Master Sciences, Technologies, Santé
Mention Mathématiques, spécialité Enseignement des mathématiques

MHT063

Algorithmique et graphes, thèmes du second degré

ALGORITHMIQUE ET GRAPHES, THEMES DU SECOND DEGRE

Éric SOPENA
Eric.Sopena@labri.fr

SOMMAIRE

Chapitre 1. Notions de base d'algorithmique

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

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 :

- 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

[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.

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
variables   a, 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
```

```
variables   a, 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
variables   v1, 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
variables   v1, 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
variables   v1, 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
variables   a, 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
variables   a, 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.
variables   n, 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
variables   a, 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).

1.10. Les listes

Une *liste* est une collection d'objets de même type (on dit que c'est une structure *homogène*[3]), ordonnée (les objets sont rangés dans l'ordre où ils ont été ajoutés à la liste) et offrant un *accès direct* à ces objets en fonction de leur *rang* (le premier élément de la liste a pour rang 0, le deuxième a pour rang 1, etc.).

Lors de la déclaration d'une variable de type liste, on précise le type des objets qui la composeront :

```
variables   listeEntiers : liste d'entiers naturels
            listeRéels : liste de réels
            listeCaractères : liste de caractères
            listeChaînes : liste de chaînes
```

On peut affecter une collection ordonnée d'objets à une liste en écrivant par exemple :

```
  listeEntiers ← [ 8, 0 ]
  listeRéels ← [ 2.5, 3.0, 8.12, -54.987 ]
  listeCaractères ← [ 'a', 'b', 'c' ]
  listeChaînes ← [ "bonjour", "tout", "le", "monde" ]
```

Pour affecter une liste vide à la liste `L`, nous écrirons `L ← [ ]`.

On peut accéder aux objets contenus dans une liste à partir de leur rang qui est un entier naturel. Ainsi, par rapport à l'exemple précédent, nous aurons :

[3] Certains langages de programmation, Python par exemple, permettent de manipuler des listes *hétérogènes*, composées d'objets quelconques.

`listeEntiers [0]` correspond à l'entier `8`

`listeRéels [3]` correspond au réel `-54.987`

Une liste peut contenir un nombre quelconque d'objets. Le nombre d'objets contenus dans une liste s'obtient à l'aide de la primitive `NombreÉléments`. Ainsi,

`NombreEléments ( listeCaractères )` renvoie l'entier `3`

`listeCaractères [ NombreEléments ( listeCaractères ) – 1 ]` renvoie le caractère `'c'`

Pour afficher le contenu d'une liste, on écrira ainsi par exemple :

```
…
Pour I de 0 à NombreEléments ( listeEntiers ) – 1
   Afficher ( listeEntiers [i] )
fin_pour
…
```

On peut *concaténer* (coller) deux listes d'objets de même type à l'aide du symbole '+'. Ainsi, après l'opération suivante :

`listeEntiers ← listeEntiers + [7, 1, 9] + listeEntiers`

la liste `listeEntiers` aura pour contenu `[ 8, 0, 7, 1, 9, 8, 0 ]`.

Exemple 8. L'algorithme suivant permet de construire une liste d'entiers strictement positifs, à partir d'une suite entrée au clavier et terminée par l'entier 0, puis de l'afficher en sens inverse (du dernier au premier) :

```
Algorithme exempleListe

# cet algorithme construit une liste d'entiers strictement positifs,
# à partir d'une suite entrée au clavier et terminée par l'entier 0,
# puis l'affiche en sens inverse

variables   listeEntiers : liste d'entiers naturels
            n, i : entiers naturels
début

        # initialisation
   listeEntiers ← [ ]

        # boucle de lecture des valeurs
   Entrer ( n )
   tantque ( n ≠ 0 )

             # on rajoute l'entier n en fin de liste
        listeEntiers ← listeEntiers + [ n ]

             # on lit la valeur suivante
        Entrer ( n )
   fin_tantque

        # boucle de parcours à l'envers pour affichage
   pour i de NombreEléments ( listeEntiers ) – 1 à 0 par pas de -1 faire
        Afficher ( listeEntiers [ i ] )
   fin_pour

fin
```

Il est possible d'extraire une *sous-liste* d'une liste `L` donnée, c'est-à-dire une liste composée des éléments de `L` situés entre deux rangs donnés. Ainsi, si le contenu de la liste `L` est `[8, 1, 5, 0, 4]`, nous avons :

`L [ 0 : 2 ]` renvoie la liste `[ 8, 1, 5 ]`

`L [ 1 : 4 ]` renvoie la liste `[ 1, 5, 0, 4 ]`

`L [ 2 : NombreEléments (L) – 1 ]` renvoie la liste `[ 0, 4 ]`

`L [ 3 : 3 ]` renvoie la liste `[ 3 ]`

1.11. Primitives graphiques

La plupart des langages de programmation usuels proposent des instructions (primitives) permettant de « dessiner ». Ces primitives peuvent être très différentes d'un langage à l'autre.

Les dessins seront réalisés dans un plan dont les points sont repérés par leurs coordonnées habituelles. Afin de pouvoir écrire des *algorithmes de dessin*, nous utiliserons les primitives suivantes :

Primitives graphiques	
syntaxe	**rôle**
`DessinerPoint ( X, Y )`	dessine le point de coordonnées (X, Y)
`DessinerSegment ( XA, YA, XB, YB )`	dessine un segment de droite reliant les points de coordonnées (XA, YA) et (XB, YB)
`DessinerCercle ( X, Y, rayon )`	dessine un cercle de rayon "rayon" centré en (X, Y)
`CouleurTrait ( <couleur> )`	définit la couleur du trait ("noir" par défaut, <couleur> est une chaîne de caractères)
`EpaisseurTrait ( <épaisseur> )`	définit l'épaisseur du trait (1 par défaut, <épaisseur> est un entier naturel)

Exemple 9. Voici par exemple un algorithme permettant de dessiner en rouge un rectangle centré en l'origine, dont les hauteur et largeur sont entrées au clavier :

```
Algorithme dessinRectangle
variables   hauteur, largeur : réels
            X1, X2, Y1, Y2 : réels
début
      # lecture des données
  Entrer ( hauteur, largeur )
      # calcul des coordonnées
  X1 ← - largeur / 2
  X2 ← - X1
  Y1 ← - hauteur / 2
  Y2 ← - Y1
      # dessin du rectangle
  CouleurTrait ( "rouge" )
  DessinerSegment ( X1, Y1, X2, Y1 )
  DessinerSegment ( X2, Y1, X2, Y2 )
  DessinerSegment ( X2, Y2, X1, Y2 )
  DessinerSegment ( X1, Y2, X1, Y1 )
fin
```

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…
variables   a, 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 façons 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.

