ALGORITHMIQUE POUR LE LYCÉE

Éric SOPENA
Eric.Sopena@labri.fr

SOMMAIRE

Chapitre 1. Notions de base d'algorithmique

1.1. Qu'est-ce qu'un algorithme ?

De façon intuitive, un algorithme décrit un enchaînement d'opérations permettant, en un temps fini, de résoudre toutes les instances d'un problème donné. Partant d'une instance du problème (les données en entrée), il fournit un résultat correspondant à la solution du problème sur cette instance.

La définition du Larousse est la suivante : « *ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations. Un algorithme peut être traduit, grâce à un langage de programmation, en un programme exécutable par un ordinateur* ».

On se doit cependant d'y apporter les compléments suivants :

- un algorithme décrit un traitement sur un nombre fini de données structurées (parfois aucune). Ces données peuvent avoir une structure élémentaire (nombres, caractères, etc.), ou une structure plus élaborée (liste de nombres, annuaire, etc.).
- un algorithme est composé d'un nombre fini d'opérations[1]. Une opération doit être bien définie (rigoureuse, non ambiguë), effective, c'est-à-dire réalisable par un ordinateur (la division entière est par exemple une opération effective alors que la division réelle, avec un nombre éventuellement infini de décimales, ne l'est pas).
- un algorithme doit toujours se terminer après exécution[2] d'un nombre fini d'opérations et donner un résultat.

Ainsi, l'expression d'un algorithme nécessite un langage clair (compréhension), structuré (décrire des enchaînements d'opérations), non ambigu (la programmation ne supporte pas l'ambiguïté !). Il doit de plus être « universel » : un (vrai) algorithme doit être indépendant du langage de programmation utilisé par la suite (e.g. l'algorithme Euclide !).

En particulier, une recette de cuisine est un très mauvais exemple d'algorithme, du fait de l'imprécision notoire des instructions qui la composent (rajouter une « pincée de sel », verser un « verre de farine », faire mijoter « à feu doux », placer au four « 45 mn environ », …).

Algorithme est un terme dérivé du nom du mathématicien Muhammad ibn Musa al-Khwarizmi (Bagdad, 783-850) qui a notamment travaillé sur la théorie du système décimal (il est l'auteur d'un précis sur l'Al-Jabr qui, à l'époque, désignait la théorie du calcul, à destination des architectes, astronomes, etc.) et sur les techniques de résolution d'équations du 1er et 2ème degré (*Abrégé du calcul par la restauration et la comparaison*, publié en 825).

La notion d'algorithme est cependant plus ancienne : Euclide (3e siècle av. JC, pgcd, division entière), Babyloniens (1800 av. JC, résolution de certaines équations).

Lors de la conception d'un algorithme, on doit être attentif aux points suivants :

[1] Nous utiliserons le terme d'opération en algorithmique, er réserverons le terme d'instruction pour désigner leur équivalent en programmation.

[2] On s'autorisera fréquemment cet abus de langage. Il s'agit bien évidemment ici de l'exécution d'un programme implantant l'algorithme considéré.

- Adopter une méthodologie de conception : l'analyse descendante consiste à bien penser l'architecture d'un algorithme. On décompose le problème, par affinements successifs, en sous-problèmes jusqu'à obtenir des problèmes simples à résoudre ou dont la solution est connue. On obtient ainsi un schéma général de découpage du problème. On résout les sous-problèmes et, en composant ces différentes solutions, on obtient une solution du problème général.
- Utiliser la modularité : spécification claire des modules construits, réutilisation de modules existants, en évitant les modules trop spécifiques, afin de garantir un bon niveau de réutilisabilité (cet aspect se situe cependant au-delà du contenu du programme de la classe de seconde).
- Être attentif à la lisibilité, la « compréhensibilité » : en soignant en particulier la mise en page, la qualité de la présentation, en plaçant des commentaires pertinents, et en choisissant des identificateurs parlants.
- Se soucier du coût de l'algorithme : notion de complexité en temps (nombre d'opérations nécessaires à la résolution d'un problème de taille donnée), de complexité en espace (taille mémoire nécessaire à la résolution d'un problème de taille donnée).
- Ne pas chercher à « réinventer la roue » : cela nécessite une bonne culture algorithmique (problèmes et solutions standards, techniques usuelles de résolution, etc.).

Le schéma suivant permet de situer la place de l'algorithmique dans le cadre général du développement (traditionnel, maintenant dépassé) d'une application informatique :

Lors de la conception d'un algorithme, on doit avoir à l'esprit trois préoccupations essentielles :

- La *correction* de l'algorithme.

  Il s'agit ici de s'assurer (il est souvent possible d'en donner une « preuve mathématique ») que les résultats produits par l'algorithme sont corrects (l'algorithme réalise bien ce pour quoi il a été conçu) et ce, quelles que soient les (valeurs des) données de départ.

