INITIATION À L'ALGORITHMIQUE EN CLASSE DE SECONDE

Coordonné par Éric Sopena

IREM d'Aquitaine - Groupe Algorithmique

Jean-Yves Boyer, Jérémy Canouet, Ludovic Faure, Pascal Grandjean, Yann-Michaël Guidez, François Petit, Chloé Ubera

INTRODUCTION

Ce document présente et illustre les notions de base de l'algorithmique nécessaires à la mise en œuvre du nouveau programme de mathématiques de la classe de seconde, en vigueur depuis la rentrée 2009. Nous nous sommes volontairement limités dans ce document aux notions présentes dans ce programme. Ainsi par exemple, nous n'évoquons pas la notion de liste, bien que celle-ci nous paraisse indispensable pour la mise en œuvre d'algorithmes plus élaborés et, probablement, plus intéressants…

La suite de ce document est organisée de la façon suivante :

- Le premier chapitre présente l'ensemble de ces notions : variables, opérations d'entrée-sortie, structure conditionnelle, structures répétitives.
- Le deuxième chapitre est une présentation rapide du logiciel libre AlgoBox qui permet d'exécuter des algorithmes simples, afin d'en vérifier la correction.
- Le troisième chapitre est un premier pas dans l'univers de la programmation, à travers l'utilisation du langage Python.
- Enfin, les deux derniers chapitres proposent un corpus d'exercices généraux et d'exercices liés au programme de la classe de seconde, chaque exercice étant accompagné d'un corrigé.

SOMMAIRE

Chapitre 1. Notions de base d'algorithmique 7

1.1. Qu'est-ce qu'un algorithme ? 7
1.2. Structure d'un algorithme 8
1.3. La notion de variable, l'affectation 9
1.4. Opérations d'entrée-sortie 10
1.5. Initialisation de variables 11
1.6. Enchaînement séquentiel 11
1.7. Structure conditionnelle 12
1.8. Structures répétitives 13
1.8.1. La structure Tant que 13
1.8.2. La structure Pour 14
1.9. Exécution « manuelle » d'un algorithme 16
1.10. Primitives graphiques 17
1.11. Répertoire des types et opérations de base 18

Chapitre 2. Exécution d'algorithmes avec AlgoBox 19

2.1. Introduction 19
2.2. Installation du logiciel 19
2.3. Premiers pas 19
2.4. Quelques compléments 22
2.4.1. Le type NOMBRE. 22
2.4.2. Définir et utiliser une fonction numérique 22
2.4.3. Dessin 22
2.5. Quelques exemples illustratifs 22
2.5.1. Déterminer si un nombre est ou non premier 22
2.5.2. Dessin d'une étoile 23

Chapitre 3. Programmation avec Python 25

3.1. Introduction 25
3.2. Python pour la classe de seconde 26
3.2.1. Éléments du langage 26
3.2.2. Types de données élémentaires 27
3.2.3. Affectation et opérations d'entrée-sortie 28
3.2.4. Structures de contrôle 28
3.2.5. Quelques exemples de scripts Python 29
3.2.6. Traduction d'algorithmes en Python - Tableau de synthèse 30
3.2.7. Dessiner en Python 31

Chapitre 4. Corpus d'exercices généraux 35

4.1. Affectation et opérations d'entrée-sortie 35
Exercice 1. Lecture d'algorithme 35
Exercice 2. Décomposition d'un montant en euros 36
Exercice 3. Somme de deux fractions 36

4.2. Structures conditionnelles 37
Exercice 4. Valeur absolue 37
Exercice 5. Résolution d'une équation du 1er degré 37
Exercice 6. Résolution d'une équation du 2nd degré 38
Exercice 7. Minimum de trois nombres 38
Exercice 8. Durée d'un vol d'avion avec conversion 39
Exercice 9. Durée d'un vol d'avion sans conversion 40
Exercice 10. Lendemain d'une date donnée 40
4.3. Structures répétitives 41
Exercice 11. Lecture d'algorithme 41
Exercice 12. Lecture d'algorithme 42
Exercice 13. Multiplication par additions successives 43
Exercice 14. Exponentiation par multiplications successives 43
Exercice 15. Calcul de factorielle 43
Exercice 16. Somme des entiers de 1 à n 44
Exercice 17. Afficher les diviseurs d'un entier 45
Exercice 18. Nombres parfaits 45
Exercice 19. Maximum d'une suite d'entiers 46
Exercice 20. Moyenne d'une suite d'entiers terminée par 0 46
Exercice 21. Vérifier qu'une suite entrée au clavier est croissante 47
Exercice 22. Calcul du PGCD et du PPCM 48
Exercice 23. Nombre premier 48
Exercice 24. Nombres premiers strictement inférieurs à 100 49
Exercice 25. Nombres premiers jumeaux inférieurs à 1000 49
Exercice 26. Calcul du nième nombre d'une suite 50
Exercice 27. Calcul du nième nombre de Fibonnacci 50
Exercice 28. Nombres à trois chiffres 51

Chapitre 5. Corpus d'exercices liés au programme de la classe de seconde 53

5.1. Fonctions 53
5.1.1. Images, antécédents 53
Exercice 29. Calcul d'image 53
Exercice 30. Calcul d'antécédent par une fonction affine 53
Exercice 31. Calcul d'antécédent 54
5.1.2. Résolution d'équations 55
Exercice 32. Résolution d'une équation du premier degré 55
Exercice 33. Encadrer une racine par dichotomie 56
5.1.3. Fonctions de référence 57
Exercice 34. Tracé de courbe 57
5.1.4. Polynômes de degré 2 58
Exercice 35. Tracé de courbe d'un polynôme de degré 2 58
5.1.5. Fonctions homographiques 58
Exercice 36. Tracé de courbe d'une fonction homographique 58
5.1.6. Inéquations 58
Exercice 37. Résolution graphique d'inéquation 1 58
Exercice 38. Résolution graphique d'inéquation 2 60
Exercice 39. Résolution d'inéquation 61
5.1.7. Trigonométrie 62
Exercice 40. Sinus et cosinus dans un triangle rectangle 62
5.2. Géométrie 63
5.2.1. Coordonnées d'un point du plan 63
Exercice 41. Longueur d'un segment 63
Exercice 42. Coordonnées du milieu d'un segment 63
5.2.2. Configurations du plan 64
Exercice 43. Périmètre et aire d'un rectangle 64
Exercice 44. Périmètre et aire d'autres figures 64
Exercice 45. Est-ce un triangle rectangle ? 65
Exercice 46. Est-ce un triangle équilatéral ? 66
Exercice 47. Est-ce un triangle isocèle ? 67
Exercice 48. Triangle des milieux 67

Exercice 49. Est-ce un rectangle ? 69
5.2.3. Droites 70
Exercice 50. Équation de droite donnée par deux points 70
Exercice 51. Équation de droite perpendiculaire....71
Exercice 52. Équation de droite parallèle....72
Exercice 53. Droites parallèles ou sécantes ? 73
Exercice 54. Droites perpendiculaires ?....74
Exercice 55. Trois points sont-ils alignés ? 75
5.2.4. Vecteurs 76
Exercice 56. Coordonnées d'un vecteur 76
Exercice 57. Vecteurs égaux....77
Exercice 58. Vecteurs colinéaires 78
5.3. Statistiques et probabilités 78
Exercice 59. Lancer de pièces (1)....78
Exercice 60. Lancer de pièces (2)....79
Exercice 61. Lancer de pièces (3)....80
Exercice 62. Lancer de dés (1) 81
Exercice 63. Lancer de dés (2) 82
Exercice 64. Boules rouges et noires (1) 83
Exercice 65. Boules rouges et noires (2) 84
5.4. Divers 86
5.4.1. Intervalles....86
Exercice 66. Appartenance à un intervalle....86
Exercice 67. Inclusion d'intervalles 86
Exercice 68. Intersection de deux intervalles....87
Exercice 69. Réunion d'intervalles 88
5.4.2. Approximations de Pi 89
Exercice 70. Approximation de Pi (1)....89
Exercice 71. Approximation de Pi (2)....90
Exercice 72. Approximation de Pi (3)....91
Exercice 73. Approximation de Pi (4)....91

Chapitre 1. Notions de base d'algorithmique

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

Un algorithme décrit un enchaînement d'opérations permettant, en un temps fini, de résoudre toutes les instances d'un problème donné. Un algorithme permet donc, à partir d'une instance du problème (les données en entrée), d'obtenir un résultat correspondant à la solution du problème sur cette instance. Ce résultat est obtenu en réalisant « pas à pas » une succession d'opérations[1] élémentaires.

