\frac{D}{d} = q$$

et, éventuellement un reste r

La division entière est souvent notée DIV et le reste MOD.

- a DIV b est le quotient de la division et
- a MOD b — nous dirons a *modulo* b — est le reste de cette division.

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

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

Exemples :

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

4.4.4.1 Utilité - Tester la divisibilité

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

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

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

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

Et, en langage java :

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

4.4.4.2 Utilité - 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

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

Exemple : (65536 DIV 100) MOD 10 = 5... qui est le chiffre des centaines.

4.4.5 Le hasard et les nombres aléatoires

Il existe de nombreuses applications qui font intervenir le hasard.

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

Le vrai hasard n'existe pas en informatique puisqu'il s'agit de suivre des étapes précises dans un ordre fixé et que la machine est déterministe. Pourtant, on peut concevoir des algorithmes et des programmes qui *simulent* le hasard [29]. À partir d'un nombre donné [30] (appelé *graine* ou *seed* en anglais) ils fournissent une suite de nombres qui *ont l'air* aléatoires.

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

En Java, il suffit d'écrire [31]

```java
double number = Math.random();
```

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

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

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

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

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

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

▷ un entier entre 0 inclus et n exclu

```
(la partie entière de) random() * n
```

langage naturel

▷ un entier entre min et max inclus

```
(la partie entière de) min + random() * (max-min+1)
```

langage naturel

En langage Java, il existe une méthode `Math.random()` comme nous l'avons dit mais également une autre manière de faire en utilisant la classe **Random** comme ceci [32] :

```java
Random R = new Random();

// Une valeur entière comprise entre 0 et 13 strictement
int value = R.nextInt(13);

// Une valeur entière comprise entre 10 et 20 strictement
int value2 = 10 + R.nextInt(10);

// Une valeur pseudo-réelle entre 0 et 1 strictement
double value3 = R.nextDouble();
```

java

4.5 Des algorithmes et des programmes de qualité

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

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

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

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

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

32. Le code n'est pas fonctionnel en l'état. Il manque un *import*. Nous y reviendrons

4.5.1 L'efficacité

L'**efficacité**[33] désigne le fait que l'algorithme (le programme) résoutbien le problème donné. C'est un minimum !

4.5.2 La lisibilité

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

- ▷ celles qui doivent se convaincre de sa validité avant de passer à la programmation ;
- ▷ celles qui doivent trouver les causes d'une erreur lorsque celle-ci a été rencontrée[34] ;
- ▷ celles qui doivent faire évoluer l'algorithme ou le programme suite à une modification du problème ;
- ▷ et, accessoirement, celles qui doivent le coter ;)

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

Comparer la lisibilité de deux algorithmes ou de deux programmes n'est pas une tâche évidente car c'est une notion subjective. Il faut se demander quelle version va être le plus facilement comprise par la majorité des lecteurs. La section 4.6 page suivante explique ce qui peut être fait pour rendre ses algorithmes plus lisibles. À ces recommandations s'ajouteront les **conventions d'écriture** propres à chaque langage qu'il faudra aussi respecter.

4.5.3 La rapidité

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

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

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

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

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

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

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

4.5.4 La taille

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

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

4.5.5 Conclusion

Tous ces critères n'ont pas le même poids. Le point le plus important est bien sûr d'écrire des algorithmes et des programmes corrects mais ne vous arrêtez pas là ! Demandez-vous s'il n'est pas possible de le retravailler pour améliorer sa lisibilité [35].

4.6 Améliorer la lisibilité

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

4.6.1 Mise en page des algorithmes et des programmes

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

- ▷ les mots imposés ou, tout au moins importants, sont mis en évidence (en gras [36]) ;
- ▷ une seule instruction se trouve par ligne ;
- ▷ les instructions à l'intérieur de l'algorithme sont *indentées* (décalées vers la droite). On indentera également les instructions à l'intérieur des choix et des boucles ;
- ▷ Le début et la fin de quelque chose — un bloc — sont marqués par des accolades () ou une ligne.

35. 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.

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

Exemples

On n'écrira pas :

```java
public static double travelTime(double speedMS, double distanceKm) {
double distanceM;
distance M = 1000 * distanceKM;
return distanceM / speedMS;
}
```

ni :

```java
public static double travelTime(double speedMS, double distanceKm) {
  double distanceM;
    distance M = 1000 * distanceKM; return distanceM / speedMS; }
```

mais plutôt :

```java
public static double travelTime(double speedMS, double distanceKm) {
  double distanceM;
  distance M = 1000 * distanceKM;
  return distanceM / speedMS;
}
```

Plus spécifiquement pour les programmes, l'utilisation d'un éditeur de code [37] ou d'un IDE (Environnement de Développement Intégré) [38] aide à l'écriture de programmes lisibles et respectant les conventions.

- ▷ Le code est indenté comme les algorithmes.
- ▷ Une seule instruction par ligne.
- ▷ La longueur des lignes n'excède pas 80 caractères [39].
- ▷ S'il est nécessaire de couper une ligne trop longue, celle-ci est coupée avant un opérateur ou après une virgule.

37. Un éditeur de code est un programme aidant à l'édition de code qu'il ne faut pas confondre avec un éditeur de texte. Des éditeurs de codes connus ; *gVim*, *Notepad++*
38. Des IDE connus ; *Netbeans*, *Eclipse*...
39. Ce n'est pas la largeur de l'écran qui détermine la longueur des lignes. La limite à 80 caractères permet un code plus lisible. Elle permet aussi de pouvoir lire plus vite le code en limitant le mouvement des yeux de gauche à droite.

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

java

4.6.2 Choix des noms

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

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

4.6.3 Les commentaires

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

Habituellement, on distingue deux sortes de commentaires :

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

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

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

Dans nos algorithmes, les commentaires commencent par //.

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

1. // en début de ligne ;
2. le commentaire sur plusieurs lignes en commençant par /* et en le terminant par */ ;
   /* <commentaire> */

3. le commentaire sur plusieurs lignes en commençant par /** et en le terminant par */. Dans ce cas, c'est un commentaire destiné à la documentation *javadoc*. Nous y reviendrons (dans la section **??** p.**??**).

Voici quelques mauvais commentaires

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

```

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

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

sans commentaires est bien préférable aux lignes suivantes :

```langage naturel
c1 = c0 * ( 1 + r/100)
Calcul du nouveau capital en appliqant le taux
```

Pour résumer :

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

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

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

```

4.6.4 Constantes

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

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

L'utilisation de constantes dans vos algorithmes présente les avantages suivants :

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

Exemples

```java
public static final double PI = 3.1415;
public static final int PASS_LEVEL = 10;
public static final String ESI = "École␣supérieure␣d'Informatique";
```

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

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

4.7 Appel d'algorithme, appel de méthode

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

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

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

En Java, la contrainte est un peu plus forte. Les programmes sont généralement écrits dans plusieurs fichiers qui doivent « se retrouver »... mais dans un premier temps, nous nous contenterons d'un seul. Ce fichier représentera une *classe*.

- ▷ Les instructions qui définissent une classe sont :

```java
public class MyClass{
  // statements
}
```

- ▷ La classe MyClass doit se trouver dans le fichier `MyClass.java`[40]

Pour pouvoir appeler une méthode en Java qui se trouve dans la même classe, il suffit d'écrire :

40. Cette contrainte peut être relâchée à partir de JDK11. Voir `http://namok.be/blog/?post/2019/04/06/JDK10-11-12`

```java
  area = rectangleArea(122, 3.78);
2
```

java

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

4.8 Interagir avec l'utilisateur

4.8.1 Afficher un résultat

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

```
print 2
print rectangleArea(122, 3.78)

afficher 3
```

langage naturel

En langage naturel, nous pouvons choisir d'écrire afficher, print, écrire à l'écran... Dans ces notes, nous écrirons **print**.

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

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

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

java

Les lettres *ln* qui suivent le mot *print* signifient *new line* et indiquent un passage à la ligne après l'affichage. Il existe aussi une instruction print sans les lettres *ln*... qui ne passe pas à la ligne.

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

4.8.2 Demander des valeurs

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

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

demander la valeur de length
```

langage naturel

En langage naturel, nous pouvons choisir d'écrire lire, read, demander la valeur de... Dans ces notes, nous écrirons **read**.

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

En langage Java, trois instructions seront nécessaires pour pouvoir faire une lecture au clavier pour un programme s'exécutant dans la console [41].

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

java

4.8.3 Préférer les paramètres

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

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

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

```java
import java.util.Scanner;

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

  public static void main(String[] args){
    double length;
    double width;
    Scanner keyboard = new Scanner(System.in);

    System.out.println("Entre la longueur: ");
    length = keyboard.nextDouble();
    System.out.println("Entre la largeur: ");
    width = keyboard.nextDouble();

    System.out.println("Surface: " + rectangleArea(length, width));
  }
}

```

java

Chapitre 5

Premiers programmes

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

Contenu

5.1 Introduction

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

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

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

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

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

langage orienté objets

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

— Que faut-il faire ?

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

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

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

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

langage fonctionnel

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

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

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

La langage Java est un langage strict. Il impose, par exemple, que l'on déclare les variables que l'on utilise et il vérifie que les données assignées aux variables soient du « bon » type. Il possède un *garbage collector* (ramasse-miettes en français [42]) qui gère la mémoire à la place du développeur. Ces aspects facilitent l'acquisition de bonnes pratiques de développement.

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

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

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

5.1.1 Compilation - interprètation

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

42. Vous noterez la traduction.

Définition

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

Ces langages nécessitent un **compilateur**.

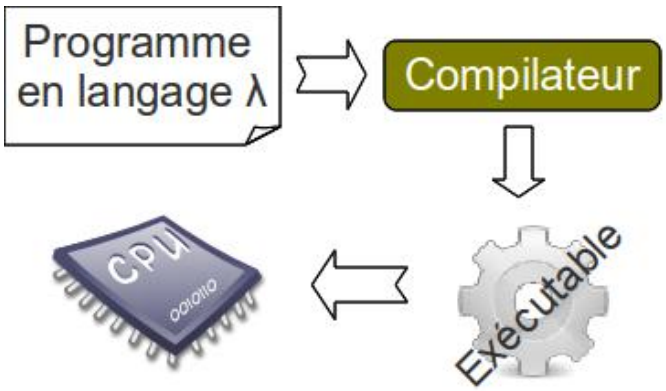


FIGURE 5.1 – Langage compilé

Définition

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

Ces langages nécessitent un **interprèteur** [43].

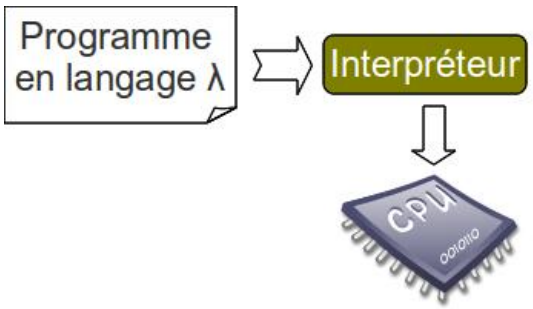


FIGURE 5.2 – Langage interprété

Les deux approches ont leurs avantages et leurs inconvénients.

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

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

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

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

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

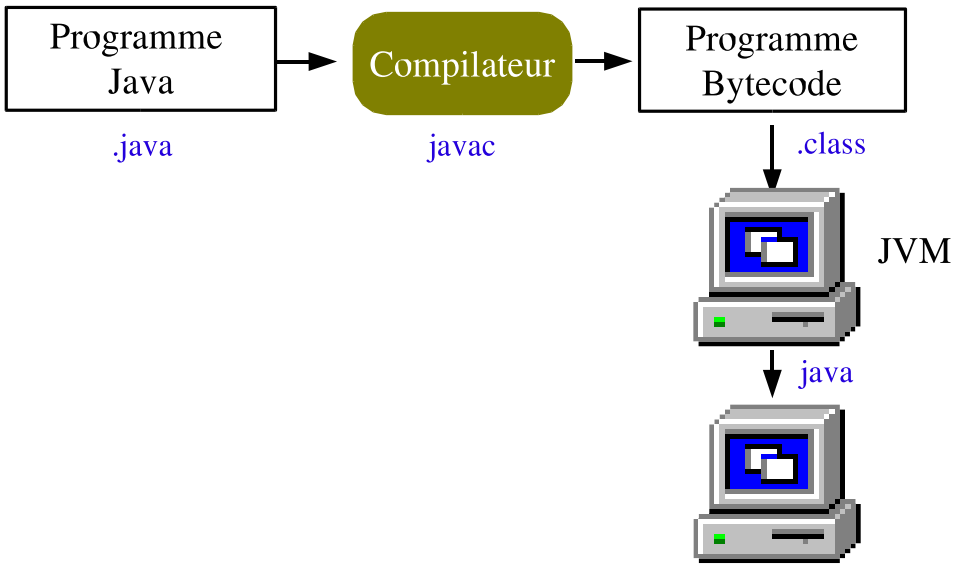


FIGURE 5.3 – Java est compilé et ensuite interprété

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

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

```
$
javac MyProgram.java
```
terminal

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

```
$
java MyProgram
```
terminal

Remarques

1. Notez que `javac` prend le nom d'un fichier en paramètre (avec son extension) alors que `java` prend le nom d'un programme — nous dirons une classe bientôt — en paramètre (sans extension donc).
2. Avec JDK11, il est possible de passer un fichier à la commande `java`. Dans ce cas, la commande compilera le fichier, conservera le bytecode en RAM et l'exécutera. Il n'y aura donc pas création d'un fichier contenant le *bytecode* et le code source sera exécutable en une seule commande (voir [Bet19]).

```
$
java MyProgram.java
```
terminal

Attention de ne pas confondre.

5.2 Environnement de développement

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

5.2.1 La base

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

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

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

5.2.2 Écrire le code

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

1. **éditeur de code**
   Un éditeur de code est un éditeur de texte — à ne pas confondre avec un traitement de texte — augmenté c'est-à-dire offrant des fonctionnalités supplémentaires. Citons par exemple :
   - ▷ la coloration syntaxique. Le programme reconnait les mots clés du langage, les structures et les écrit en couleur pour accroitre la lisibilité du code ;
   - ▷ l'indentation automatique et la réindentation du code ;

- ▷ une certaine autocomplétion pour les mots connus du langage et les variables.

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

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

2. **IDE**

   Un environnement de développement intégré est un programme servant à écrire les programmes. En plus d'un éditeur de code, un IDE compile en arrière plan, a un débogueur, organise les fichiers, crée des fichiers (en partie) pré-complétés sur base de *templates*...

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

Connaitre et utiliser correctement un éditeur de code et un IDE sont deux compétences essentielles d'un bon développeur. L'une n'allant pas sans l'autre.

5.2.3 Exécuter le programme

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

Il en existe deux :

1. Java est — actuellement — la propriété d'Oracle et c'est Oracle qui fournit le JDK officiel.

   Télécharger Java chez Oracle [44]

2. OpenJDK est une alternative libre au JDK officiel Java.

   Télécharger OpenJDK [45]

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

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

5.2.4 Hello world

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

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

44. `http://www.oracle.com/technetwork/java/javase/downloads`
45. `http://openjdk.java.net/`

Grâce à mon éditeur de code, je crée le fichier `Hello.java`[46]qui contient :

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

java

Je compile ma classe :

```
$
javac Hello.java
```

terminal

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

```
$
java Hello
```

terminal

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

Remarque À partir de JDK11, voir [] ou au paragraphe 5.1.1 page 61, on peut se contenter d'une seule commande `java Hello.java`.

5.3 La grammaire du langage

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

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

JLS Télécharger *The Java Language Specification*[47]

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

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

46. Si vous êtes un utilisateur MS Windows, désactivez la propriété «*Hide extension for know files types*». Si vous ne le faites pas, vous allez — probablement — voir `Hello.java` alors que le fichier que vous créez est `Hello.java.txt`.
47. `https://docs.oracle.com/javase/specs/`

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

> *NumericType:*
> *IntegralType*
> *FloatingPointType*
>
> *IntegralType:*
> `int`
>
> *FloatingPointType:*
> `double`

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

Nous sommes incomplets à ce stade. Les personnes curieuses peuvent aller voir la grammaire complète de ces règles dans JLS10 [48] p. 42.

48. `https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf`

Chapitre 6

Une question de choix

Les **alternatives** permettent de n'exécuter des instructions que si une certaine *condition* est vérifiée. Par exemple, si le filtre à café est vide, le remplir. Dans la vie, nous testons notre environnement, dans nos algorithmes et dans nos programmes, nous allons tester les données.

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

Contenu

6.1 Le si (*if-then*)

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

```
si une certaine condition est vraie alors
    exécuter une ou plusieurs actions
fin si
continuer l'algorithme
```

langage naturel

Dans la grammaire du langage Java, cette instruction s'écrit :

```
if ( Expression ) Statement
```

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

Remarque

Attention, 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.

Exemple

Supposons que la variable nb contienne un nombre positif ou négatif. Et supposons que l'on veuille le rendre positif. Il faudra tester son signe et, s'il est négatif, l'inverser. Par contre, s'il est positif, il n'y a rien à faire.

Voici comment écrire un algorithme sous différentes formes :

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

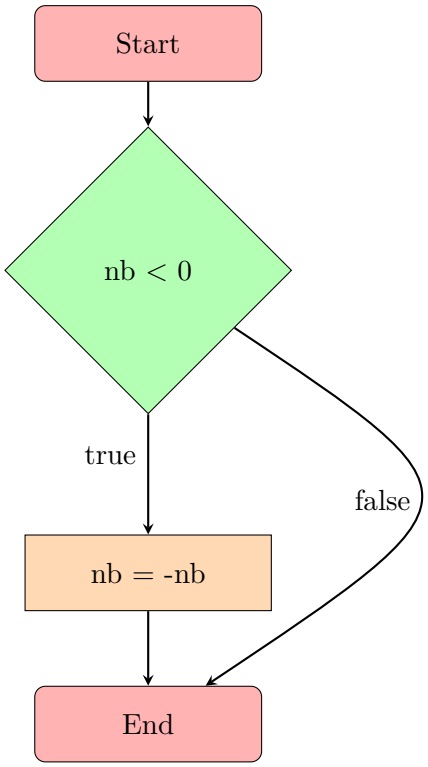
langage naturel

ou

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

langage naturel

Un organigramme aurait cette forme :



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

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

Pour s'éviter des erreurs, nous utiliserons toujours le bloc d'instructions. Nous conseillons de faire de même et nous écrirons :

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

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

#	nb	test
	-3	
1	|	vrai
2	3	

#	nb	test
	3	
1	|	faux

Exercice de compréhension Tracez l'algorithme écrit en langage Java avec les valeurs fournies et donnez la valeur de retour.

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

java

▷ exercice(2, 5) = _____

▷ exercice(4, 1) = _____

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

La construction si-sinon permet d'exécuter certaines instructions ou d'autres en fonction d'un test.

```
si une certaine condition est vraie alors
    exécuter une ou plusieurs actions
sinon (la condition est alors fausse)
    exécuter d'autres actions
fin si
continuer l'algorithme
```

langage naturel

Dans la grammaire du langage Java, cette instruction s'écrit :

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

Exemple

Pour illustrer cette instruction, nous allons écrire un algorithme qui recherche le maximum de deux nombres.

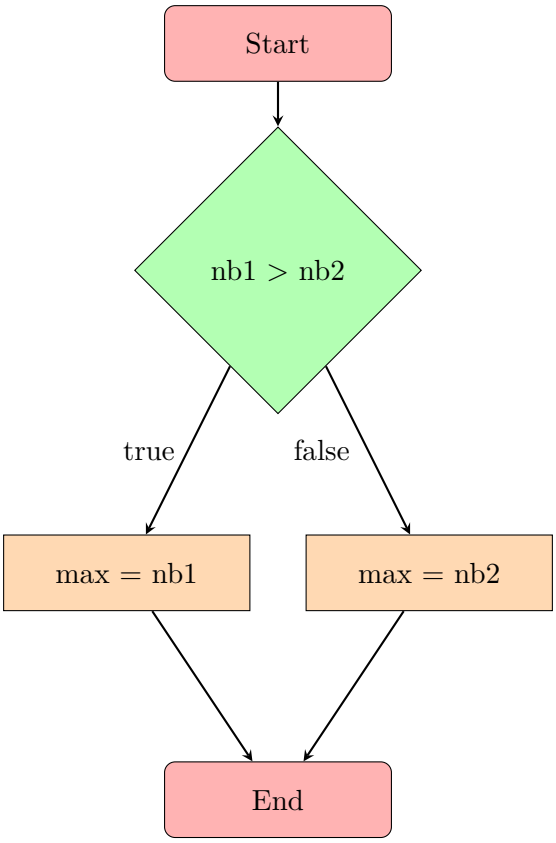
Pour déterminer le maximum de deux nombres, c'est-à-dire la plus grande des deux valeurs, il y aura deux chemins possibles. Le maximum devra prendre la valeur du premier nombre ou du second selon que le premier est plus grand que le second ou pas.

Voici comment écrire cet algorithme sous différentes formes :

```
if (nb1 > nb2)
    max = nb1
else
    max = nb2
```

langage naturel

Un organigramme aurait cette forme :



En supposant les variables déclarées, le *si-alors*, s'écrit comme suit en langage Java :

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

Traçons l'algorithme dans différentes situations.

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

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

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

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

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

```java
public static int exercice(int a, int b){
  int c;
  if (a > b){
    c = a/b;
  } else {
    c = b%a;
  }
}
```

java

▷ exercice(2, 3) = _____ ▷ exercice(4, 1) = _____

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

java

▷ exercice(2, 3) = _____ ▷ exercice(4, 1) = _____

6.3 Le si-sinon-si

Avec cette construction, il est possible d'indiquer à un endroit de l'algorithme plus de deux chemins possibles et dès lors que la première condition est fausse, tester une condition supplémentaire à chaque étape.

```
si une certaine condition est vraie alors
    exécuter une ou plusieurs actions
sinon si une autre condition est vraie alors
    exécuter d'autres actions
sinon
    exécuter encore d'autres actions
fin si
continuer l'algorithme
```

langage naturel

Dans la grammaire du langage Java, il n'y a pas d'instruction supplémentaire. Un *si-sinon-si* n'étant que des *si-sinon* imbriqués un peu comme suit :

```
if ( Expression )
    Statement
else
    if ( Expression )
        Statement
    else
        Statement
```

. . . qui seront généralement indentés comme suit :

```
if ( Expression )
    Statement
else if ( Expression )
    Statement
else
    Statement
```

. . . mais voyons cela sur un exmeple.

Exemple

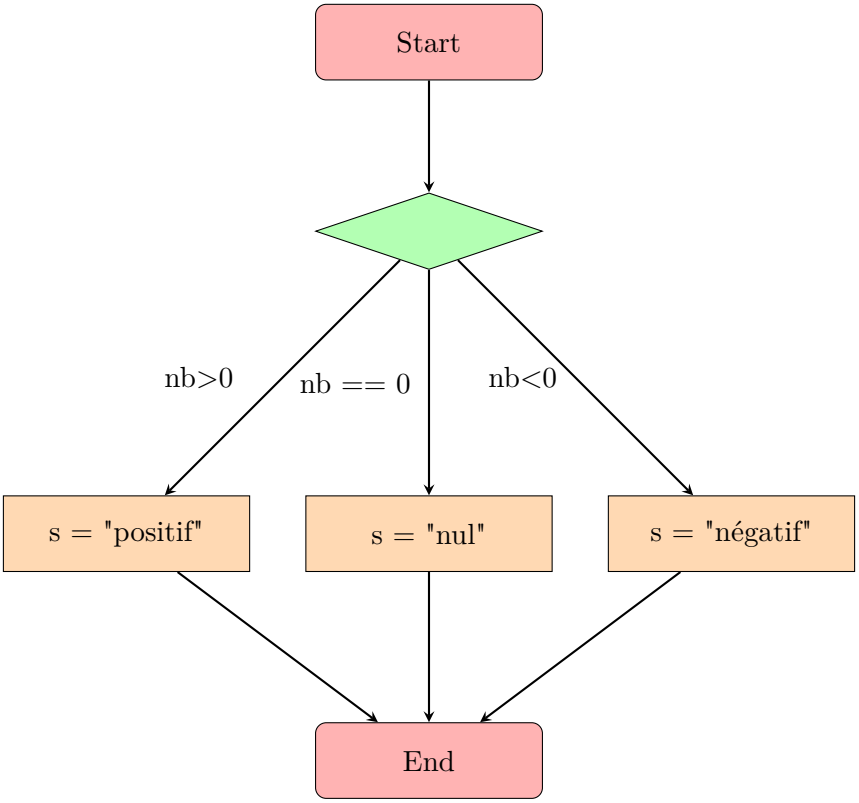
Supposons que l'on veuille mettre dans la chaine signe la valeur "positif", "négatif" ou "nul" selon qu'un nombre donné est positif, négatif ou nul.

Voici comment écrire cet algorithme sous différents formats :

```
if (nb > 0)
      signe = "positif"
else if (nb = 0)
      signe = "nul"
else
      signe = "négatif"
```

langage naturel

Un organigramme aurait cette allure :



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

```java
if (nb > 0){
  s = "positif";
} else if (nb == 0) {
  s = "nul";
} else {
  s = "négatif";
}
```

java

Traçons l'algorithme dans différentes situations.

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

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

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

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.
- ▷ 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.