- La *terminaison* de l'algorithme.

  Tout algorithme doit effectuer ce pour quoi il a été conçu en un temps fini. Il est donc nécessaire de s'assurer que l'algorithme termine toujours et, là encore, quelles que soient les données de départ.

- La *complexité* de l'algorithme.

  La complexité *en espace* fait référence à l'espace mémoire nécessaire à l'exécution d'un algorithme (directement lié à l'espace mémoire nécessaire pour stocker les différentes données) et la complexité *en temps* au temps nécessaire à celle-ci.

  En réalité, la complexité permet de mesurer l'évolution, de l'espace ou du temps nécessaires, en fonction de l'évolution de la taille des données de départ. Ainsi, un algorithme *linéaire* en temps est un algorithme dont le temps d'exécution dépend linéairement de la taille des données (pour traiter 10 fois plus de données, il faut 10 fois plus de temps).

On se doit de garder à l'esprit la distinction indispensable entre *algorithme* et *programme*. L'algorithme décrit une méthode de résolution d'un problème donné et possède un caractère *universel*, qui permet de l'implanter dans la plupart (sinon tous) des langages de programmation. Un programme n'est alors que la traduction de cet algorithme dans un certain langage et n'a de signification que pour un compilateur, ou un interpréteur, du langage en question.

1.2. Structure d'un algorithme

Il n'existe pas de langage universel dédié à l'écriture des algorithmes. En règle générale, on utilisera donc un langage « communément accepté » permettant de décrire les opérations de base et les structures de contrôle (qui précisent l'ordre dans lequel doivent s'enchaîner les opérations) nécessaires à l'expression des algorithmes, et ce de façon *rigoureuse*.

Ce langage possèdera donc une syntaxe et une sémantique précises, permettant à chacun de produire des algorithmes lisibles et compréhensibles par tous ceux qui utiliseront le même langage algorithmique. Bien que ce langage ne soit destiné à être lu que par des être humains, il est me semble-t-il important de

contraindre ses utilisateurs à utiliser une syntaxe précise (ce sera de toute façon nécessaire lorsque l'on passera au stade de la programmation, et il n'est jamais trop tard pour prendre de bonnes habitudes).

Pour chaque élément composant un algorithme, nous proposerons donc une syntaxe formelle et une sémantique précise.

La présentation générale d'un algorithme sera la suivante :

```
Algorithme monPremierAlgorithme

# ceci est mon premier algorithme
# il a pour but d'illustrer la syntaxe générale d'un algorithme

début
...
fin
```

- Les termes Algorithme, début et fin sont des *mots-clés* de notre langage algorithmique (mots spéciaux ayant une sémantique particulière). Le *corps* de l'algorithme sera placé entre les mots-clés début et fin.
- Le terme monPremierAlgorithme est un *identificateur*, terme permettant de désigner de façon unique (identifier donc) l'algorithme que nous écrivons. Il est très fortement recommandé (sinon indispensable) de choisir des identificateurs *parlants*, dont la lecture doit suffire pour comprendre le sens et le rôle de l'objet désigné (même lorsqu'on le revoit six mois plus tard… où lorsqu'il a été choisi par quelqu'un d'autre). Naturellement, les identificateurs que nous choisissons ne peuvent être des mots-clés utilisés par notre langage algorithmique qui, eux, ont une sémantique spécifique et sont donc *réservés*. L'usage consistant à utiliser des identificateurs commençant par une lettre minuscule, et à insérer des majuscules à partir du second mot composant l'identificateur, est une recommandation que l'on retrouve dans plusieurs langages de programmation (monPremierAlgorithme en est un bon exemple).
- Les lignes débutant par un « # » sont des lignes de commentaire qui permettent ici de préciser le but de l'algorithme (il s'agit à ce niveau d'une *spécification* de l'algorithme : on décrit ce qu'il fait, sans dire encore *comment* il le fait).

La syntaxe précise et complète du langage algorithmique que nous utilisons sera décrite de façon formelle au chapitre suivant.

Le corps de l'algorithme sera composé *d'opérations élémentaires* (affectation, lecture ou affichage de valeur) et de *structures de contrôle* qui permettent de préciser la façon dont s'enchaînent ces différentes opérations.

Nous allons maintenant décrire ces différents éléments.

1.3. La notion de variable, l'affectation

Un algorithme agit sur des données concrètes dans le but d'obtenir un résultat. Pour cela, il manipule un certain nombre *d'objets* plus ou moins complexes (nous dirons *structurés*).

Exemple 1. Division entière par soustractions successives.

Le problème consiste à déterminer q et r, quotient et reste de la division entière de a par b. Sur un exemple (a=25, b=6) le principe intuitif est le suivant :

```
25 - 6 = 19        q = 1
19 - 6 = 13        q = 2
13 - 6 = 7         q = 3
7 - 6 = 1          q = 4       résultat : q = 4 et r = 1.
```

Ici, on a utilisé des objets à valeur entière : a et b pour les données de départ, q et r pour les données résultats, ainsi que certaines données (non nommées ici) pour les calculs intermédiaires.