Prenons un exemple simple, le rendu de monnaie. Une instance de ce problème est la donnée de deux valeurs, le prix à payer, disons 22,30 €, et la somme fournie par le client, disons 25 €. Le résultat attendu est la valeur de la monnaie à rendre, soit 2,70 € dans notre cas. Naturellement, le résultat s'obtient en retranchant le prix à payer de la somme fournie. De façon informelle, cet algorithme peut être transcrit ainsi :

```
Algorithme de rendu de monnaie
début de l'algorithme
   1. demander le prix à payer
   2. demander la somme fournie par le client
   3. retrancher le prix à payer de la somme fournie par le client
   4. afficher le résultat ainsi obtenu
fin de l'algorithme
```

Dans cet exemple, l'ordre dans lequel les opérations doivent être réalisées est donné par leur numérotation. Les valeurs des données en entrée sont « récupérées » (lignes 1 et 2), puis le calcul du résultat est effectué (ligne 3) et, enfin, celui-ci est affiché (ligne 4).

L'expression d'un algorithme nécessite un langage clair (compréhensible par tous), structuré (pour décrire des enchaînements d'opérations élaborés) et non ambigu (pour ne pas être sujet à différentes interprétations). 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 », etc.).

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, quelle que soit l'instance du problème considérée (i.e. les valeurs des données en entrée).

- 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, quelle que soit l'instance du problème considérée.

- La *complexité* de l'algorithme.

  La complexité *en espace* fait référence à l'espace mémoire nécessaire à l'exécution[2] d'un algorithme (directement lié à l'espace mémoire nécessaire pour mémoriser les différentes données)

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

[2] Par abus de langage, on parlera d'exécution d'un algorithme pour faire référence à l'exécution d'un programme qui serait la traduction de cet algorithme.

et la complexité *en temps* au temps nécessaire à cette exécution. La complexité permet de mesurer l'évolution, de l'espace ou du temps nécessaire, en fonction de l'évolution de la taille des données en entrée. Ainsi, un algorithme *linéaire* en temps est un algorithme dont le temps d'exécution dépend linéairement de la taille des données en entrée (pour traiter 10 fois plus de données, il faudra 10 fois plus de temps).

La finalité d'un algorithme est généralement d'être traduit dans un langage de programmation, afin d'être « exécuté » sur un ordinateur. On se doit cependant 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 le traduire à l'aide de n'importe quel langage 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.

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. J.-C., pgcd, division entière), Babyloniens (1800 av. J.-C., résolution de certaines équations).

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 êtres humains, il est important de contraindre ses utilisateurs à utiliser une syntaxe rigoureuse (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 et ne pouvant être utilisés dans un autre contexte). Le *corps* de l'algorithme (qui décrit l'enchaînement des opérations élémentaires) 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 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).

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*.

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`, en n'utilisant que des opérations d'addition ou de soustraction. Sur un exemple (`a=25`, `b=6`) le principe intuitif est le suivant :

```
r = 25                q = 0
r = 25 - 6 = 19       q = 0 + 1 = 1
r = 19 - 6 = 13       q = 1 + 1 = 2
r = 13 - 6 = 7        q = 2 + 1 = 3
r = 7 - 6 = 1         q = 3 + 1 = 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 chaîne de caractères 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 1.11 présente les principaux types de base de notre langage, ainsi que les opérations associées.

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, naturellement…

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. Lorsque nous avons déclaré la constante `PI`, nous lui avons ainsi associé pour valeur la valeur du littéral 3.14.

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 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 parenthèses 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 :

```
…
     # 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 qui avait été
                            # mémorisée dans temporaire
…
```

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_d'identificateurs_de_variables> )
Afficher ( <liste_d'expressions> )
```

Une <liste_d'identificateurs_de_variables> est simplement une liste d'identificateurs séparés par des virgules (e.g. `Entrer ( a, b, c )`). De même, une <liste_d'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   valeur1, valeur2, somme : entiers naturels
début
      # lecture des deux valeurs
  Entrer ( valeur1, valeur2 )
      # calcul de la somme
  somme ← valeur1 + valeur2
      # affichage de la somme
  Afficher (somme)
fin
```

Cet algorithme utilise trois variables, valeur1, valeur2 et somme. Il demande à l'utilisateur de donner deux valeurs entières (rangées dans valeur1 et valeur2), 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   valeur1, valeur2 : entiers naturels
début
      # lecture des deux valeurs
  Entrer ( valeur1, valeur2 )
      # affichage de la somme
  Afficher ( valeur1 + valeur2 )
fin
```

1.5. Initialisation de variables

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. Structure conditionnelle

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 »).

L'alternative simple, ou `si-alors-sinon`, 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 :

```
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 d'un 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.

La syntaxe que nous adopterons est donc la suivante :

```
si ( <expression_logique> )
alors <bloc_alors>
[ sinon <bloc_sinon> ]
fin_si
```

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[3]. 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   valeur1, valeur2 : entiers naturels
début
       # lecture des deux valeurs
  Entrer ( valeur1, valeur2 )
       # affichage de la valeur minimale
  si ( valeur1 < valeur2 )
  alors   Afficher ( valeur1 )
  sinon   Afficher ( valeur2 )
  fin_si
fin
```

1.8. Structures répétitives

Les structures répétitives permettent d'exécuter plusieurs fois un bloc d'opérations, tant qu'une condition (de *continuation*) est 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. La structure Tant que

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).

Exemple 5. L'algorithme suivant permet de calculer le reste de la division entière d'un entier naturel `a` par un entier naturel `b` en effectuant des soustractions successives :

[3] Cette situation est tout à fait différente en programmation Python où c'est justement l'indentation qui permet de délimiter les blocs d'instruction (voir Chapitre 3).

```
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… Pour remédier à cela, il faut donc vérifier si la valeur de b est nulle ou non :

```
Algorithme resteDivisionEntièreVersion2
# 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 )
      # cas de la division par 0
  si ( b = 0 )
  alors
      Afficher ( "division par 0, calcul impossible" )
  sinon
          # 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_si
fin
```

1.8.2. La structure 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 6. 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
```

Pour n = 7, l'exécution de cet algorithme produira l'affichage suivant :

```
   7 * 0 = 0
   7 * 1 = 7
   7 * 2 = 14
   7 * 3 = 21
   7 * 4 = 28
   7 * 5 = 35
   7 * 6 = 42
   7 * 7 = 49
   7 * 8 = 56
   7 * 9 = 63
   7 * 10 = 70
```

Exemple 7. L'algorithme suivant est une variante du précédent qui permet d'afficher les tables de multiplication de 0 à 9 (l'utilisateur n'intervient plus). On utilise pour cela deux structures pour *imbriquées* (i.e. l'une contenant l'autre).

```
Algorithme afficheDixTablesDeMultiplication
# cet algorithme affiche les tables de multiplication de 0 à 9.
variables   n, i, produit : entiers naturels
début
        # boucle principale pour afficher les 10 tables
   pour n de 0 à 9 faire
             # calcul et affichage de la table de multiplication par n
        pour i de 0 à 10 faire
             produit ← n * i
             Afficher ( n, '*', i, '=', produit )
        fin_pour
   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 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> qui, également, ne doivent pas être modifiées dans le corps de boucle.

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 et de les rectifier.

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, b )
      # cas de la division par 0
  si ( b = 0 )
  alors
      Afficher ( "division par 0, calcul impossible" )
  sinon
          # 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_si
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		
Entrer (b)		4	
b = 0 ? faux			
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 soit 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. 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 sont généralement réalisés dans un plan dont les points sont repérés par leurs coordonnées cartésiennes. 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 8. 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
```

1.11. 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, selon les besoins : `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)

Chapitre 2. Exécution d'algorithmes avec AlgoBox

2.1. 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 0.7.2 du 21 juin 2012.

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`),
- les opérations élémentaires d'affectation, de lecture et d'affichage (de variables ou de chaînes de caractères),
- 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 présentons dans ce document de façon directe, à une exception près : Algobox ne permet pas en effet l'affichage d'expressions, du type « `Afficher ( 2 * a + b )` ». Il sera donc nécessaire d'utiliser une variable dans laquelle sera rangée l'expression souhaitée pour pouvoir en afficher sa valeur.

Notons enfin qu'Algobox offre également la possibilité de manipuler des *listes de nombres* (type `LISTE`), mais cette notion ne figurant pas au programme du lycée (ce que l'on peut regretter…), nous ne l'évoquerons pas ici.