```java
if (quantité < 10){
  prixUnitaire = 10;
} else if (quantité < 100) {
  // On sait que la quantité est plus grande
  // ou égale à 10. Inutile de tester.
  prixUnitaire = 8;
} else {
  prixUnitaire = 6;
}

```

6.4 Expression booléenne

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

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

Une telle expression se compose grâce :

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

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

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

   EqualityOperator:
   (one of)
   `== !=`

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

   LogicalComplementOperator:
   `!`
   ConditionalOperator:
   (one of)
   `|| &&`

6.5 Le selon-que (*switch*)

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

```
selon que la variable vale :
    — une valeur :
        exécuter une ou plusieurs actions
    — une autre valeur :
        exécuter d'autres actions
    — encore une autre valeur :
        exécuter d'autres actions
    — et une dernière valeur :
        exécuter d'autres actions
fin selon que
```

langage naturel

Dans la grammaire du langage Java, cette instruction, *switch*, s'écrit (grammaire simplifiée) :

> *SwitchStatement:*
> `switch (` *Expression* `)` *SwitchBlock*
>
> *SwitchBlock:*
> *SwitchLabels Statement*
>
> *SwitchLabel:*
> `case` *ConstantExpression*`:`
> `default:`

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

. . . mais voyons ça sur un exemple.

Exemple.

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

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

Voyons comment écrire un algorithme sous différentes formes :

langage naturel

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

En langage Java :

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

```

java

Remarques.

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

Chapitre 7

Module et références

Contenu

7.1 Décomposer le problème

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

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

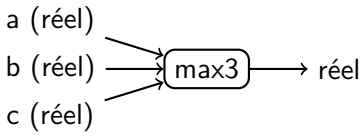
- **Cela permet de libérer l'esprit.** L'esprit humain ne peut pas traiter trop d'informations à la fois (*surcharge cognitive*). Lorsqu'un sous-problème est résolu, il peut se libérer l'esprit et attaquer un autre sous-problème.
- **On peut réutiliser ce qui a été fait.** Si un même sous-problème apparait 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, un algorithme, appelle un autre algorithme pour résoudre un sous-problème, le lecteur ou la lectrice verra un nom d'algorithme qui peut être plus parlant que les instructions qui se cachent derrière, même s'il y en a peu. Par exemple, dizaine(nb) est plus parlant que nb MOD 100 DIV 10 pour calculer le nombre de 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.

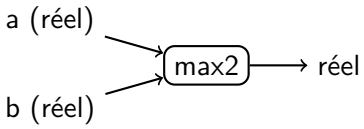
Un algorithme qui résout une partie de problème est parfois appelé **fonction**, **procédure**, **méthode** ou encore **module** en fonction du langage et du contexte. Il y a quelques nuances mais elles importent peu ici.

7.2 Exemple

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



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

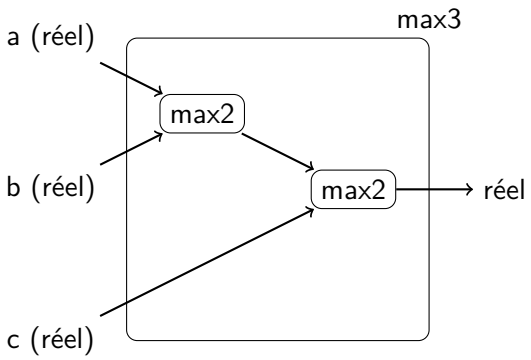



```java
public static double
    max2(double a, double b){
  double max;
  if (a > b){
    max = a;
  } else {
    max = b;
  }
  return max;
}
```

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

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

Elle s'illustre comme suit :



Sur base de cette idée, on voit que calculer le maximum de trois nombres peut se faire en calculant deux fois le maximum de deux nombres. On ne va évidemment pas *recopier*[49] dans notre solution ce qu'on a écrit pour le maximum de deux nombres ; on va plutôt y faire référence, c'est-à-dire appeler l'algorithme max2. Ce qui donne :

```java
public static double max3(double a, double b, double c){
  double maxab, max;
  maxab = max2(a, b);
  max = max2(maxab, c);
  return max;
}
```

qui peut se simplifier en :

```java
public static double max3(double a, double b, double c){
  return max2(max2(a, b), c);
}
```

7.3 Paramètres et valeur de retour

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

Il s'agit d'une situation fréquente mais pas obligatoire que nous pouvons généraliser. De manière générale nous pouvons concevoir trois sortes de paramètres. Nous verrons ensuitece qu'il en est en langage Java.

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

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

elle ne sera pas modifiée. On pourrait faire suivre le nom du paramètre d'une flèche vers le bas (↓) pour rappeler son rôle mais ce n'est pas obligatoire lorsqu'il n'y a pas d'ambiguïté.

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

```
// Code
// appelant
myAlgo(expr)
```

pseudocode

```
// Code appelé
algorithm myAlgo(par↓ : integer)
    ...
```

pseudocode

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

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

```
1:  algorithm test()                                          // Code appelant
2:      real max
3:      max = max3(3, 2, 5)
4:      print max
5:
6:
7:
8:  algorithm max3(a↓, b↓, c↓ : reals) → real                 // Code appelé
9:      reals maxab, max
10:     maxab = max2(a,b)
11:     max = max2(maxab,c)
12:     return max
13:
14:
```

pseudocode

Traçons son exécution.

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

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

7.3.2 Le paramètre en entrée-sortie

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

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

Cela signifie que l'algorithme a pour but de modifier le paramètre. Un tel paramètre parrait être suivi d'une double flèche (↓↑).

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

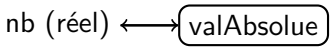
```
// Code
// appelant

myAlgo(variable)
```
pseudocode

```
// Code appelé
algorithm myAlgo(par↓↑ : entier)
    ...
```
pseudocode

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

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




```
1:  algorithm valAbsolue(nb↓↑ : réel)
2:      if nb<0 then
3:          nb = -nb
4:
5:
6:
7:
8:  algorithm test()
9:      réel température
10:     température = -12.5
11:     valAbsolue(température)
12:     print température
13:
14:
```
pseudocode

Traçons-le.

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

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

7.3.3 La valeur de retour

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

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

pseudocode

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

pseudocode

Remarques

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

7.4 Type primitif et type référence

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

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

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

7.4.1 Type primitif

Définition : Une variable de type primitif est une variable qui contient directement la valeur qui lui est assignée. Cette variable a une taille fixe qui dépend de son type. L'emplacement mémoire qui lui est attribué se trouve sur la pile (*stack*[50]).

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

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

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

PrimitiveType:
    NumericType
    `boolean`

NumericType:
    IntegralType
    FloatingPointType

IntegralType:
    (one of)
    `byte short int long char`

FloatingPointType:
    (one of)
    `float double`

Chaque type a une taille déterminée.

Les entiers sont codés en notation en complément à deux, excepté le type char qui est un entier non-signé de 16 bits représentant le code Unicode codé en UTF-16 du caractère[51].

Voici les tailles et les intervalles.

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

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

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

Les nombres pseudo-réel, ou encore les nombres à virgule flottante, sont codés suivant la norme IEEE 754 [52]. Selon cette norme, un nombre est représenté avec un signe, une mantisse et un exposant. Le tout en base 2. Un bit est utilisé pour le signe.

$$nombre = signe\ mantisse_2\ 2^{exposant_2}$$

type	taille (*bit*)	exposant	mantisse
float	32	8	23
double	64	11	52

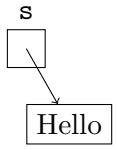
Pour les booléens, bien qu'un bit suffirait, la taille dépend de l'architecture et de la *jvm*.

7.4.2 Type référence

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

String est un type référence.

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

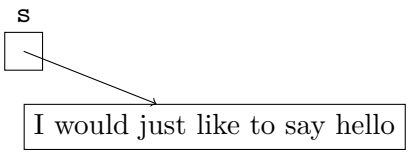


La même variable `s` peut recevoir une autre valeur, par exemple beaucoup plus grande I would just like to say

52. `https://fr.wikipedia.org/wiki/IEEE_754`

hello.

```java
String s = "Hello";
s = "I would just like to say hello";
```



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

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

7.4.3 Les paramètres en Java

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

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

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

Traduisons l'exemple de la section 7.3.1 (p.81) en Java. Les valeurs 3, 2 et 5 sont passées en argument à la méthode max3. Les variables a, b et c, locales à la méthode max3, reçoivent les trois valeurs.

```java
public class Test{
  public static void main(String[] args){
    double max;
    max = max3(3,2,5);
    System.out.println("Maximum: " + max);
  }

  public double max2(double a, double b){
    //
  }

  public double max3(double a, double b, double c){
    double maxab;
    double max;
    maxab = max2(a,b);
    max = max2(maxab, c);
    return max;
  }
}
```

java

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

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

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 si vous avez utilisé *studio.code.org* comme proposé dans la section 1.3 « Ressources » (cfr. p 14)

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

Contenu

8.1 La notion de travail répétitif

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

- le travail à répéter ;

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

Examinons quelques exemples pour fixer notre propos.

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

▷ La tâche à répéter est : « traiter un dossier ».

▷ Quand continuer : « s'il reste encore un dossier à traiter ».

Nous aurions pu dire de manière semblable :

▷ La tâche à répéter est : « traiter un dossier ».

▷ Quand s'arrêter : « dès qu'il n'y a plus de dossier à traiter ».

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

▷ La tâche à répéter est : « calculer la cote d'un · e étudiant · e ».

▷ Combien de fois faire le travail : « autant qu'il y a d'étudiant · es »
Il faut le faire pour tous les étudiants et les étudiantes. Nous pourrions être plus précis et dire qu'il faut commencer au premier, passer à chaque fois au suivant et s'arrêter lorsque c'est terminé avec le dernier.

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

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

▷ Nous indiquons qu'il faut le faire pour tous les nombres de 1 à 100. Il faut commencer à 1, passer au suivant et s'arrêter après avoir afficher 100.

8.2 Une même instruction, des effets différents

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

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

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

8.2.1 Exemple - Afficher les nombres de 1 à 5

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

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

langage naturel

Ces cinq instructions sont proches mais pas tout-à-fait identiques. En l'état, nous ne pouvons pas encore en faire une boucle [53] ; il va falloir ruser. Nous pouvons obtenir le même résultat avec l'algorithme suivant :

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

langage naturel

ou encore

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

langage naturel

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

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

langage naturel

Cette tâche doit être effectuée cinq fois dans notre exemple. Il existe plusieurs structures répétitives qui vont se distinguer par la façon dont on va contrôler le nombre de répétitions. Voyons-les une à une [54].

53. 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 ?

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

8.3 Le « tant que » (*while*)

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

```
tant que une certaine condition est vraie, répeter
    une ou plusieurs actions
fin tant que
continuer l'algorithme
```

langage naturel

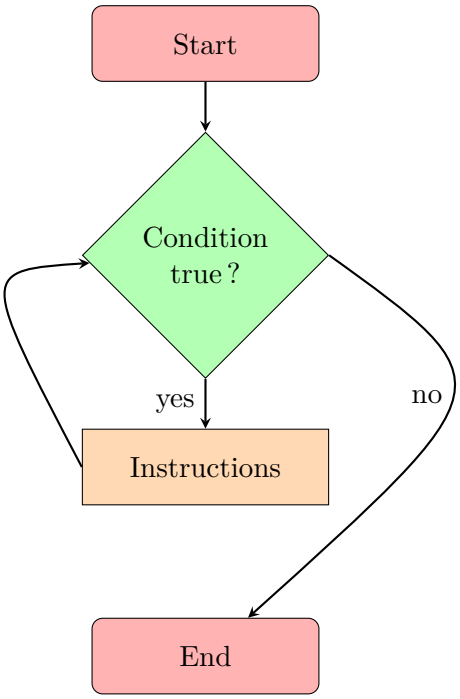
À la fin de chaque exécution des actions, la condition est retestée. Tant qu'elle est vraie les actions sont exécutées à nouveau.

Dans la grammaire du langage, cette instruction s'écrit :

```
while ( Expression )
    Statement
```

Expression est une expression booléenne comme nous l'avons vu (cf. section 6.4 p. 74) et *Statement* peut être une seule instruction ou un bloc d'instructions délimitées par des accolades.

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



Remarques

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

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

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 :

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

langage naturel

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`) :

```java
public static void count5(){
  int nb = 1;
  while (nb <= 5){
    System.out.println(nb);
    nb = nb + 1;
  }
  System.out.println(
    "nb vaut: " + nb);
}

```

java

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

8.3.2 Exemple - Généralisation à n nombres

Généraliser l'algorithme pour qu'il reçoive le nombre d'itérations en paramètres se fait simplement comme suit :

```java
public static void count(int n){
  int i = 1;
  while (i <= n){
    System.out.println(i);
    i = i + 1;
  }
  System.out.println(
    "i vaut: " + i);
}
```

java

8.4 Le « pour » (*for*)

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

```
pour une variable allant d'une valeur de départ à une valeur finale
    exécuter une ou plusieurs actions
fin pour
continuer l'algorithme
```

langage naturel

Ou, en étant un peu plus précis et en précisant un pas différent de 1 :

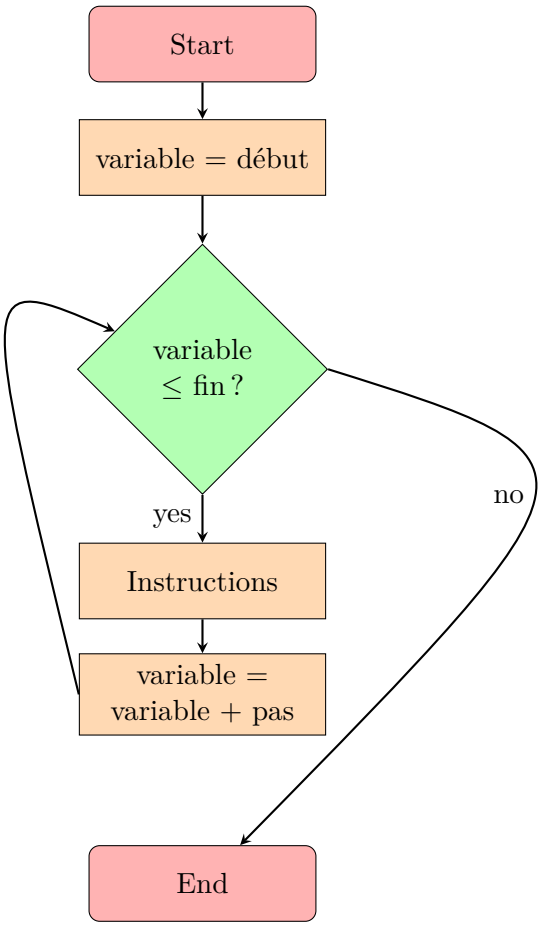
```
pour variable = début jusqu'à fin par pas
    exécuter une ou plusieurs actions
fin pour
continuer l'algorithme
```

langage naturel

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

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

L'organigramme ci-contre illustre le fonctionnement d'une boucle for.



Dans la grammaire du langage Java, l'instruction *for* est un peu plus large que de dire qu'une variable parcourt les valeurs allant de début à fin. Voici sa grammaire (simplifiée).

> *ForStatement:*
> `for (` *ForInit*`;` *Expression*`;` *ForUpdate*`)`
> *Statement*

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

Expression
est la condition de fin ;

ForUpdate
est l'incrément, le passage à l'itération suivante ;

Par exemple variable = variable + pas

Une utilisation simple et fréquent du pour à cette forme :

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

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

8.4.1 Exemples

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

```java
public static void countTo5({
  for (int nb=1; nb<= 5; nb = nb+1){
    System.out.printl(i);
  }
  System.out.println(
    "nb␣n'existe␣plus");
}

```

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

Pour généraliser à n nombres, nous pouvons écrire :

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

Remarques

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

8.4.2 Un pas négatif

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

Exemple : Compte à rebours à partir de n.

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

8.4.3 Remarques

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

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

8.4.4 Exemple – Afficher uniquement les nombres pairs

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

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

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

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

8.5 « faire – tant que »

À l'inverse du « tant que - faire » qui peut ne pas exécuter la séquence d'instructions si le test est faux dès le départ, il existe une structure qui exécutera au moins une fois la séquence d'instructions avant de tester la condition. Cette instruction peut-être :

- ▷ « faire - jusqu'à ce que » ou ;
- ▷ « faire - tant que ».

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

```
faire
    une ou plusieurs actions
jusqu'à ce qu'une certaine condition soit vraie
continuer l'algorithme
```

langage naturel

ou

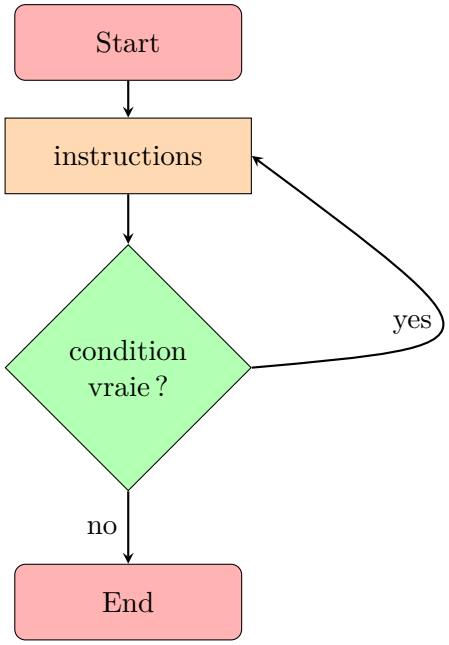
```
faire
    une ou plusieurs actions
tant que une certaine condition est vraie
continuer l'algorithme
```

langage naturel

En Java, c'est le second choix qui est retenu

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

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



8.5.1 Exemple

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

```java
public static void count5(){
  int nb = 1;
  do {
    System.out.println(nb);
    nb = nb + 1;
  } while (nb <= 5);
  System.out.println("nb␣
    vaut␣" + nb);
}
```

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

Exercice Tracer l'algorithme si l'instruction 2 est `int nb = 10;`.

8.6 Quel type de boucle choisir ?

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

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

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

La figure fig.8.1 ci-dessous propose un récapitulatif.

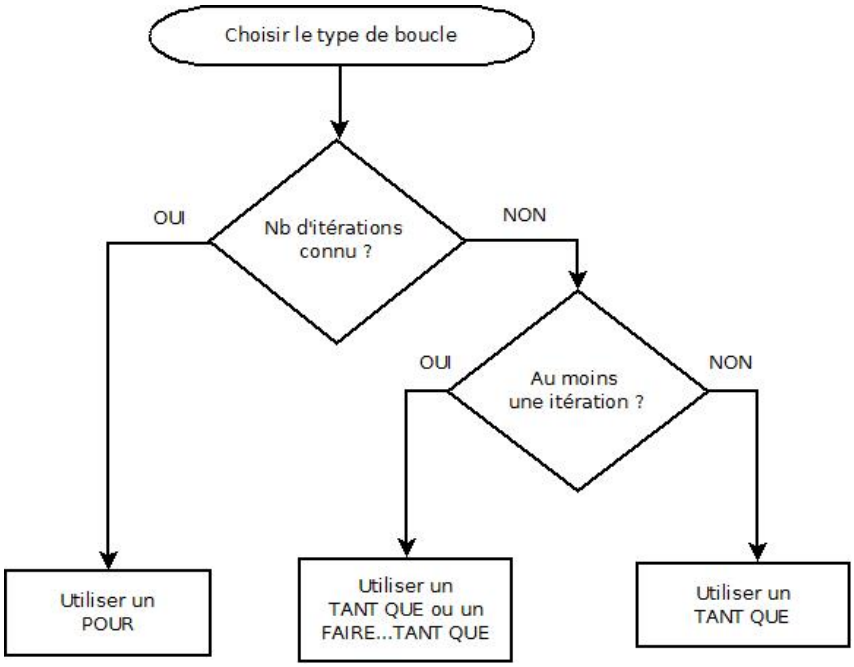


FIGURE 8.1 – Récapitulatif, choix d'une structure répétitive

8.7 Exercices résolus

8.7.1 Saisie des données par l'utilisateur

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

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

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

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

8.7.1.1 Variante 1 : le nombre de valeurs est connu

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

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

Une solution en Java peut-être :

```java
public static int sum(){
  int nValues;     // Nombre de valeurs à additionner
  int value;       // Terme de l'addition
  int sum = 0;     // Somme construite au fur et à mesure
  Scanner keyboard = new Scanner(System.in); // nécessite un import

  nValues = keyboard.nextInt();
  for  (int i = 0; i < nValues; i = i + 1){
    value = keyboard.nextInt();
    sum = sum + value;
  }
  return sum;
}
```

java

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

```java
value = keyboard.nextInt();
```

java

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

```java
System.out.print("Entrer␣un␣entier:␣");
value = keyboard.nextInt();
```

java

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

```java
System.out.print("Entrer␣un␣entier:␣");
while (!keyboard.hasNextInt()){
  keyboard.next();
}
value = keyboard.nextInt();
```

java

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

- ▷ Si c'est `true`, c'est un entier. Le programme n'entre pas dans la boucle et se rend directement à l'instruction suivante. Cette instruction consomme la valeur

lue précédemment et la place dans la variable value.

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

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

8.7.1.2 Variante 2 : stop ou encore

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

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

Si nous demandons en fin de boucle s'il reste encore un nombre à additionner l'algorithme aura l'allure suivante où nous utilisons un « do - while » :

```java
public static int sum(){
  /* Est-ce qu'il reste une variable à additionner (hasMore) */
  boolean hasMore;
  int value;
  int sum = 0;
  Scanner keyboard = new Scanner(System.in); // nécessite un import

  do {
    value = keyboard.nextInt();
    sum = sum + value;
    hasMore = keyboard.nextBoolean();
  } while (hasMore);
  return sum;
}

```

java

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

```java
public static int sum(){
  /* Est-ce qu'il reste une variable à additionner (hasMore) */
  boolean hasMore;
  int value;
  int sum = 0;
  Scanner keyboard = new Scanner(System.in); // nécessite un import

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

java

8.7.1.3 Variante 3 : valeur sentinelle

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

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

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

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

```java
public static int sum(){
  int value;      // Un des termes de l'addition
  int sum = 0;    // La somme
  Scanner keyboard = new Scanner(System.in); // nécessite un import

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

java

8.7.2 Les suites

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

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

8.7.2.1 Exemple 1 - Afficher les carrés

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

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

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

L'algorithme qui en découle est :

```java
public static void squarres(int n){
  for (int i = 1; i <= n; i++){
    System.out.printl(Math.pow(i, 2));
  }
}
```

Dans cette solution, la variable de contrôle compte simplement le nombre d'itérations. Le calcul du nombre à affiche se fait en fonction de cette variable de contrôle (ici le carré convient). Par une vieille habitude des programmeurs [55], une variable de contrôle qui se contente de compter les passages dans la boucle est souvent nommée i. Elle es appelée « indice » de parcours de la boucle.

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

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

$$i \longrightarrow f(i)$$

8.7.2.2 Exemple 2 - Une suite un peu plus complexe

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

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

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

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

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

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

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

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

Mathématiquement, cette suite s'écrirait

$$x_0 = 1$$
$$x_i = x_{i-1} + i$$

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

```
value = première valeur à afficher
for i de 1 à n
    print value
    value = valeur suivante calculée à partir de la précédente
```

langage naturel

qui, dans notre exemple précis, devient :

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

java

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

Chapitre 9

Les tests

« J'ai fait tourner le programme, il fonctionne bien. »

Tous les programmes contiennent des erreurs, des *bugs*, des défauts... Ces « *bugs* » entrainent, au mieux, un inconfort pour l'utilisateur ou l'utilisatrice. Ils peuvent occasionner une perte de temps, d'argent, de données, de matériel... ou, pire, un danger pour la vie humaine dans un environnement industriel par exemple.