Exercice 7. Minimum de trois nombres

Écrire un algorithme permettant d'afficher le plus petit de trois nombres entrés au clavier.

Exercice 8. Intersection de cercles

Écrire un algorithme permettant de déterminer si deux cercles (donnés par leur rayon et les coordonnées de leur centre) ont une intersection non vide.

Exercice 9. Intersection de rectangles

Écrire un algorithme permettant de déterminer si deux rectangles *horizontaux* (donnés par les coordonnées de leurs coins Nord-Ouest et Sud-Est) ont une intersection non vide.

Exercice 10. Durée d'un vol d'avion avec conversion

Écrire un algorithme permettant de calculer la durée d'un vol d'avion, connaissant l'horaire de départ (heures et minutes) et l'horaire d'arrivée (heures et minutes), en convertissant les horaires en minutes. On suppose que le vol dure moins de 24 heures.

Exercice 11. Durée d'un vol d'avion sans conversion

Écrire un algorithme permettant de calculer la durée d'un vol d'avion, connaissant l'horaire de départ (heures et minutes) et l'horaire d'arrivée (heures et minutes), sans convertir les horaires en minutes. On suppose que le vol dure moins de 24 heures.

Exercice 12. Intersection de deux intervalles d'entiers

Écrire un algorithme permettant de calculer l'intersection de deux intervalles d'entiers [a, b] et [c, d].

Exercice 13. Lendemain d'une date donnée

Écrire un algorithme permettant de calculer le lendemain d'une date donnée de l'année 2010 (en 2010, le mois de février compte 28 jours).

Exercice 14. Calculette sommaire

Écrire un algorithme permettant, à partir de deux entiers relatifs entrés au clavier et d'un opérateur entré au clavier (de type caractère), d'afficher le résultat de l'opération correspondante. On se limitera aux opérations '+', '-', '*' et '/' (division entière).

2.3. Structures répétitives

Exercice 15. Lecture d'algorithme

Que fait l'algorithme suivant ?

```
Algorithme mystèreBoucle1
# c'est à vous de trouver ce que fait cet algorithme…
variables   a, b, c : entiers naturels
début
       # lecture des données
  Entrer ( a, b )
```

```
        # initialisation et calculs
   c ← 0
   tantque ( a ≠ 0 )
   faire    si ( ( a mod 2 ) ≠ 0 )
            alors    c ← c + b
            fin_si
            a ← a div 2
            b ← b * 2
   fin_tantque
        # affichage résultat
   Afficher ( c )
fin
```

Exercice 16. Lecture d'algorithme

Que fait l'algorithme suivant ?

```
Algorithme mystèreBoucle2
# c'est à vous de trouver ce que fait cet algorithme…
variables   a, b, c : entiers naturels
début
        # lecture des données
   Entrer ( a, b )
        # initialisation et calculs
   c ← 1
   tantque ( b ≠ 0 )
   faire    si ( ( b mod 2 ) = 1 )
            alors    c ← c * a
            fin_si
            a ← a * a
            b ← b div 2
   fin_tantque
        # affichage résultat
   Afficher ( c )
fin
```

Exercice 17. Multiplication par additions successives

Écrire un algorithme permettant de calculer le produit de deux entiers naturels entrés au clavier en effectuant des additions successives (a * b = a + a + … + a (b fois)).

Exercice 18. Exponentiation par multiplications successives

Écrire un algorithme permettant de calculer la valeur de a puissance b (a et b sont deux entiers naturels entrés au clavier) en effectuant des multiplications successives (a^b = a * a * … * a (b fois)).

Exercice 19. Calcul de factorielle

Écrire un algorithme permettant de calculer la factorielle d'un entier naturel entré au clavier.

Exercice 20. Somme des entiers de 1 à n

Écrire un algorithme permettant de calculer la somme des entiers naturels compris entre 1 et n.

Exercice 21. Afficher les diviseurs d'un entier

Écrire un algorithme permettant d'afficher les diviseurs d'un entier naturel par ordre croissant.

Exercice 22. Nombres parfaits

Un nombre est parfait s'il est égal à la somme de ses diviseurs stricts (différents de lui-même). Ainsi par exemple, l'entier 6 est parfait car 6 = 1 + 2 + 3. Écrire un algorithme permettant de déterminer si un entier naturel est un nombre parfait.

Exercice 23. Maximum d'une suite d'entiers

Écrire un algorithme permettant de saisir une suite d'entiers naturels terminée par 0 et d'afficher ensuite la valeur maximale de la suite (attention, la suite peut être vide !...).

Par exemple, si l'utilisateur entre la suite `8, 4, 11, 4, 0`, l'algorithme affichera la valeur 11.

Exercice 24. Moyenne d'une suite d'entiers terminée par 0

Écrire un algorithme permettant de saisir une suite d'entiers naturels terminée par 0 et d'afficher ensuite la valeur moyenne de la suite (attention, la suite peut être vide !...)

Exercice 25. Vérifier qu'une suite entrée au clavier est croissante

Écrire un algorithme permettant de vérifier si une suite d'entiers naturels terminée par 0 est ou non croissante.

Exercice 26. Calcul du PGCD et du PPCM

Écrire un algorithme permettant de calculer le PGCD et le PPCM de deux entiers naturels non nuls entrés au clavier.

Exercice 27. Nombre premier

Écrire un algorithme permettant de déterminer si un entier naturel entré au clavier est premier.

Exercice 28. Nombres premiers inférieurs à 100

Écrire un algorithme permettant d'afficher la liste de tous les nombres premiers inférieurs à 100.

Exercice 29. Nombres premiers jumeaux inférieurs à 1000

Deux nombres premiers sont jumeaux si leur différence vaut 2 (par exemple, 5 et 7 sont deux nombres premiers jumeaux). Écrire un algorithme permettant d'afficher tous les couples de nombres premiers jumeaux inférieurs à 1000.

Exercice 30. Calcul du $n^{ième}$ nombre d'une suite

Écrire un algorithme permettant de calculer la $n^{ième}$ valeur d'une suite de la forme $u_n = a_{un-1} + b$, $u_0 = c$ (a, b et c sont des entiers naturels entrés au clavier).

Exercice 31. Calcul du $n^{ième}$ nombre de Fibonnacci

Écrire un algorithme permettant de calculer le nombre de Fibonacci `F(n)` : `F(0) = 0`, `F(1) = 1`, et `F(n) = F(n-1) + F(n-2)`.

Exercice 32. Nombres à trois chiffres

Écrire un algorithme permettant d'afficher par ordre croissant tous les nombres à 3 chiffres dont la somme des chiffres est multiple de 5.