2.2. Installation du logiciel

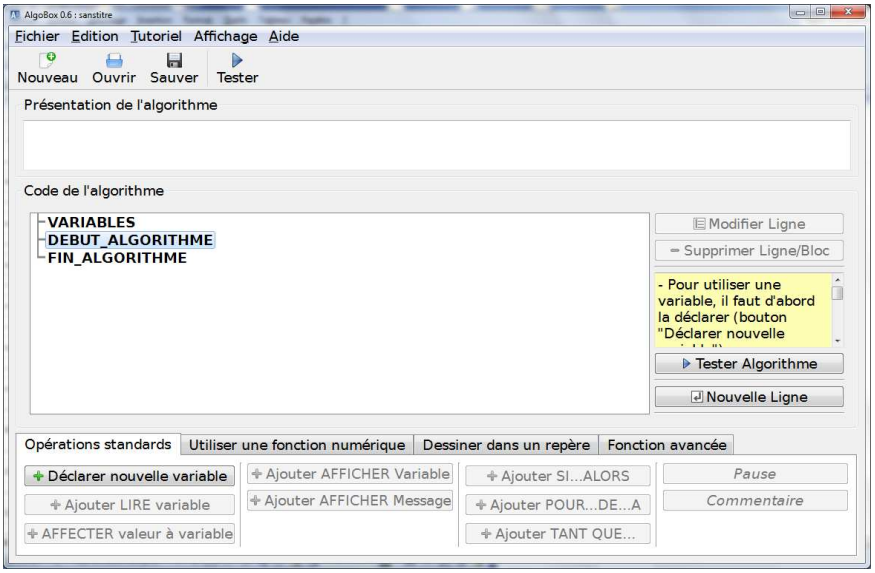
Le logiciel AlgoBox peut être téléchargé gratuitement sur le site http://www.xm1math.net/algobox/.

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 :

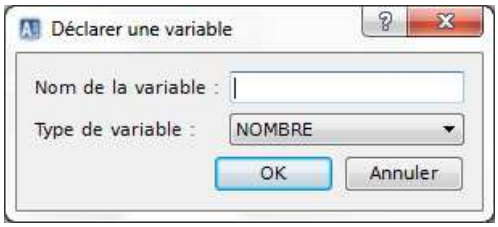
- une animation flash présentant le fonctionnement d'AlgoBox sur un exemple simple,
- un tutoriel d'initiation à l'algorithmique avec AlgoBox,
- des exemples d'algorithmes de tous niveaux réalisés avec AlgoBox.

2.3. Premiers pas

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

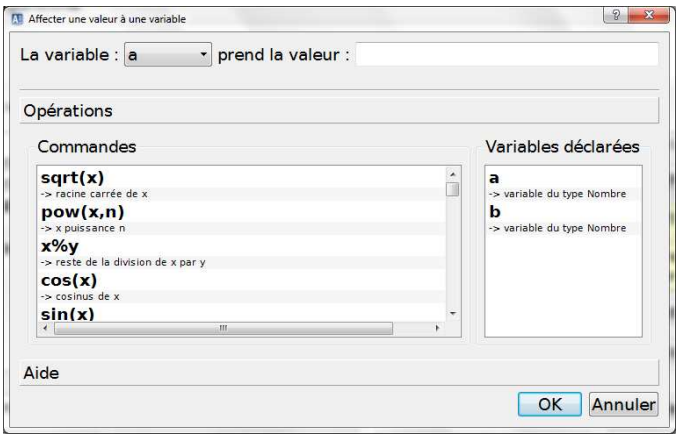


Le bouton `Déclarer nouvelle variable` ouvre la fenêtre ci-contre qui permet de donner un nom à la variable et de définir son type (`NOMBRE`, `CHAÎNE` ou `LISTE`).



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.

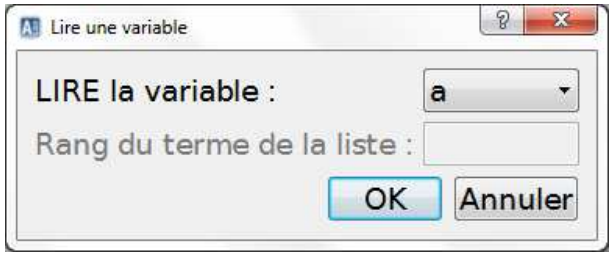
Les fenêtres affichées par AlgoBox dans chacun de ces cas sont les suivantes :



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).

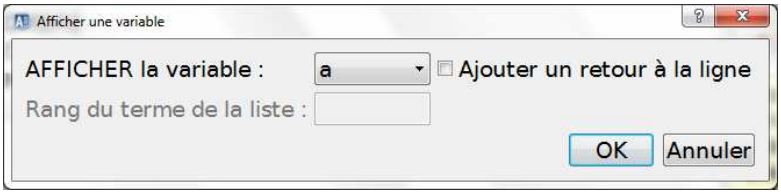
La zone `Variables déclarées` permet d'insérer directement un nom de variable par simple-clic.



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.

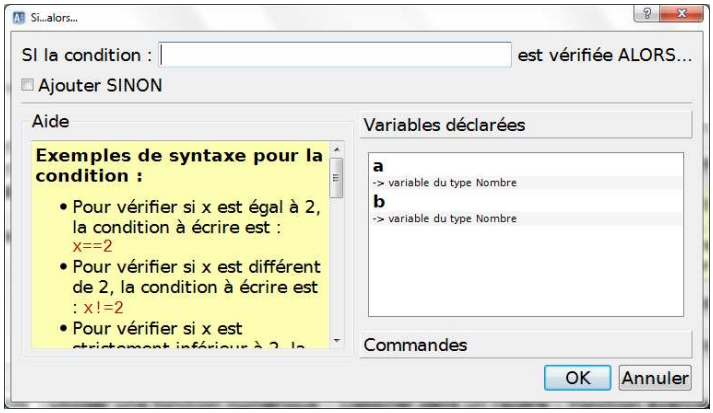
La case à cocher permet de provoquer un changement de ligne après l'affichage de la variable.

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

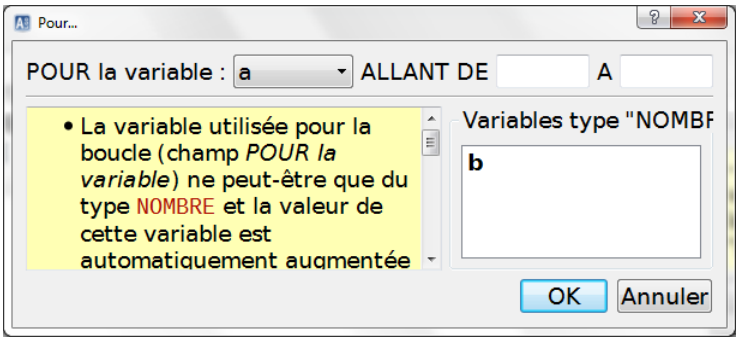
On entre directement le message à afficher.

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



Structure SI

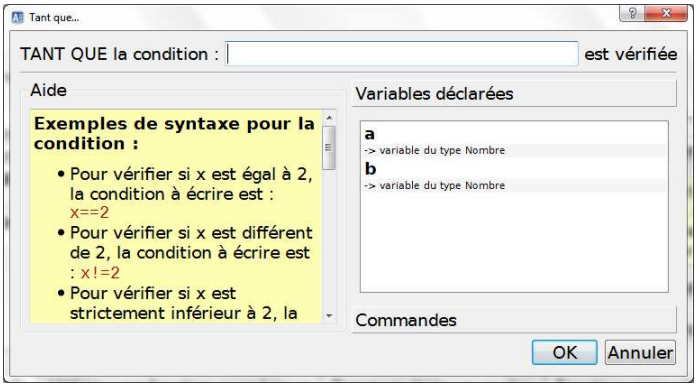
On entre la condition dans la zone prévue (la zone `Variables déclarées` permet d'insérer directement un nom de variable par simple-clic) et on coche la case `Ajouter SINON` si nécessaire.



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 500 000.



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 500 000.

2.4. Quelques compléments

2.4.1. 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 (`+, -, *, /`), diverses fonctions sont disponibles (partie entière, valeur absolue, arrondi, exponentielle, fonctions trigonométriques, etc.). Ces fonctions sont décrites dans la fenêtre « Affecter une valeur à une variable ».

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.

2.4.2. 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 fonction de x>`. Une fois la fonction définie par l'utilisateur, elle s'utilise comme n'importe quelle fonction prédéfinie au sein d'expressions numériques.

2.4.3. 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.