Le travail de développeur ou de la développeuse est de réduire le nombre d'erreurs dans son programme. Pour ce faire, il ou elle le **testera** abondamment. Ce chapitre s'intéresse aux tests.

Contenu

9.1 Les types d'erreurs et de tests

Les défauts d'un programme peuvent être de différentes sortes :

1. les erreurs de compilation ;
   Ces erreurs sont détectées par le compilateur et rapidement corrigées.
2. les erreurs d'exécution
   ... et le programme s'arrête ;
   ... et le programme ne fournit pas les bonnes valeurs ;
   Ces erreurs sont difficiles à corriger. Elles nécessitent que le programme soit **testé** avec un nombre suffisant de situations différentes. Elles peuvent être dues à des erreurs de programmation mais aussi à des erreurs dans nos algorithmes.

3. les erreurs de programmation qui entrainent une lenteur du programme ou une consommation excessive de mémoire.

   Pour éviter ces erreurs, respecter les bonnes pratiques de développement est une première étape qu'il ne faut pas négliger. Ensuite, il faudra détecter les endroits dans le code où les ralentissements ont lieu ou la consommation de mémoire augmente. Il existe des logiciels dédiés à ce genre de cas.

Pour montrer — garantir, prouver — qu'un programme fonctionne bien, il ne suffit pas de montrer qu'il fournit les bons résultats dans certains cas, il faut convaincre, qu'il fournira les bons résultats dans **tous** les cas :

- ▷ les cas généraux ;
- ▷ les cas particuliers ;
- ▷ lors d'une défaillance de l'environnement ;

  Par exemple, si un capteur de distance est défectueux, il faut que la chaine de production s'arrête.
- ▷ lors d'une utilisation non conforme du programme.

Pour répondre à ces attentes, le développeur ou la développeuse sera attentive [56] à utiliser une méthodologie éprouvée dans toutes les étapes de développement de son projet ; de la phase d'analyse au déploiement. Il mettra en place différents types de tests [57] ; des tests unitaires, d'intégration, fonctionnels, de non-régression... Dans ce premier cours de développement, nous nous intéressons aux **tests unitaires**. Pour ces tests unitaires, le développeur ou la développeuse fournira un ensemble de valeurs à tester représentatives et convaincantes.

Définition

Test unitaire Procédure permettant de tester le bon fonctionnement d'une unité de code, une méthode Java dans notre cas. En fonction des paramètres qu'elle reçoit en entrée, la méthode fournit-elle les bons résultats ?

L'idée sous-jacente des tests unitaires est que si chaque partie est correcte, l'ensemble sera correct.

Pour tester régulièrement — très régulièrement — notre programme nous avons besoin d'outils qui vont automatiser ces tests mais commençons par les planifier.

9.2 Planifier les tests

Dans la méthode de résolution de problèmes que nous avons présenté dans le chapitre 1 p. 9 et dans l'exercice résolu de la section 4.1 p. 31 nous précisions que l'écriture d'une solution complète dans ce premier cours de développement consistait en :

1. spécifier le problème ;
2. fournir des exemples ;

56. Je m'autorise un accord de proximité.
57. Voir `https://fr.wikipedia.org/wiki/Test_(informatique)`

3. écrire un algorithme ;
4. vérifier les exemples (en traçant l'algorithme) ;
5. écrire un programme exécutable correspondant à l'algorithme ;
6. tester le programme et constater qu'il fournit bien les résultats attendus.

Les points 2 et 4 sont les prémisses à l'écriture des tests puisque ces exemples vont nous convaincre que l'algorithme — et le programme — fournit bien les « bons » résultats. Ces exemples constituent le **plan de tests** que nous allons utiliser. Dans ces différents cas, outre les cas généraux, les cas particuliers, les valeurs limites, apparaitront aussi des cas montrant les erreurs de programmation fréquentes. Par exemple :

- ▷ commencer ou arrêter trop tôt ou trop tard une boucle ;
- ▷ ne pas initialiser ou mal initialiser une variable ;
- ▷ confondre $<$ et $\leq$ ou $>$ et $\geq$... voire $<$ et $>$;
- ▷ confondre AND et OR ;
- ▷ mal écrire la négation d'une proposition ;
- ▷ ...

C'est l'expérience qui permettra d'écrire un plan de tests efficace et convaincant.

Un **plan de test** est donc une liste ou un tableau reprenant un certain nombre d'exemples qui mettent en œuvre la méthode testée sur des cas tels que ceux décrits ci-dessus et permettent de faire apparaitre des problèmes lors de l'exécution automatique. Chacun des tests sera défini par une méthode qui porte un nom qui explique l'objectif du test sans la nécessité de commentaires additionnels. Autant que possible, ce nom décrit aussi la portée ou le contexte de ce qui est vérifié.

Remarque Dans certaines méthodologies, la mise en exergue de l'importance des tests se fait en écrivant d'abord les tests avant le code. Voir par exemple Wikipedia [58] ou cette article d'*extreme programming* [59][Wel00].

Dès lors que le plan de tests est écrit, nous pouvons nous intéresser aux outils qui vont permettre d'automatiser l'exécution des tests et faciliter leur écriture. En effet pour que nos tests soient utiles il faudra tester souvent — idéalement dès qu'une méthode est écrite — et « tout ». Ce sont toutes les méthodes qui sont testées parce que l'écriture d'une méthode pourrait amener une erreur dans une autre méthode. Dans ces conditions, pour que le développeur ou la développeuse teste son programme, ce doit être automatique.

Cette automatisation est fournie par JUnit.

9.3 JUnit

Remarque préalable JUnit n'est pas fourni avec le JDK (*java development kit*), c'est un *framework* de tests indépendant que l'on trouve sur `junit.org`. Il est donc nécessaire de l'installer.

58. `https://fr.wikipedia.org/wiki/Test_driven_development`
59. `http://www.extremeprogramming.org/rules/testfirst.html`

Par contre, il est fourni avec Netbeans.

JUnit est un *framework* simple pour l'écriture de tests répétables.

Pour tester la méthode — `foo` — dans la classe `MyClass`, il faut écrire une classe `MyClassTest` qui contiendra les méthodes de tests. Le nom de la classe de test porte le nom de la classe qu'elle teste, suffixé par `Test`. Ensuite, il faut demander à JUnit — dès lors qu'il est installé — d'exécuter cette classe. Un clic dans un IDE ou une commande du style fait l'affaire :

```
$
java org.junit.runner.JUnitCore my.package.MyClassTest
```
terminal

La classe `MyClassTest` contient plusieurs méthodes. Une méthode de test par cas. Une méthode de test :

- ▷ est autonome. Elle ne reçoit pas de paramètre et ne retourne rien ;
- ▷ contient des affirmations qui seront évaluées ;
  La méthode doit me retourner « vrai ». : assertTrue()
  La méthode doit me retourner « faux ». : assertFalse()
  La méthode doit me donner cette valeur si ses paramètres sont ceux-ci. : assertEquals(<valeur attendue>, <valeur>)
  Ces trois méthodes assertTrue, assertFalse, assertEquals sont définies dans la classe org.junit.Assert. On peut les utiliser directement après les avoir préalablement importées :

```java
import static org.junit.Assert.*;
```
java

- ▷ est précédée de l'annotation @Test permettant de la faire reconnaitre comme un test unitaire ;
- ▷ n'a pas de mot clé static.

Un test aura donc l'allure suivante :

```java
@Test
public void max2_cas1(){
  assertEquals(2, max2(-1, 2));
}
```
java

Après lancement des tests, JUnit fournira un rapport l'allure suivante :

```
$
JUnit version 4.12
.

Time: 0,012

OK (1 test)
```

terminal

9.4 Exemple

Reprenons l'exemple du calcul du maximum de 2 nombres décrit dans la fiche 4 p 212. Voici un plan de tests :

test n°	description	nb1	nb2	réponse attendue
1	maximum en position 2 et nombre négatif	-3	4	4
2	maximum en position 1	7	4	7
3	tous égaux	4	4	4
4	avec un nul	0	-4	0

Et la classe de tests `MaximumTest` sans les commentaires.

```java
package be.he2b.esi.cours.dev1;

import org.junit.Test;
import static org.junit.Assert.*;

public class MaximumTest {


  @Test
  public void max2_maximumEnPosition2EtUnNégatif(){
    assertEquals(4, max2(-3, 4));
  }

  @Test
  public void max2_maximumEnPosition1(){
    assertEquals(7, max2(7, 4));
  }

  @Test
  public void max2_tousÉgaux(){
    assertEquals(4, max2(4, 4));
  }

  @Test
  public void max2_avecUnNul(){
    assertEquals(0, max2(0, -4));
  }
}
```

java

```
$
JUnit version 4.12
....

Time: 0,012

OK (4 tests)
```

terminal

Troisième partie

Les tableaux

Chapitre 10

Les tableaux

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

Contenu

10.1 Utilité des tableaux

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

Exemple. Statistiques de 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 -1 sera introduite ;
- ▷ le traitement final, consistant à communiquer les valeurs des compteurs pour chaque produit.

Vous trouverez sur la page suivante une version possible de cet algorithme sans utiliser les tabeaux.

```java
import java.util.Scanner;
/*
 Calcule et affiche la quantité vendue de 10 produits
*/
public static void statistics(){
  Scanner keyboard = new Scanner(System.in);
  int cpt0, cpt1, cpt2, cpt3, cpt4, cpt5, cpt6, cpt7, cpt8, cpt9;
  int productId, quantity;
  cpt0 = 0;
  cpt1 = 0;
  cpt2 = 0;
  cpt3 = 0;
  cpt4 = 0;
  cpt5 = 0;
  cpt6 = 0;
  cpt7 = 0;
  cpt8 = 0;
  cpt9 = 0;
  System.out.print("Introduire␣le␣numéro␣de␣produit␣:␣");
  productId = keyboard.nextInt();
  while (productId != -1){
    System.out.print("Quantité:");
    quantity = keyboard.nextInt();
    switch (productId){
      case 0:
        cpt0 = cpt0 + quantity;
        break;
      case 1:
        cpt1 = cpt1 + quantity;
        break;
      case 2:
        cpt2 = cpt2 + quantity;
        break;
      case 3:
        cpt3 = cpt3 + quantity;
        break;
      case 4:
        cpt4 = cpt4 + quantity;
        break;
      case 5:
        cpt5 = cpt5 + quantity;
        break;
      case 6:
        cpt6 = cpt6 + quantity;
        break;
      case 7:
        cpt7 = cpt7 + quantity;
        break;
      case 8:
        cpt8 = cpt8 + quantity;
        break;
      case 9:
        cpt9 = cpt9 + quantity;
        break;
    }
    System.out.print("Introduire␣le␣numéro␣de␣produit␣:␣");
    productId = keyboard.nextInt();
  }

```

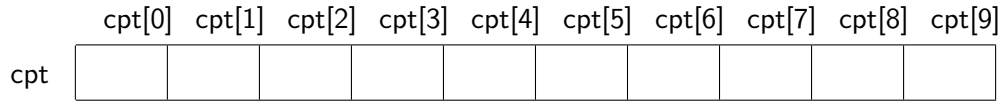
java

```java
    System.out.println("Quantité␣vendue␣de␣produit␣0" + cpt0);
    System.out.println("Quantité␣vendue␣de␣produit␣1" + cpt1);
    System.out.println("Quantité␣vendue␣de␣produit␣2" + cpt2);
    System.out.println("Quantité␣vendue␣de␣produit␣3" + cpt3);
    System.out.println("Quantité␣vendue␣de␣produit␣4" + cpt4);
    System.out.println("Quantité␣vendue␣de␣produit␣5" + cpt5);
    System.out.println("Quantité␣vendue␣de␣produit␣6" + cpt6);
    System.out.println("Quantité␣vendue␣de␣produit␣7" + cpt7);
    System.out.println("Quantité␣vendue␣de␣produit␣8" + cpt8);
    System.out.println("Quantité␣vendue␣de␣produit␣9" + cpt9);
}
```

java

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

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



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 de notre solution avec tableau.

```java
import java.util.Scanner;
public static void statisticsWithArray(){
  Scanner keyboard = new Scanner(System.in);
  int[] cpt = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  int i;
  int productId;
  int quantity;
  System.out.print("Introduire le numéro de produit : ");
  productId = keyboard.nextInt();
  while (productId != -1){
    System.out.print("Quantité:");
    quantity = keyboard.nextInt();

    cpt[productId] = cpt[productId] + quantity;

    System.out.print("Introduire le numéro de produit : ");
    productId = keyboard.nextInt();
  }

  for(int i=0; i<10; i++){
    System.out.println("Quantité vendue de produit " i + ": " +
    cpt[i]);
  }
}
```

java

10.2 Définitions

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

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

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

Un tableau est de type **référence**.

10.3 Déclaration - création - initialisation

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

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

- l'**initialisation** du tableau consiste à donner ses premières valeurs aux différents éléments du tableau.

Nous supposerons dans la suite que length est le nombre d'éléments — la longueur — et Type est le type des éléments que l'on trouvera dans le tableau.

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

10.3.1 Déclaration

Pour **déclarer** un tableau :

> *ArrayDeclarationExpression:*
> *Type*`[]` *Identifier*

```java
int[] myArray;
String[] strings;
```

Seule la variable est déclarée. Aucun espace mémoire n'est réservé pour recevoir les éléments du tableau. Ceci sera fait lors de la création du tableau.

10.3.2 Création

Pour **créer** un tableau, préalablement déclaré, nous écrirons :

> *ArrayCreationExpression:*
> `new` *Type*`[`*Expression*`]`

```java
myArray = new int[10];
strings = new String[3];

```

L'emplacement mémoire nécessaire pour stocker la valeur de chaque élément du tableau est réservé.

10.3.3 Initialisation

Pour **initialiser** les éléments du tableau, nous pouvons le faire en accédant à chaque élément du tableau. Un par un.

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

```java
    myArray[0] = 5;
    strings[1] = "Hello";

```

La première case du tableau porte l'indice 0 la dernière l'indice size-1. C'est une erreur d'indiquer un indice qui ne correspond pas à une case du tableau (trop petit ou trop grand). Par exemple, si on déclare et crée le tableau myArray :

```java
int[] myArray = new int[100];
```

c'est une erreur d'utiliser myArray[-1] ou myArray[100].

Il est également possible d'initialiser un tableau en une seule fois, en utilisant un *array initializer*. Une notation entre accolades.

```
myArray = {2, 3, 7}
```

langage naturel

Voici la grammaire Java.

ArrayInitializer:
{ *[VariableInitializerList]* }

VariableInitializerList:
Expression [, *Expression*]

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

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

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

```

ligne 7
Pour utiliser un *ArrayInitializer*, il faut créer et initialiser le tableau dans la même instruction. Dans ce cas, le langage interdit de renseigner le nombre d'éléments du tableau... le compilateur peut le déduire.
Cette instruction crée un nouveau tableau qui écrase l'autre.

ligne 9
Lorsque l'on initialise le tableau avec un *ArrayInitializer* lors de la déclaration, la partie création peut être omise.

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

10.4 Tableau et paramètres

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

Nous avons vu dans la section 7.3 p. 81 les différentes sortes de paramètres. Voyons ce qu'il en est pour chacun d'eux lorsque ce paramètre est un tableau.

10.4.1 Un tableau comme paramètre en entrée

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

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




// Utilisations possible
array of 10 integers anArray
anArray = {2,3,5,7,11,13,17,19,23,29}
display(anArray)
```

pseudocode

10.4.2 Un tableau comme paramètre en entrée-sortie

↓↑ : indique que l'algorithme va consulter/modifier les valeurs du tableau reçu en paramètre. Exemple :

```
// Inverse le signe des éléments du tableau
algorithm reverseSigne(myArray↓↑ : array of 10 integers)
    for i from 0 to 9
        myArray[i] = -myArray[i]




// Utilisation possible
array of  integers anArray
anArray = array of  integers {2,-3,5,-7,11,13,17,-19,23,29}
reverseSigne(anArray)
```

pseudocode

10.4.3 Un tableau comme paramètre en langage Java

En langage Java, le passage de paramètres se fait toujours par valeur. Comme un tableau est de type référence, c'est bien la valeur de la référence qui est passée en

paramètre. En ce sens, il sera possible de modifier les valeurs d'un tableau reçu mais pas de remplacer le tableau reçu par un autre. Exemples :

```java
public static void reverseSigne(int[] is){
  for (int i=0; i < is.length; i = i + 1){
    is[i] = -is[i];
  }
}

// Utilisation
int[] myArray = {2,-3,5,-7,11,13,17,-19,23,29};
reverseSigne(myArray);
```

java

```java
public static void fail(double[] ds){
  ds = new double[] {2, 2.61};
}


// Utilisation
double[] myArray = {1, 3.14};
fail(myArray);
// 3.14 et pas 2.61
System.out.println("Value:␣" + myArray[1]);
```

java

10.4.4 Un tableau comme type de retour

Comme pour n'importe quel autre type, un algorithme ou un programme peut retourner un tableau. Ce sera à lui de le déclarer et de lui donner des valeurs. Exemple :

```java
public static int[] create(){
  int[] myArray = new int[10];
  for (int i=0; i < 10; i = i + 1){
    myArray[i] = i;
  }
  return myArray;
}


// Utilisation possible
int[] anArray = create();
```

java

10.4.5 Paramétrer la taille

Un tableau connait sa taille dès lors qu'elle existe c'est-à-dire dès que le tableau est créé. La taille est accessible grâce au mot length comme suit :

```java
double[] myArray;
int size = myArray.length;

double[] otherArray = {3.14, 2.71};
int otherSize = otherArray.length;
```

Un tableau peut être passé en paramètre à un algorithme ou à une méthode et ce, quelle que soit la taille du tableau passé en paramètre. L'algorithme ou la méthode aura comme paramètre un tableau qui connait sa taille. Lorsque l'algorithme ou la méthode recevra le paramètre effectif, la taille sera déterminée.

Exemple :

```java
public static void display(int[] myArray){
  System.out.println("Tableau de "
    + myArray.length
    + " éléments." );
  for (int i=0; i< myArray.length; i++){
    System.out.println(myArray[i]);
  }

  // Utilisations possibles
  int[] integers = new int[2];
  integers[0] = 56;
  integers[1] = -3;
  display(integers);

  int[] myBeautifulArray = {6, 12, 18};
  display(myBeautifulArray);
}

```

La fiche **??** page ?? récapitule tout ça.

10.5 Parcours d'un tableau

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

Parcours complet

Nous avons déjà vu dans les exemples comment parcourir tous les éléments d'un tableau. Une boucle for fait généralement l'affaire comme ceci :

```java
int[] myArray = { /* insert values */ };
for (int index = 0; index < myArray.length; index++){
  // do something with myArray[index]
}
```

Java propose une autre forme de for — l'*enhanced for* ou *foreach* — permettant de parcourir un tableau [60]

```java
int[] myArray = { /* insert values */ };
for (int value: myArray){
  // do something with value
}
```

java

La fiche 5 page 215 décrit comment afficher tous les éléments d'un tableau. Il est possible de faire autre chose avec ces éléments : les sommer, les comparer...

Parcours partiel

Parfois, il n'est pas nécessaire de parcourir le tableau jusqu'au bout car un arrêt prématuré survient dès qu'une condition est remplie.

Par exemple : recherche d'un élément et arrêt dès qu'il est trouvé ; vérification que toutes les valeurs sont non nulles mais l'on vient d'en trouver une ; vérification que la suite des éléments est positives alors que l'on vient d'en trouver un négatif ou nul.

Pour résoudre ce problème, la structure tant que avec un test d'arrêt fait bien l'affaire.

La fiche 7 page 220 détaille un parcours partiel.

10.6 Taille logique et taille physique

Parfois, la taille du tableau n'est pas connue à l'avance.

Imaginons, par exemple, qu'il s'agisse de demander des valeurs à l'utilisateur et de les stocker dans un tableau pour un traitement ultérieur. Supposons que l'utilisateur va indiquer la fin des données par une valeur sentinelle. Impossible de savoir, à priori, combien de valeurs il va entrer et, par conséquent, la taille à donner au tableau.

Une solution est de créer un tableau suffisamment grand pour tous les cas [61] quitte à n'en n'utiliser qu'une partie.

Mais comment savoir quelle est la partie du tableau qui est effectivement utilisée ?

Comprenez bien qu'il **n'y a pas de concept de case vide**. Rien ne différencie une case utilisée d'une case non utilisée. La manière de s'en sortir est de stocker les valeurs dans la partie basse du tableau (à gauche) et de retenir dans une variable le nombre de cases effectivement utilisées.

La **taille physique** d'un tableau est le nombre de cases qu'il contient. Sa **taille logique** est le nombre de cases actuellement utilisées.

Exemple. Voici un tableau qui ne contient pour l'instant que quatre nombres.

10	4	3	7	?	?	?	?	?	?

60. Nous verrons en DEV2, que cette structure permet de parcourir d'autres choses que les tableaux.

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

tailleLogique = 4 tailleLogique = 4

Les cases indiquées par " ?" ne sont pas vides ; elles peuvent être non initialisées (et il n'est pas possible de tester si une case est non initialisée) ou bien contenir une valeur qui n'est pas pertinente.

Exemple. L'algorithme suivant demande des valeurs positives à l'utilisateur et les stocke dans un tableau (maximum 1000). Toute valeur négative ou nulle est une valeur sentinelle.

```java
public static void stockValues(){
  Scanner keyboard = new Scanner(System.in);
  int[] myArray = new int[1000];
  int nbValues = 0;
  int value;
  value = keyboard.nextint();
  while (value > 0 && nbValues < myArray.length){
    myArray[nbValues] = value;
    nbValues++;
    value = keyboard.nextint();
  }
  // Test de value et pas de nbValues. Pourquoi ?
  if (value > 0){
    System.out.println("La limite physique a été atteinte");
  }
}
```

java

En langage Java, essayer d'ajouter un élément en dehors de la taille du tableau va générer une erreur à l'exécution. Une ArrayIndexOutOfBoundException.

10.7 Les tableaux ont comme premier indice 0

Les tableaux ont comme premier indice 0, Un tableau de taille n a toujours ses indices allant de 0 à n-1.

Parfois le problème à résoudre donne envie d'avoir des indices dont le premier ne commence pas à 0. Par exemple allant de 1 à 7. Dans ces cas, il est toujours possible de s'arranger pour commencer à partir de 0.

Exemple. Imaginons l'algorithme qui permet de convertir un numéro de jour en son intitulé : 1 donne "lundi", 2 donne "mardi", ... Une solution possible, sans tableau, serait :

```java
public static String dayName(int dayNumber){
  String dayName;
  switch (dayNumber){
    case 1:
      dayName = "lundi";
      break;
    case 2:
      dayName = "mardi";
      break;
    case 3:
      dayName = "mercredi";
      break;
    case 4:
      dayName = "jeudi";
      break;
    case 5:
      dayName = "vendredi";
      break;
    case 6:
      dayName = "samedi";
      break;
    case 7:
      dayName = "dimanche";
      break;
    default:
      dayName = "inconnu";
  }
  return dayName;
}
```

java

Mais une solution plus élégante passe par l'utilisation d'un tableau. Appelons-le dayString et stockons-y les noms des jours comme illustré ci-dessous.

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

Si le tableau commence à l'indice 1 jusqu'à l'indice 7, pour obtenir le nom d'un jour, il suffirait d'écrire : dayString[dayNumber].

Comme les tableaux commencent toujours à l'indice 0, il faudra plutôt considérer le tableau ci-dessous et écrire dayString[dayNumber-1].

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

Chapitre 11

Algorithmes d'insertions et de recherches

Souvent lorsque les applications stockent des données. Ces données changent. Il est nécessaire d'en ajouter, de modifier certaines, d'en rechercher, d'en supprimer.

Contenu

Exemple Imaginons qu'une école organise un événement libre et gratuit pour les étudiants et les étudiantes. Mais pour y assister, il est nécessaire de s'inscrire. Nous voulons fournir ce qu'il faut pour :

- ▷ inscrire un ou une étudiant · e ;
- ▷ vérifier l'inscription ;
- ▷ désinscrire ;
- ▷ afficher la liste des personnes inscrites.