2.4. Manipulations de listes

Exercice 33. Lecture et affichage d'une liste

Écrire un algorithme permettant de construire une liste d'entiers naturels strictement positifs à partir d'une suite entrée au clavier et terminée par 0, puis de l'afficher une fois construite.

Exercice 34. Retournement d'une liste

Écrire un algorithme permettant de retourner une liste (son premier élément deviendra dernier, son deuxième avant-dernier, etc.) et d'afficher la liste ainsi retournée.

Exercice 35. Nombre d'occurrences d'un élément

Écrire un algorithme permettant de compter le nombre d'occurrences (d'apparitions) d'un élément donné dans une liste.

Exercice 36. La liste est-elle triée ?

Écrire un algorithme permettant de déterminer si la liste obtenue est ou non triée par ordre croissant (au sens large).

Exercice 37. La liste est-elle monotone ?

Écrire un algorithme permettant de déterminer si une liste est ou non triée par ordre croissant ou décroissant au sens large (une telle liste est dite monotone, croissante ou décroissante respectivement).

Exercice 38. Tri par insertion

Écrire un algorithme permettant de construire une liste *triée par ordre croissant* d'entiers naturels strictement positifs à partir d'une suite entrée au clavier et terminée par 0. Ainsi, chaque nouvel élément devra être inséré en bonne position dans la liste en cours de construction.

Exercice 39. Fusion de deux listes triées

Écrire un algorithme permettant, à partir de deux listes triées, de construire « l'union » triée de ces deux listes. À partir des listes [3, 6, 9] et [1, 6, 8, 12, 15], on obtiendra la liste [1, 3, 6, 6, 8, 9, 12, 15]. On supposera que l'utilisateur entre correctement les deux listes triées…

Exercice 40. Suppression des doublons

Écrire un algorithme permettant de supprimer les doublons (éléments déjà présents) dans une liste triée donnée. À partir de la liste [3, 3, 6, 9, 9, 9, 9, 11], on obtiendra la liste [3, 6, 9, 11].

Chapitre 3. Corpus d'exercices liés au programme de la classe de seconde

Nous présentons ici un ensemble de suggestions d'exercices directement liés au programme de mathématiques de la classe de seconde. Ces exercices peuvent servir de source d'inspiration pour la construction de séances spécifiques.

3.1. Fonctions

3.1.1. Images, antécédents

Exercice 41. Calcul d'image

Écrire un algorithme qui calcule l'image d'un nombre `x` par une fonction du type $f(x) = ax^2 + bx + c$ (`a`, `b` et `c` donnés).

Exercice 42. Calcul d'antécédent par une fonction affine

Écrire un algorithme qui calcule, s'il existe, l'antécédent d'un nombre `y` par une fonction affine $f(x) = ax + b$ (`a` et `b` donnés).

Exercice 43. Calcul d'antécédent

Écrire un algorithme qui calcule, s'il existe, l'antécédent d'un nombre `y` par une fonction du type $f(x) = ax^2 + b$ (`a` et `b` donnés).

3.1.2. Étude qualitative de fonctions

Exercice 44. Maximum d'une fonction sur un intervalle donné

Écrire un algorithme qui détermine l'entier maximisant une fonction du type $f(x) = ax^2 + bx + c$ sur un intervalle d'entiers `[n1,n2]` (`a`, `b`, `c`, `n1` et `n2` donnés).

3.1.3. Résolution d'équations

Exercice 45. Résolution d'une équation du premier degré

Écrire un algorithme permettant de résoudre une équation du premier degré, $ax + b = c$ (`a`, `b` et `c` donnés).

Exercice 46. Encadrer une racine par dichotomie

Écrire un algorithme permettant de donner un encadrement de `racine(x)`, `x` donné, en procédant par dichotomie (la précision souhaitée sera également donnée).

3.1.4. Fonctions de référence

Exercice 47. Tracé de courbe

Écrire un algorithme permettant de tracer les courbes des fonctions carré, cube, inverse, …, sur un intervalle de la forme `[a, b]` (`a` et `b` donnés).

3.1.5. Polynômes de degré 2

Exercice 48. Tracé de courbe d'un polynôme de degré 2

Écrire un algorithme permettant de tracer la courbe de la fonction polynôme de degré 2 $f(x) = ax^2 + bx + c$ (a, b et c donnés).

3.1.6. Fonctions homographiques

Exercice 49. Tracé de courbe d'une fonction homographique

Écrire un algorithme permettant de tracer la courbe d'une fonction homographique $f(x) = (ax + b)/(cx + d)$.

3.1.7. Inéquations

Exercice 50. Résolution graphique d'inéquation 1

Écrire un algorithme permettant de résoudre graphiquement l'inéquation $f(x) < k$. (On pourra par exemple tracer la courbe de la fonction f en noir, et en rouge pour la partie satisfaisant l'inéquation…).

Exercice 51. Résolution graphique d'inéquation 2

Écrire un algorithme permettant de résoudre graphiquement l'inéquation $f(x) < g(x)$. (On pourra par exemple tracer les courbes des fonctions f et g en noir, et en rouge pour la partie satisfaisant l'inéquation…).

Exercice 52. Résolution d'inéquation

Écrire un algorithme permettant de ésoudre une inéquation de la forme $ax + b < cx + d$ (a, b, c et d donnés).

3.1.8. Trigonométrie

Exercice 53. Sinus et cosinus dans un triangle rectangle

Soit un triangle rectangle de la forme OAB avec $O = (0,0)$, $A = (a,0)$ et $B = (0,b)$ (a et b donnés). Écrire un algorithme permettant de calculer les sinus et cosinus des angles <OAB> et <OBA>.

3.2. Géométrie

3.2.1. Coordonnées d'un point du plan

Exercice 54. Longueur d'un segment

Écrire un algorithme permettant de calculer la longueur d'un segment donné par les coordonnées de ses deux extrémités.

Exercice 55. Coordonnées du milieu d'un segment

Écrire un algorithme permettant de calculer les coordonnées du milieu d'un segment donné par les coordonnées de ses deux extrémités.

3.2.2. Configurations du plan

Exercice 56. Périmètre et aire d'un rectangle

Écrire un algorithme permettant de calculer le périmètre et l'aire d'un rectangle donné par ses longueur et largeur.

Exercice 57. Périmètre et aire d'autres figures

Écrire un algorithme permettant de calculer le périmètre et l'aire de différentes figures…

Exercice 58. Est-ce un triangle rectangle ?

Écrire un algorithme permettant de vérifier si un triangle, donné par trois points, est un triangle rectangle ou non.

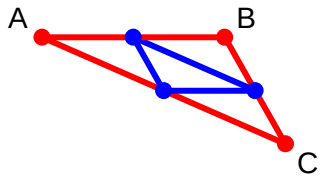
Exercice 59. Est-ce un triangle équilatéral ?