2.5. Quelques exemples illustratifs

2.5.1. 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 :

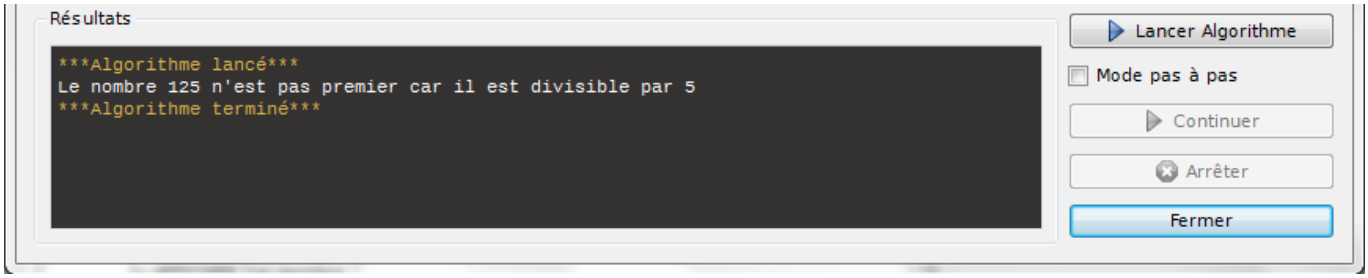
```
est_premier  -  01.11.2010

****************************************
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 :



2.5.2. 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

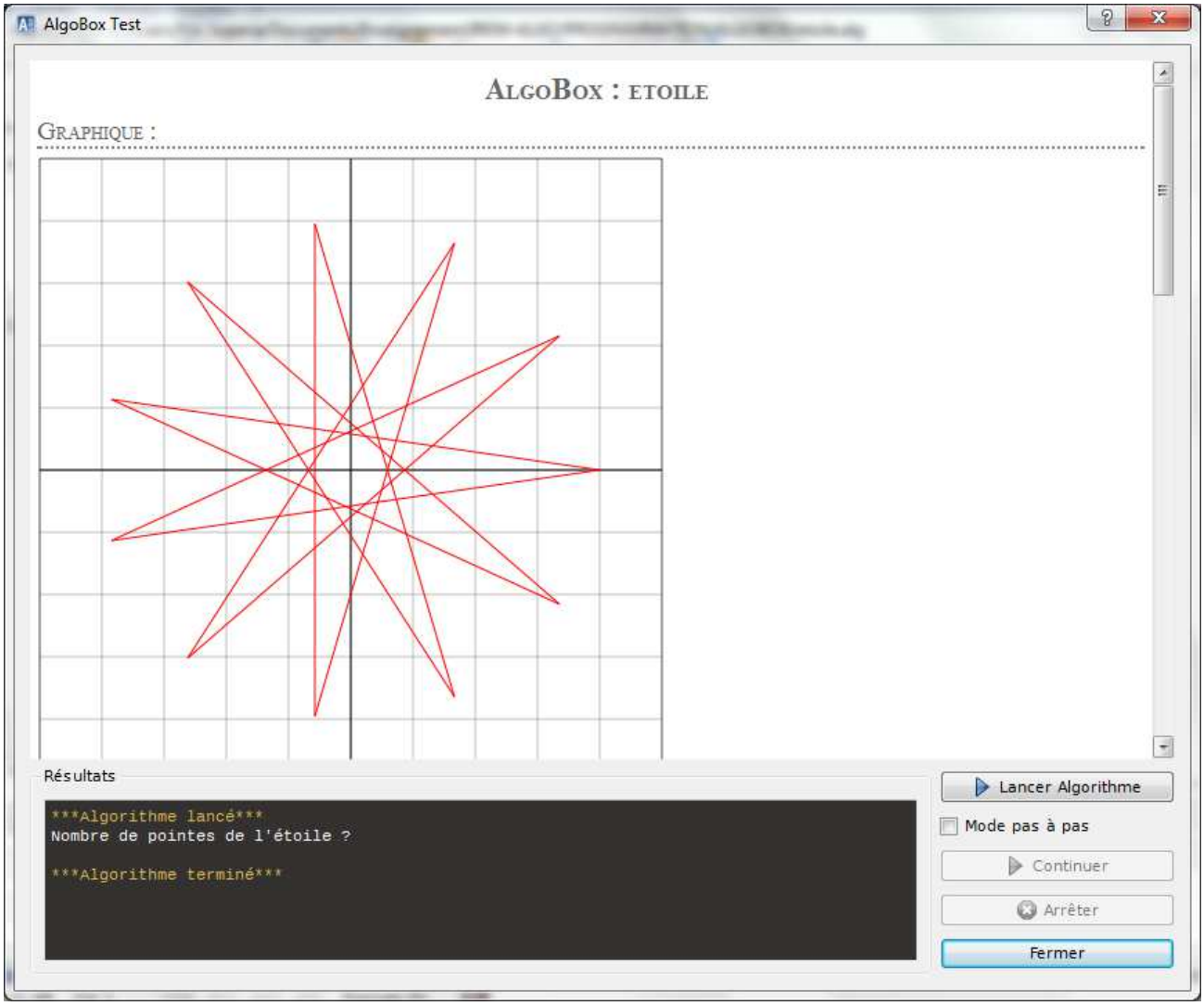
```
etoile  -  01.11.2010

*****************************************
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) :



Chapitre 3. Programmation avec Python

3.1. Introduction

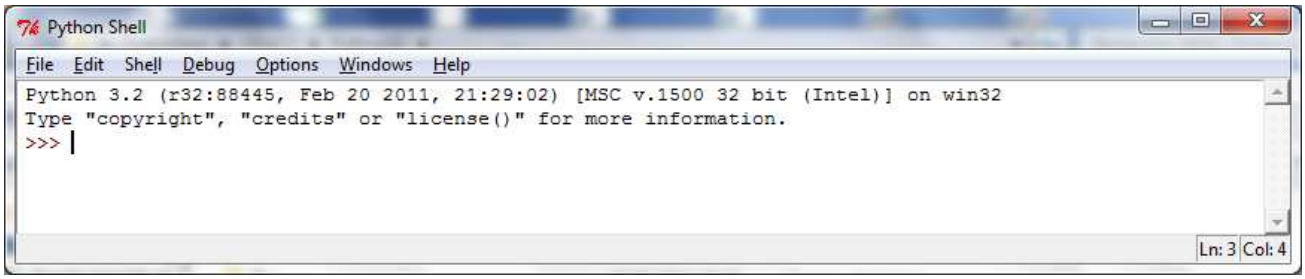
Le langage Python est né dans les années 1990, au CWI Amsterdam, développé par Guido van Rossum. Il a été nommé ainsi par référence à l'émission de la BBC « Monty Python's Flying Circus », et de nombreuses références aux dialogues des Monty Python parsèment les documentations officielles…

Python est un langage libre et gratuit, facile d'accès (il possède une syntaxe très simple), puissant, et est utilisé comme langage d'apprentissage par de nombreuses universités. Dans le cadre de l'initiation au lycée, seule une partie des possibilités de ce langage sera exploitée (en particulier, tous les aspects liés à la programmation par objets se situent hors du cadre de cet enseignement).

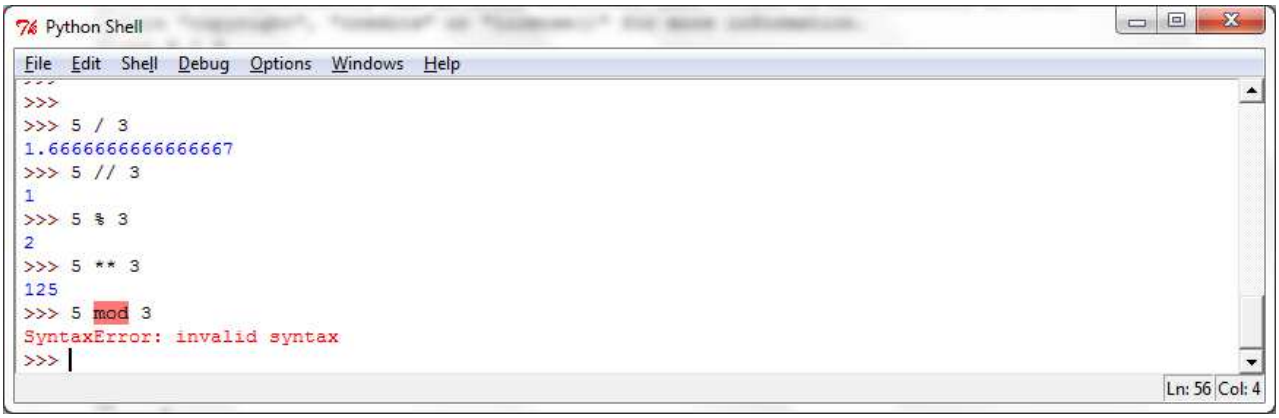
Le langage Python peut être utilisé en mode *interprété* (chaque ligne du code source est analysée et traduite au fur et à mesure en instructions directement exécutées) ou en mode *mixte* (le code source est compilé et traduit en *bytecode* qui est interprété par la *machine virtuelle* Python).