Nous allons, pour l'exemple, stocker les numéros d'étudiants et d'étudiantes dans un tableau [62]. Comme le tableau ne sera généralement pas rempli, nous allons créer un tableau « trop grand » et gérer sa taille logique. Nous avons deux variables :

- ▷ registered : le tableau des numéros d'étudiants
- ▷ nRegistered : le nombre d'étudiant · es actuellement inscrit · es (la taille logique du tableau)

Nous allons envisager deux cas :

1. les numéros seront placés dans le tableau sans ordre imposé ;
2. les numéros seront placés dans l'ordre.

11.1 Insertion dans un tableau non trié

Les valeurs sont stockées dans le tableau dans un ordre quelconque. Par exemple :

registered =

42010	41390	43342	42424	?	?	?	?	...

nRegistered = 4

Il y a pour le moment quatre personnes inscrites dont les numéros sont ceux repris dans les quatre premières cases du tableau.

11.1.1 Inscription

Pour inscrire un étudiant ou une étudiante, il suffit de l'ajouter à la suite dans le tableau. Par exemple, en partant de la situation décrite ci-dessus, l'inscription de l'étudiant numéro 42123 aboutit à la situation suivante :

registered =

42010	41390	43342	42424	42123	?	?	?	...

nRegistered = 5

62. Ce n'est bien sûr pas la bonne manière de faire pour une application réelle pour laquelle nous utiliserions probablement une base de données... mais c'est une autre histoire... et un autre cours.

L'algorithme est assez simple :

```
algorithme register(registered,nRegistered, number)
    registered[nRegistered] = number
    incrémenter nRegistered
```

langage naturel

En langage Java, il n'est pas possible de passer nRegistered en entrée sortie puisque le passage de paramètre se fait par valeur et que int est un type primitif. Il faudra que ce soit en valeur de retour. Nous pourrions écrire :

```java
public static int register(int[] registered, int nRegistered,
    int number){
  // On peut imaginer vérifier que l'étudiant n'est pas déjà
  unscrit
  // On peut imaginer vérifier qu'il reste de la place
  // (c-à-d que le tableau n'est pas plein)
  registered[nRegistered] = number;
  return nRegistered + 1;
}

```

java

Dans ce cas, c'est bien le code appelant qui devra maintenir la valeur de la taille logique à jour.

11.1.2 Vérifier une inscription

Pour vérifier si un étudiant est déjà inscrit, il faut le rechercher dans la partie utilisée du tableau. Cette recherche se fait via un parcours avec sortie prématurée.

L'algorithme ne va pas se contenter de retourner un booléen précisant si le numéro est trouvé mais va retourner l'indice dans le tableau de ce numéro, -1 sinon.

```java
public static int check(
  int[] registered,
  int nRegistered,
  int number){
    int i = 0;
    while (i < nRegistered && registered[i] != number){
      i++;
    }
    if (i<nRegistered){
      return i;
    } else {
      return -1,
    }
}

```

java

La fiche 8 page 223 reprend cet algorithme dans un cadre plus général.

11.1.3 Désinscription

Pour désinscrire un étudiant, il faut le trouver dans le tableau et l'enlever. Attention, il ne peut pas y avoir de trou dans un tableau (il n'y a pas de case vide). Le plus simple est d'y déplacer le dernier élément. Par exemple, reprenons la situation après inscription de l'étudiant numéro 42123.

registered =

42010	41390	43342	42424	42123	?	?	?	...

nRegistered = 5

La désinscription de l'étudiant numéro 41390 donnerait ce qui suit.

Nous avons volontairement indiqué le 42123 en 5e position. Il est toujours là mais ne sera pas considéré par les algorithmes puisque cette case est au-delà de la taille logique (donnée par la variable nRegistered).

registered =

42010	42123	43342	42424	42123	?	?	?	...

nRegistered = 4

L'algorithme est assez simple à écrire en utilisant la recherche écrite juste avant :

```java
public static int unsubscribe(int[] registered,
    int nRegistered, int number){
  int pos = check(registered, nRegistered, number);
  registered[pos] = registered[nRegistered - 1];
  return nRegistered - 1;
}
```

11.1.4 Exercices

1. Éviter la double inscription.
   Comment modifier l'algorithme d'inscription pour s'assurer qu'un étudiant ne s'inscrive pas deux fois ?
2. Limiter le nombre d'inscriptions.
   Comment modifier l'algorithme d'inscription pour refuser une inscription si le nombre maximal de participant est atteint en supposant que ce maximum est égal à la taille physique du tableau ?
3. Vérifier la désinscription.
   Que se passerait-il avec l'algorithme de désinscription tel qu'il est si on demande à désinscrire un étudiant non inscrit ? Que suggérez-vous comme changement ?
4. Optimiser la désinscription.
   Dans l'algorithme de désinscription tel qu'il est, voyez-vous un cas où on effectue une opération inutile ? Avez-vous une proposition pour l'éviter ?

11.2 Insertion dans un tableau trié

Imaginons à présent que nous maintenons un ordre dans les données du tableau. En reprenant l'exemple utilisé pour les données non triées, nous avons :

registered =

41390	42010	42424	43342	?	?	?	?	...

nRegistered = 4

— Qu'est-ce que ça change au niveau des algorithmes ?
— Beaucoup !

Par exemple, plus question de placer un nouvel inscrit à la suite des autres ; il faut trouver sa place.

11.2.1 Rechercher la position d'une inscription

Tous les algorithmes que nous allons voir dans le cadre de données triées ont une partie en commun : la recherche de la position du numéro, s'il est présent ou de la position où il aurait dû être s'il est absent. Commençons par cela.

L'algorithme est assez proche de celui de la vérification d'une inscription vu précédemment si ce n'est que l'algorithme peut s'arrêter dès qu'un numéro plus grand est trouvé. Nous allons décomposer l'algorithme en deux parties : la première pour la recherche de la position à laquelle devrait se trouver le numéro et la seconde pour vérifier que c'est bien le bon numéro qui se trouve à cette position.

```java
/*
 * Recherche d'un étudiant ou d'une étudiante par son numéro d'id.
 * return: l aposition où a été trouvé le numéro
 * ou -1 si non présent
 */
public static int find(
  int[] registered,
  int nRegistered,
  int number){
    int pos = findPosition(registered, nRegistered, number);
    if (registered[pos] == number){
      return pos;
    } else {
      return -1;
    }
}

/*
 * Recherche d'une position
 * return: la position à laquelle est ou devrais être le numéro
 */
public static int findPosition(
    int[] registered,
    int nRegistered,
    int number){
  int pos = 0,
  while (pos < nRegistered && registered[pos] < number){
    pos++;
  }
  return pos;
}

```

java

Comprenez-vous pourquoi :

- ▷ registered[nRegistered] $<$ number contient un '$<$' dans la condition du tant que et pas un '$\neq$' ?
- ▷ la condition du if est composée de deux parties et utilise un $=$?

La liste étant triée il faut s'arrêter de chercher dès que la valeur n'est plus inférieure à la valeur cherchée. Dès qu'elle est plus grande c'est que la valeur **n'est pas** dans la liste.

De même pour la condition du if, il faut d'abord tester si la recherche n'est pas arrivée à la fin du nombre d'inscrits et d'inscrites. Ensuite, il faut voir si le numéro à la position trouvée est bien le bon.

Le if tel que présenté dans l'algorithm peut se réécrire grâce à l'opérateur ternaire de Java. Profitons en pour le présenter.

> *ConditionalExpression:*
> *Condition* ? *Expression* : *Expression*

Si la condition est vraie, l'expression prend la première valeur et si la condition est fausse, ce sera la seconde valeur. Par exemple : heure < 12 ? "matin" : "soir" vaut "soir" si heure vaut 15.

```java
/**
 * Recherche une inscription dans le tableau.
 *
 * @param registered le tableau d'inscrit·es
 * @param nRegistered le nombre d'inscrit·es
 * @param number le numéro à chercher
 * @return la position dans le tableau, -1 si non présent
 */
public static int find(int[] registered, int nRegistered,
    int number){
  int pos = findPosition(registered, nRegistered, number);
  return registered[pos] == number ? pos : -1;
}

/**
 * Recherche la position que le nombre soit présent ou pas.
 * @param registered le tableau d'inscrit·es
 * @param nRegistered le nombre d'inscrit·es
 * @param number le numéro à chercher
 * @return la position à laquelle est ou devrait être le numéro.
 */
public static int findPosition{int[] registered, int nRegistered,
    int number){
  int pos = 0;
  while (pos < nRegistered && registered[pos] < number){
    pos = pos + 1;
  }
  return pos;
}
```

java

11.2.2 Inscrire un étudiant

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

1. recherche de l'étudiant via l'algorithme vu à l'instant, ce qui nous donne la place où le placer ;

   Nous ne vérifions pas si l'étudiant ou l'étudiante est déjà présent · e dans le tableau.

2. libération de la place pour l'y mettre ;

   Cette opération n'est pas triviale. Si vous tenez des cartes en main, il est facile d'insérer une nouvelle carte à n'importe quelle position. Avec un tableau, il en va tout autrement ; pour insérer un élément à un endroit donné, il faut décaler tous les suivants d'une position sur la droite.

3. placer l'élément à la position libérée.

Ce qui nous donne :

```java
/**
 * Inscrit un ou une étudiant·e.
 *
 * @param registered le tableau d'inscrit·es
 * @param nRegistered le nombre d'inscrit·es
 * @param number le numéro à chercher
 */
public static void subscribe(int[] registred, int nRegistered,
        int number){
  int pos = findPosition(registered, nRegistered, number);
  shiftRight(registred,pos, nRegistered);
  registered[pos] = number;
  nRegistered = nRegistered + 1;
}

/**
 * Décale les éléments du tableau de begin à end un à droite.
 * @param tab le tableau
 * @param begin l'indice de départ
 * @param end l'indice de fin
 */
public shiftRight(int[] tab, int begin, int end){
  for (int i = end, i >= begin; i = i - 1){
    tab[i + 1] = tab[i];
  }
}

```

java

11.2.3 Désinscription

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

```java
/**
 * Désinscrit un ou une étudiant·e.
 *
 * @param registered le tableau d'inscrit·es
 * @param nRegistered le nombre d'inscrit·es
 * @param number le numéro à chercher
 */
public static void unsubscribe(int[] registred, int nRegistered,
        int number){
  int pos = findPosition(registered, nRegistered, number);
  shiftLeft(registred,pos, nRegistered);
  nRegistered = nRegistered - 1;
}

/**
 * Décale les éléments du tableau de begin à end un à gauche
 * (et écrase donc le premier).
 * @param tab le tableau
 * @param begin l'indice de départ
 * @param end l'indice de fin
 */
public shiftLeft(int[] tab, int begin, int end){
  for (int i = begin, i <= end; i = i + 1){
    tab[i - 1] = tab[i];
  }
}

```

java

La fiche 9 page 227 reprend cet algorithme dans un cadre plus général.

11.2.4 Intérêt de trier les valeurs ?

Est-ce que trier les valeurs est vraiment intéressant ?

La recherche est un peu plus rapide. En moyenne deux fois plus rapide si l'élément ne s'y trouve pas. Par contre, l'ajout et la suppression d'un élément sont plus lents. Les algorithmes sont plus complexes à écrire et à comprendre. Le gain ne parait pas évident et en effet, en l'état, il ne l'est pas.

Mais c'est sans compter un autre algorithme de recherche, beaucoup plus rapide, la recherche dichotomique, que nous allons voir maintenant.

11.3 La recherche dichotomique

La recherche dichotomique est un algorithme de recherche d'une valeur dans un tableau trié. Il a pour essence de réduire à chaque étape la taille de l'ensemble de recherche de moitié, jusqu'à ce qu'il ne reste qu'un seul élément dont la valeur devrait être celle recherchée, sauf bien entendu en cas d'inexistence de cette valeur dans l'ensemble de départ.

Pour l'expliquer, nous allons considérer un tableau d'entiers complètement rempli. Il sera facile d'adapter la solution à un tableau partiellement rempli (avec une taille logique) ou un tableau contenant tout autre type de valeurs qui peut s'ordonner.

Description de l'algorithme

Soit value la valeur recherchée dans une zone délimitée par les indices leftIndex et rightIndex. Il faut commencer par déterminer l'élément médian, c'est-à-dire celui qui se trouve « au milieu » de la zone de recherche [63] ; son indice sera déterminé par la formule

$$\text{medianIndex} = (\text{leftIndex} + \text{rightIndex}) \text{ DIV } 2$$

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

Exemple de recherche fructueuse

Supposons que l'on recherche la valeur **23** dans un tableau de 20 entiers. La zone ombrée représente à chaque fois la partie de recherche, qui est au départ le tableau trié dans son entièreté. Au départ, leftIndex vaut 0 et rightIndex vaut 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Première étape : medianIndex = (0 + 19) DIV 2, c'est-à-dire 9.

Comme la valeur en position 9 est 15, la valeur cherchée doit se trouver à sa droite, et on prend comme nouvelle zone de recherche celle délimitée par leftIndex qui vaut 10 et rightIndex qui vaut 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Deuxième étape : medianIndex = (10 + 19) DIV 2, c'est-à-dire 14.

Comme on y trouve la valeur 25, on garde les éléments situés à la gauche de celui-ci ; la nouvelle zone de recherche est [10, 13].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Troisième étape : medianIndex = (10 + 13) DIV 2, c'est-à-dire 11 où se trouve l'élément 20. La zone de recherche devient leftIndex vaut 12 et rightIndex vaut 13.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

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

Quatrième étape : medianIndex = (12 + 13) DIV 2, c'est-à-dire 12 où se trouve la valeur cherchée ; la recherche est terminée.

Exemple de recherche infructueuse

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

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Arrivé à ce stade, la zone de recherche s'est réduite à un seul élément. Ce n'est pas celui qu'on recherche mais c'est à cet endroit qu'il se serait trouvé ; c'est donc là qu'on pourra éventuellement l'insérer. Le paramètre de sortie prend la valeur 5.

Algorithme

```java
public static int dichotomousSearch(int[] myArray, int value){
  int leftIndex = 0;
  int medianIndex;
  int rightIndex = myArray.length;
  int candidate;
  boolean isFound = false;

  while (!isFound && leftIndex < rightIndex){
    medianIndex = (leftIndex + rightIndex) / 2;
    candidate = myArray[medianIndex];
    if (candidate == value){
      isFound = true;
    } else if (candidate < value){
      leftIndex = medianIndex + 1;
    } else {
      rightIndex = medianIndex - 1;
    }
  }

  if (isFound){
    return medianIndex;
  } else {
    return -1,
  }
}
```

java

11.4 Introduction à la complexité

> L'algorithme de recherche dichotomique est beaucoup plus rapide que l'algorithme de recherche linéaire.

Mais qu'est-ce que cela veut dire exactement ?
En est-on sûr ?
Comment le mesure-t-on ?
Quels critères utilise-t-on ?

De façon générale, comment comparer la vitesse de deux algorithmes différents qui résolvent le même problème.

11.4.1 Une approche pratique : la simulation numérique

Une manière naïve de comparer la rapidité des algorithmes serait de les traduire dans un langage de programmation, de les exécuter et de comparer les temps d'exécution.

Cette technique pose toutefois quelques problèmes :

- ▷ il faut que les programmes soient exécutés dans des environnements strictement identiques ce qui n'est pas toujours le cas ou facile à vérifier ;
- ▷ certains environnements peuvent être favorables à un algorithme par rapport à l'autre ce qui ne serait pas le cas d'un autre environnement ;

  Par exemple, certaines architectures sont plus rapides dans les calculs entiers mais moins dans les calculs flottants.
- ▷ le test ne porte que sur un (voir quelques uns) jeu(x) de tests.

  Comment en tirer un enseignement général ?
  Que se passerait-il avec des données plus importantes ?
  Avec des données différentes ?

En fait, un chiffre précis ne nous intéresse pas. Ce qui est vraiment intéressant, c'est de savoir comment l'algorithme va se comporter avec de grandes données. C'est ce qu'apporte l'approche suivante.

11.4.2 Une approche théorique : la complexité

Le principe de cette approche est de déduire de l'algorithme le nombre d'opérations de base à effectuer en fonction de la **taille** du problème à résoudre et en déduire comment il va se comporter sur de « gros » problèmes ?

Dans le cas de la recherche dans un tableau, la taille du problème est la taille du tableau dans lequel l'élément est cherché. Nous pouvons considérer que l'opération de base est la comparaison avec un élément du tableau.

La question est alors la suivante : pour un tableau de taille n, à combien de comparaisons faut-il procéder pour trouver l'élément (ou se rendre compte de son absence) ?

Pour la recherche linéaire

Cela dépend évidemment de la position de la valeur à trouver. Dans le meilleur des cas c'est 1, dans le pire c'est n mais on peut dire, qu'en moyenne, cela entraine « $n/2$ » comparaisons (que ce soit pour la trouver ou se rendre compte de son absence).

Pour la recherche dichotomique

Ici, la zone de recherche est divisée par 2, à chaque étape. Imaginons une liste de 64 éléments : après 6 étapes, on arrive à un élément isolé. Pour une liste de taille n, on peut en déduire que le nombre de comparaisons est au maximum l'exposant qu'il faut donner à 2 pour obtenir n, soit « $\log_2(n)$ ».

Comparaisons

Voici un tableau comparatif du nombre de comparaisons en fonction de la taille n

n	10	100	1000	10.000	100.000	1 million
recherche linéaire	5	50	500	5.000	50.000	500.000
recherche dichotomique	4	7	10	14	17	20

Nous pouvons voir que c'est surtout pour des grands problèmes que la recherche dichotomique montre toute son efficacité. Et nous voyons aussi que l'important pour mesurer la complexité est l'ordre de grandeur du nombre de comparaisons.

Nous dirons que la recherche simple est un algorithme de complexité linéaire (c'est-à-dire que le nombre d'opérations est de l'ordre de n ou proportionnel à n ou encore que doubler la taille du problème fait doubler aussi le temps de calcul). Ceci se note en langage plus mathématique : $O(n)$ (prononcé « grand O de n »).

Pour la recherche dichotomique, la complexité est logarithmique. Elle se note $O(\log_2(n))$ ce qui veut dire que doubler la taille du problème augmente d'une seule itération la boucle. C'est un fameux gain.

Comparons les complexités les plus courantes.

n	10	100	1000	10^4	10^5	10^6	10^9
$O(1)$	1	1	1	1	1	1	1
$O(\log_2(n))$	4	7	10	14	17	20	30
$O(n)$	10	100	1000	10^4	10^5	10^6	10^9
$O(n^2)$	100	10^4	10^6	10^8	10^10	10^{12}	10^{18}
$O(n^3)$	1000	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{27}
$O(2^n)$	1024	10^{30}	10^{301}	10^{3010}	10^{30102}	10^{301029}	$10^{3010299995}$

Nous pouvons en conclure que si trouver un algorithme qui résout un algorithme est sans doute bien, trouver un algorithme qui a une complexité exponentielle ne sert à rien en pratique.

Par exemple un algorithme de recherche de complexité exponentielle, exécuté sur une machine pouvant effectuer un milliard de comparaisons par secondes, prendrait

plus de dix mille milliards d'années pour trouver une valeur dans un tableau de 100 éléments.

Chapitre 12

Algorithmes de tris

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

Contenu

12.1 Motivation

Dans le chapitre précédent, nous avons vu que la recherche d'information est beaucoup plus rapide dans un tableau trié grâce à l'algorithme de recherche dichotomique. Nous avons vu comment **garder** un ensemble trié en insérant toute nouvelle valeur au *bon* endroit.

Dans certaines situations, nous serons amené à devoir trier toute une collection de valeurs. Par exemple un tableau.

1. D'abord les situations impliquant le classement total d'un ensemble de données « brutes », c'est-à-dire complètement désordonnées.
   Prenons pour exemple les feuilles récoltées en vrac à l'issue d'un examen ; il y a peu de chances que celles-ci soient remises à l'examinateur de manière ordonnée ; celui-ci devra donc procéder au tri de l'ensemble des copies, par exemple par ordre alphabétique des noms des étudiants ou des étudiantes, ou par numéro de groupe, etc.
2. Enfin, les situations qui consistent à devoir retrier des données préalablement ordonnées sur un autre critère.
   Prenons l'exemple d'un paquet de copies d'examen déjà triées sur l'ordre alphabétique des noms des étudiants et des étudiantes, et qu'il faut retrier cette

fois-ci sur les numéros de groupe. Il est clair qu'une méthode efficace veillera à conserver l'ordre alphabétique déjà présent dans la première situation afin que les copies apparaissent dans cet ordre dans chacun des groupes.

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

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

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

- ▷ le nombre de tests effectués (comparaisons de valeurs) et ;
- ▷ le nombre de transferts de valeurs réalisés.

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

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

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

12.2 Tri par insertion

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

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

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

Prenons comme exemple un tableau `tab` de 20 entiers. Au départ, le sous-tableau trié est formé du premier élément, `tab[0]`, qui vaut 20 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	12	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

L'étape suivante consiste à insérer `tab[1]`, qui vaut 12, dans ce sous-tableau de taille 2 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
12	20	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

Ensuite, c'est au tour de `tab[2]`, qui vaut 18, d'être inséré :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
12	18	20	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

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

Algorithme. L'algorithme combine la recherche de la position d'insertion et le décalage concomitant de valeurs.

```java
/**
 * Tri par insertion.
 *
 * @param myArray tableau à trier
 */
public static void insertionSort(int[] myArray){
  int j, value;
  for (int i = 1; i < myArray.length; i = i + 1){
    value = myArray[i];
    j = i - 1;
    while (j >= 0 && value < myArray[j]){
      myArray[j + 1] = myArray[j];
      j = j - 1;
    }
    myArray[j + 1] = value;
  }
}
```

java

12.3 Tri par sélection des minima successifs

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

Description de l'algorithme. Prenons par exemple un tableau de 20 entiers.

La première étape consiste à rechercher la valeur minimale du tableau. Il s'agit de l'élément d'indice 9 et de valeur 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	52	61	47	82	64	95	66	84	17	32	24	46	48	75	55	19	61	21	30

Celui-ci devrait donc apparaitre en 1re position du tableau. Or cette position est occupée par la valeur 20. Comment procéder dès lors pour ne perdre aucune valeur et sans faire appel à un second tableau ? La solution est simple, il suffit d'échanger le contenu des deux éléments d'indices 0 et 9 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
17	52	61	47	82	64	95	66	84	20	32	24	46	48	75	55	19	61	21	30

Le tableau se subdivise à présent en deux sous-tableaux, un sous-tableau déjà trié (pour le moment réduit au seul élément $tab[0]$) et le sous-tableau des autres valeurs non encore triées (de l'indice 1 à 19). On recommence ce processus dans ce second sous-tableau : le minimum est à présent l'élément d'indice 16 et de valeur 19. Celui-ci viendra donc en 2e position, échangeant sa place avec la valeur 52 qui s'y trouvait :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
17	19	61	47	82	64	95	66	84	20	32	24	46	48	75	55	52	61	21	30

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

Et ainsi de suite, jusqu'à obtenir un sous-tableau trié comprenant tout le tableau. En pratique l'arrêt se fait une étape avant car lorsque le sous-tableau non trié n'est plus que de taille 1, il contient nécessairement le maximum de l'ensemble.

```java
/**
 * Trie le tableau reçu en paramètre via un tri
 * par sélection des minima successifs
 *
 * @param myArray le tableau à trier
 */
public static void selectionSort(int[] myArray){
  int i;
  int minIndex;
  for (int i=0; i<myArray.length-1; i++){
    minIndex = searchMinIndex(myArray, i, myArray.length-1);
    // swap values
    int value = myArray[i];
    myArray[i] = myArray[minIndex];
    myArray[minIndex] = value;
  }
}

/**
 * Retourne l'indice du minimum entre les indices
 * début et fin du tableau reçu.
 *
 * @param myArray le tableau d'entiers
 * @param l'indice de début
 * @param l'indice de fin
 */
public static int searchMinIndex(
    int[] myArray,
    int begin,
    int end){
  int minIndex;
  int i;
  minIndex = begin;
  for (int i = begin + 1; i<end; i++){
    if (myArray[i] < myArray[minIndex]) {
      minIndex = i;
    }
  }
  return minIndex;
}
```

java

12.4 Tri bulle

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

Description de l'algorithme Prenons pour exemple un tableau de taille 14. En partant de la fin du tableau, on le parcourt vers la gauche en comparant chaque couple de valeurs consécutives. Quand deux valeurs sont dans le désordre, on les

permute. Le premier parcours s'achève lorsqu'on arrive à l'élément d'indice 0 qui contient alors la « bulle », c'est-à-dire la plus petite valeur du tableau, soit 1 dans l'exemple :

0	1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	12	15	4	8	1	7	12	11	3	6	5	4
10	5	12	15	4	8	1	7	12	11	3	6	4	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	3	11	4	6	5
10	5	12	15	4	8	1	7	3	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	1	8	3	7	12	11	4	6	5
10	5	12	15	1	4	8	3	7	12	11	4	6	5
10	5	12	1	15	4	8	3	7	12	11	4	6	5
10	5	1	12	15	4	8	3	7	12	11	4	6	5
10	1	5	12	15	4	8	3	7	12	11	4	6	5
1	10	5	12	15	4	8	3	7	12	11	4	6	5

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

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

Et ainsi de suite. Au total, il y aura $n - 1$ étapes.