Un objet sera caractérisé par :

- un *identificateur*, c'est-à-dire un nom utilisé pour le désigner (rappelons que ce nom devra être « parlant » et distinct des mots-clés de notre langage algorithmique).
- un *type*, correspondant à la nature de l'objet (entier naturel, entier relatif ou caractère par exemple). Le type détermine en fait l'ensemble des valeurs possibles de l'objet, et par conséquence l'espace mémoire nécessaire à leur représentation en machine, ainsi que les opérations (appelées *primitives*) que l'on peut lui appliquer.
- une *valeur* (ou contenu de l'objet). Cette valeur peut varier au cours de l'algorithme ou d'une exécution à l'autre (l'objet est alors une *variable*), ou être défini une fois pour toutes (on parle alors de *constante*).

La section 1.12 présente les principaux types de base de notre langage, ainsi que les opérations associées. On est parfois amené à manipuler des objets dont la structure est plus complexe (une liste, un annuaire, un graphe, …). Ces objets peuvent être construits à l'aide de ce que l'on appelle des *constructeurs de types* (voir par exemple la section 1.10 traitant des listes).

Les objets manipulés par un algorithme doivent être clairement définis : identificateur, type, et valeur pour les constantes. Ces déclarations se placent avant le corps de l'algorithme :

```
Algorithme monDeuxièmeAlgorithme

# commentaire intelligent

constantes     pi = 3.14
variablesa, b : entiers naturels
                car1 : caractère
                r : réel
                adresse : chaîne de caractères

début
...
fin
```

Remarque (la notion de littéral). Lorsque nous écrivons 3.14, 3.14 est un objet, de type réel, dont la valeur (3.14 !) n'est pas modifiable. C'est donc une constante, mais une constante qui n'a pas de nom (i.e. d'identificateur), et que l'on désigne simplement par sa valeur (3.14) ; c'est ce que l'on appelle un *littéral* (de type réel ici). Autres exemples : 28 est un littéral de type entier (naturel ou relatif), 'c' un littéral de type caractère, "Bonjour" un littéral de type chaîne de caractères.

L'affectation est une opération élémentaire qui permet de donner une valeur à une variable. La syntaxe générale de cette opération est la suivante :

<identificateur_variable> ← <expression>

La sémantique intuitive de cette opération est la suivante : l'expression est évaluée (on calcule sa valeur) et la valeur ainsi obtenue est affectée à (rangée dans) la variable. L'ancienne valeur de cette variable est perdue (on dit que la nouvelle valeur *écrase* l'ancienne valeur).

Naturellement, la variable et la valeur de l'expression doivent être du même type. Voici quelques exemples d'affectation (basés sur les déclarations de variables précédentes) :

```
a ← 8                          # a prend la valeur 8
b ← 15                         # b prend la valeur 15
a ← 2 * b + a                  # a prend la valeur 38
b ← a – b + 2 * ( a – 1 )      # b prend la valeur 97
```

Notons au passage l'utilisation des parenthèses dans les expressions, qui permettent de lever toute ambiguïté. Les règles de priorité entre opérateurs sont celles que l'on utilise habituellement dans l'écriture mathématique. Ainsi, l'expression « 2 * b + a » est bien comprise comme étant le double de b

auquel on rajoute a. L'expression « 2 * (b + a) », quant à elle, nécessite la présence de parenthèse pour être correctement interprétée.

Le fait que l'ancienne valeur d'une variable soit écrasée par la nouvelle lors d'une affectation conduit nécessairement à l'utilisation d'une troisième variable lorsque l'on souhaite *échanger* les valeurs de deux variables :

```
Algorithme échangeDeuxValeurs

# cet algorithme permet d'échanger les valeurs de deux
# variables a et b

variablesa, b, temporaire : entiers naturels

début

   ...
         # échange des valeurs de a et b
   temporaire ← a              # temporaire « mémorise » la valeur de a
   a ← b                       # a reçoit la valeur de b
   b ← temporaire              # b reçoit la valeur de a mémorisée dans
                                     # temporaire
   ...
fin
```

Par convention, au début d'un algorithme, les valeurs des variables sont *indéfinies* (d'une certaine façon, elles contiennent « n'importe quoi »). Il est ainsi souvent nécessaire, en début d'algorithme, de donner des valeurs initiales à un certain nombre de variables. On parle d'*initialisation* pour désigner ces affectations particulières.