Le site officiel du langage Python est http://www.python.org/. On peut y télécharger gratuitement la dernière version du logiciel[4], pour différentes plateformes (Windows, Mac, Linux), ainsi que de nombreuses documentations. Notons que les versions 3.x constituent une réelle rupture avec les versions précédentes (2.x) qui ne sont plus maintenues. Ce document fait donc référence à la série des versions 3.x.

L'installation de Python ne pose aucun problème particulier. L'environnement de développement IDLE est fourni (<répertoire_Python>\Lib\idlelib\idle.bat) qui suffit amplement à l'utilisation du langage. Le lancement de IDLE ouvre la fenêtre suivante :

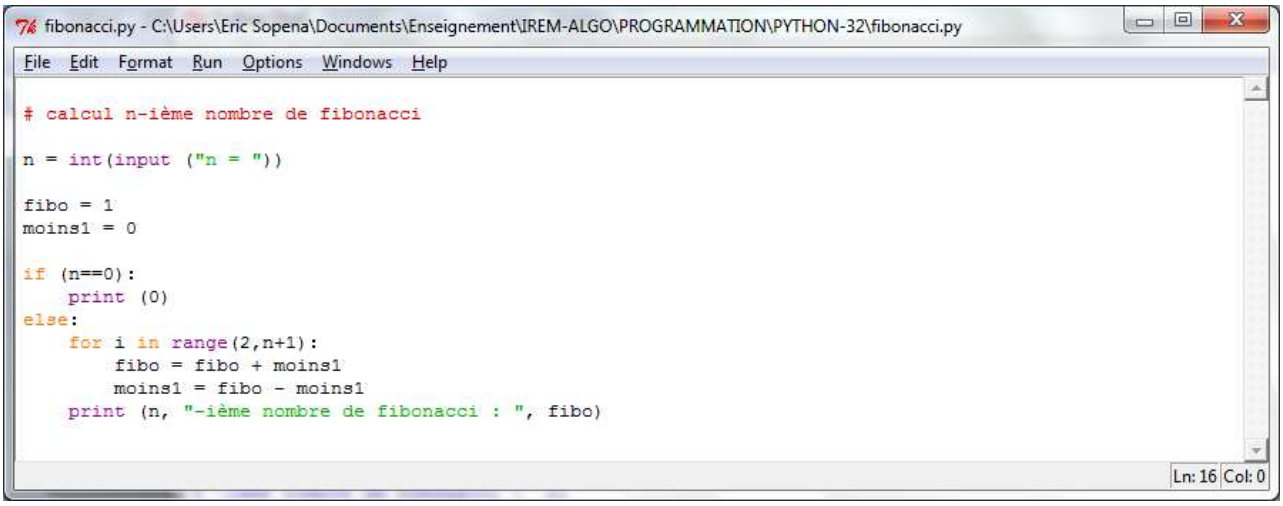


Il est alors possible d'utiliser directement Python en mode interprété (dans ce cas, Python lit, évalue et affiche la valeur) :

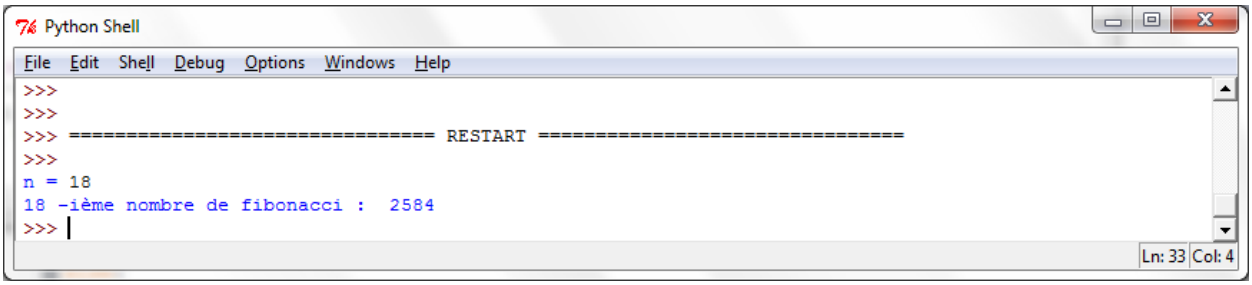


[4] À la date d'édition de ce document, il s'agit de la version 3.3.0.

ou d'éditer un *script* Python (menu File/New Window) :



Les scripts Python doivent être sauvegardés avec l'extension « .py » (dans le cas contraire, l'éditeur IDLE n'est plus « associé » à la syntaxe Python). On peut lancer l'exécution d'un script ouvert dans IDLE à l'aide de la touche F5 (ou par le menu Run/Run Module) :



Parmi les caractéristiques du langage Python, on notera en particulier les suivantes, qui diffèrent de celles que nous avons adoptées pour notre « langage algorithmique » :

- il n'est pas nécessaire de déclarer les variables (attention, cela peut être source de problème en phase d'initiation : une variable mal orthographiée sera considérée comme une nouvelle variable !…),
- les séquences d'instructions (ou blocs) sont définies en fonction de *l'indentation* (décalage en début de ligne) et non à l'aide de délimiteurs de type début-fin.

Ressources documentaires en ligne :

- www.python.org : site officiel de Python
- www.inforef.be/swi/python.htm : ressources didactiques (Gérard Swinnen), dont en particulier le livre *Apprendre à programmer avec Python 3* en téléchargement libre.
- http://hebergement.u-psud.fr/iut-orsay/Pedagogie/MPHY/Python/courspython3.pdf : cours de Robert Cordeau, *Introduction à Python 3*.

3.2. Python pour la classe de seconde

3.2.1. Éléments du langage

Un identificateur Python est une suite non vide de caractères, de longueur quelconque, à l'exception des *mots réservés* du langage : `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`. Notons que Python fait la distinction entre majuscules et minuscules. Ainsi, `nombre` et `Nombre` correspondent à des

identificateurs différents. Une excellente habitude consiste à nommer les constantes en `MAJUSCULES` et les variables en `minuscules`.

Un commentaire Python commence par le caractère # et s'étend jusqu'à la fin de la ligne :

```
# petit exemple de script Python

PI = 3.14
nombre = 1          # la variable nombre est initialisée à 1
```

Une expression est une portion de code que Python peut évaluer, composée d'identificateurs, de littéraux et d'opérateurs : `3 * nombre + 5`, `(math.sqrt(3) - 1 ) / 2`, etc.

3.2.2. Types de données élémentaires

Les types de données prédéfinis qui nous seront utiles sont les types `bool` (booléen), `int` (entier), `float` (flottant) et `str` (chaîne de caractères).

Les expressions de type `bool` peuvent prendre les valeurs `True` ou `False` (i.e. `Vrai` ou `Faux`), et être combinées à l'aide des opérateurs `and`, `or` et `not` (i.e. `et`, `ou` et `non`). On peut notamment combiner des expressions booléennes utilisant les opérateurs de comparaison : `==, !=, <, >, <=, >= :`

```
monBooleen = (( a > 8 ) and ( a <= 20 )) or (b != 15)
```

Les valeurs des expressions de type `int` ne sont limitées que par la taille mémoire, et celles des expressions de type `float` ont une précision finie. Les littéraux de type `float` sont notés avec un point décimal ou en notation exponentielle : `3.14`, `.09` ou `3.5e8` par exemple.