Exercice 60. Est-ce un triangle isocèle ?

Écrire un algorithme permettant de vérifier si un triangle, donné par trois points, est un triangle isocèle ou non.

Exercice 61. Triangle des milieux

Soit un triangle ABC donné par les coordonnées des points A, B et C. Écrire un algorithme permettant de dessiner en rouge le triangle ABC et en bleu le triangle joignant les milieux des 3 segments AB, BC et AC.



Exercice 62. Est-ce un parallélogramme ?

Soient A, B, C et D quatre points donnés ; écrire un algorithme permettant de déterminer si le quadrilatère ABCD est ou non un parallélogramme.

Exercice 63. Est-ce un rectangle ?

Écrire un algorithme permettant de déterminer si quatre points A, B, C et D forment ou non un rectangle.

3.2.3. Droites

Exercice 64. Équation de droite donnée par deux points

Écrire un algorithme permettant de déterminer l'équation d'une droite donnée par deux de ses points.

Exercice 65. Équation de droite perpendiculaire

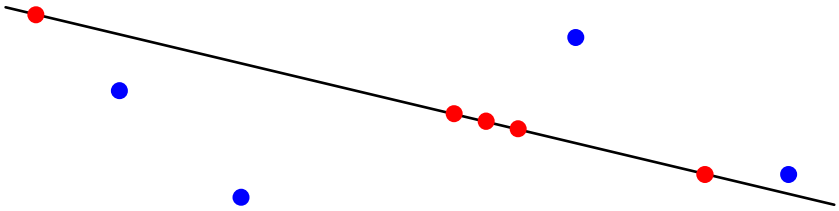
Soient deux points A et B ; écrire un algorithme permettant de déterminer l'équation de la droite passant par A et perpendiculaire au segment AB.

Exercice 66. Équation de droite parallèle

Soient trois points A, B et C ; écrire un algorithme permettant de déterminer l'équation de la droite passant par A et parallèle au segment BC.

Exercice 67. Droite et liste de points

Écrire un algorithme permettant de dessiner la droite d'équation $y = ax + b$ (a et b donnés) et, à partir d'une liste de points donnés, de dessiner en rouge les points appartenant à la droite et en bleu les points n'appartenant pas à la droite.



Exercice 68. Droites parallèles ou sécante ?

Écrire un algorithme permettant de déterminer si deux droites AB et CD données par deux de leurs points sont parallèles ou sécantes.

Exercice 69. Droites perpendiculaires ?

Écrire un algorithme permettant de déterminer si deux droites AB et CD données par deux de leurs points sont ou non perpendiculaires.

Exercice 70. Trois points sont-ils alignés ?

Soient trois points A, B et C ; écrire un algorithme permettant de déterminer si ces trois points sont ou non alignés.

Exercice 71. Intersection de deux droites

Soient quatre points A, B, C et D donnés par leurs coordonnées. Écrire un algorithme permettant de tracer les droites définies par les segments AC et BD, de calculer les coordonnées du point d'intersection de ces deux droites et de l'afficher pour pouvoir vérifier visuellement que c'est correct.

Exercice 72. Théorème des milieux

Soient trois points A, B et C ; écrire un algorithme permettant de dessiner le triangle ABC et le segment IJ, où I est le milieu du segment AB et J le milieu du segment AC, et de calculer les coefficients directeurs des droites passant respectivement par les points I et J et par les points B et C.

3.2.4. Vecteurs

Exercice 73. Dessin de vecteurs

Soient A et B deux points donnés ; écrire un algorithme permettant de dessiner les vecteurs $\overrightarrow{OA}$, $\overrightarrow{OB}$ et $\overrightarrow{OC} = \overrightarrow{OA} + \overrightarrow{OB}$ (O désigne le point origine).

Exercice 74. Coordonnées d'un vecteur

Soient A et B deux points donnés ; écrire un algorithme permettant de déterminer les coordonnées du vecteur $\overrightarrow{AB}$.

Exercice 75. Vecteurs égaux

Soient A, B, C et D quatre points donnés ; écrire un algorithme permettant de déterminer si les vecteurs $\overrightarrow{AB}$ et $\overrightarrow{CD}$ sont ou non égaux.

Exercice 76. Vecteurs colinéaires

Soient A, B et C trois points donnés ; écrire un algorithme permettant de déterminer si les vecteurs $\overrightarrow{AB}$ et $\overrightarrow{BC}$ sont ou non colinéaires.

Exercice 77. Vecteurs colinéaires bis

Soient A, B, C et D quatre points donnés ; écrire un algorithme permettant de déterminer si les vecteurs $\overrightarrow{AB}$ et $\overrightarrow{CD}$ sont ou non colinéaires.

3.2.5. Géométrie dans l'espace

Exercice 78. Calcul de différents volumes…

Écrire un algorithme permettant de calculer différents volumes (cubes, parallélépipèdes, sphères, …).

Exercice 79. Dessin d'un cube

Écrire un algorithme permettant de dessiner un cube en perspective cavalière, pour une longueur d'arête, un angle et un coefficient de fuite donnés.

3.3. Statistiques et probabilités

Exercice 80. Lancer de pièces (1)

Écrire un algorithme permettant de simuler n lancers de pièces, n donné, et d'afficher la fréquence de l'événement « la pièce tombe sur Pile ».

Exercice 81. Lancer de pièces (2)

Écrire un algorithme permettant de répéter n fois, n donné, la simulation de 100 lancers de pièces, et d'afficher la fréquence de l'événement « au moins 60 Piles sur 100 lancers ».

Exercice 82. Lancer de pièces (3)

Écrire un algorithme permettant de simuler n lancers de pièces, n donné, et d'afficher la fréquence de l'événement « deux Piles successifs ».

Exercice 83. Lancer de dés (1)

Écrire un algorithme permettant de simuler n lancers de deux dés, n donné, et d'afficher la fréquence de l'événement « la somme de deux dés est paire ».

Exercice 84. Lancer de dés (2)

Écrire un algorithme permettant de simuler n lancers de deux dés, n donné, et d'afficher la fréquence de l'événement « les sommes de deux lancers consécutifs sont identiques ».

Exercice 85. Boules rouges et noires (1)

Une urne contient R boules rouges et N boules noires, R et N donnés. Écrire un algorithme permettant de simuler k tirages d'une boule avec remise, k donné, et d'afficher la fréquence de l'événement « la boule tirée est rouge (ou, autre exemple, « on a tiré exactement deux boules noires »).

Exercice 86. Boules rouges et noires (2)

Même exercice que le précédent, mais cette fois sans remise (l'algorithme vérifiera que nous avons bien $k \leq R + N$).