1.4. Opérations d'entrée-sortie

L'opération de lecture (ou entrée) de valeur permet d'affecter à une variable, en cours d'exécution, une valeur que l'utilisateur entrera au clavier. Au niveau algorithmique, on supposera toujours que l'utilisateur entre des valeurs *acceptables*, c'est-à-dire respectant la contrainte définie par le type de la variable. Les différents contrôles à appliquer aux valeurs saisies sont du domaine de la programmation. Elles surchargeraient inutilement les algorithmes, au détriment du « cœur » de ceux-ci.

À l'inverse, l'opération d'affichage d'une valeur permet d'afficher à l'écran la valeur d'une variable ou, plus généralement, d'une expression (dans ce cas, l'expression est dans un premier temps évaluée, puis sa valeur est affichée à l'écran).

Nous utiliserons pour ces opérations la syntaxe suivante :

```
Entrer ( <liste_identificateurs_variable> )
Afficher ( <liste_expressions> )
```

Une <liste_identificateurs_variable> est simplement une liste d'identificateurs séparés par des virgules (e.g. Entrer (a, b, c)). De même, une <liste_expressions> désigne une liste d'expressions séparées par des virgules (e.g. Afficher ("Le résultat est : ", somme)).

Exemple 2. Nous sommes maintenant en mesure de proposer notre premier exemple complet d'algorithme :

```
Algorithme calculSommeDeuxValeurs

# cet algorithme calcule et affiche la somme de deux valeurs entrées
# par l'utilisateur au clavier

variablesv1, v2, somme : entiers naturels

début
```

```
        # lecture des deux valeurs
    Entrer ( v1, v2 )

        # calcul de la somme
    somme ← v1 + v2

        # affichage de la somme
    Afficher (somme)

fin
```

Cet algorithme utilise trois variables, v1, v2 et somme. Il demande à l'utilisateur de donner deux valeurs entières (rangées dans v1 et v2), en calcule la somme (rangée dans somme) et affiche celle-ci.

Nous avons ici volontairement fait apparaître les trois parties essentielles d'un algorithme : acquisition des données, calcul du résultat, affichage du résultat. Dans le cas de cet exemple simple, il est naturellement possible de proposer une version plus « compacte » :

Exemple 3.

```
Algorithme calculSommeDeuxValeursVersion2

# cet algorithme calcule et affiche la somme de deux valeurs entrées
# par l'utilisateur au clavier

variablesv1, v2 : entiers naturels

début

        # lecture des deux valeurs
    Entrer ( v1, v2 )

        # affichage de la somme
    Afficher ( v1 + v2 )

fin
```

1.5. Initialisation de variables

Par convention, au début d'un algorithme, les valeurs des variables sont *indéfinies* (d'une certaine façon, elles contiennent « n'importe quoi »). Il est ainsi souvent nécessaire, en début d'algorithme, de donner des valeurs initiales à un certain nombre de variables.

Cela peut être fait par des opérations d'affectation ou de lecture de valeur. On parle *d'initialisation* pour désigner ces opérations.

1.6. Enchaînement séquentiel

De façon tout à fait naturelle, les opérations écrites à la suite les unes des autres s'exécutent *séquentiellement*. Il s'agit en fait de la première *structure de contrôle* (permettant de contrôler l'ordre dans lequel s'effectuent les opérations) qui, du fait de son aspect « naturel », ne nécessite pas de notation particulière (on se contente donc d'écrire les opérations à la suite les unes des autres…).

Nous allons maintenant présenter les autres structures de contrôle nécessaires à la construction des algorithmes.

1.7. Structures conditionnelles

Cette structure permet d'effectuer telle ou telle séquence d'opérations selon la valeur d'une condition. Une condition est une expression *logique* (on dit également *booléenne*), dont la valeur est *vrai* ou *faux*.

Voici quelques exemples d'expressions logiques :

```
a < b
( a + 2 < 2 * c + 1 ) ou ( b = 0 )
( a > 0 ) et ( a ≤ 9 )
non ( ( a > 0 ) et ( a ≤ 9 ) )
( a ≤ 0 ) ou ( a > 9 )
```

Ces expressions sont donc construites à l'aide d'opérateurs de comparaison (qui retournent une valeur logique) et des opérateurs logiques et, ou, et non (qui effectuent des opérations sur des valeurs logiques). Remarquons ici que la 4ème expression est la *négation* de la 3ème (si l'une est vraie l'autre est fausse, et réciproquement) et que les 4ème et 5ème expressions sont équivalentes (les lois dites de *De Morgan* expriment le fait que « la négation d'un et est le ou des négations » et que « la négation d'un ou est le et des négations »…).

1.7.1. Alternative simple

L'alternative simple permet d'exécuter une parmi deux séquences d'opérations selon que la valeur d'une condition est vraie ou fausse.

La syntaxe générale de cette structure est la suivante :

```
<alternative_simple>    ::=   si ( <expression_logique> )
                              alors <bloc_alors>
                              [ sinon <bloc_sinon> ]
```

Les crochets entourant la partie sinon signifient que celle-ci est *facultative*. Ainsi, le « sinon rien » se matérialise simplement par l'absence de partie sinon.