Les principales opérations définies sur les expressions numériques sont les suivantes :

```
print(17 + 4)      # 21
print(21 - 6.2)    # 14.8
print(3.3 * 2)     # 6.6
print(5 / 2)       # 2.5
print(5 // 2)      # 2 (division entière)
print(5 % 2)       # 1 (modulo)
print(5 ** 2)      # 25 (exponentiation)
print(abs(3-8))    # 5 (valeur absolue)
```

L'import du module `math` donne accès à l'ensemble des fonctions mathématiques usuelles :

```
import math

print(math.cos(math.pi/3))          # 0.5000000000000001 (et oui…)
print(math.factorial(6))            # 720
print(math.log(32,2))               # 5.0
```

Les littéraux de type `str` sont délimités par des « ' » ou des « " » :

```
chaine1 = "Bonjour"
chaine2 = 'coucou les "amis"'
chaine3 = "aujourd'hui"
```

Les chaînes peuvent être manipulées à l'aide des opérations suivantes :

```
chaine1 = 'bonjour'
print(len(chaine1))             # 7 (len = longueur)

chaine2 = chaine1 + ' monde'    # + = concaténation de chaînes
print(chaine2)                  # bonjour monde

chaine2 = chaine1.upper()       # passage en majuscules
print(chaine2)                  # BONJOUR

chaine2 = chaine2.lower()       # passage en minuscules
print(chaine2)                  # bonjour

print(chaine2[0])               # b       (les indices débutent à 0)
print(chaine2[3])               # j
print(chaine2[1:4])             # onj     (de l'indice 1 à l'indice 4
                                #         non compris)
print(chaine2[:3])              # bo      (du début à l'indice
                                #         3 non compris)
```

```
print(chaine2[4:])                # our     (de l'indice 4 à la fin)
```

3.2.3. Affectation et opérations d'entrée-sortie

L'affectation se note à l'aide du symbole '=', qu'il ne faudra pas confondre avec l'opérateur d'égalité (opérateur de comparaison), noté '==' :

```
nombre1 = 2 * 18 - 5              # nombre1 ← 31
print(nombre1 == 30)              # False
```

Il est possible de modifier (on dit également « transtyper »), le type d'une expression de la façon suivante :

```
nombre1 = int(2*PI)               # nombre1 ← 6
flottant1 = float(17*2)           # flottant1 ← 34.0
```

Pour saisir une valeur au clavier, on utilise l'instruction `input`, éventuellement complétée par un message à afficher. Il s'agit d'une lecture *en mode texte*, c'est-à-dire que la valeur retournée par cette instruction est de type str ; il est donc la plupart du temps nécessaire de transtyper cette valeur :

```
nbTours = int (input('nombre de tours ? '))   # saisie d'un entier
msg = input('message ? ')                     # saisie d'une chaîne
```

L'instruction `print` (que nous avons déjà plusieurs fois utilisée) permet d'afficher une séquence d'expressions (séparées par des virgules) :

```
i = 3
print('resultat :',3*i)      # resultat : 9
```

Par défaut, l'instruction `print` rajoute un espace entre les différentes valeurs de la liste d'expressions, et un saut de ligne après affichage, à moins d'utiliser les options `sep` (caractères à rajouter entre les expressions de la liste) et/ou `end` (caractères à rajouter en fin de ligne) de la façon suivante :

```
i = 5
print(i,end="")
print(i+1)
                          # affiche 56 avant de changer de ligne
print(3,4,5,sep='-',end='***')   # affiche 3-4-5***
```

Les littéraux de type `str` peuvent contenir des *caractères spéciaux*, préfixés par un '\' (antislash), dont la signification est la suivante :

caractère	signification
\'	apostrophe
\"	guillemet
\n	saut de ligne
\a	bip (sonnerie)
\t	tabulation horizontale

3.2.4. Structures de contrôle

Rappelons que les instructions ayant la même indentation (même « décalage » par rapport au début de la ligne) appartiennent à un même bloc. Les décalages s'effectuent généralement de 4 en 4 (touche tabulation), et sont automatiquement gérés dans l'éditeur IDLE.

Alternative

La structure `if-else` est l'équivalent Python de notre « `si ... alors ... sinon ... fin_si` » :

```
if a == b:
    print('egalite pour',a,'et',b)
    print('----------------------')
```

```
nombre = 8                    # fin du if…, pas de partie « else »

if nombre > 5:
     print('grand')
else:
     print('petit')
a = 3                         # fin du else…
```

Structure à choix multiple

La structure `if-elif-…-else` permet de simplifier l'écriture des « `si-alors-sinon` » imbriqués, grâce à la contraction du « `else if` » qui évite d'avoir une indentation excessive :

```
note = float(input('Note du baccalaureat : '))
if note >= 16:
     print('mention TB')
elif note >= 14:
     print('mention B')
elif note >= 12:
     print('mention AB')
elif note>= 10:
     print('mention Passable')
else:
     print('collé...')
```

Boucle while

La boucle `while` est l'équivalent Python de notre « `tantque faire ... fin_tantque` » :

```
n = int (input('Donnez un entier compris entre 1 et 10 : ') )
while (n<1) or (n>10):
     print('J\'ai dit compris entre 1 et 10 !')
     n = int (input('Allez, recommencez : ') )
print('Merci !...')
```

Boucle for

La boucle `for` est l'équivalent Python de notre « `pour .. de .. à .. faire .. fin_pour` ». L'expression `range(i)` retourne la liste des entiers allant de `0` à `i` non compris. L'expression `range(i,j)` retourne la liste des entiers allant de `i` à `j` non compris. L'expression `range(i,j,k)` retourne la liste des entiers allant de `i` à `j` non compris *par pas de* `k`.

```
for i in range(7):
     print(2*i,end=' ')       # affiche : 0 1 2 3 4 5 6

for i in range(1,6):
     print(2*i,end=' ')       # affiche : 2 4 6 8 10

for i in range(1,13,3)
     print(i,end=' ')         # affiche : 1 4 7 10
```

3.2.5. Quelques exemples de scripts Python

Ce premier script calcule le produit de deux nombres en utilisant l'algorithme dit de la « multiplication russe » :

```
# multiplication russe de a par b

# initialisations
a = int(input("a = "))
b = int(input ("b = "))
c = 0;

# boucle de calcul
while (a != 0):
    if (a % 2 == 1):
        c = c + b
    a = a // 2
    b = b * 2

# affichage du résultat
```

```
print ("Résultat :", c)
```

Ce deuxième script permet de déterminer si un entier n est ou non premier :

```
# ce script détermine si l'entier N est premier ou non

import math

# lecture de N
N = int(input("N ? "))

# initialisations
racineDeN = int(math.sqrt(N))
diviseur = 2

# tant qu'on n'a pas trouvé de divisieur, on avance...
while ((N % diviseur != 0) and (diviseur <= racineDeN)):
    diviseur = diviseur + 1

# si diviseur est allé au delà de racineDeN, N est premier
if (diviseur > racineDeN):
    print ("Le nombre", N, "est premier.")
else:
    print ("Le nombre", N, "est divisible par", diviseur)
```

Remarque : on pourrait naturellement traiter à part le cas des nombres pairs et ne tester ensuite que les diviseurs 3, 5, 7, 9, 11, etc.

3.2.6. Traduction d'algorithmes en Python - Tableau de synthèse

Langage algorithmique	Script Python
Algorithme XXX début … fin	*rien de particulier (à notre niveau), le script commence à la première instruction…*
# ceci est un commentaire	# ceci est un commentaire
var a, b : entiers	*pas de déclaration de variables*
Entrer (a)	a = int(input("valeur de a ? "))
Afficher ("résultat : ", res)	print ("résultat : ", res)
a ← a + 2	a = a + 2
quotient division entière	//
reste division entière	%
opérateurs de comparaison : <, >, ≤, ≥, =, ≠	<, >, <=, >=, ==, !=
opérateurs logiques : et, ou, non	and, or, not
si (a = b) c = 2 * c fin_si	if (a == b): c = 2 * c

si (a = b)
  c = 2 * c
sinon
  a = a + 1
  b = b div 2
fin_si
	if (a == b):
	  c = 2 * c
	else:
	  a = a + 1
	  b = b / 2

tantque (a ≠ 0) faire Afficher (2 * a) a ← a - 1 fin_tantque	while (a != 0): print 2*a a = a - 1
pour i de 1 à n faire a = 3 * i Afficher (a) fin_pour	for i in range(1,n+1): a = 3 * i print a *range(a,b) permet de parcourir les valeurs a, a+1, …, b-1 (b non compris)*
pour i de 3 à 20 par pas de 2 faire a = 3 * i Afficher (a)	for i in range(3,21,2): a = 3 * i print a

fin_pour	
Manipulation de chaînes de caractères	- ch[i] : i-ième caractère de ch (les indices démarrent à 0) - len(ch) : nombre de caractères de ch - ch1 + ch2 : concaténation - ch[i:j] : la sous-chaîne de ch allant du i-ième au (j-1)-ième caractère - ch[i:] : la fin de ch, à partir du i-ième caractère - ch[:i] : le début de ch, jusqu'au (i-1)-ième caractère