3.4. Divers

3.4.1. Intervalles

Exercice 87. Appartenance à un intervalle

Écrire un algorithme permettant de déterminer si un nombre appartient ou non à un intervalle donné.

Exercice 88. Inclusion d'intervalles

Écrire un algorithme permettant de déterminer si un intervalle est ou non inclus dans un autre.

Exercice 89. Intersection d'intervalles

Écrire un algorithme permettant de calculer l'intersection de deux intervalles donnés.

Exercice 90. Réunion d'intervalles

Écrire un algorithme permettant de déterminer si la réunion de deux intervalles donnés est ou non un intervalle.

3.4.2. Approximations de Pi

Exercice 91. Approximation de Pi (1)

Écrire un algorithme permettant de calculer une approximation de π à partir de la formule de Leibniz (on demandera le nombre d'étapes de calcul) :

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots\right)$$

Exercice 92. Approximation de Pi (2)

Écrire un algorithme permettant de calculer une approximation de π à partir de la formule suivante (on demandera le nombre d'étapes de calcul) :

$$\pi = \frac{6}{\sqrt{3}}\left(1 - \frac{1}{3\times 3} + \frac{1}{5\times 3^2} - \frac{1}{7\times 3^3} + \frac{1}{9\times 3^4} + \ldots\right)$$

Exercice 93. Approximation de Pi (3)

Écrire un algorithme permettant de calculer une approximation de π en utilisant le produit de Wallis (on demandera le nombre d'étapes de calcul) :

$$\pi = 2\left(\frac{2}{1}\times\frac{2}{3}\times\frac{4}{3}\times\frac{4}{5}\times\frac{6}{5}\times\frac{6}{7}\times\frac{8}{7}\times\frac{8}{9}\times\ldots\right)$$

Exercice 94. Approximation de Pi (4)

Écrire un algorithme permettant de calculer une approximation de π à partir de la formule suivante (on demandera le nombre d'étapes de calcul) :

$$\pi = 6\sqrt{\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2}\ldots}$$

Chapitre 4. Exécution d'algorithmes avec AlgoBox

4.1. Introduction

AlgoBox est un logiciel libre, multiplateforme et gratuit d'aide à l'élaboration et à l'exécution d'algorithmes, dans l'esprit des nouveaux programmes de mathématiques du lycée. Il a été développé par Pascal Brachet, professeur de mathématiques au lycée Bernard Palissy à Agen.

La version concernée par ce document est la version 0.5 du 6 avril 2010.

Algobox permet de créer de façon interactive (en minimisant les sources potentielles d'erreurs syntaxiques et de structuration) un algorithme et de l'exécuter. Il est également possible d'exécuter un algorithme *pas à pas*, en suivant l'évolution des valeurs des variables, outil très utile en phase de mise au point !...

AlgoBox permet la création d'algorithmes utilisant :

- des déclarations de variables de type nombre, entier ou décimal (type NOMBRE), chaîne de caractères (type CHAÎNE), ou liste de nombres (type LISTE),
- les opérations élémentaires d'affectation, de lecture et d'affichage (de variables ou de messages),
- la structure de contrôle `Si Alors Sinon`,
- les boucles `Pour` et `Tantque`.

Il est ainsi possible de mettre en œuvre l'ensemble des algorithmes que nous avons présentés dans ce document de façon directe, à l'exception de ceux utilisant la boucle `répéter jusqu'à`, non offerte par Algobox. Il sera dans ce cas nécessaire de *simuler* la boucle `répéter jusqu'à` à l'aide de la boucle `tantque` selon le schéma général suivant :

Structure Répéter Jusqu'à	Simulation
répéter <séquence_opérations> jusqu'à (<condition_arrêt>)	<séquence_opérations> tantque (non <condition_arrêt>) faire <séquence_opérations> fin_tantque

En effet, la boucle `répéter jusqu'à` exécute toujours le corps de boucle (<séquence_opérations>) au moins une fois, et s'interrompt lorsque la condition `<condition_arrêt>` retourne la valeur `vrai`. Il est donc nécessaire d'exécuter une fois `<séquence_opérations>` avant la boucle `tantque` dont la condition de continuation sera naturellement la négation de la condition d'arrêt (non `<condition_arrêt>`).

Voici un exemple plus concret (quoique…) de transformation :

Structure Répéter Jusqu'à	Simulation

répéter
  B ← 2 * B + 1
  A ← A - 1
jusqu'à (A = 0)
	B ← 2 * B + 1
	A ← A - 1
	tantque (A ≠ 0)
	faire
	  B ← 2 * B + 1
	  A ← A - 1
	fin_tantque

Notons également qu'Algobox ne permet pas l'affichage d'expressions. Il sera donc nécessaire d'utiliser une variable dans laquelle sera rangée l'expression souhaitée pour pouvoir en afficher sa valeur.

4.2. Installation du logiciel

Le logiciel AlgoBox peut être téléchargé gratuitement sur le site http://www.xm1math.net/algobox/.

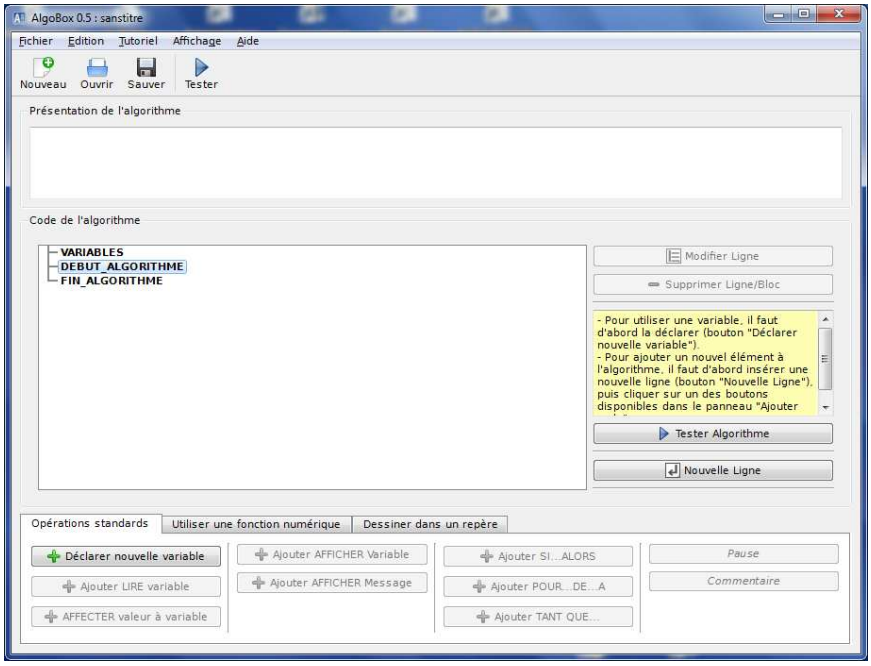
L'installation ne pose aucun problème particulier (il suffit d'exécuter le fichier téléchargé et de se laisser guider) et ne sera donc pas détaillée ici.