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

```java
public static void bubbleSort(int[] myArray){
  for (int bubbleIndex = 0;
      bubbleIndex < myArray.length - 1;
      bubbleIndex++){
    for (int i = myArray.length - 2; i <= bubbleIndex; i--){
      if (myArray[i] > myArray[i + 1]){
        // swap values
        int value = myArray[i];
        myArray[i] = myArray[i+1];
        myArray[i+1] = value;
      }
    }
  }
}
```

java

12.5 Cas particuliers

Tri partiel

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

Sélection des meilleurs

Une autre situation courante est la suivante : un ensemble de valeurs sont traitées (lues, calculées...) et il ne faut garder que les k plus petites (ou plus grandes).

L'algorithme est plus simple à écrire si on utilise une position supplémentaire en fin de tableau. Notons aussi qu'il faut tenir compte du cas où il y a moins de valeurs que le nombre de valeurs voulues ; c'est pourquoi on ajoute une variable indiquant le nombre exact de valeurs dans le tableau.

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

```java
public static int bests(int[] smallers){
  Scanner keyboard = new Scanner(System.in);
  int val;
  nbValues = 0;
  val = keyboard.nextInt();
  while (val >= 0){
    nbValues = insert(val, smallers, nbValues);
    val = keyboard.nextInt();
  }
}

public static int insert(int val, int[] smallers, nbValues){
  int i = nbValues - 1;
  while ( i >= 0 && val < smallers[i]){
    smallers[i+1] = smallers[i];
    i = i - 1;
  }
  smallers[i+1] = val;
  if (nbValues < smallers.length){
    nbValues++;
  }
  return nbValues;
}

```

java

12.6 Références

▷ `http://interstices.info/jcms/c_6973/les-algorithmes-de-tri`

Ce site [VE04] permet de visualiser les méthodes de tris en fonctionnement. Il présente tous les tris vus en première plus quelques autres comme le « tri rapide » (« quicksort ») que vous verrez en deuxième année.

Quatrième partie

Compléments

Chapitre 13

Les chaines et les `String`

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

Très tôt dans ce cours, nous avons introduit le type chaine mais nous n'en avons encore fait que des usages basiques, essentiellement des affichages. Il est temps d'aller plus loin et de les *manipuler*, c'est-à-dire d'examiner et de modifier, déplacer... le contenu de chaines.

Procédons par étapes.

Contenu

13.1 Interface de programmation applicative

Une interface de programmation applicative, *application programming interface* en anglais ou encore simplement **API** est un ensemble de méthodes (de classes et de constantes) fournies — dans ce cas par le langage Java — au développeur.

L'API Java complète peut-être découverte à la lecture de la documentation fournie avec le langage à cette adresse par exemple : `https://docs.oracle.com/en/java/javase/11/docs/api/index.html` [tea18]

Un des éléments mesurant la force d'un langage est la taille de son API à laquelle s'ajoute la qualité de sa documentation. Dans cette API se retrouvent les « actions ordinaires et habituelles » que demandent les développeurs et développeuses. Intéressons-nous à la partie concernant les chaines de caractères.

13.2 Chaine et caractère

Certains langages, comme Java, distinguent clairement le type chaine (String) et le type caractère (char). Ainsi, en Java, le littéral 'A' représente un caractère qui est différent du littéral "A" qui représente une chaine (composée d'un seul caractère). Le type String est un type référence tandis que le type char est un type primitif.

Cette distinction est essentiellement dictée par des détails d'implémentation ; un caractère pouvant se représenter de façon plus économe qu'une chaine. En langage Java toujours essayer de comparer — par un "A" == 'A' — un chaine et un caractère génère une erreur de compilation.

D'autres langages, comme Python, n'ont pas de type caractère spécifique. D'autres encore, comme C, ont un type caractère mais pas de type chaine à proprement parler ; ils sont vus comme des tableaux de caractères.

13.3 Longueur

La **longueur** d'une chaine est le nombre de caractères qu'elle contient. Pour la connaitre on utilisera la notation pointée uneChaine.length().

Par exemple :

- "Bonjour".length() vaut 7.
- "Une chaine".length() vaut 10 (l'espace compte pour 1).
- "A".length() vaut 1.
- "".length() est égal à 0.

```java
"Bonjour".length() // vaut 7
```

Remarque Notez la présence des parenthèses.

13.4 Le contenu d'une chaine

Pour pouvoir manipuler une chaine, il faut pouvoir accéder aux caractères qui la composent. Le langage Java, l'API, propose une méthode — char charAt(int) — permettant d'accéder à une lettre d'un mot et la retournant.

```java
char c = "Hello".charAt(2);   // le caractère 'l'
```

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

```java
public static boolean contains(String word, char letter){
  int i = 1;
  while (i <= word.length() && word.charAt(i) != letter){
    i = i + 1;
  }
  return i <= word.length();
}
```

Remarque L'exemple est rhétorique car l'API Java propose une méthode contains répondant à cette question.

13.5 La concaténation

Il est fréquent de devoir rassembler plusieurs chaines pour former une seule chaine plus grande, il s'agit de l'opération de **concaténation**. En Java, c'est simplement l'opérateur + qui est utilisé. Dès lors que les opérandes sont des chaines, l'opérateur concatène au lieu d'additionner.

```java
String text;
text = "al" + "go" + "rithmique";
```

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

```java
public static String mirror(String word){
  String mirror = "";
  for (int i = 0; i < word.length(); i = i + 1){
    mirror = word.charAt(i) + mirror;
  }
  return mirror;
}
```

Remarque L'instruction de la ligne 4 concatène un caractère (word.charAt(i)) avec une chaine (mirror) ce qui n'est, à priori, pas possible. Dans ce cas, Java convertit le caractère en une chaine avant d'effectuer la concaténation de manière tout à fait transparente.

Dans cette instruction à la ligne 4 toujours, il y a création d'une nouvelle chaine de caractère à chaque itération de la boucle. Si l'on veut gagner un peu de place en mémoire, il est possible de créer une chaine de la bonne taille dès le début et d'y placer les caractères dans l'ordre qui nous convient au fur et à mesure. Ce serait mieux n'est-ce pas ?

En langage Java, ce n'est pas immédiat de remplacer un caractère par un autre dans une chaine. Ceci parce que character et String sont deux types différents. De plus, l'un est de type primitif comme nous l'avons déjà dit et l'autre de type référence.

Une manière élégante de résoudre le problème, est l'utilisation de StringBuilder qui, comme son nom l'indique, permet de manipuler plus finement une chaine de caractères. Nous pouvons écrire :

```java
public static String mirror(String word){
  StringBuilder sb = new StringBuilder(word);
  for (int i = 0; i < word.length(); i = i + 1){
    sb.setCharAt(word.length() - i - 1, word.charAt(i));
  }
  return sb.toString();
}
```

java

- ▷ instruction 2 : création d'un espace de la mème longueur que la chaine word ;
- ▷ instruction 4 : placement du caractère à la position d'indice i dans word à la position d'indice « taille du mot - i -1 » dans le StringBuilder.

13.6 Manipuler les caractères

Voici quelques méthodes de l'API Java pratiques pour la manipulation de chaines.

Nous pourrions écrire ces algorithmes et ces méthodes de l'API mais ce serait long et fastidieux. Vous pouvez y réfléchir à titre d'exercice.

Certaines méthodes sont des méthodes de la classe String, d'autres de la classe Character. Nous vous encourageons à consulter la Javadoc [tea18].

isLetter

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

boolean Character.isLetter(char c)

```java
Character. isLetter('#');  // false
char c = 'A';
Character.isLetter(c);   // true

```

isLowerCase

Permet de savoir si le caractère **est une lettre minuscule**.

boolean Character.isLowerCase(char c)

```java
 Character. isLowerCase('A');  // false
char c = 'a';
 Character.isLowerCase(c);   // true
```

isUpperCase

Permet de savoir si le caractère **est une lettre majuscule**.

boolean Character.isUpperCase(char c)

```java
 Character. isUpperCase('a');  // false
char c = 'A';
 Character.isUpperCase(c);   // true
```

isDigit

Permet de savoir si un caractère **est un chiffre**. Elle retourne vrai pour les dix caractères '0', '1', '2', '3', '4', '5', '6', '7', '8' et '9' (et aussi pour les chiffres dans d'autres langues, cfr. javadoc [64]).

```java
 Character. isDigit('a');  // false
char c = '1';
 Character.isDigit(c);   // true
```

toUpperCase

Convertit la chaine en passant tous les caractères en majuscules.

String.toUpperCase()

```java
 String s = "hello";
s.toUpperCase();
 System.out.println(s);    // affiche HELLO
```

toLowerCase

Convertit la chaine en passant tous les caractères en minuscules.

String.toLowerCase()

64. https://docs.oracle.com/javase/9/docs/api/java/lang/Character.html#isDigit-char-

```java
String s = "WORLD";
s.toLowerCase();
System.out.println(s);    // affiche world
```

13.7 L'alphabet

Il peut aussi être pratique de connaitre la position d'une lettre dans l'alphabet pour résoudre des exercices. Cette méthode retournerait toujours un entier entre 1 et 26 (ou entre 0 et 25). Par exemple, letterIndex('E') donnerait 5. Cette fonction traiterait de la même manière les lettres majuscules et les lettres minuscules. Ainsi, letterIndex('e') retournerait également 5.

Il n'existe pas de telle méthode dans l'API Java.

Pour les caractères non accentués, il est assez simple de connaitre la position d'une lettre dans l'alphabet dès lors que l'on sait qu'à chaque lettre est associée un code (Unicode) et que ces codes se suivent. La table unicode contenant les lettres de l'alphabet [65] nous montre que le code associé à la lettre 'A' est 0x0041 ou 65 en base 10, 'B' 0x0042 et ainsi de suite [Bet09].

Cette table nous montre également que les lettres minuscules arrivent ensuite ; 'a' a comme code 0x0061 ou 97 en base 10, 'b' 0x0062...

Retourner la position d'une lettre devient aussi simple qu'une soustraction. En supposant qu'il ne faille pas vérifier que le caractère reçu est bien une lettre, nous pourrions écrire :

```java
public static int letterIndex(char c){
  return Character.toLowerCase(c) - 0x61 + 1;
}
```

Pour traiter les caractères accentués, c'est plus complexe et ça sort un peu du cadre d'un premier cours de développement. Bien sûr, nous pouvons nous en sortir avec une série de if de la forme if(c == 'é' || c== 'è'...) mais ce serait fastidieux.

L'API Java peut faire ce travail pour nous si c'est demandé gentiment. Il existe une classe de l'API pour « normaliser » les caractères d'une chaine. Cette normalisation consiste à vérifier que les caractères accentués le sont sous la forme « lettre-accent ». C'est-à-dire qu'un à est bien codé a‘. Dans ce cas, replaceAll peut supprimer les accents et l'on pourrait écrire une méthode letterIndex qui fonctionne avec des lettres accentuées comme suit [66] :

65. https://www.unicode.org/charts/PDF/U0000.pdf
66. Un import java.text.Normalizer est nécessaire.

```java
public static int letterIndex(char c){
  c = Normalizer
    .normalize(""+c, Normalizer.Form.NFD)
    .replaceAll("[\u0300-\u036F]", "")
    .charAt(0);
  return Character.toLowerCase(c) - 0x61 + 1;
}

```

java

... mais ça sort un peu du cadre de ce premier cours de développement.

Il peut être utile d'avoir un outil qui fait l'opération inverse, à savoir associer la lettre de l'alphabet correspondant à une position donnée. C'est immédiat avec une simple soustraction :

indexToUpperChar
: Retourne la forme majuscule de la n^{e} lettre de l'alphabet (où n sera obligatoirement compris entre 1 et 26). Par exemple, lettreMaj(13) retourne "M".
C'est immédiat par return 0x41 + n.

indexToLowerChar
: Idem pour les minuscules.
C'est immédiat par return 0x61 + n.

13.8 Chaine et nombre

Si une chaine contient un nombre, il doit être possible de convertir la chaine en nombre et inversement.

Nombre vers chaine
: La conversion d'un nombre en chaine est immédiate lors l'une affectation dans une variable de type String

```java
        String s = 42;

```

java

Lors d'une concaténation, les conversions d'un nombre vers une chaine sont immédiates à ceci près qu'il faut parfois être attentif à l'ordre des opérandes.

```java
        String s1 = 1 + "2";    // "12"
        String s2 = 1 + 2;      // "3"
        String s3 = 1 + "2" + "3";  // "123"
        String s4 = 1 + 2 + "3";  // "33"

```

java

Chaine vers nombre
: Transforme une chaine contenant des caractères numériques en nombre. Pour chaque type, il existe une méthode qui fait cette transformation.

```java
String s = "3.14";
double d = Double.parseDouble(s);
s = "5";
int i = Integer.parseInt(s);
```

13.9 Extraction de sous-chaines

substring

Permet d'extraire une sous-chaine d'une chaine à partir d'une position donnée jusqu'à une autre.

```java
String s = "Karine, Jenny et Vicky paraissent belles";
System.out.println(s.substring(8,40));
  // Jenny et Vicky paraissent belles

```

Il faut bien sûr être vigilant pour ne pas sortir des bornes de la chaine sinon, une erreur est générée.

Cette fonction est très utile pour sélectionner des portions d'une chaine contenant des informations codées sous un certain format. Prenons par exemple une date stockée dans une chaine *stringDate* de format "JJ/MM/AAAA" (de longueur 10). Pour extraire :

- le jour, substring(stringDate, 0, 2) ;
- le mois, substring(stringDate, 3, 5) ;
- le jour, substring(stringDate, 6, 10) ;

13.10 Recherche de sous-chaine

indexOf

Permet de savoir si une sous-chaine donnée est présente dans une chaine donnée. Elle permet d'éviter d'écrire le code correspondant à une recherche. La valeur de l'entier renvoyé est la position où commence la sous-chaine recherchée. Si la sous-chaine ne s'y trouve pas, la fonction retourne -1.

```java
String s = "Des 3 nombrils, "
  + "c'est Karine la plus intelligente.";
s.indexOf("Karine"); // 22
```

Remarque Pour simplement savoir si la sous-chaine est présente sans demander sa position, la fonction contains retourne un booléen et prend les mêmes paramètres.

Cinquième partie

Annexes

Annexe A

Exercices

Contenu

A.1 Exercices : spécifier le problème

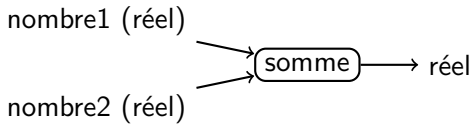
Pour ces premiers exercices, nous vous demandons d'imiter la démarche décrite dans le chapitre 1, à savoir :

- ▷ déterminer quelles sont les données ; leur donner un nom et un type ;
- ▷ déterminer quel est le type du résultat ;
- ▷ déterminer un nom pertinent pour l'algorithme ;
- ▷ fournir un résumé graphique ;
- ▷ donner des exemples.

1 Somme de 2 nombres

Calculer la somme de deux nombres donnés.

Solution. [67] 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 HTVA (hors tva), il faut lui ajouter 21% pour obtenir le prix TTC (toutes taxes comprises). Écrire un algorithme qui permet de passer du prix HTVA au prix TTC.

67. Nous allons de temps en temps fournir des solutions. En algorithmique, il y a souvent **plusieurs** solutions possibles. Ce n'est donc pas parce que vous avez trouvé une autre façon de faire qu'elle est fausse. Mais il peut y avoir des solutions **meilleures** que d'autres ; n'hésitez jamais à montrer la vôtre à votre professeur pour avoir son avis.

7 Les intérêts

Calculer les intérêts reçus après 1 an pour un montant placé en banque à du 2% d'intérêt.

8 Placement

Étant donné le montant d'un capital placé (en €) et le taux d'intérêt annuel (en %), calculer la nouvelle valeur de ce capital après un an.

9 Prix total

Étant donné le prix unitaire d'un produit (hors TVA), le taux de TVA (en %) et la quantité de produit vendue à un client, calculer le prix total à payer par ce client.

10 Durée de trajet

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

11 Allure et vitesse

L'allure d'un coureur est le temps qu'il met pour parcourir 1 km (par exemple, $4'37''$). On voudrait calculer sa vitesse (en km/h) à partir de son allure. Par exemple, la vitesse d'un coureur ayant une allure de $4'37''$ est de 13 km/h.

12 Somme des chiffres

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

13 Conversion HMS en secondes

Étant donné un moment dans la journée donné par trois nombres, à savoir, heure, minute et seconde, calculer le nombre de secondes écoulées depuis minuit.

14 Conversion secondes en heures

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "heure".

Ex : 10000 secondes donnera 2 heures.

15 Conversion secondes en minutes

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "minute".

Ex : 10000 secondes donnera 46 minutes.

16 Conversion secondes en secondes

Étant donné un temps écoulé depuis minuit. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "seconde".

Ex : 10000 secondes donnera 40 secondes.

17 Cote moyenne

Étant donné les résultats (cote entière sur 20) de trois examens passés par un étudiant (exprimés par six nombres, à savoir, la cote et la pondération de chaque examen), calculer la moyenne globale exprimée en pourcentage.

A.2 Exercices : premiers algorithmes et programmes

Dans ces exercices, nous vous proposons d'écrire un algorithme et de le traduire en un programme Java. Certains exercices ont déjà été travaillés dans la section précédente (voir annexe A.1, p.166).

18 Moyenne de 2 nombres

Calculer la moyenne de deux nombres donnés.

19 Surface d'un triangle

Calculer la surface d'un triangle connaissant sa base et sa hauteur.

20 Périmètre d'un cercle

Calculer le périmètre d'un cercle dont on donne le rayon.

21 Surface d'un cercle

Calculer la surface d'un cercle dont on donne le rayon.

22 TVA

Si on donne un prix hors TVA, il faut lui ajouter 21% pour obtenir le prix TTC. Écrire un algorithme qui permet de passer du prix HTVA au prix TTC.

23 Les intérêts

Calculer les intérêts reçus après 1 an pour un montant placé en banque à du 2% d'intérêt.

24 Placement

Étant donné le montant d'un capital placé (en €) et le taux d'intérêt annuel (en %), calculer la nouvelle valeur de ce capital après un an.

25 Conversion HMS en secondes

Étant donné un moment dans la journée donné par trois nombres, à savoir, heure, minute et seconde, calculer le nombre de secondes écoulées depuis minuit.

26 Prix total

Étant donné le prix unitaire d'un produit (hors TVA), le taux de TVA (en %) et la quantité de produit vendue à un client, calculer le prix total à payer par ce client.

A.3 Exercices : tracer un algorithme

Nous vous proposons de tracer vos algorithmes comme présenté dans le chapitre 4 dans la section 4.3.3, page 39.

27 Tracer des bouts de code

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

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

pseudocode

#	a	b	c
1			
2			
3			
4			
5			
6			
7			

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

pseudocode

#	a	b	c
1			
2			
3			
4			
5			
6			

28 Calcul de vitesse

Soit le problème suivant : « Calculer la vitesse (en km/h) d'un véhicule dont on donne la durée du parcours (en secondes) et la distance parcourue (en mètres). ».

Voici *une* solution :

```java
public static double vitesseKMH(double distanceM, double duréeS){
  double distanceKMH, duréeH;
  distanceKM = 1000 * distanceM;
  duréeH = 3600 * duréeS;
  return distanceKM / duréeH;
}
```

java

L'algorithme, s'il est correct, devrait donner une vitesse de 1 km/h pour une distance de 1000 mètres et une durée de 3600 secondes. Testez cet algorithme avec cet exemple.

#	
1	
2	
3	
4	
5	

Si vous trouvez qu'il n'est pas correct, voyez ce qu'il faudrait changer pour le corriger.

29 Allure et vitesse

L'allure d'un coureur est le temps qu'il met pour parcourir 1 km (par exemple, $4'37''$ [68]). On voudrait calculer sa vitesse (en km/h) à partir de son allure.

Par exemple, la vitesse d'un coureur ayant une allure de $4'37''$ est de $12,996389892$ km/h.

30 Cote moyenne

Étant donné les résultats (cote entière sur 20) de trois examens passés par un étudiant (exprimés par six nombres, à savoir, la cote et la pondération de chaque examen), calculer la moyenne globale exprimée en pourcentage.

68. 4 minutes et 37 secondes

A.4 Exercices : division entière, quotient et reste

31 Calculs

Voici quelques petits calculs à compléter faisant intervenir la division entière et le reste. Par exemple : « 14 DIV 3 = 4 reste 2 » signifie que 14 DIV 3 = 4 et 14 MOD 3 = 2.

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

32 Les prix ronds

Voici un algorithme qui reçoit une somme d'argent exprimée en centimes et qui calcule le nombre (entier) de centimes qu'il faudrait ajouter à la somme pour tomber sur un prix rond en euros. Testez-le avec des valeurs numériques. Est-il correct ?

```java
public static int versPrixRond(int prixCentimes){
  retun 100 - (prixCentimes \% 100);
}
```

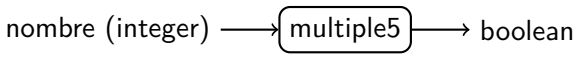
test n°	prixCentimes	réponse correcte	valeur retournée	Correct ?
1	130	70		
2	40	60		
3	99	1		
4	100	0		

Remarque Les exercices qui suivent n'ont pas tous été déjà analysés et ils demandent des calculs faisant intervenir des divisions entières, des restes et/ou des expressions booléennes. Comme d'habitude, écrivez la spécification si ça n'a pas encore été fait, donnez des exemples, rédigez un algorithme et vérifiez-le.

33 Nombre multiple de 5

Calculer si un nombre entier donné est un multiple de 5.

Solution. Dans ce problème, il y a une donnée, le nombre à tester. La réponse est un booléen qui est à vrai si le nombre donné est un multiple de 5.



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 :

```
return nombre 5 = 0
```

langage naturel

Vérifions sur nos exemples :

test n°	nombre	réponse correcte	valeur retournée	Correct ?
1	4	faux	faux	✓
2	15	vrai	vrai	✓
3	0	vrai	vrai	✓
4	-10	vrai	vrai	✓

34 Nombre entier positif se terminant par un 0

Calculer si un nombre donné se termine par un 0.

35 Les centaines

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

36 Somme des chiffres

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

37 Conversion secondes en heures

Étant donné un temps écoulé depuis minuit en secondes. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "heure".

Ex : 10000 secondes donnera 2 heures.

Aide : L'heure n'est qu'un nombre exprimé en base 60 !

38 Conversion secondes en minutes

Étant donné un temps écoulé depuis minuit en secondes. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "minute".

Ex : 10000 secondes donnera 46 minutes.

39 Conversion secondes en secondes

Étant donné un temps écoulé depuis minuit en secondes. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "seconde".

Ex : 10000 secondes donnera 40 secondes.

40 Un double aux dés

Écrire un algorithme qui simule le lancer de deux dés et indique s'il y a eu un double (les deux dés montrant une face identique).

41 Année bissextile

Écrire un algorithme qui vérifie si une année est bissextile.

Pour rappel, les années bissextiles sont les années multiples de 4. Font exception, les multiples de 100 (sauf les multiples de 400 qui sont bien bissextiles). Ainsi 2012 et 2400 sont bissextiles mais pas 2010 ni 2100.

42 Conversion en heures-minutes-secondes

Écrire un algorithme qui permet à l'utilisateur de donner le nombre de secondes écoulées depuis minuit et qui affiche le moment de la journée correspondant en heures-minutes-secondes. Par exemple, si on est 3726 secondes après minuit alors il est 1h2'6".