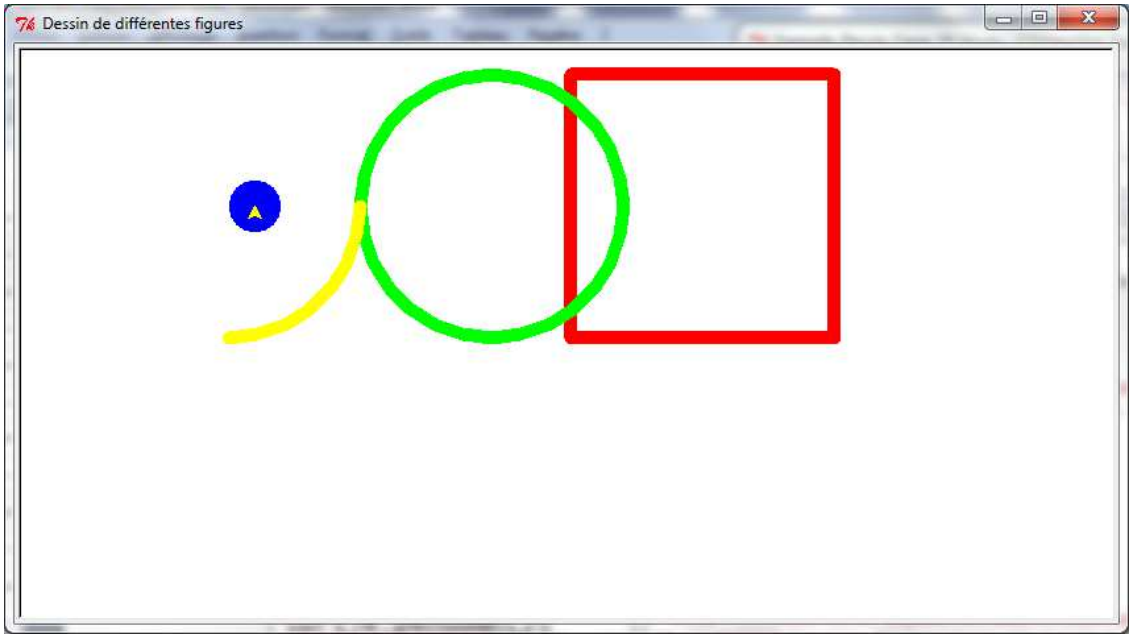
3.2.7. Dessiner en Python

Le module `turtle` permet de dessiner dans une fenêtre graphique à l'aide d'un ensemble de primitives dédiées.

L'exemple suivant, aisément compréhensible, permet de dessiner différentes figures :

```
# script exemple turtle : dessin de différentes figures
import turtle
    # reinitialise la fenêtre
turtle.reset()
    # titre de la fenêtre
turtle.title("Dessin de différentes figures")
    # paramètres de dessin
turtle.color('red')
turtle.width(10)
turtle.speed(5)
    # dessin d'un carré
monCote = 200
turtle.pendown()
for i in range(4):
    turtle.forward(monCote)
    turtle.left(90)
    # partons ailleurs dessiner un cercle vert
turtle.penup()
turtle.goto(-60,0)
turtle.pendown()
turtle.color('green')
turtle.circle(100)
    # puis un arc de cercle jaune
turtle.penup()
turtle.goto(-260,0)
turtle.pendown()
turtle.color('yellow')
turtle.circle(100,90)
    # et enfin un gros point bleu
turtle.penup()
turtle.goto(-240,100)
turtle.pendown()
turtle.dot(40,'blue')
```

Le résultat obtenu est le suivant :



Les principales primitives graphiques qui nous seront utiles sont les suivantes :

PYTHON - Utilisation du module `turtle`	
`import turtle`	permet d'importer les fonctionnalités du module turtle
`turtle.title(<chaîne>)`	donne un titre à la fenêtre turtle (par défaut : Turtle Graphics)
`turtle.reset()`	efface l'écran, recentre la tortue en (0,0)
`turtle.color(<chaîne>)`	détermine la couleur du tracé (noir par défaut). Couleurs prédéfinies : 'red', 'blue', 'green', etc.
`turtle.width(<épaisseur>)`	détermine l'épaisseur du tracé
`turtle.speed(<vitesse>)`	détermine la vitesse du tracé (valeur entière)
`turtle.forward(<distance>)`	avance d'une distance donnée
`turtle.backward(<distance>)`	recule d'une distance donnée
`turtle.left(<angle>)`	tourne à gauche d'un angle donné (exprimé en degrés)
`turtle.right(<angle>)`	tourne à droite d'un angle donné (exprimé en degrés)
`turtle.circle(<rayon>{,<angle>})`	dessine un cercle de rayon donné, ou un arc de cercle de rayon et angle donnés
`turtle.dot(<diamètre>,<couleur>)`	dessine un point (cercle plein) de diamètre et couleur donnés
`turtle.penup()`	relève le crayon (pour pouvoir se déplacer sans dessiner)
`turtle.pendown()`	abaisse le crayon (pour pouvoir se déplacer en dessinant)
`turtle.goto(x,y)`	se déplace jusqu'au point de coordonnées (x,y)
`turtle.xcor(), turtle.ycor()`	retourne la coordonnée courante (abscisse ou

	ordonnée) de la tortue
`turtle.write(<chaîne>)`	écrit la chaîne de caractères

Remarque. Il peut arriver que la fenêtre graphique, une fois le programme terminé, ne réponde pas… Pour remédier à ce problème, il suffit que le programme `idle.pyw` soit lancé avec l'option `-n`. Une façon simple de réaliser cela est d'effectuer une copie `idle2.bat` du fichier `idle.bat` (situé dans le répertoire `Lib\idlelib` de l'installation Python), et de le modifier ainsi :

Fichier `idle.bat` (à modifier) :

```
@echo off
rem Start IDLE using the appropriate Python interpreter
set CURRDIR=%~dp0
start "%CURRDIR%..\..\pythonw.exe" "%CURRDIR%idle.pyw" %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Fichier `idle2.bat` (modifié) :

```
@echo off
rem Start IDLE using the appropriate Python interpreter
set CURRDIR=%~dp0
start "%CURRDIR%..\..\pythonw.exe" "%CURRDIR%idle.pyw" -n %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Il suffit alors de lancer `idle2.bat` au lieu de `idle.bat`…

Chapitre 4. 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.

4.1. 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
```

Réponse. *Cet algorithme échange le contenu des variables a et b. Pour voir cela, il suffit de « faire tourner » l'algorithme à la main, en suivant l'évolution du contenu des variables :*

opération	*valeur de a*	*valeur de b*	
Entrer (a)	*17*		*par exemple…*
Entrer (b)		*28*	*par exemple…*
a ← a + b	*45*		
b ← a - b		*17*	
a ← a - b	*28*		
Afficher (a, b)	*28*	*17*	

Attention, l'algorithme ci-dessus pose un problème : si les valeurs de a et b sont « trop grandes », la première affectation peut entraîner un débordement… (la valeur de la somme étant au-delà de la limite supérieure des entiers naturels gérés par le langage de programmation utilisé…).

Il est possible d'éviter cet inconvénient en utilisant l'opérateur binaire XOR (ou exclusif), en écrivant :

```
      # calcul mystère - version 2
  a ← a XOR b
  b ← a XOR b
  a ← a XOR b
```

On peut en effet vérifier que cette séquence échange bien le contenu des variables a et b. Prenons par exemple a = 28 (et donc '11100' en binaire) et b = 17 (et donc '10001' en binaire).

Nous avons alors :

opération	*valeur de a (binaire)*	*valeur de b (binaire)*	*valeur de a (décimal)*	*valeur de b (décimal)*
	11100	*10001*	*28*	*17*
a ← a XOR b	*01101*	*10001*	*13*	*17*
b ← a XOR b	*01101*	*11100*	*13*	*28*
a ← a XOR b	*10001*	*11100*	*17*	*28*

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.

Réponse. *Rappelons que l'opérateur `div` permet d'obtenir le quotient et l'opérateur `mod` le reste de la division entière. L'idée consiste ici à déterminer dans un premier temps le nombre de billets de 20 euros nécessaires (qui correspond au quotient de la division du montant par 20) puis, pour la somme restante (à calculer…), le nombre de billets de 10 euros nécessaires et ainsi de suite.*

L'algorithme est le suivant :