Notons cependant que des ressources complémentaires sont également disponibles sur ce site :

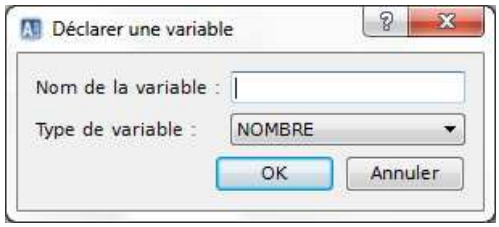
- une animation flash présentant le fonctionnement d'AlgoBox sur un exemple simple,
- un tutoriel d'initiation à l'algorithmique avec AlgoBox,
- des exemples d'algorithmes de tous niveaux réalisés avec AlgoBox.

4.3. Premiers pas

Lorsqu'on lance AlgoBox, la fenêtre principale suivante apparaît :

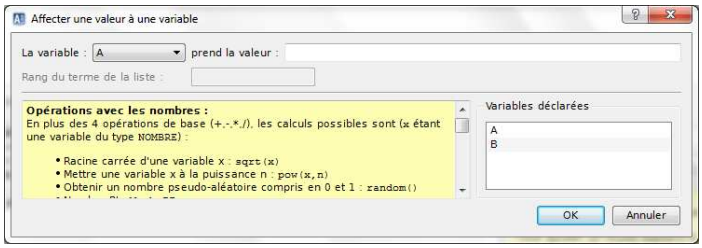


Le bouton `Déclarer nouvelle variable` ouvre la fenêtre ci-contre qui permet de donner un nom à la variable et de définir son type (`NOMBRE`, `CHAÎNE` ou `LISTE`).



L'onglet `Opérations standards` permet d'inclure les opérations de base (`AFFECTER`, `LIRE` ou `AFFICHER`), les structures de contrôle (`SI`, `POUR`, `TANTQUE`), ou encore des lignes de commentaires dans l'algorithme. Pour cela, il faut dans un premier temps insérer une ligne vide dans l'algorithme (bouton `Nouvelle ligne` ou, plus simplement, les touches `Ctrl-Entrée`), puis utiliser le bouton correspondant à l'opération ou à la structure.

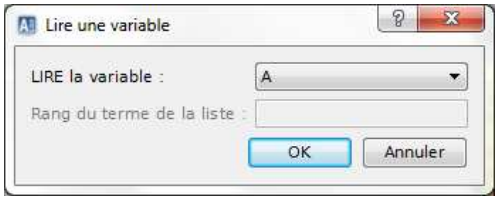
Les fenêtres affichées par AlgoBox dans chacun de ces cas sont les suivantes :



Affectation

On choisit dans la liste la variable à affecter, puis on inscrit la valeur souhaitée dans la zone `prend la valeur`. Cette valeur peut être une constante, une variable ou n'importe quelle expression (exemples fournis).

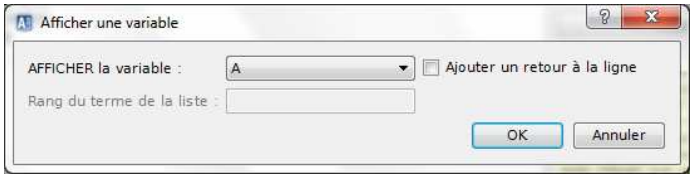
La zone `variables déclarées` permet d'insérer directement un nom de variable par simple-clic.



Lecture

On choisit simplement la variable dans la liste déroulante.

Dans le cas d'une variable de type liste, on précise également le rang dans la liste auquel sera affecté l'élément lu.

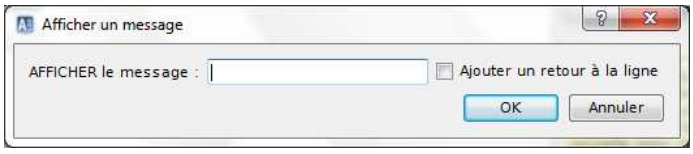


Affichage de la valeur d'une variable

Même principe que pour la lecture d'une variable.

La case à cocher permet de provoquer un changement de ligne après l'affichage de la variable.

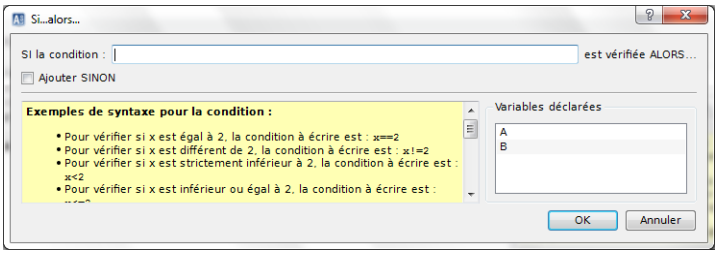
Pour afficher la valeur d'une expression, il est nécessaire de passer par une variable dédiée (on affecte l'expression à la variable, puis on affiche la variable).



Affichage d'un message

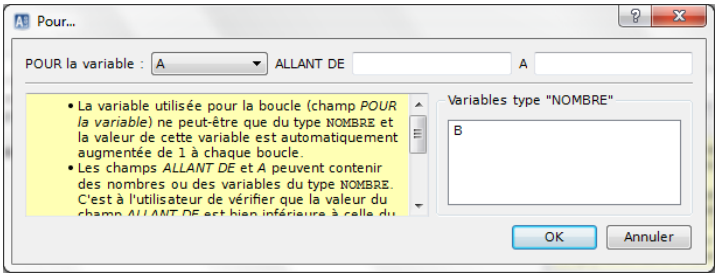
On entre directement le message à afficher.

La case à cocher permet de provoquer un changement de ligne après l'affichage du message.



Structure SI

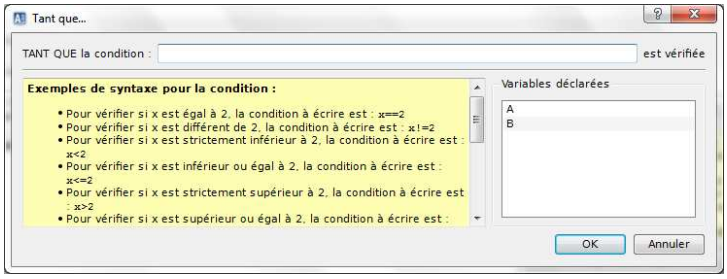
On entre la condition dans la zone prévue (la zone `Variables déclarées` permet d'insérer directement un nom de variable par simple-clic) et on coche la case `Ajouter SINON` si nécessaire.