Les éléments <bloc_alors> et <bloc_sinon> doivent être des séquences d'opérations parfaitement délimitées. On trouve dans la pratique plusieurs façons de délimiter ces blocs (rappelons que nous ne disposons pas de langage algorithmique universel…).

En voici deux exemples :

```
si ( a < b )
alors   début
        c ← b – a
        afficher (c)
        fin
sinon   début
        c ← 0
        afficher (a)
        afficher (b)
        fin
```

```
si ( a < b )
alors   c ← b – a
        afficher (c)
sinon   c ← 0
        afficher (a)
        afficher (b)
fin_si
```

Dans le premier exemple, on utilise des *délimiteurs* de bloc (début et fin). Ces délimiteurs sont cependant considérés comme facultatifs lorsque le bloc concerné n'est composé que d'une seule opération.

Dans le second exemple, le bloc de la partie alors se termine lorsque le sinon apparaît et le bloc de la partie sinon (ou le bloc de la partie alors en cas d'absence de la partie sinon) se termine lorsque le délimiteur fin_si apparaît. Nous utiliserons plutôt cette seconde méthode, plus synthétique.

Remarquons également *l'indentation* (décalage en début de ligne) qui permet de mettre en valeur la structure de l'algorithme. Attention, l'indentation ne supplante pas la syntaxe ! Il ne suffit pas de décaler certaines lignes pour qu'elles constituent un bloc… Il s'agit simplement d'une aide « visuelle » qui permet de repérer plus rapidement les blocs constitutifs d'un algorithme.

Exemple 4. Voici un algorithme permettant d'afficher le minimum de deux valeurs lues au clavier :

```
Algorithme calculMinimum

# cet algorithme affiche le minimum de deux valeurs entrées au clavier
```

```
variablesv1, v2 : entiers naturels
début
        # lecture des deux valeurs
   Entrer ( v1, v2 )
        # affichage de la valeur minimale
   si ( v1 < v2 )
   alors  Afficher ( v1 )
   sinon Afficher ( v2 )
   fin_si
fin
```

1.7.2. Structure à choix multiple

La structure à choix multiple n'est qu'un « raccourci d'écriture » qui a l'avantage de rendre plus lisible les *imbrications* de structures alternatives. Ainsi, la séquence :

```
si ( a = 1 )
alors     Afficher ( "jonquille" )
sinon     si ( a = 2 )
          alors  Afficher ( "cyclamen" )
          sinon si ( a = 3 )
                    alors     Afficher ( "dahlia" )
                    sinon     Afficher ( "pas de fleur" )
                    fin_si
          fin_si
fin_si
```

est beaucoup plus lisible (et donc compréhensible) sous la forme suivante :

```
selon que
   a = 1 : Afficher ( "jonquille" )
   a = 2 : Afficher ( "cyclamen" )
   a = 3 : Afficher ( "dahlia" )
   sinon : Afficher ( "pas de fleur" )
fin_selon
```

Le fonctionnement de cette structure est équivalent au fonctionnement des structures si imbriquées :

- la 1ère condition est évaluée, si elle est vraie on exécute les opérations associées et on quitte la structure (on poursuit derrière le fin_selon), sinon on passe à la condition suivante ;
- la 2ème condition est évaluée, si elle est vraie on exécute les opérations associées et on quitte la structure, sinon on passe à la condition suivante ;
- on continue de façon identique ; si on atteint la partie sinon (facultative), on exécute les opérations associées.

Cette structure ne se retrouve pas sous cette forme dans tous les langages de programmation. Ceci n'est pas du tout gênant ! Le langage algorithmique se doit d'être universel et compréhensible. C'est le rôle du programmeur de traduire cette structure à l'aide des structures à sa disposition dans le langage de programmation considéré. La structure alternative existant dans tous les langages impératifs, au pire, il pourra toujours utiliser la traduction basée sur les structures si-alors-sinon imbriquées…

1.8. Structures répétitives

Les structures répétitives permettent d'exécuter plusieurs fois un bloc d'opérations, tant qu'une condition (de *continuation*) est satisfaite, jusqu'à ce qu'une condition (de *terminaison*) soit satisfaite ou en faisant varier automatiquement une *variable de boucle*.

Ce sont ces structures qui, par leur nature, peuvent engendrer des algorithmes qui ne s'arrêtent jamais (on dit qu'ils *bouclent* indéfiniment). Nous devrons donc être attentifs au problème de la *terminaison* de l'algorithme dès que nous utiliserons ces structures.

1.8.1. Tant que faire

Cette première structure permet de répéter un bloc d'opérations tant qu'une condition de continuation est satisfaite (c'est-à-dire retourne la valeur *vrai* lorsqu'elle est évaluée).