A.5 Exercices : structures alternatives

Remarque Dans certains des exercices qui suivent vous aurez besoin d'obtenir une valeur « au hasard ». Vous pouvez supposer qu'il existe une instruction random(n) qui donne une valeur au hasard comprise entre 0 et n (strictement).

Cette notion est abordée dans la section 4.4.5, p.46.

43 Compréhension

Tracez cet algorithme avec les valeurs fournies et donnez la valeur de retour.

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

java

▷ exerciceA(2, 5) = _____ ▷ exerciceA(4, 1) = _____

44 Compréhension

Tracez ces algorithmes avec les valeurs fournies et donnez la valeur de retour.

```java
public static int exerciceB(int a, int b){
  int c;
  if (a > b){
    c = a/b;
  } else {
    c = a%b;
  }
  return c;
}
```

java

▷ exerciceB(2, 3) = _____ ▷ exerciceB(4, 1) = _____

```java
public static int exerciceC(int x1, int x2){
  boolean isBigger;
  isBigger = x1 > x2;
  if (isBigger){
    isBigger = isBigger && x1 ==  4;
  } else {
    isBigger = isBigger || x2 ==  3;
  }
  if (isBigger){
    x1 = x1 * 1000;
  }
  return x1 + x2;
}

```

java

▷ exerciceC(2, 3) = _____ ▷ exerciceC(4, 1) = _____

45 Simplification d'algorithmes

Voici deux extraits d'algorithmes que vous pouvez supposer 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.

```java
if (condition){
  b = true;
} else {
  b = false;
}
```

java

```java
if (n1>n2){
  b = false;
} else {
  if (n1<=n2){
    b = true;
  }
}
```

java

46 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 212.

47 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 ?

48 Fonction de Syracuse

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

49 Tarif réduit ou pas

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

50 Maximum de 3 nombres

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

51 Le signe

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

52 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.

53 Dés identiques

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

54 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.

55 Numéro du jour

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

56 Tirer une carte

Écrire un algorithme qui affiche l'intitulé d'une carte tirée au hasard dans un paquet de 52 cartes. Par exemple, "As de cœur", "3 de pique", "Valet de carreau" ou encore "Roi de trèfle".

Remarque. Il est plus facile de déterminer séparément chacune des deux caractéristiques de la carte : couleur et valeur.

57 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.

58 Moyenne pondérée

Un examen se déroule en plusieurs parties ;

- ▷ la première partie a une pondération de 5% ;
- ▷ la seconde, 25% et ;
- ▷ la dernière 70%.

Écrire un algorithme qui reçoit les 3 cotes (/20) et qui indique si l'étudiant a réussi l'examen ou non. L'algorithme affiche également la cote de l'examen.

59 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 ?

60 Le prix des photocopies

Un magasin de photocopies facture 0,10 € les dix premières photocopies, 0,09 € les vingt suivantes et 0,08 € au-delà. Écrivez un algorithme qui reçoit le nombre de photocopies effectuées et qui affiche la facture correspondante.

61 Le stationnement alternatif

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

A.6 Exercices : modules et méthodes

62 Tracer des algorithmes

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

```java
public static void ex1(){
  int x, y;
  x = add(3, 4);
  System.out.println(x);
  x = 3;
  y = 5;
  y add(x, Y);
  System.out.println(y);
}

public static int add(int a, int b){
  int sum;
  sum = a + b;
  return sum;
}

```

java

```java
public static void ex2(){
  int a, b;
  a = add(3, 4);    // cfr. ci-dessus
  System.out.println(a);
  a = 3;
  b = 5;
  b = sub(b, a);
  System.out.println(b);
}

public static int sub(int a, int b){
  return a - b;
}

```

java

63 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 ?

```java
public static int pgcd(int a, int b){
  // ...
}

```

java

- ☐ a = pgcd(24, 32)
- ☐ a = pgcd(a, 24)

- ☐ b = 3 * pgcd(a + b, 2 * c) + 120
- ☐ pgcd(20, 30)
- ☐ a = pgcd(a, b, c)
- ☐ a = pgcd(a, b) + pgcd(a, c)
- ☐ a = pgcd(a, pgcd(a, b))
- ☐ System.out.println(pgcd(a, b))
- ☐ pgcd(a, b) = c

64 Maximum de 4 nombres

Écrivez un algorithme qui calcule le maximum de 4 nombres.

65 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.

66 Tirer une carte

L'exercice suivant a déjà été résolu. Refaites une solution modulaire.

Écrire un algorithme qui affiche l'intitulé d'une carte tirée au hasard dans un paquet de 52 cartes. Par exemple, "As de cœur", "3 de pique", "Valet de carreau" ou encore "Roi de trèfle".

67 Nombre de jours dans un mois

Écrire un algorithme qui donne le nombre de jours dans un mois. Il reçoit en paramètre le numéro du mois (1 pour janvier...) ainsi que l'année. Pour le mois de février, il faudra répondre 28 ou 29 selon que l'année fournie est bissextile ou pas. Vous devez réutiliser au maximum ce que vous avez déjà fait lors d'exercices précédents (cf. exercice 41 page 173).

68 Valider une date

Écrire un algorithme qui valide une date donnée par trois entiers : l'année, le mois et le jour. Vous devez réutiliser au maximum ce que vous avez déjà fait lors d'exercices précédents.

69 Généraliser un algorithme

Dans l'exercice 33 page 171, 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.

Écrivez un seul algorithme, plus général, qui résoud tous ces problèmes en une seule fois.

A.7 Exercices : structures itératives

A.7.1 while

70 Compréhension d'algorithmes

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

```java
public static void while1(){
  int x;
  x = 0;
  while (x < 12){
    x = x + 2;
  }
  System.out.println(x);
}
```

java

```java
public static void while2(){
  boolean shouldContinue = true;
  int x = 5;
  while (shouldContinue) {
    x = x + 7;
    shouldContinue = x % 11 != 0
  }
  System.out.println(x);
}
```

java

```java
public static void while3(){
  boolean isDone = false;
  int count = 0,
    x = 10;
  while (!isDone && count < 3){
    if (x % 2 == 0){
      x = x + 1;
      isDone = x < 20;
    } else {
      x = x + 3
      count++
    }
  }
  System.out.println(x);
}
```

java

71 Afficher des nombres

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

a) les nombres de 1 à n ;

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

c) les nombres impairs de 1 à n ;

d) les nombres de -n à n ;

e) les multiples de 5 de 1 à n ;

f) les multiples de n de 1 à 100.

A.7.2 for

72 Compréhension d'algorithmes

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

```java
public static void for1(){
  int x;
  boolean b;
  x = 3;
  b = true;
  for (int i = 0; i < 5; i++){
    x = x + i + 1;
    b = b && x % 2 == 0;
  }
  if (b){
    System.out.println(x);
  } else {
    System.out.println(2 * x);
  }
}
```

java

```java
public static void for2(){
  int end;
  for (int i = 0; i < 3; i++){
    end = 6 * (i + 1) - 11;
    for (int j = 0; j < end; j = j + 3){
      System.out.println(10 * i + j);
    }
  }
}
```

java

73 Afficher des nombres

Reprenons un exercice déjà donné avec le **while**. En utilisant un **for**, écrire un algorithme qui reçoit un entier n positif et affiche

a) les nombres de 1 à n ;

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

c) les nombres de -n à n ;

d) les multiples de 5 de 1 à n ;

e) les multiples de n de 1 à 100.

A.7.3 do while

74 Compréhension d'algorithmes

Quels sont les affichages réalisés lors de l'exécution de l'algorithme suivant ?

```java
public static void dowhile1(){
  boolean isOdd, isBig;
  int p, x;
  x = 1;
  p = 1;
  do {
    p = 2 * p;
    x = x + p;
    isOdd = x % 2 == 0;
    isBig = x > 15;
  } while (!isOdd && !isBig);
```

java

75 Afficher des nombres

Reprenons un exercice déjà fait avec le **while** et le **for** en utilisant cette fois un **do while**. Écrire un algorithme qui reçoit un entier n positif et affiche

a) les nombres de 1 à n ;

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

c) les nombres de -n à n ;

d) les multiples de 5 de 1 à n ;

e) les multiples de n de 1 à 100.

A.7.4 Exercices récapitulatifs sur les structures répétitives

76 Afficher les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières à l'utilisateur et qui affiche celles qui sont impaires. L'algorithme commence par demander à l'utilisateur le nombre de valeurs qu'il désire donner.

77 Compter les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières à l'utilisateur et qui lui affiche le nombre de valeurs impaires qu'il a donné. Après chaque valeur entrée, l'algorithme demande à l'utilisateur s'il y en a encore d'autres.

78 Choix de la valeur sentinelle

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

79 Afficher les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières non nulles à l'utilisateur et qui affiche celles qui sont impaires. La fin des données sera signalée par la valeur sentinelle 0.

80 Compter le nombre de réussites

Écrire un algorithme qui demande une série de cotes (entières, sur 20) à l'utilisateur et qui affiche le pourcentage de réussites. La fin des données sera signalée par une valeur sentinelle que vous pouvez choisir.

81 Suites

Écrire les algorithmes qui affichent les n premiers termes des suites suivantes. À vous de voir quel est, parmi les 2 modèles décrits dans les exemples précédents, le modèle de solution le plus adpaté.

a) -1, -2, -3, -4, -5, -6...

b) 1, 3, 6, 10, 15, 21, 28...

c) 1, 0, 1, 0, 1, 0, 1, 0...

d) 1, 2, 0, 1, 2, 0, 1, 2...

e) 1, 2, 3, 1, 2, 3, 1, 2...

f) 1, 2, 3, 2, 1, 2, 3, 2...

Remarque Pour les exercices qui suivent, nous vous donnons peu d'indications sur la solution à mettre en œuvre. À vous de jouer...

82 Lire un nombre

Écrire un algorithme qui demande à l'utilisateur un nombre entre 1 et n et le retourne. Si la valeur donnée n'est pas dans l'intervalle souhaité, l'utilisateur est invité à rentrer une nouvelle valeur jusqu'à ce qu'elle soit correcte.

83 Lancé de dés

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

```java
/**
 * Simule le lancé d'un dé et compte
 * le nombre de fois ou value apparait.
 *
 * @param n le nombre de lancés
 * @param value la valeur comptée
 */
public static int rollDice(int n, int value){
  // ...
}
```

84 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, $0\,! = 1$.

85 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.

86 Table de multiplication

Écrire un algorithme qui affiche la table de multiplication des nombres de 1 à 10 (cf. l'exemple ci-contre).

```
1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
2 x 1 = 2
...
10 x 9 = 90
10 x 10 = 100
```

87 Double 6

Écrire un algorithme qui lance de façon répétée deux dés. Il s'arrête lorsqu'il obtient un double 6 et retourne le nombre de lancés effectués.

88 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.

89 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.

90 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 $1 + 3 + 3 + 0 + 4 + 5 = 16$.

91 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 $28 = 1 + 2 + 4 + 7 + 14$.

92 Décomposition en facteurs premiers

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

93 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.

94 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.

95 Jeu de la fourchette

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

A.8 Exercices : tableaux

96 Déclarer et initialiser

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

97 Initialiser un tableau à son indice

Écrire un algorithme qui déclare un tableau de 100 entiers et initialise chaque élément à la valeur de son indice. Ainsi, la case numéro i contiendra la valeur i.

98 Initialiser un tableau aux valeurs de 1 à 100

Écrire un algorithme qui déclare un tableau de 100 entiers et y met les nombres de 1 à 100.

99 Compréhension

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

100 Trouver les entêtes

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

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

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

c) Écrire un algorithme qui détermine si un tableau d'entiers contient au moins un nombre négatif.

d) Écrire un algorithme qui détermine si un tableau de chaines contient une chaine donnée en paramètre.

e) Écrire un algorithme qui détermine si un tableau de chaines contient au moins deux occurrences de la même chaine, quelle qu'elle soit.

f) Écrire un algorithme qui retourne un tableau donnant les n premiers nombres premiers, où n est un paramètre de l'algorithme.

g) Écrire un algorithme qui reçoit un tableau d'entiers et retourne un tableau de booléens de la même taille où la case i indique si oui ou non le nombre reçu dans la case i est strictement positif.

101 Inverser le signe des éléments

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

102 Somme

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

103 Nombre d'éléments d'un tableau

Écrire un algorithme qui reçoit en paramètre le tableau `doubles` de n réels et qui retourne le nombre d'éléments du tableau.

104 Compter les éléments négatifs

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

105 Y a-t-il une copie valant 20/20 ?

1. Écrire un algorithme qui reçoit en paramètre le tableau cotes de n entiers représentant les cotes des étudiants et qui retourne un booléen indiquant s'il contient **au moins** une fois la valeur 20.
2. Écrire un algorithme qui reçoit en paramètre le tableau cotes de n entiers représentant les cotes des étudiants et qui retourne un booléen indiquant s'il contient **exactement** une fois la valeur 20.

106 Lancers de deux dés

Écrire un algorithme qui lance n fois deux dés et compte le nombre de fois que chaque somme apparait. Cet algorithme retourne un tableau d'entiers ; un élément par somme.

107 Nombre de jours dans un mois

Écrire un algorithme qui reçoit un numéro de mois (de 1 à 12) ainsi qu'une année et donne le nombre de jours dans ce mois (en tenant compte des années bissextiles). N'hésitez pas à réutiliser des algorithmes déjà écrits.

108 Inscription / désinscription

1. Comment modifier l'algorithme d'inscription (cfr. 11.1 p 130) pour s'assurer qu'un étudiant ne s'inscrive pas deux fois ?
2. Comment modifier l'algorithme d'inscription pour refuser une inscription si le nombre maximal de participants est atteint en supposant que ce maximum est égal à la taille physique du tableau ?
3. Que se passerait-il avec l'algorithme de désinscription tel qu'il est si on demande à désinscrire un étudiant non inscrit ? Que suggérez-vous comme changement ?

109 Calcul de complexités

Quelle est la complexité d'un algorithme qui :

a) recherche le maximum d'un tableau de n éléments ?

b) remplace par 0 toutes les occurrences du maximum d'un tableau de n éléments ?

c) vérifie si un tableau contient deux éléments égaux ?

d) vérifie si les éléments d'un tableau forment un palindrome ?

e) cherche un élément dans un tableau en essayant des cases au hasard jusqu'à le trouver ?

110 Réflexion

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

Remarque Voici quelques exercices qui reprennent les différentes notions vues sur les tableaux. Pour chacun d'entre-eux :

- ▷ demandez-vous quel est le problème le plus proche déjà rencontré et voyez comment adapter la solution ;
- ▷ indiquez la complexité de votre solution ;
- ▷ décomposez votre solution en plusieurs algorithmes si ça peut améliorer la lisibilité ;
- ▷ testez votre solution dans le cas général et dans les éventuels cas particuliers.

111 Renverser un tableau

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

112 Tableau symétrique ?

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

113 Maximum

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

114 Minimum

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

115 Indice du maximum/minimum

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

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

116 Tableau ordonné ?

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

117 Remplir un tableau

On souhaite remplir un tableau de 20 éléments avec les entiers de 1 à 5, chaque nombre étant répété quatre fois. On voudrait deux variantes :

a) D'abord une version où les nombres identiques sont groupés : d'abord tous les 1 puis tous les 2, ...

1	1	1	1	2	2	2	2	3	3	3	3	4	4
4	4	5	5	5	5								

b) Ensuite une version où on trouve dans le tableau les valeurs 1, 2, 3, 4, 5 qui se suivent, quatre fois.

1	2	3	4	5	1	2	3	4	5	1	2	3	4
5	1	2	3	4	5								

Remarque : Il existe de nombreuses façons de résoudre ce problème. On peut par exemple utiliser deux boucles imbriquées. On peut aussi adapter un algorithme de génération de suites.

118 Positions du minimum

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

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

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

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

119 Occurrence des chiffres

Écrire un algorithme qui reçoit un nombre entier positif ou nul en paramètre et qui retourne un tableau de 10 entiers indiquant, pour chacun de ses chiffres, le nombre de fois qu'il apparait dans ce nombre.

Ainsi, pour le nombre 10502851125, on retournera le tableau [2,3,2,0,0,3,0,0,1,0].

120 Les doublons

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

121 Le crible d'Ératosthène

> « Le crible d'Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné N. L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers. On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant. On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant, car dans ce cas, tous les non-premiers ont déjà été rayés précédemment. » (source : Wikipédia)

Le tableau dont il est question peut être un simple tableau de booléens ; le booléen en position "i" indiquant si le nombre "i" est premier ou pas.

Écrire un algorithme qui reçoit un entier n et affiche tous les entiers premiers de 1 à n [69].

122 Mastermind

> Dans le jeu du Mastermind, un joueur A doit trouver une combinaison de n pions de couleur, choisie et tenue secrète par un autre joueur B. Cette combinaison peut contenir éventuellement des pions de même couleur. À chaque proposition du joueur A, le joueur B indique le nombre de pions de la proposition qui sont corrects et bien placés et le nombre de pions corrects mais mal placés.
>
> Les propositions du joueur A, ainsi que la combinaison secrète du joueur B sont contenues dans des tableaux de n composantes de type chaine.

Écrire deux algorithmes :

- ▷ l'un qui renvoie le nombre de pions de la bonne couleur et bien placés et ;
- ▷ l'autre qui renvoie le nombre de pions de la bonne couleur et mal placés.

69. Tester votre programme avec de « grandes » valeurs de n.

A.9 Exercices : chaine et String

123 Calcul de fraction

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

124 Conversion de nom

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

125 Gauche et droite

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

126 Grammaire

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

127 Normaliser une chaine

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

128 Les palindromes

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

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

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

129 Le chiffre de César

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

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

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

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

On vous demande également d'écrire l'algorithme de déchiffrement. Cet algorithme reçoit un message chiffré et la valeur de k qui a été utilisée pour le chiffrer et retourne le message en clair.

Remarque Ce second algorithme est **très simple** dès lors que l'algorithme de chiffrement est écrit.

130 Remplacement de sous-chaines

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

A.10 Exercices récapitulatifs

La plupart des exercices qui suivent sont inspirés d'anciens examens. Ils sont assez conséquents et reprennent la plupart des notions vues dans ce cours.

131 AEBBCLRS

Le Scrabble est un jeu de lettres, où le but est de former des mots ayant le score le plus élevé possible. Pour calculer le score d'un mot, une valeur est associée à chaque lettre de l'alphabet. Par exemple la lettre `A` vaut 1, `V` vaut 4, `J` vaut 8,... Le score d'un mot est la somme de la valeur de ses lettres [70].

Par exemple le score du mot `JAVA` est $8 + 1 + 4 + 1 = 14$.

Les joueurs tirent chacun 7 lettres qu'ils placent sur un *chevalet*. Le but du jeu est de former des mots à partir des lettres du chevalet et de les placer sur le plateau de jeu.

70. Dans le vrai jeu c'est un peu plus compliqué.

Valeur des lettres

Écrire un algorithme, `valeurLettre`, qui reçoit un caractère et retourne sa valeur. Les valeurs des lettres sont les suivantes :

- ▷ A,E,I,L,N,O,R,S,T,U : 1 point
- ▷ D,G,M : 2 points
- ▷ B,C,P : 3 points
- ▷ F,H,V : 4 points
- ▷ J,Q : 8 points
- ▷ K,W,X,Y,Z : 10 points

L'algorithme lance une erreur avec un message adéquat si le caractère passé en paramètre n'est pas une lettre majuscule.

Score d'un mot

Écrire un algorithme, `scoreMot`, qui reçoit une chaine de caractères, `mot`, et retourne la valeur de ce mot, c'est-à-dire la somme de la valeur de chacune de ses lettres.

Exemple : si l'algorithme reçoit le mot `JAVA`, il retourne 14 (=8+1+4+1).

Mot possible

Écrire un algorithme, `motPossible`, qui reçoit un tableau de caractères, `chevalet`, et une chaine de caractères, `mot`, et retourne vrai si les lettres du chevalet permettent d'obtenir le mot donné, et retourne faux dans le cas contraire.

Exemple : si l'algorithme reçoit le tableau `[A, O, V, G, G, J, L]` et le mot `"ALGO"`, l'algorithme retourne vrai. Par contre si le mot donné est `"JAVA"`, alors l'algorithme retourne faux car la lettre `A` n'apparait pas 2 fois dans le chevalet.

Aide : Une solution possible est de faire une copie du chevalet et pour chaque lettre du mot de vérifier que la lettre s'y trouve, si elle s'y trouve de la remplacer par un symbole (par exemple un '-'). Pour cela on définit un algorithme `copieChevalet` permettant de copier un chevalet et un algorithme `indiceLettre` permettant de connaitre l'indice d'une lettre dans un chevalet (cet algorithme retourne -1 si la lettre ne s'y trouve pas).

Meilleur mot

Écrire un algorithme, `meilleurMot`, qui reçoit :

- ▷ un tableau de caractères, `chevalet`, qui contient les lettres disponibles pour former un mot ;
- ▷ un tableau de chaines de caractères, `dico`, qui contient par exemple tous les mots du dictionnaire.

L'algorithme retourne le mot du dictionnaire que l'on peut obtenir avec les lettres du chevalet et qui a le score le plus élevé. Si aucun mot n'est possible avec le chevalet, l'algorithme retourne un mot vide `""`.

132 Serpents et échelles

> Serpents et échelles est un jeu de société populaire, se jouant à plusieurs, consistant à déplacer les jetons sur un tableau de cases avec un dé en essayant de monter les échelles et en évitant de trébucher sur les serpents.
>
> Pour l'histoire, le jeu peut être perçu comme une représentation d'un chemin spirituel que les humains prennent pour atteindre le ciel. Avec des bons gestes, le chemin est raccourci (ce que symbolisent les échelles), tandis qu'avec de mauvais gestes, le chemin est allongé (d'où vient le symbolisme des serpents).
>
> Extrait de Wikipedia.

Représentation du chemin

Nous représenterons le chemin par un tableau d'entiers. Chaque case du tableau contiendra une valeur :

- ▷ positive pour représenter une échelle et permettre d'avancer plus vite ;
- ▷ négative pour représenter le serpent et ralentir (voire reculer).

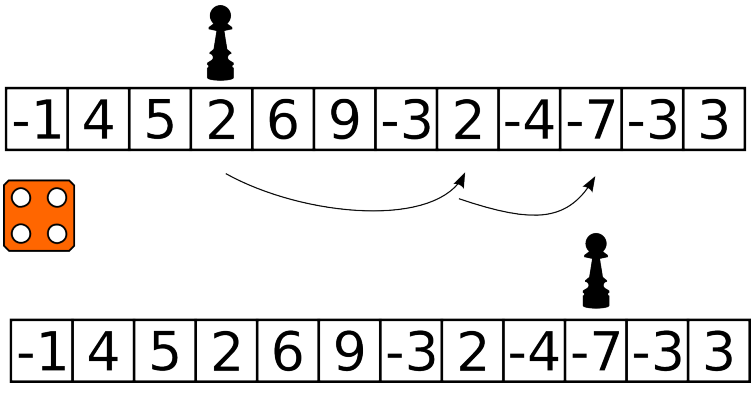
Règles du jeu

Pour jouer, le joueur qui a la main lance le dé et avance de la valeur indiquée par le dé. À partir de cette nouvelle position, il vérifie la valeur de la case sur laquelle il se trouve pour continuer à avancer ou au contraire reculer. Si la valeur est positive, il avance et si elle est négative, il recule de la valeur contenue dans la case.

Si la case est déjà occupée, le joueur se placera sur la case suivante.

Le joueur passe la main au joueur suivant.

Exemple. Le joueur se trouvant à la position 3 lance le dé. Il obtient la valeur 4. Il avance de 4 cases et se retrouve sur la case en position 7 qui contient la valeur 2. Il avance encore de 2 cases ... et termine donc son tour en position 9. Il passe alors la main au joueur suivant.



Le premier joueur à atteindre ou dépasser la dernière case a gagné. Nous supposerons que cette dernière case est en position 42. Nous supposons également que la première et la dernière case n'ont ni échelle ni serpent (valeur nulle).

Positions des joueurs

Les positions des joueurs seront mémorisées dans un tableau d'entiers. Toutes les cases de ce tableau sont initialisées à 0.

Le plateau de jeu, un tableau d'entiers s'appellera **chemin**.

Le tableau d'entiers contenant la position courante des joueurs s'appellera **positionsJoueurs**.