Boucle Pour

On choisit la variable de boucle et les valeurs de départ et d'arrivée, qui peuvent être des expressions (attention, le pas de parcours vaut toujours 1).

Le nombre de répétitions du corps de boucle est limité à 200 000.



Boucle Tantque

On entre la condition de continuation dans la zone prévue à cet effet…

Le nombre de répétitions du corps de boucle est limité à 200 000.

4.4. Quelques compléments

4.4.1. Le type NOMBRE.

Le point est utilisé comme marque décimale. Ainsi, `3` et `5.124` sont deux valeurs de type `NOMBRE`. En plus des 4 opérations de base (`+`, `-`, `*`, `/`), les fonctions suivantes sont disponibles (`x` étant une variable du type `NOMBRE`) :

- Racine carrée d'une variable `x` : `sqrt(x)`
- Mettre une variable `x` à la puissance `n` : `pow(x, n)`
- Obtenir un nombre pseudo-aléatoire compris en 0 et 1 : `random()`
- Nombre PI : `Math.PI`
- Partie entière d'une variable `x` : `floor(x)`
- Cosinus d'une variable `x` (en radians): `cos(x)`
- Sinus d'une variable `x` (en radians): `sin(x)`
- Tangente d'une variable `x` (en radians): `tan(x)`
- Exponentielle d'une variable `x` : `exp(x)`
- Logarithme népérien d'une variable `x` : `log(x)`
- Valeur absolue d'une variable `x` : `abs(x)`
- Arrondi d'une variable `x` à l'entier le plus proche : `round(x)`
- Reste de la division de la variable `x` par la variable `y` : `x % y`

Attention !... Il ne faut pas rajouter d'espaces entre le nom d'une fonction et la parenthèse ouvrante qui lui est associée. Dans le cas contraire, AlgoBox produit une erreur à l'exécution.

4.4.2. Le type LISTE.

Les listes AlgoBox sont des listes de nombres, dont chaque élément est repéré par son rang au sein de la liste (le premier élément ayant pour rang 1). Ainsi, `L[4]` désigne le 4ème élément de la liste `L`.

Il est possible d'affecter une valeur à plusieurs éléments d'une liste en une seule instruction :

```
L[4] prend la valeur 6 : 8 : 11 : 5
```

Les valeurs 6, 8, 11 et 5 sont respectivement affectés aux éléments L[4], L[5], L[6] et L[7].

Algobox n'offre pas de fonction permettant de connaître le nombre d'éléments d'une liste et il est donc nécessaire de gérer soi-même une variable dédiée. De même, il n'y a pas d'opération prédéfinie de concaténation de listes.

L'exemple suivant illustre la façon de réaliser soi-même une telle concaténation :

```
****************************************
Cet algorithme lit 2 listes au clavier, puis concatène la deuxième à la fin
de la première.
```

```
****************************************

1   VARIABLES
2     L1 EST_DU_TYPE LISTE
3     L2 EST_DU_TYPE LISTE
4     NB1 EST_DU_TYPE NOMBRE
5     NB2 EST_DU_TYPE NOMBRE
6     ELEM EST_DU_TYPE NOMBRE
7     IND EST_DU_TYPE NOMBRE
8   DEBUT_ALGORITHME
9     //lecture liste 1
10    AFFICHER "Entrer les éléments de la première liste, 0 pour terminer"
11    NB1 PREND_LA_VALEUR 0
12    LIRE ELEM
13    TANT_QUE (ELEM != 0) FAIRE
14      DEBUT_TANT_QUE
15      NB1 PREND_LA_VALEUR NB1 + 1
16      L1[NB1] PREND_LA_VALEUR ELEM
17      LIRE ELEM
18      FIN_TANT_QUE
19    AFFICHER "Première liste lue, Nombre d'éléments : "
20    AFFICHER NB1
21    //lecture liste 2
22    AFFICHER "Entrer les éléments de la première liste, 0 pour terminer"
23    NB2 PREND_LA_VALEUR 0
24    LIRE ELEM
25    TANT_QUE (ELEM != 0) FAIRE
26      DEBUT_TANT_QUE
27      NB2 PREND_LA_VALEUR NB2 + 1
28      L2[NB2] PREND_LA_VALEUR ELEM
29      LIRE ELEM
30      FIN_TANT_QUE
31    AFFICHER "Deuxième liste lue, Nombre d'éléments : "
32    AFFICHER NB2
33    //On concatène L2 à la fin de L1
34    POUR IND ALLANT_DE 1 A NB2
35      DEBUT_POUR
36      NB1 PREND_LA_VALEUR NB1 + 1
37      L1[NB1] PREND_LA_VALEUR L2[IND]
38      FIN_POUR
39    //Affichage de la liste L1 résultat
40    POUR IND ALLANT_DE 1 A NB1
41      DEBUT_POUR
42      AFFICHER L1[IND]
43      AFFICHER " - "
44      FIN_POUR
45  FIN_ALGORITHME
```

AlgoBox offre les fonctions de calcul suivantes sur les listes :

- Somme :
  `ALGOBOX_SOMME(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Moyenne :
  `ALGOBOX_MOYENNE(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Variance :
  `ALGOBOX_VARIANCE(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Ecart-type :
  `ALGOBOX_ECART_TYPE(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Médiane :
  `ALGOBOX_MEDIANE(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Premier quartile :
  `ALGOBOX_QUARTILE1(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`

(définition calculatrice : médiane de la sous-série inférieure – la série doit comporter au moins 4 valeurs, sinon une erreur est déclarée)

- Troisième quartile :
  `ALGOBOX_QUARTILE3(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
  (définition calculatrice : médiane de la sous-série supérieure – la série doit comporter au moins 4 valeurs, sinon une erreur est déclarée)
- Premier quartile Bis :
  `ALGOBOX_QUARTILE1_BIS(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
  (autre définition : plus petite valeur telle qu'au moins 25% des données lui soient inférieures – la série doit comporter au moins 4 valeurs, sinon une erreur est déclarée)
- Troisième quartile Bis :
  `ALGOBOX_QUARTILE3_BIS(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
  (autre définition : plus petite valeur telle qu'au moins 75% des données lui soient inférieures – la série doit comporter au moins 4 valeurs, sinon une erreur est déclarée)
- Minimum :
  `ALGOBOX_MINIMUM(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Maximum :
  `ALGOBOX_MAXIMUM(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Rang du minimum :
  `ALGOBOX_POS_MINIMUM(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`
- Rang du maximum :
  `ALGOBOX_POS_MAXIMUM(nom_de_la_liste, rang_premier_terme, rang_dernier_terme)`

4.4.3. Définir et utiliser une fonction numérique

L'onglet `Utiliser une fonction numérique` permet de définir une fonction à variable entière du type `F(x) = <expression>`. Une fois la fonction définie par l'utilisateur, elle s'utilise comme n'importe quelle fonction prédéfinie au sein d'expressions numériques.