La syntaxe générale de cette structure est la suivante :

```
tantque ( <condition> ) faire
    <bloc_opérations>
fin_tantque
```

La condition de continuation <condition> est une expression logique qui, lorsqu'elle est évaluée, retourne donc l'une des valeurs vrai ou faux.

Cette structure fonctionne de la façon suivante :

- La condition de continuation est évaluée. Si sa valeur est faux, le bloc d'opérations n'est pas exécuté, et l'exécution se poursuit à la suite du fin_tantque. Si sa valeur est vrai, le bloc d'opérations est exécuté *intégralement* (c'est-à-dire que l'on ne s'arrête pas en cours de route, même si la condition de continuation devient fausse).
- À la fin de l'exécution du bloc, on « remonte » pour évaluer à nouveau la condition de continuation, selon la règle précédente.

Ainsi, cette structure de contrôle peut *éventuellement* conduire à une situation dans laquelle le bloc d'opérations n'est jamais exécuté (lorsque la condition de continuation est évaluée à faux dès le départ). C'est une caractéristique essentielle de cette structure et c'est cette particularité qui nous conduira à choisir (ou non) cette structure plutôt que la structure répéter jusqu'à (section suivante).

Exemple 5. L'algorithme suivant permet de calculer le reste de la division entière d'un entier naturel a par un entier naturel b :

```
Algorithme resteDivisionEntière

# cet algorithme calcule le reste de la division entière
# d'un entier naturel a par un entier naturel b

variablesa, b, reste : entiers naturels

début

        # entrée des données
    Entrer ( a, b )

        # initialisation du reste
    reste ← a

        # boucle de calcul du reste
    tantque ( reste ≥ b ) faire
        reste ← reste – b
    fin_tantque

        # affichage du résultat
    Afficher ( reste )
```

```
fin
```

Remarquons dans cet exemple que si les valeurs entrées pour a et b sont telles que a est strictement inférieur à b alors le corps de la boucle tantque n'est pas exécuté (et le reste est donc égal à a).

Quant à la terminaison de cet algorithme, que se passe-t-il si l'utilisateur entre la valeur 0 pour l'entier naturel b ? Le programme boucle indéfiniment, car l'opération reste ← reste – b ne modifie plus la valeur de reste qui, ainsi, ne décroit jamais. La condition de continuation, reste ≥ b, sera donc toujours satisfaite et le corps de boucle sera indéfiniment répété.

La structure suivante va nous permettre de remédier à cette anomalie.

1.8.2. Répéter jusqu'à

Cette structure permet de répéter un bloc d'opérations jusqu'à ce qu'une condition d'arrêt soit satisfaite (c'est-à-dire retourne la valeur *vrai* lorsqu'elle est évaluée).

La syntaxe générale de cette structure est la suivante :

```
répéter
    <bloc_opérations>
jusqu'à ( <condition> )
```

La condition de continuation <condition> est là aussi une expression logique.

Cette structure fonctionne de la façon suivante :

- Le bloc d'opérations est exécuté *intégralement* (c'est-à-dire que l'on ne s'arrête pas en cours de route, même si la condition de terminaison devient vraie).
- La condition de terminaison est évaluée. Si sa valeur est faux, le bloc d'opérations est exécuté à nouveau, comme décrit précédemment. Si sa valeur est vrai, la répétition s'arrête et l'exécution se poursuit à la suite du jusqu'à (<condition>).

Ainsi, cette structure de contrôle entraîne nécessairement l'exécution du bloc d'opérations, au moins une fois (une seule fois lorsque la condition d'arrêt est évaluée à vrai à la fin du premier passage). C'est une caractéristique essentielle de cette structure et c'est cette particularité qui nous conduira à choisir (ou non) cette structure plutôt que la structure tantque faire.

Cette structure permet notamment de s'assurer qu'une valeur entrée par l'utilisateur satisfait une condition particulière. Dans l'algorithme défaillant de la section précédente, nous devions nous assurer que l'utilisateur entrait une valeur non nulle pour l'entier b. On peut s'en assurer en écrivant :

```
répéter Entrer ( b ) jusqu'à ( b ≠ 0 )
```

Ainsi, si l'utilisateur entre une valeur insatisfaisante, il lui sera demandé d'entrer une nouvelle valeur, et ce jusqu'à ce qu'il entre une valeur correcte.

Exemple 6. L'algorithme suivant permet de calculer et afficher la somme de deux entiers naturels lus au clavier et ce, de façon répétitive jusqu'à ce que l'utilisateur souhaite arrêter son exécution.

```
Algorithme sommeDeuxEntiersAVolonté

# cet algorithme permet de calculer et afficher la somme de deux entiers
# naturels lus au clavier et ce, de façon répétitive jusqu'à ce que
# l'utilisateur souhaite arrêter son exécution

variablesa, b, somme : entiers naturels
                réponse : caractère

début

   répéter
```

```
            # lecture des données
        Entrer ( a, b )

            # calcul et affichage de la somme
        somme ← a + b
        Afficher ( somme )

            # l'utilisateur souhaite-t-il continuer ?
        Afficher ( "On continue (o/n) ?" )
        Entrer ( réponse )

    jusqu'à ( réponse = 'n' )

fin
```

1.8.3. Boucle pour

La structure « pour » permet de répéter un bloc d'opérations un nombre de fois connu au moment d'entrer dans la boucle, et ce en faisant varier automatiquement la valeur d'une variable (dite *variable de boucle*).