S'il y a 4 joueurs, le tableau *positionsJoueurs* contient [2,5,1,7] alors :

- ▷ le joueur 0 est en position 2 sur le chemin ;
- ▷ le joueur 1 est en position 5 sur le chemin ;
- ▷ le joueur 2 est en position 1 sur le chemin ;
- ▷ le joueur 3 est en position 7 sur le chemin ;

Le premier jour, initialiser le chemin

Écrivez un algorithme

```java
public static int[] créerChemin(int n)
```

Cet algorithme *créerChemin* créera un tableau d'entiers dont les valeurs seront des valeurs aléatoires comprises strictement entre -10 et 10 (inclus).

Nous supposons que l'entier n est un naturel strictement supérieur à 0. Vous ne devez pas le vérifier.

Qui est en tête ?

Écrivez un algorithme

```java
public static int enPremièrePosition(int[] positionsJoueurs)
```

Cet algorithme retourne le numéro du joueur en tête de la course. Il recherche donc le maximum dans le tableau passé en argument et retourne l'indice de ce maximum.

Jouer un tour de jeu

Cet algorithme permettra de jouer un tour de jeu pour un joueur donné. Jouer un tour de jeu consiste à le faire avancer ou reculer (c'est-à-dire changer sa position) du bon nombre de cases.

```java
public static void jouer(int[] chemin,
    int[] positionsJoueurs,
    int joueurCourant,
    int valeurDé)
```

Cet algorithme fait changer de position le joueur d'indice *joueurCourant* de la valeur donnée par le dé (*valeurDé*) en respectant les règles du jeu.

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

Comme le joueur ne peut se placer sur une case déjà occupée, il peut être utile d'écrire un module `estOccupé` précisant si la case est libre ou occupée.

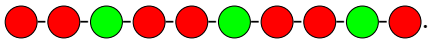
133 Les algorithmes en maternelle

En maternelle, déjà, les enfants font des algorithmes mais il s'agit d'une chose un peu différente. On leur propose un collier[71] dessiné sur une feuille de papier et ils doivent le colorier en répétant un *motif* donné[72], une séquence précise de couleurs.

Par exemple, on donne ce collier

qui est précolorié avec deux perles rouges et une perle verte, c'est le motif[73].

Le résultat attendu est :

 .

Comme on peut le constater sur cet exemple, un motif peut comporter plusieurs perles de la même couleur et, en fin de collier, il est possible qu'on ne puisse appliquer qu'une partie du motif.

Nous allons représenter chaque perle par un caractère indiquant sa couleur et un collier comme un tableau de caractères. Une perle non coloriée sera indiquée par un point (`'.'`).

Par exemple, le collier ci-dessus serait représenté ainsi :

`'R'`	`'R'`	`'V'`	`'.'`	`'.'`	`'.'`	`'.'`	`'.'`	`'.'`	`'.'`

Créer un collier

Écrivez un algorithme **créerCollier** qui reçoit une taille et crée un collier de cette taille où toutes les perles sont non coloriées (rappel : une perle non coloriée est représentée par un point).

On suppose que la taille reçue est bien un entier non négatif.

Créer un motif

Écrivez un algorithme **créerMotif** qui reçoit un collier dont aucune perle n'est coloriée et qui colorie les premières en fonction des indications de l'utilisateur.

Concrètement, l'utilisateur entre les couleurs des perles en spécifiant à chaque fois, après, s'il y a encore une perle à colorier. Dans notre exemple de la première page, l'utilisateur entrerait successivement : 'R', vrai, 'R', vrai, 'V', faux.

L'algorithme doit vérifier que l'utilisateur ne demande pas à colorier plus de perles qu'il n'y en a dans le collier.

71. Par exemple, mais ça peut aussi être une chenille, un escargot, une simple suite de cases...
72. Ce qu'ils appellent un *algorithme*.
73. Pour cet exercice, le lecteur est content s'il a la version couleurs des notes.

Taille du motif

Écrivez un algorithme **tailleMotif** qui reçoit un collier dont seulement les premières perles sont coloriées (c'est le motif qu'il faudra suivre) et qui donne la taille de ce motif. Dans l'exemple de la première page, il faudra retourner la valeur 3. Votre algorithme doit générer une erreur si il n'y a pas de motif à suivre.

Suivre un motif

Écrivez un algorithme **colorier** qui reçoit un collier dont seulement les premières perles sont coloriées (c'est le motif qu'il faut suivre) et qui colorie le reste du collier en suivant ce motif.

Vérifier un collier

Écrivez un algorithme **vérifier** qui reçoit un collier complètement colorié et la taille du motif de départ et qui vérifie si le collier respecte ce motif.

Trouver le motif

Écrivez un algorithme **trouverMotif** qui reçoit un collier complètement colorié et qui détermine la taille du motif. C'est la plus petite séquence qui se répète. Comme cas extrème, ce pourrait être le collier tout entier.

Aide : vous avez déjà tout pour que cet exercice soit facile.

Annexe B

Exercices résolus

Contenu

les exercices résolus sont différents des fiches. les fiches sont des problèmes « connus » ou « habituels»

lire un nombre / lecture robuste

B.1 Exercice résolu : le blackjack

Le blackjack est un jeu de casino dans lequel le joueur ou la joueuse joue contre la banque. Le but du jeu est que la somme de ses cartes soit le plus proche de 21 sans pour autant dépasser cette valeur. Chaque carte a pour valeur son rang, sauf le valet (11), la dame (12) et roi (13) qui valent **dix** et l'as (1) qui vaut **1 ou 11**.

Écrivons un algorithme qui reçoit les cartes représentant la main du joueur ou de la joueuse dans un tableau d'entiers (de 1 à 13) et qui retourne la somme la plus avantageuse pour la personne. C'est à dire, celle qui est le plus proche de 21 sans pour autant dépasser cette valeur. Il faut donc attribuer aux as la valeur 1 ou 11 selon le jeu de la personne.

Exemples :

- ▷ pour le tableau de cartes [5,12,1,4], l'algorithme renvoie 20 (5 + 10 (la dame vaut 10) + 1 + 4) ;
- ▷ pour le tableau de cartes [1,1,7] l'algorithme renvoie 19 (1 + 11 + 7) ;
- ▷ pour le tableau de cartes [2,3,11,1,1] l'algorithme renvoie 17 (2 + 3 + 10 + 1 + 1) ;
- ▷ pour le tableau de cartes [13,3,11,1,1] l'algorithme renvoie 25 (10 + 3 + 10 + 1 + 1).

Stratégie de résolution Avant d'espérer une solution correcte à un problème, il faut s'assurer de bien le comprendre, d'identifier tous les cas particuliers et de savoir soi-même calculer la bonne réponse dans ces cas là.

Lorsque le problème semble complexe, une stratégie peut être d'attaquer d'abord une version plus simple du problème pour incorporer par la suite les éléments mis de coté. C'est ce que nous allons faire ici.

B.1.1 Une version simplifiée : tous les as valent 1

Commençons donc par considérer une version plus simple où tous les as valent 1. Si on arrive à résoudre ce problème on réintroduira la possibilité pour un as de valoir 11.

Dans cette version simplifiée, il s'agit de calculer la somme des valeurs des cartes. Ce qu'il ne faut pas confondre avec la carte elle-même. Car si un 5 vaut 5, un 12 vaut 10.

La somme Comme chaque carte compte, on doit effectuer un parcours complet du tableau. On peut schématiser la solution ainsi :

```
Algorithme : somme simplifié d'un tableau de cartes.

Soit cartes le tableau des cartes
Initialiser une somme à 0
Pour toutes les cartes
    Calculer la valeur de la carte
    Ajouter cette valeur à la somme
```

langage naturel

Dans l'algorithme que l'on vient d'écrire, il existe un seul élément qui n'est pas trivial : le **calcul de la valeur d'une carte**. Il s'agit d'un problème en soi, qu'il est bon de regarder et de résoudre séparément sans polluer l'algorithme que l'on vient d'écrire

La valeur d'une carte Dans la plupart des cas, la valeur de la carte est égale à elle-même. Les seules exceptions sont les figures (valet, dame et roi) qui valent 10. On peut l'exprimer ainsi :

```
Algorithme : Calcul de la valeur d'une carte

Soit carte une carte
Si la carte est > 10 alors
    sa valeur est 10
sinon
    sa valeur est égale à elle-même
```

langage naturel

Solution en Java Si on exprime le tout en Java, ça donne :

```java
public static int valeur(int carte) {
    int valeur;
    if(carte > 10) {
        valeur = 10;
    } else {
        valeur = carte;
    }
    return valeur;
}

public static int sommeSimplifiée(int[] cartes) {
    int somme = 0;
    for(int carte : cartes) {
        somme += valeur(carte);
    }
    return somme;
}
```

java

Si on veut pouvoir le tester, il faut écrire une méthode principale qui crée le tableau

et appelle la méthode calculant la somme. Pour le premier exemple de l'énoncé, cela donne

```java
public static void main(String[] args) {
    int[] cartes = {5,12,1,4};
    int somme = sommeSimplifiée(cartes);
    System.out.println(somme);
}
```

java

B.1.2 Version complète : un as vaut 1 ou 11

Nous avons une solution mais rappelons-nous que nous avons simplifié le problème. Réintroduisons la règle qui dit qu'un as vaut 1 ou 11.

1 ou 11 ? Comment décider ? Il faut bien comprendre les points suivants :

- ▷ un as va compter pour 11 si et seulement si ça ne fait pas dépasser la valeur 21 ;
- ▷ même si plusieurs as sont présents dans la main du joueur, il ne peut y en avoir qu'un seul maximum qui compte pour 11, sinon le total dépasserait 21 ;
- ▷ on ne peut pas décider qu'un as vaut pour 11 tant qu'on n'a pas calculé la somme complète.
  Par exemple avec [8,1,7], lorsqu'on rencontre le 1, on pourrait être tenté de le compter comme un 11 ($8 + 11 = 19 < 21$) mais ce sera un problème par la suite pour ajouter le 7 ($19 + 7 = 26 > 21$).

Des constatation ci-dessus, on peut tirer un algorithme assez simple

```
Algorithme : somme non simplifié d'un tableau de cartes.

Soit cartes le tableau des cartes
Calculer la somme simplifiée, c'est à priori la bonne réponse
Mais s'il existe (au moins) un as dans les cartes
                et que la somme simplifiée est < 12 alors
    Ajouter 10 à la somme simplifiée
```

langage naturel

Dans cet algorithme le seul élément non trivial qu'il reste est de savoir s'il y a (au moins) un as dans la main du joueur. Voyons comment le résoudre.

Au moins un as Se demander s'il y a un as dans la main du joueur (le tableau de cartes) est un problème très proche d'autres qu'on a déjà résolu (ex : "est-ce que le tableau contient un 0 ?"). On a vu que la solution mettait en oeuvre un **parcours partiel** du tableau. On a vu aussi qu'il y a deux solutions types : avec et sans variable booléenne. Il suffit donc de reprendre et d'adapter une des 2 approches déjà écrite par ailleurs.

Solution en Java La solution en Java peut utiliser les 2 méthodes déja écrites.

```java
public static int somme(int[] cartes) {
    int somme = sommeSimplifiée(cartes);
    if (existeAs(cartes) && somme<12) {
        somme += 10;
    }
    return somme;
}

public static boolean existeAs(int[] cartes) {
    int i = 0;
    boolean asTrouvé = false;
    while (i<cartes.length && !asTrouvé) {
        asTrouvé = (cartes[i]==1);
        i++;
    }
    return asTrouvé;
}
```

java

Revoyez l'algorithme de parcours partiel si vous ne comprenez pas bien la méthode **existeAs**. Notez qu'on aurait pu utiliser l'autre approche pour **existeAs** ce qui aurait donné :

```java
public static boolean existeAs(int[] cartes) {
    int i = 0;
    while (i<cartes.length && cartes[i]!=1) {
        i++;
    }
    return i<cartes.length;
}
```

java

Vous vous rappelez pourquoi le test du return doit être celui-là ?

Il reste une dernière chose à faire : modifier la méthode principale afin qu'elle appelle la nouvelle version de la somme :

```java
public static void main(String[] args) {
    int[] cartes = {5,12,1,4};
    int somme = somme(cartes);
    System.out.println(somme);
}
```

java

Ce n'est pas tout-à-fait fini. Comme tout programmeur est faillible, il reste une chose importante à faire : tester. Au minimum, en testant que le programme donne la bonne réponse dans tous les exemples donnés dans l'énoncé.

B.1.3 Conclusion

Il y trois enseignements à tirer de cet exercice :

1. Ne pas hésiter à attaquer d'abord une version simplifiée du problème. Cela permet d'arriver plus facilement à la solution finale.

   Ici, il n'y a même pas eu de travail inutile puisque nous avons pu réutiliser tel quel le code écrit pour la version simplifiée.

2. Décomposer la solution en plusieurs algorithme. Si on identifie un sous-problème, il mérite une méthode à part plutôt que d'incorporer toutes les lignes de code dans une même méthode. On obtiendrait une longue méthode qui a beaucoup de défauts (plus difficile à lire/comprendre/corriger) et aucun avantage décisif (peut-être un rien plus rapide mais c'est négligeable).

3. Ne pas réinventer la roue ! Toujours se demander si le problème que l'on doit résoudre ne fait pas partie d'une catégorie de problèmes déjà résolu. Si c'est la cas, reprendre et adapter la solution déjà trouvée plutôt que de tout recommencer du début.

   Par exemple, ici, on a compris que rechercher la présence d'un as est un parcours partiel de tableau à la recherche d'un élément donné. Un problème déjà étudié et résolu.

Annexe C

Les fiches

Vous trouverez ici des fiches récapitulatives d'algorithmes analysés dans ce cours et présentant des problèmes habituellement rencontrés en développement.

Vous êtes invités à y faire référence.

En tant que référence — éventuellement — pour une année ultérieure, nous avons exprimé les algorithmes en pseudocode également. Lors d'une première lecture, vous pouvez laisser cette formulation de côté pour y revenir plus tard.

Fiche n° 1 – Un calcul simple

Le problème

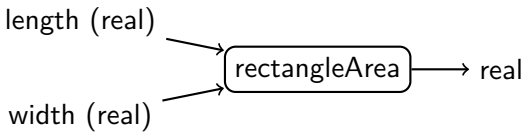
Calculer la surface d'un rectangle à partir de sa longueur et sa largeur.

Spécification

Données (toutes réelles et non négatives) :

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

Résultat : un réel représentant la surface du rectangle.



Exemples

- ▷ rectangleArea(3, 2) donne 6
- ▷ rectangleArea(3.5, 1) donne 3.5
- ▷ rectangleArea(4, 0) donne 0

Solution

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

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

```
surface = longueur * largeur
```

langage naturel

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

pseudocode

```java
public static double rectangleArea(double length, double width){
    return length * width;
}
```

java

Vérification / tests

test n°	longueur	largeur	réponse attendue	réponse fournie	
1	3	2	6	6	✓
2	3.5	1	3.5	3.5	✓
3	4	0	0	0	✓

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 suivante)

Fiche n° 2 – Un calcul complexe

Problème

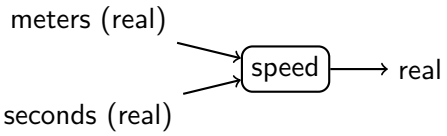
Calculer la vitesse moyenne (en km/h) d'un véhicule dont on donne la distance parcourue (en mètres) et la durée du parcours (en secondes).

Spécification

Données (toutes réelles et non négatives) :

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

Résultat : la vitesse moyenne du véhicule (en km/h).



Exemples

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

Solution

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

$$\text{vitesse} = \frac{\text{distance}}{\text{durée}}$$

pour autant que les unités soient cohérentes.

Ainsi pour obtenir une vitesse en km/h, il faut convertir la distance en kilomètres et la durée en heures. Ce qui donne :

```
convertir la distance reçue en mètres, en km : / 1000
convertir la durée reçue en secondes, en heure : / 3600
la vitesse est la distance / durée
```

langage naturel

```
algorithm speed(meters, seconds : real) → real
    real km, hours
    km = meters / 1000
    hours = seconds / 3600
    return km / hours
```

pseudocode

```java
public static double speed(double meters, double seconds){
  double km;
  double hours;
  km = meters / 1000;
  hours = seconds / 3600;
  return km / hours;
}
```

java

Vérification / tests

test n°	dist. (m)	durée (s)	rép. attendue	rép. fournie	
1	100	10	36	36	✓
2	10000	3600	10	10	✓

Remarque

Nous n'avons pas tenu compte des cas où la distance serait nulle et dans ce cas, la vitesse l'est aussi et où la durée est nulle... dans ce cas la vitesse serait infinie !

Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois que la réponse s'obtient par un calcul complexe sur les données qu'il est bon de décomposer pour aider à sa lecture. Si le calcul est plutôt simple, on peut le garder en une seule expression (cf. fiche 1 page 206).

Fiche n° 3 – Un nombre pair

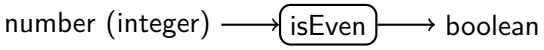
Problème

Un nombre reçu en paramètre est-il pair [74] ?

Spécification

Données : le nombre entier dont on veut savoir si il est pair.

Résultat : un booléen à *vrai* si le *nombre* est pair et *faux* sinon.



Exemples

- ▷ isEven(16) donne *vrai*
- ▷ isEven(15) donne *faux*
- ▷ isEven(0) donne *vrai*

Solution

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

```
n est pair si n modulo 2 = 0
```

langage naturel

```
algorithm isEven(number : integer) → boolean
    return number MOD 2 == 0
```

pseudocode

```java
public static boolean isEven(int n){
  return n%2 == 0;
}

```

java

Remarque C'est mieux de respecter les bonnes pratiques d'écriture et d'écrire directement un return sans if dans ce cas. Voir annexe D.3 page 240.

74. *Even* en anglais et *odd* pour impair.

Vérification / tests

test n°	n	rép. attendue	rép. fournie	
1	16	true	true	✓
2	15	false	false	✓
3	0	true	true	✓

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 208.

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

Problème

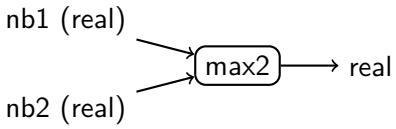
Quel est le maximum de deux nombres ?

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'ambigüité 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
- ▷ max2(-4, -8) donne -4

Solution

Pour trouver le maximum des deux nombres, il suffit de les comparer en utilisant un si.

```
si nb1 > nb2 alors
    nb1 est le maximum
sinon
    nb2 est le maximum
```

langage naturel

```
algorithm max2(nb1 : real, nb2 : real) → real
    real max
    if nb1 > nb2 then
        max = nb1
    else
        max = nb2

    return max
```

pseudocode

```java
public static double max2(double nb1, double nb2){
  double max;
  if (nb1 > nb2){
    max = nb1;
  } else {
    max = nb2;
  }
  return max;
}
```

java

Nous pourrions raisonner différement et se dire que le maximum des deux nombres est le premier... sauf si le deuxième est plus grand. C'est une autre manière de penser.

```
le maximum est nb1
si nb2 > nb1 alors
    le maximum est nb2
```

langage naturel

```java
public static double max2(double nb1, double nb2){
  double max = nb1
  if (nb2 > nb1){
    max = nb2;
  }
  return max;
}

```

java

Remarques

- C'est mieux de respecter les bonnes pratiques d'écriture et d'écrire directement un return sans if dans ce cas. Voir annexe D.4 page 241.
- Nous avons rapidement montré l'opérateur ternaire (cfr. Section 11.2.1 page 134) permettant d'écrire cette méthode en une seule expression.

```java
return nb1 > nb2 ? nb1 : nb2;
```

java

Vérification / tests

test n°	nb1	nb2	rép. attendue	rép. fournie	
1	-3	4	4	4	✓
2	7	4	7	7	✓
3	4	4	4	4	✓
4	-4	-8	-4	-4	✓

Quand l'utiliser ?

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

Fiche n° 5 – Parcours complet d'un tableau

Problème

Parcours complet d'un tableau pour afficher tous les éléments. Par exemple un tableau de double.

Spécification

- ▷ **Données** : le tableau à afficher
- ▷ **Résultat** : aucun.
- ▷ **Affiche** : les éléments du tableau, dans l'ordre.

Solution

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

```
algorithm display(ds : array of real)
    for i from 0 to ds.length - 1
        print ds[i]
```

pseudocode

```java
// Utilisation d'un for
public static void display (double[] ds){
  for (int i=0; i < ds.length; i++){
    System.out.println(ds[i]);
  }
}

```

java

Si l'on n'a pas besoin de modifier la valeur dans le tableau et que l'on veut simplement utiliser la valeur — dans ce cas, pour l'afficher — on peut utiliser une boucle *enhanced for* comme suit :

```java
// Utilisation d'un enhanced for (foreach)
public static void display (double[] ds){
  for (double d: ds){
    System.out.println(d);
  }
}
```

java

Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois qu'il est nécessaire d'examiner **tous** les éléments d'un tableau, quel que soit le traitement voulu : les afficher, les sommer, les comparer...

L'*enhanced for* ne permet pas de modifier les éléments du tableau.

Fiche n° 6 – Maximum dans un tableau

Problème

Parcours complet d'un tableau pour rechercher la valeur maximale. Par exemple dans un tableau d'entiers.

Spécification

Données : le tableau à analyser

Résultat : la valeur du maximum

Solution

Une bonne manière de faire est d'initialiser la valeur du maximum à la première valeur du tableau et d'ensuite parcourir le reste du tableau à la recherche d'une valeur plus grande.

```
int max
int[] is
maximum = première valeur du tableau, is[0]
pour chaque valeur à partir de la deuxième
    si max > valeur alors
        max = valeur
    fin si
fin pour
```

langage naturel

```
algorithm max(is : array of integer) → integer
    integer max
    max = is[0]
    for i from 1 to is.length - 1
        if is[i] > max then
            max = is[i]


    return max
```

pseudocode

```java
public static int max(int[] is){
  int max = is[0];
  for (int i=1; i < is.length; i++){
    if (is[i] > max){
      max = is[i];
    }
  }
  return max;
}
```

java

Si l'on accepte de tester la première valeur, cet algorithme peut s'écrire avec un *enhanced for*.

```java
public static int max(int[] is){
  int max = is[0];
  for (value: is){
    if (value > max){
      max = value;
    }
  }
  return max,
}
```

java

Variante

Si c'est la position et non la valeur qui nous intéresse, une solution serait :

```
algorithm max(is : array of  integer) → integer
    integer maxIndex
    maxIndex = 0
    for i from 1 to is.length - 1
        if is[i]>is[maxIndex] then
            maxIndex = i


    return maxIndex
```

pseudocode

```java
public static int max(int[] is){
  int maxIndex = 0;
  for (int i=1; i < is.length; i++){
    if (is[i] > is[maxIndex]){
      maxIndex = i;
    }
  }
  return max;
}
```

java

Fiche n° 7 – Parcours partiel d'un tableau

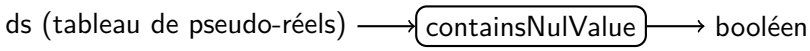
Problème

Parcours partiel d'un tableau pour rechercher un zéro. Par exemple avec un tableau de double.

Spécification

Données : le tableau à tester

Résultat : un booléen à vrai si il existe une valeur nulle dans le tableau.



Solution

Contrairement au parcours complet (cf. fiche 5 page 215) nous allons utiliser un *tant que* (while) car nous voulons arrêter l'algorithme dès que la valeur cherchée est trouvée.

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

[A] Sans variable booléenne

```
algorithm containsNulValue(ds : array of  double) → boolean
    integer i
    i = 0
    while i < ds.length AND ds[i] ≠ 0
        i = i + 1

    return i < n                                  // Si i<n -> arrêt prématuré
                                                  // c-à-d que l'on a trouvé un 0.
```

pseudocode

```
public static boolean containsNulValue(double[] ds){
  int i = 0;
  while (i < ds.length && ds[i] != 0){
    i++;
  }
  /* Si i < n c'est un arrêt prématuré et c'est que l'on a trouvé */
  return i < n
}
```

java

Il faut être attentif à **ne pas inverser** les deux parties du test sous peine de générer une erreur parce que l'on essaie d'accéder à un élément hors du tableau. Il faut absolument vérifier que l'indice est bon avant de tester la valeur à cet indice. Se

ce n'est pas clair, revoir la notion d'évaluation paresseuse (court-circuit). Voir 4.4.3 page 44.