4.4.4. Dessin

L'onglet `Dessiner dans un repère` permet d'accéder aux fonctions permettant de dessiner. Pour cela, il est nécessaire dans un premier temps de cocher la case `Utiliser un repère`, puis de définir les paramètres de la zone de dessin : `XMin`, `XMax` et `Graduations X`, puis `YMin`, `YMax` et `Graduations Y`.

On peut ensuite utiliser les deux opérations de dessin proposées, `TRACERPOINT` (on indique les coordonnées du point) et `TRACERSEGMENT` (on indique les coordonnées du point de départ et celles du point d'arrivée). Dans chaque cas, on peut également choisir la couleur de dessin utilisée.

4.5. Quelques exemples illustratifs

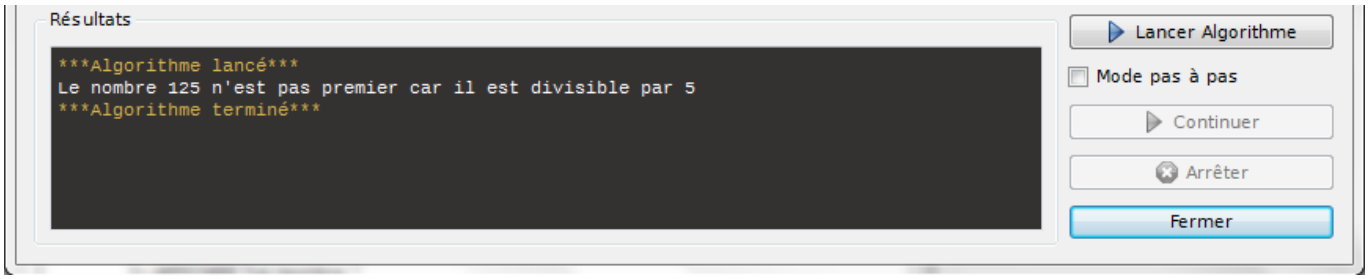
4.5.1. Déterminer si un nombre est ou non premier

Pour déterminer si un entier `N` est premier, on cherche un diviseur de `N` compris entre `2` et `Racine(N)`. L'algorithme exprimé en AlgoBox est le suivant :

```
est_premier  -  01.11.2010

****************************************
Cet algorithme détermine si un entier N lu au clavier est ou non premier.
****************************************

1   VARIABLES
2     N EST_DU_TYPE NOMBRE
3     DIVISEUR EST_DU_TYPE NOMBRE
4     RACINE_N EST_DU_TYPE NOMBRE

5   DEBUT_ALGORITHME
```

```
6     //Lecture de N
7     LIRE N
8     //Initialisations
9     DIVISEUR PREND_LA_VALEUR 2
10    RACINE_N PREND_LA_VALEUR round(sqrt(N))
11    //Boucle de recherche d'un diviseur
12    TANT_QUE (((N % DIVISEUR) != 0) ET (DIVISEUR <= RACINE_N)) FAIRE
13      DEBUT_TANT_QUE
14      DIVISEUR PREND_LA_VALEUR DIVISEUR + 1
15      FIN_TANT_QUE
16    //Affichage de la réponse
17    SI (DIVISEUR <= RACINE_N) ALORS
18      DEBUT_SI
19      //On a trouvé un diviseur, N n'est pas premier
20      AFFICHER "Le nombre "
21      AFFICHER N
22      AFFICHER " n'est pas premier car il est divisible par "
23      AFFICHER DIVISEUR
24      FIN_SI
25      SINON
26        DEBUT_SINON
27        //Aucun diviseur trouvé, le nombre N est premier
28        AFFICHER "Le nombre "
29        AFFICHER N
30        AFFICHER " est premier"
31        FIN_SINON
32  FIN_ALGORITHME
```

La fenêtre suivante montre un exemple d'exécution de cet algorithme :



4.5.2. Dessin d'une étoile

L'algorithme suivant permet de dessiner une étoile dont le nombre de branches est impair. Les paramètres de la zone de dessin ont été définis ainsi :

XMin = -10 ; XMax = 10 ; Graduations X = 2

YMin = -10 ; YMax = 10 ; Graduations Y = 2

```
etoile  -  01.11.2010

*****************************************
Cet algorithme permet de dessiner une étoile ayant un nombre impair de
branches
*****************************************

1   VARIABLES
2     N EST_DU_TYPE NOMBRE
3     angle EST_DU_TYPE NOMBRE
4     angleParcours EST_DU_TYPE NOMBRE
5     I EST_DU_TYPE NOMBRE
6     pointX EST_DU_TYPE NOMBRE
7     pointY EST_DU_TYPE NOMBRE
8     suivantX EST_DU_TYPE NOMBRE
9     rayon EST_DU_TYPE NOMBRE
10    suivantY EST_DU_TYPE NOMBRE
11  DEBUT_ALGORITHME
```

```
12    // lecture des données - N doit être impair
13    AFFICHER "Nombre de pointes de l'étoile ?"
14    LIRE N
15    TANT_QUE (N % 2 == 0) FAIRE
16      DEBUT_TANT_QUE
17      AFFICHER "N doit être impair !!!"
18      LIRE N
19      FIN_TANT_QUE
20    // Initialisations
21    rayon PREND_LA_VALEUR 8
22    angle PREND_LA_VALEUR (2.0 * Math.PI)/N
23    angleParcours PREND_LA_VALEUR 0.0
24    suivantX PREND_LA_VALEUR rayon
25    suivantY PREND_LA_VALEUR 0
26    // Dessin de l'étoile
27    POUR I ALLANT_DE 1 A N
28      DEBUT_POUR
29      pointX PREND_LA_VALEUR suivantX
30      pointY PREND_LA_VALEUR suivantY
31      angleParcours PREND_LA_VALEUR angleParcours + ((N+1) * angle / 2.0)
32      suivantX PREND_LA_VALEUR rayon * cos(angleParcours)
33      suivantY PREND_LA_VALEUR rayon * sin(angleParcours)
34      TRACER_SEGMENT (pointX,pointY)->(suivantX,suivantY)
35      FIN_POUR
36  FIN_ALGORITHME
```

La fenêtre suivante montre un exemple d'exécution de cet algorithme (étoile à 11 branches) :