La syntaxe de cette structure est la suivante :

```
pour <identificateur_variable> de <valeur_début> à <valeur_fin> faire
        <bloc_opérations>
fin_pour
```

<identificateur_variable> est l'identificateur d'une variable de type *entier* (naturel ou relatif). Les deux valeurs, <valeur_début> et <valeur_fin>, sont deux expressions entières (c'est-à-dire dont l'évaluation retourne une valeur entière).

La valeur de <identificateur_variable> est *automatiquement* initialisée à <valeur_début> avant la première exécution du bloc d'opérations. À la fin de chaque exécution de ce bloc, la valeur de <identificateur_variable> est *automatiquement* incrémentée de 1 (sa valeur est augmentée de 1). Lorsque la valeur de <identificateur_variable> dépasse <valeur_fin> la répétition s'arrête.

Attention, le corps de boucle n'est jamais exécuté si <valeur_début> est strictement supérieure à <valeur_fin> !...

Exemple 7. L'algorithme suivant affiche la table de multiplication par un entier naturel n, où la valeur de n est choisie par l'utilisateur.

```
Algorithme afficheTableDeMultiplication

# cet algorithme affiche la table de multiplication par un entier naturel
# n, où la valeur de n est choisie par l'utilisateur.

variablesn, i, produit : entiers naturels

début

        # lecture de la donnée
    Entrer ( n )

        # calcul et affichage de la table
    pour i de 0 à 10 faire
        produit ← n * i
        Afficher ( n, '*', i, '=', produit )
    fin_pour

fin
```

Notons qu'il est également possible de définir un *pas* (valeur d'incrément) différent de 1, et même éventuellement négatif (dans ce cas, on doit avoir <valeur_début> supérieure ou égale à <valeur_fin>, sinon le corps de boucle n'est jamais exécuté).

La structure se présente alors de la façon suivante :

```
pour i de 15 à 12 par pas de -1 faire
...                    # i prendra successivement les valeurs 15, 14, 13 et 12
fin_pour
...
pour i de 1 à 10 par pas de 2 faire
...                    # i prendra successivement les valeurs 1, 3, 5, 7 et 9
fin_pour
```

Remarque. La variable de boucle ne doit absolument pas être modifiée dans le bloc d'opérations composant le corps de boucle ! Cette variable est en effet gérée *automatiquement* par la structure pour elle-même et doit donc être considérée comme « réservée ». Dans le cas contraire, il s'agit d'une erreur grossière de conception algorithmique. Il en va de même pour les bornes de l'intervalle à parcourir, <valeur_début> et <valeur_fin>.

1.9. Exécution « manuelle » d'un algorithme

La finalité d'un algorithme est d'être traduit sous la forme d'un programme exécutable sur un ordinateur. Il est donc indispensable d'avoir une idée précise (bien plus qu'une idée en réalité !) de la façon dont va « fonctionner » le programme en question.

Pour cela, il est très formateur « d'exécuter soi-même » (on dit *faire tourner*) l'algorithme que l'on conçoit. Cela permet de mieux comprendre le fonctionnement des différentes opérations et structures et, bien souvent, de découvrir des anomalies dans l'algorithme que l'on a conçu.

Pour exécuter un algorithme, il suffit de conserver une trace des valeurs en cours des différentes variables et d'exécuter une à une les opérations qui composent l'algorithme (en respectant la sémantique des structures de contrôle) en reportant leur éventuel impact sur les valeurs des différentes variables.

Nous allons par exemple faire tourner l'algorithme, vu précédemment, de calcul du reste de la division entière d'un entier naturel a par un entier naturel b. Voici pour mémoire l'algorithme en question (avec contrôle de la saisie de la donnée b) :

```
Algorithme resteDivisionEntière

# cet algorithme calcule le reste de la division entière
# d'un entier naturel a par un entier naturel b non nul

variablesa, b, reste : entiers naturels
début

        # entrée des données, b doit être non nul
    Entrer ( a )
    répéter Entrer ( b ) jusqu'à ( b ≠ 0 )

        # initialisation du reste
    reste ← a

        # boucle de calcul du reste
    tantque ( reste >= b )
    faire  reste ← reste – b
    fin_tantque

        # affichage du résultat
    Afficher ( reste )
```

```
fin
```