[B] Avec variable booléenne

```
algorithm containsNulValue(is : array of integer) → boolean
    integer i
    boolean isZero
    isZero = false
    i = 0
    while i < is.length AND NON isZero
        isZero = is[i] == 0
        i = i + 1

    return isZero
```

pseudocode

```java
public static boolean containsNulValue(double[] ds){
  int i = 0;
  boolean isZero = false;
  while (i < ds.length && !isZero){
    isZero = ds[i] == 0;
    i++;
  }
  return isZero;
}
```

java

Au sortir de la boucle, l'indice i ne désigne pas l'élément qui nous a permis d'arrêter mais le suivant. Si nécessaire, on peut remplacer l'intérieur de la boucle par :

```java
if (ds[i] == 0){
  isZero = true;
} else {
  i++;
}
```

java

Dans notre exemple, nous cherchons un élément particulier (un 0). Dans le cas où l'on vérifie si tous les éléments possèdent une certaine propriété (être positifs par exemple), nous veillerons à adapter le nom du booléen et son utilisation (par exemple un booléen appelé areAllPositives, initialisé à true avec un . . .AND areAllPositives dans le test.

Dans tous les cas, faites attention à ne pas accéder à l'élément — utiliser ds[i] — si vous n'êtes pas sûr de l'indice i. C'est particulièrement vrai après la boucle.

Remarque C'est mieux de respecter les bonnes pratiques d'écriture et d'écrire directement un return sans if dans ce cas. Voir annexe D.5 page 242.

Quand l'utiliser ?

Ce type de solution peut être utilisé pour tout parcours d'un tableau où un arrêt prématuré est possible en se posant les questions suivantes :

- ▷ Est-ce que tous les éléments sont positifs ?
- ▷ Est-ce que les élément sont triés ?
- ▷ Est-ce qu'un élément précis est présent ?
- ▷ ...

Fiche n° 8 – Tableau non trié

Problème

Rechercher, ajouter, supprimer des données dans un tableau non trié. Par exemple dans un tableau d'entiers

Rechercher

Spécification

Retourner l'indice d'une donnée trouvée dans un tableau non trié ou -1 si elle n'est pas trouvée.

Nous supposons que le tableau n'est pas rempli, l'algorithme recevra donc une valeur représentant le nombre d'éléments du tableau. Cette valeur est inférieure à la taille du tableau bien sûr.

Données : le tableau à analyser, le nombre d'éléments dans ce tableau (taille logique), la valeur à rechercher

Résultat : la position de l'élément si il est dans le tableau et -1 sinon.

Solution

Il suffit de parcourir le tableau jusqu'à ce que l'on trouve la valeur ou bien que l'on arrive à la fin.

```
tant que l'on n'est pas à la fin et la valeur n'est pas trouvée
    incrémenter l'indice
fin tant que
si on n'est pas à la fin alors
    on a trouvé
sinon
    on n'a pas trouvé et on retourne -1
fin si
```

langage naturel

```
// Vérifie si un nombre est dans un tableau et donne sa position (-1 sinon)
algorithm indexValue(is↓ : array of integers,
                     nValues↓ : integer, value↓ : integer) → integer
    integer i
    i = 0
    while i < nValues AND is[i] ≠ value
        i = i + 1

    if i < nValues then
        return i
    else
        return -1
```

pseudocode

```java
public static int indexValue(int[] is, int nValues, int value){
  int i = 0;
  while (i < nValues && is[i] != value){
    i++;
  }
  if (i < nValues){
    return i;
  } else {
    return -1;
  }
}

```

java

Ajouter

Spécification

Ajouter une donnée non encore présente dans le tableau de données non triées.

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à ajouter

Résultat : le tableau reçu est modifié en lui ajoutant la valeur si elle n'y était pas déjà

Solution

Si l'algorithme reçoit le tableau, le nombre d'éléments et la valeur à ajouter, il suffit d'insérer la valeur dans le tableau... après avoir vérifier qu'elle n'est pas présente. Cette vérification peut se faire simplement en utilisant l'algorithme indexValue. Si celui-ci retourne -1, c'est que la valeur n'est pas présente.

```
if nValues < taille du tableau AND indexValue = -1
    ajouter la valeur à la fin du tableau
```

langage naturel

```
// Ajoute un nombre non encore présent dans le tableau.
algorithm add(is↓↑ : array of  integer,
                                        nValues↓↑ : integer, value↓ : integer)
    if nValues < is.length AND indexValue(is, nValues, value) = -1 then
        is[nValues] = value
        nValues = nValues + 1
```

pseudocode

Remarque En langage Java, le passage de paramètre se faisant par valeur (cfr. Chapitre 11 page 129), la taille logique du tableau doit être retournée et maintenue à jour par le code appelant. En langage Java, cet algorithme aurait l'allure suivante :

```
public static int add(int[] is, int nValues, int value){
  if (nValues < is.length()
      && indexValue(is, nValues, value) == -1){
    is[nValues] = value;
    nValues = nValues + 1;
  }
  return nValues;
}
```

java

Supprimer

Spécification

Supprimer une donnée d'un tableau de données non triées.

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à supprimer

Résultat : le tableau reçu est modifié en lui supprimant la valeur

Solution

Comme le tableau n'est pas trié, pour supprimer une valeur, il suffit de trouver l'index de cette valeur et de placer la dernière valeur à sa place.

```
index = l'index de la valeur à supprimer
if index != 1
    is[index] = is[nValues-1]
    nValues–
end if
```

langage naturel

```
// Supprime un nombre donné dans le tableau.
algorithm delete( is↓↑ : array of n integer,
                                        nValues↓↑ : integer, value↓ : integer )
    integer index
    index = indexValue(is, nValues, value)
    if index ≠ -1 then
        is[index] = is[nValues-1]
        nValues = nValues - 1
```

pseudocode

```
public static int delete(int[] is, int nValues, int value){
  int index = indexValue(is, nValues, value);
  if (index != -1){
    is[index] = is[nValues];
    nValues = nValues - 1;
  }
  return nValues;
}
```

java

Fiche n° 9 – Tableau trié

Problème

Rechercher, ajouter, supprimer des données triées dans un tableau trié. Par exemple dans un tableau d'entiers.

Rechercher

Spécification

Rechercher la position où a été trouvé l'élément ou la position où il aurait dû être.

Données : le tableau à analyser, le nombre d'éléments dans ce tableau, la valeur à rechercher

Résultat : la position où a été trouvée la valeur ou la position où elle aurait dû être

Solution

L'algorithme retourne la position où a été trouvée la valeur **ou** celle où elle devrait être. En parcourant le tableau, il suffit de s'arrêter dès que l'élément du tableau est plus grand ou égal à la valeur recherchée.

```
// Recherche un élément.
// - index : indique la position où a été trouvée la valeur
// ou la position où elle aurait dû être
algorithm findIndex( is↓ : array of  integer, nValues↓ : integer,
                                              value↓ : integer, ) → index integer
    integer index = 0
    while index < nValues AND is[index] < value
        index = index + 1

    return index
```

pseudocode

```
publci static int findIndex(int[] is, int nValues, int value){
  int index = 0;
  while (index < nValues && is[index] < value){
    index++;
  }
  return index;
}
```

java

Ajouter

Spécification

Ajouter une donnée non encore présente dans le tableau de données triées.

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à ajouter

Résultat : le tableau reçu est modifié en lui ajoutant la valeur si elle n'y était pas déjà

Solution

Pour faire un ajout sans doublon dans un tableau triée, il faut :

1. vérifier que la valeur n'est pas déjà présente ;
2. rechercher l'endroit ou elle devrait être placée ;
3. déplace les valeurs plus grande d'une position vers la droite ;
4. insère la valeur à sa place.

En décomposant le problème, voici une solution :

```
// Ajouter un nombre donné.
algorithm add( is↓↑ : array of  integer, nValues↓ : integer,
                                                    value↓ : integer ) → integer
    integer index
    boolean isFound
    if verify(is, nValues, value) == -1 then
        index = findIndex(is, nValues, value)
        shiftRight(is, index, nValues)
        is[index] = value
        nValues = nValues + 1

    return nValues


// Vérifie si un nombre est dans un tableau d'entiers trié
// et donne sa position (-1 si non présent)
algorithm verify( is↓ : array of  integers, nValues↓ : integer,
                                                    value↓ : integer ) → integer
    integer index
    boolean isFound
    index = findIndex( is, nValues, value)
    if is[index] == value then
        return index
    else
        return -1


// Décale d'une position à droite les éléments
// entre la position début et fin
algorithm shiftRight( is↓↑ : array of  integer, begin↓ : integer, end↓ : integer )
    for i from end to begin by -1
        is[i+1] = is[i]
```

pseudocode

```java
/**
 * Ajoute un nombre s'il n'est pas présent.
 *
 * @param is le tableau d'éléments
 * @param nValues le nombre d'éléments du tableau (taille logique)
 * @param value la valeur à insérer
 */
public static int add(int[] is, int nValues, int value){
  int index;
  if (verify(is, nValues, value) == -1){
    index = findIndex(is, nValues, value);
    shiftRight(is, index, nValues);
    is[index] = value;
    nValues++;
  }
  return nValues;
}

/**
 * Vérifie si la valeur est dans le tableau.
 *
 * @param is le tableau d'éléments
 * @param nValues le nombre d'éléments du tableau (taille logique)
 * @param value la valeur à insérer
 */
public static int verify(int[] is, int nValues, int value){
  int index = findIndex(is, nValues, value);
  if (is[index] == value) {
    return index;
  } else {
    return -1;
  }
}

/**
 * Décale d'une position vers la droite les éléments
 * entre position début et fin.
 *
 * @param is le tableau d'éléments
 * @param begin la position de début
 * @pasam end la position de fin
 */
public static void shiftRight(int[] is, int begin, int end){
  for (int i = end; i >= begin; i--){
    is[i+1] = is[i];
  }
}
```

java

Supprimer

Spécification

Supprimer une donnée d'un tableau de données triées.

Données : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à supprimer

Résultat : le tableau reçu est modifié en lui supprimant la valeur

Solution

Pour supprimer un élément, dès dès lors qu'il est trouvé, il suffit de décaler tous les élémens à sa droite vers la gauche et adapter la taille logique.

```
// supprimer l'élément donné
algorithm delete( is↓↑ : array of  integer, nValues↓↑ : integer,
                                                    value↓ : integer ) → integer
    integer index
    boolean isFound
    if verify(is, nValues, value) ≠ -1 then
        index = findIndex(is, nValues, value)
        shiftLeft(is, index + 1, nValues)
        nValues = nValues - 1

    return nValues



// Décale d'une position à gauche les éléments
// entre la position début et fin
algorithm shiftLeft( tab↓↑ : array of  integers, begin↓ : integer, end↓ : integer )
    for i from begin to end
        is[i-1] = is[i]
```

pseudocode

```java
public static int delete(int[] is, int nValues, int value){
  int index;
  if (verify(is, nValues, value) != -1){
    index = findIndex(is, nValues, value);
    shiftLeft(is, index+1, nValues);
    nValues = nValues - 1;
  }
  return nValues;
}

public static void shiftLeft(int[] is, int begin, int end){
  for (int i=begin, i < end; i++){
    is[i-1] = is[i];
  }
}
```

java

Fiche n° 10 – Recherche dichotomique

Problème

Trouver rapidement la position d'une valeur donnée dans un tableau **trié** d'entiers. Si la valeur n'est pas présente, on donnera la position où elle aurait du se trouver.

Voir Section 11.3 page 137.

Spécification

Données : le tableau à analyser et la valeur recherchée

Résultat : un booléen indiquant si la valeur a été trouvée et un entier indiquant soit la position où la valeur a été trouvée soit la position où elle aurait du être.

Solution

L'algorithme rapide que nous avons vu est la recherche dichotomique.

```java
public static int dichotomousSearch(int[] myArray, int value){
  int leftIndex = 0;
  int medianIndex;
  int rightIndex = myArray.length;
  int candidate;
  boolean isFound = false;

  while (!isFound && leftIndex < rightIndex){
    medianIndex = (leftIndex + rightIndex) / 2;
    candidate = myArray[medianIndex];
    if (candidate == value){
      isFound = true;
    } else if (candidate < value){
      leftIndex = medianIndex + 1;
    } else {
      rightIndex = medianIndex - 1;
    }
  }

  if (isFound){
    return medianIndex;
  } else {
    return -1,
  }
}

```

java

Dans le cas d'un langage permettant d'autres passages de paramètres que le passage par valeur (cfr. Section 7.3 page 81), nous pourrions écrire un algorithme :

- ▷ qui retourne un booléen présisant si la valeur est trouvée et ;
- ▷ qui donne la position à laquelle se trouve la valeur ou la position à laquelle elle devrait se trouver ce qui serait pratique en cas d'insertion.

Un tel algorithme aurait cette allure :

```
algorithm dichotomousSearch(
      myArray↓ : array of  integers, value↓ : integer, pos↓↑ : integer ) → boolean
    integer rightIndex, leftIndex, medianIndex
    integer candidate
    boolean isFound

    leftIndex = 0
    rightIndex = myArray.length - 1
    isFound = false

    while NON isFound AND leftIndex ≤ rightIndex
        medianIndex = (leftIndex + rightIndex) DIV 2
        candidate = myArray[medianIndex]
        if candidate == value then
            isFound = vrai
        else if candidate < value then
            leftIndex = medianIndex + 1                  // on garde la partie droite
        else
            rightIndex = medianIndex – 1                 // on garde la partie gauche



    if isFound then
        pos = medianIndex
    else
        pos = leftIndex                  // dans le cas où la valeur n'est pas trouvée,
          // on vérifiera que leftIndex donne la valeur où elle pourrait être insérée.


    return isFound
```

pseudocode

Fiche n° 11 – Tri d'un tableau

Trier un tableau d'entiers par ordre croissant. Les algorithmes de tri sont présentés aux sections 12.2 à 12.4 pages 145–148.

Spécification

Données : le tableau non trié, à trier.

Résultat : ce même tableau trié.

Solution

Le tri par insertion

```
// Trie le tableau reçu en paramètre (via un tri par insertion).
algorithm insertionSort(myArray↓↑ : array of integers)
    integers i, j, value
    for i from 1 to myArray.length - 1
        value = myArray[i]
        // recherche de l'endroit où insérer value dans le
        // sous-tableau trié et décalage simultané des éléments
        j = i - 1
        while j ≥ 0 AND value < myArray[j]
            myArray[j+1] = myArray[j]
            j = j – 1

        myArray[j+1] = value
```

pseudocode

```java
/**
 * Tri par insertion.
 * @param myArray tableau à trier
 */
public static insertionSort(int[] myArray){
  int j, value;
  for (int i = 1; i < myArray.length; i = i + 1){
    value = myArray[i];
    j = i - 1;
    while (j >= 0 && value < myArray[j]){
      myArray[j + 1] = myArray[j];
      j = j - 1;
    }
    myArray[j + 1] = value;
  }
}
```

java

Le tri par sélection des minima successifs

```
// Trie le tableau reçu en paramètre (via un tri par sélection des minima successifs).
algorithm selectionSort(myArray↓↑ : array of integers)
    integer i, minIndex
    for i from 0 to myArray.length – 2        // i correspond à l'étape        // de
l'algorithme
        minIndex = searchMinIndex( myArray, i, myArray.length - 1 )
        swap( myArray[i], myArray[minIndex] )



// Retourne l'indice du minimum entre les indices début et fin du tableau reçu.
algorithm searchMinIndex( myArray↓ : array of integers,
                                                  début↓, fin↓ : integers) → integer
    integers minIndex, i
    minIndex = début
    for i from début+1 to fin
        if myArray[i] < myArray[minIndex] then
            minIndex = i


    return minIndex



// Échange le contenu de 2 variables.
algorithm swap(a↓↑, b↓↑ : integers)
    integer aux
    aux = a
    a = b
    b = aux
```

pseudocode

```java
/**
 * Trie le tableau reçu en paramètre via un tri
 * par sélection des minima successifs
 *
 * @param myArray le tableau à trier
 */
public static void selectionSort(int[] myArray){
  int i;
  int minIndex;
  for (int i=0; i<myArray.length-1; i++){
    minIndex = searchMinIndex(myArray, i, myArray.length-1);
    // swap values
    int value = myArray[i];
    myArray[i] = myArray[minIndex];
    myArray[minIndex] = value;
  }
}

/**
 * Retourne l'indice du minimum entre les indices
 * début et fin du tableau reçu.
 *
 * @param myArray le tableau d'entiers
 * @param l'indice de début
 * @param l'indice de fin
 */
public static int searchMinIndex(
    int[] myArray,
    int begin,
    int end){
  int minIndex;
  int i;
  minIndex = begin;
  for (int i = begin + 1; i<end; i++){
    if (myArray[i] < myArray[minIndex]) {
      minIndex = i;
    }
  }
  return minIndex;
}
```

java

Le tri bulle

```
// Trie le tableau reçu en paramètre (via un tri bulle).
algorithm bubbleSort(myArray↓↑ : array of  integers)
    integers bubbleIndex, i
    for bubbleIndex from 0 to myArray.length - 2
        for i from myArray.length – 2 to bubbleIndex by -1
            if myArray[i] > myArray[i + 1] then
                swap( myArray[i], myArray[i + 1] )    // voir algorithme précédent
```

pseudocode

```java
public static void bubbleSort(int[] myArray){
  for (int bubbleIndex = 0;
       bubbleIndex < myArray.length - 1;
       bubbleIndex++){
    for (int i = myArray.length - 2; i <= bubbleIndex; i--){
      if (myArray[i] > myArray[i + 1]){
        // swap values
        int value = myArray[i];
        myArray[i] = myArray[i+1];
        myArray[i+1] = value;
      }
    }
  }
}
```

java

Annexe D

Bonnes pratiques

Ce chapitre recense quelques bonnes pratiques d'écriture d'algorithme et de code. Il montre également des manières d'écrire du code peu lisibles et donc à éviter, voire à proscrire.

Les bonnes pratiques d'écriture ont également un aspect culturel et dépendant d'un langage. En effet, certaines communautés de programmeurs habitués à un langage adoptent des styles de programmation qui ne sont pas partagés par d'autres programmeurs, habitués à un autre langage. Nous présentons ici des bonnes pratiques d'algorithmique générale et des bonnes pratiques liées au langage Java.

> « Lorsque je code en langage Java, je respecte les conventions du langage Java. Lorsque je code en langage C++, je respecte les conventions du langage C++.
> Lorsque je contribue à un projet, je respecte les conventions du langage et celles proposées par l'équipe en charge du projet. »

Contenu

D.1 Bonnes pratiques générales

Certaines conventions d'écriture, sont communes à tous les langages. Citons par exemple :

- ▷ préferer des lignes de longueur maximale de 80 caractères. Ceci facilite la lecture ;
- ▷ couper les lignes qui seront trop longues en suivant ces règles ;

- couper après une virqule ;
- couper avant un opérateur ;
- préférer les coupures de haut niveau de priorité aux coupures de priorité faible ;
- aligner la nouvelle ligne avec le début de l'expression se trouvant à la ligne précédente.

Nous vous invitons à consulter les conventions d'écriture du code en Java — même si le document est vieux — ceci afin d'écrire un code plus lisible. Voir [tea99]

D.2 Tester un booléen

Lorsqu'il s'agit de comparer deux nombres dans la condition d'un **if**, nous écrivons assez naturellement : **if** (nb1<nb2) et il ne vous viendrait pas à l'idée d'écrire : **if** (nb1<nb2 == true).

Avec un booléen, nous allons également éviter d'écrire == true ou == false. Par exemple, pour exécuter un traitement si la variable isAdult est vraie, certaines personnes programmeuses débutantes écrivent ceci :

```java
if (isAdult == true){
  // ...
}
```

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

```java
if (isAdult){
  // ...
}
```

C'est une bonne pratique d'utiliser cette deuxième écriture.

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

```java
if (!isAdult){
  // ...
}
```

D.3 Assigner un booléen

Pour assigner une valeur booléenne à une variable, il n'est pas utile d'utiliser un if-else, une assignation directe fait l'affaire.

Par exemple, si la variable booléenne isNul doit valoir true si un nombre est égal à 0 et false sinon, nous pourrions écrire le bout de code ci-dessous.

```java
if (nb == 0){
  isNul = true;
} else {
  isNul = false;
}
```

Ce code est correct mais inutilement compliqué. Puisque on assigne true si la condition est vraie et false si la condition est fausse, il suffit d'assigner la condition. Ce qui donne :

```java
isNul = nb == 0;
```

D.4 Assigner une valeur en fonction d'une condition

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

```java
if (isDiscount){
  price = 8;
} else {
  price = 12;
}
```

Nous rencontrons parfois le code suivant. Plus court. Est-il nécessairement mieux ?

```java
price = 12;
if (isDiscount){
  price = 8;
}
```

Cette manière d'écrire comporte effectivement (un peu) moins de lignes mais ce critère n'a que très peu d'importance. Cette version n'est pas plus rapide (elle est même plus lente en cas de tarif réduit) et elle est moins lisible car le lecteur croit d'abord que le tarif est toujours de 12 avant de constater qu'il peut être différent.

Nous privilégirons la première version.

D.5 Interrompre une boucle `for`

Certains développeurs et développeuses débutantes [75] interrompent anticipativement une boucle for en modifiant l'indice. C'est une mauvaise idée.

Dans l'exemple ci-dessous, l'indice i reçoit une « grande valeur » pour sortir de la boucle for.

```java
for (int i=0; i < n; i++){
  // ...
  if (aCondition) {
    i = n + 1;    // pour interrompre la boucle
  }
  // ...
}
```

Cette manière de faire est à proscrire absolument et ce, quelle qu'en soit la raison. Ce n'est pas une bonne idée en terme de lisibilité et dans certains langages c'est interdit par le compilateur ou cela ne fait pas ce qui est attendu.

Si vous devez interrompre un parcours répétitif, lorsqu'une condition est remplie, nous insistons pour l'utilisation d'une boucle **while** et l'utilisation d'un booléen pour contrôler la fin.

Par exemple :

```java
int i = 0;
isFinished = false;
while (i < n && !isFinished){
  // ...
  if (aCondition){
    isFinished = true;
  }
  // ...
  i = i + 1;
}
```

Si le test est la première instruction de la boucle et que la condition est courte, on peut aussi se passer de la variable booléenne.

```java
int i = 0;
while (i < n && !aCondition){
  // ...
  i = i + 1;
}
```

75. En essayant d'avoir une écriture plus inclusive, je m'autorise un accord de proximité.

Si vous tenez absolument à interrompre la boucle **for**, vous pouvez utiliser un return comme expliqué au point suivant.

D.6 Plusieurs return

Plus tôt dans ce cours nous avons énoncé comme règle :

> « Un seul return par algorithme ».

Cette règle n'est pas partagée par tous les développeurs et développeuses. Il existe un courant qui tolère plusieurs return dans certains cas précis.

En voici quelques uns.

Un cas concret est l'arrêt anticipé d'une boucle. Certains écrivent :

```java
for (int i = 0; i < n; i++){
  if (aCondition){
    return something;
  }
  // ...
  return somethingElse,
}
```

java

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

```java
if (aCondition){
  return something;
} else {
  return somethingElse;
}
```

java

Index

Références

[Bet09] Pierre Bettens. Unicode, utf8, utf16, utf32, ... et tutti quanti. http://blog.namok.be/?post/2009/11/30/unicode-UTF8-UTF16-UTF32-et-tutti-quanti, novembre 2009.

[Bet11] Pierre Bettens. Java 7 is out, quel est son lot de nouveautés ? http://blog.namok.be/?post/2011/07/11/jdk7, juillet 2011.

[Bet18] Pierre Bettens. Jdk9, les nouveautés. http://blog.namok.be/?post/2018/03/22/JDK9, mars 2018.

[Bet19] Pierre Bettens. Jdk10, 11 et 12, quelques nouveautés. http://blog.namok.be/?post/2019/04/06/JDK10-11-12, avril 2019.

[tea99] Java team. Code conventions for the java tm programming language. https://www.oracle.com/technetwork/java/codeconvtoc-136057.html, 20 avril 1999.

[tea18] Java team. Java® platform, standard edition & java development kit version 11 api specification. https://docs.oracle.com/en/java/javase/11/docs/api/index.html, 2018. essai note.

[VE04] Marion Videau and David Eck. Les algorithmes de tri. http://interstices.info/jcms/c_6973/les-algorithmes-de-tri, 2004.

[Wel00] Don Wells. Code the unit test first. http://www.extremeprogramming.org/rules/testfirst.html, 2000.