Cet algorithme utilise 3 variables, a, b et reste. Nous utiliserons donc un tableau à 3 colonnes pour matérialiser l'évolution des valeurs de ces variables. Nous devons également choisir les deux valeurs qui seront fournies par l'utilisateur au clavier, par exemple 17 pour a et 4 pour b (on appelle *jeu d'essai* les valeurs particulières ainsi choisies).

opération	valeur des variables		
	a	b	reste
Entrer (a)	17		
a >= 0 ? vrai			
Entrer (b)		4	
b > 0 ? vrai			
reste ← a			17
reste >= b ? vrai			
reste ← reste - b			13
reste >= b ? vrai			
reste ← reste - b			9
reste >= b ? vrai			
reste ← reste - b			5
reste >= b ? vrai			
reste ← reste - b			1
reste >= b ? faux			
Afficher (reste)			**1**

Notre algorithme affiche l'entier **1**, qui correspond bien au reste de la division de 17 par 4. Cela ne prouve naturellement pas que notre algorithme est correct mais, s'il avait contenu une anomalie importante, nous l'aurions très probablement détectée.

Il est par ailleurs fortement recommandé de tester plusieurs jeux d'essai, notamment pour les algorithmes ayant des comportements différents selon les situations rencontrées (en présence de structures alternatives par exemple).

1.12. Répertoire des types et opérations de base

Le tableau suivant présente les principaux types de base que nous utiliserons ainsi que les principales opérations utilisables sur ceux-ci.

TYPE	OPÉRATIONS
entier naturel	opérateurs arithmétiques : +, -, *, div, mod (quotient et reste de la division entière), ^ (a^b signifie « a à la puissance b ») opérateurs de comparaison : <, >, =, ≠, ≤, ≥
entier relatif	opérateurs arithmétiques : +, -, *, div, mod (quotient et reste de la division entière), ^ (a^b signifie « a à la puissance b ») opérateurs de comparaison : <, >, =, ≠, ≤, ≥ fonctions mathématiques : abs (n) (valeur absolue)
réel	opérateurs arithmétiques : +, -, *, / opérateurs de comparaison diverses fonctions mathématiques : RacineCarrée(r), Sinus (r), etc.
caractère	opérateurs de comparaison
chaîne de caractères	Concaténation (ch1, ch2, ...) (construit une chaîne en « collant » ch1, ch2, …) Longueur (ch) (nombre de caractères composant la chaîne) ch[i] : désigne le i-ième caractère de la chaîne ch (de type caractère donc) opérateurs de comparaison (basés sur l'ordre lexicographique)
booléen	opérations logiques : et, ou, non, xor (ou exclusif)
liste	L ← [elem1, elem2, ...], définit en extension le contenu de la liste L L[i] : retourne l'élément de rang i (le premier élément a pour rang 0) opérateur de concaténation : + NombreEléments(L) : retourne le nombre d'éléments de la liste L L[i:j] : retourne la sous-liste composée des éléments de rang i à j

Chapitre 2. Corpus d'exercices généraux

Nous déconseillons fortement d'introduire l'algorithmique au travers d'exemples tirés « de la vie courante » (recette, cafetière, etc.)… Ces situations se prêtent généralement assez mal à une expression formelle et ne peuvent donner qu'une idée faussée de la notion d'algorithmique.

2.1. Affectation et opérations d'entrée-sortie

Exercice 1. Lecture d'algorithme

Que fait l'algorithme suivant ?

```
Algorithme mystèreADécouvrir
# c'est à vous de trouver ce que fait cet algorithme...
variablesa, b : entiers naturels
début
        # lecture des données
   Entrer ( a, b )
        # calcul mystère
   a ← a + b
   b ← a – b
   a ← a – b
        # affichage résultat
   Afficher ( a, b )
fin
```

Exercice 2. Décomposition d'un montant en euros

Écrire un algorithme permettant de décomposer un montant entré au clavier en billets de 20, 10, 5 euros et pièces de 2, 1 euros, de façon à minimiser le nombre de billets et de pièces.

Exercice 3. Somme de deux fractions

Écrire un algorithme permettant de calculer le numérateur et le dénominateur d'une somme de deux fractions entières (on ne demande pas de trouver la fraction résultat sous forme irréductible).

2.2. Structures conditionnelles

Exercice 4. Valeur absolue

Écrire un algorithme permettant d'afficher la valeur absolue d'un entier relatif entré au clavier.

Exercice 5. Résolution d'une équation du 1er degré

Écrire un algorithme permettant de résoudre une équation à coefficients réels de la forme $ax + b = 0$ (a et b seront entrés au clavier).

Exercice 6. Résolution d'une équation du 2nd degré

Écrire un algorithme permettant de résoudre une équation à coefficients réels de la forme $ax^2 + bx + c = 0$ (a, b et c seront entrés au clavier). On pourra utiliser la fonction RacineCarrée(x) qui retourne la racine carrée de x.