```
Algorithme décompositionMontant

# Cet algorithme décompose un montant entré au clavier en billets
# de 20, 10, 5 euros et pièces de 2, 1 euros.

variables   montant, reste : entiers naturels
            billets20, billets10, billets5 : entiers naturels
            pièces2, pièces1 : entiers naturels

début

      # lecture donnée
  Entrer ( montant )

      # calculs
  billets20 ← montant div 20
  reste ← montant mod 20      # ou reste ← montant - (20 * billets20)
  billets10 ← reste div 10
  reste ← reste mod 10
  billets5 ← reste div 5
  reste ← reste mod 5
  pièces2 ← reste div 2
  reste ← reste mod 2
  pièces1 ← reste

      # affichage résultat
  Afficher ( billets20, billets10, billets5, pièces2, pièces1 )

fin
```

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).

Réponse. *On applique simplement la formule*

$$\frac{numA}{dénomA} + \frac{numB}{dénomB} \quad = \quad \frac{numA \times dénomB + numB \times dénomA}{dénomA \times dénomB}.$$

L'algorithme est alors le suivant :

```
Algorithme sommeDeDeuxFractions

# cet algorithme calcule le numérateur et le dénominateur d'une somme
# de deux fractions entières.

variables   numA, dénomA, numB, dénomB : entiers naturels
            numSomme, dénomSomme : entiers naturels

début
```

```
        # lecture données
   Entrer ( numA, dénomA, numB, dénomB )
        # calcul d'un dénominateur commun
   dénomSomme ← dénomA * dénomB
        # calcul du numérateur
   numSomme ← ( numA * dénomB ) + ( numB * dénomA )
        # affichage résultat
   Afficher ( numSomme, dénomSomme )
fin
```

4.2. Structures conditionnelles

Exercice 4. Valeur absolue

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

Réponse. *Il suffit de tester le signe de l'entier lu. L'algorithme est le suivant :*

```
Algorithme valeurAbsolue
# cet algorithme calcule la valeur absolue d'un entier relatif
# entré au clavier.
variables   val, valAbsolue : entiers relatifs
début
        # lecture données
   Entrer ( val )
        # calcul valeur absolue
   si ( val < 0 )
   alors    valAbsolue ← - val
   sinon    valAbsolue ← val
   fin_si
        # affichage du résultat
   Afficher (valAbsolue)
fin
```

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).

Réponse. *L'algorithme est le suivant :*

```
Algorithme equationPremierDegré
# cet algorithme résout une équation de la forme ax + b = 0
# a et b sont entrés au clavier.
variables   a, b, x : réels
début
        # lecture données
   Entrer ( a, b )
        # résolution de l'équation
             # cas où a = 0
   si ( a = 0 )
   alors    si ( b = 0 )
            alors Afficher ( "Tous les réels sont solution" )
            sinon Afficher ( "Pas de solution" )
             # cas où a ≠ 0
   sinon    x ← - ( b / a )
            Afficher ( x )
```

```
   fin_si
fin
```

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.

Réponse. *L'algorithme est le suivant :*

```
Algorithme equationSecondDegré
# cet algorithme résout une équation à coefficients réels de la
# forme ax^2 + bx + c = 0.
# a, b et c sont entrés au clavier.
variables   a, b, c, delta, x1, x2 : réels
début
       # lecture données
   Entrer ( a, b, c )
       # calcul discriminant
   delta ← ( b * b ) - ( 4 * a * c )
       # cas 1 : discriminant strictement positif
   si ( delta > 0 )
   alors     x1 ← ( - b - RacineCarrée(delta) ) / ( 2 * a )
             x2 ← ( - b + RacineCarrée(delta) ) / ( 2 * a )
             Afficher ( "Deux solutions : ", x1, " et ", x2 )
       # cas 2 : discriminant égal à 0
   sinon     si ( delta = 0 )
             alors     x1 ← - b / ( 2 * a )
                       Afficher ( "Une solution : ", x1 )
       # cas 3 : discriminant négatif
             sinon Afficher ( "Pas de solution" )
             fin_si
   fin_si
fin
```

Exercice 7. Minimum de trois nombres

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

Réponse. *Un premier algorithme est le suivant : si a est plus petit que b, il suffit de comparer a à c… sinon, il faut comparer b à c :*

```
Algorithme minimumTroisNombres
# cet algorithme permet d'afficher le plus petit de trois nombres
# entrés au clavier.
variables   a, b, c, min : entiers naturels
début
       # lecture données
   Entrer ( a, b, c )
       # comparaisons
   si ( a < b )
   alors     si ( a < c )
             alors min ← a
             sinon min ← c
             fin_si
   sinon     si ( b < c )
             alors min ← b
             sinon min ← c
             fin_si
```

```
   fin_si
        # Affichage minimum
   Afficher ( min )
fin
```

Une autre version, plus synthétique, est obtenue en calculant dans un premier temps le minimum de a et b, puis le minimum de ce minimum et c :

```
Algorithme minimumTroisNombres2
# cet algorithme permet d'afficher le plus petit de trois nombres
# entrés au clavier.
variables   a, b, c, min : entiers naturels
début
        # lecture données
   Entrer ( a, b, c )
        # comparaisons
   si ( a < b )
   alors min ← a
   sinon min ← b
   fin_si
   si ( c < min )
   alors min ← c
   fin_si
        # Affichage minimum
   Afficher ( min )
fin
```

Exercice 8. 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.

Réponse. *On convertit en minutes les deux horaires, on calcule la durée en minutes et on reconvertit en heures et minutes, en utilisant les opérateurs* `div` *et* `mod`*.*

L'algorithme est le suivant :

```
Algorithme duréeVolAvionAvecConversion
# cet algorithme permet de calculer la durée d'un vol d'avion
# en convertissant les horaires de départ et d'arrivée en minutes.
variables   h_depart, m_depart, h_arrivee, m_arrivee : entiers naturels
            duree_m, depart_m, arrivee_m : entiers naturels
            h_duree, m_duree : entiers naturels
début
        # lecture données
   Entrer ( h_depart, m_depart, h_arrivee, m_arrivee )
        # conversion des horaires
   depart_m ← ( 60 * h_depart ) + m_depart
   arrivee_m ← ( 60 * h_arrivee ) + m_arrivee
        # correction si l'arrivée a lieu le lendemain du jour de départ
   si ( arrivee_m < depart_m )
   alors     arrivee_m ← arrivee_m + ( 24 * 60 )
   fin_si
        # calcul de la durée en minutes
   duree_m ← arrivee_m – depart_m
        # conversion de la durée en h et min
   h_duree ← duree_m div 60         # quotient de la division entière
   m_duree ← duree_m mod 60         # reste de la division entière
```

```
        # affichage résultat
   Afficher ( h_duree, m_duree )
fin
```

Exercice 9. 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.

Réponse. *On effectue le calcul sans tenir compte des valeurs… et on ajuste le cas échéant. L'algorithme est alors le suivant :*

```
Algorithme duréeVolAvionSansConversion
# cet algorithme permet de calculer la durée d'un vol d'avion
# sans convertir les horaires de départ et d'arrivée en minutes.
variables   h_depart, m_depart, h_arrivee, m_arrivee : entiers naturels
            h_duree, m_duree : entiers naturels
début
        # lecture données
   Entrer ( h_depart, m_depart, h_arrivee, m_arrivee )
        # calcul de la durée
   h_duree ← h_arrivee - h_depart
   m_duree ← m_arrivee - m_depart
        # on rectifie les minutes si nécessaire
   si ( m_durée < 0 )
   alors    m_duree ← 60 + m_duree
            h_duree = h_duree - 1
   fin_si
        # on rectifie les heures si nécessaire
   si ( h_duree < 0 )
   alors    h_duree = 24 + h_duree
   fin_si
        # affichage résultat
   Afficher ( h_duree, m_duree )
fin
```

Exercice 10. 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).

Réponse. *Si l'on est en fin de mois, on passe au mois suivant, sinon on passe au jour suivant.*

L'algorithme est le suivant :

```
Algorithme lendemainDate
# cet algorithme permet de calculer le lendemain d'une date donnée
# de l'année 2013.
variables   jour, mois, jourDemain, moisDemain : entiers naturels
début
        # lecture données
   Entrer ( jour, mois )
        # test fin de mois
   si ( (jour=31)
        ou ( (jour=30) et ((mois=4) ou (mois=6) ou (mois=9) ou (mois=11)) )
        ou ( (jour=28) et (mois=2) )
   alors    jourDemain ← 1
            si ( mois = 12 )
```

```
            alors     moisDemain ← 1
            sinon     moisDemain ← mois + 1
            fin_si

       # sinon, mois en cours…
   sinon    jourDemain ← jour + 1
            moisDemain ← mois

       # affichage résultat
   Afficher ( jourDemain, moisDemain )
fin
```

4.3. Structures répétitives

Exercice 11. 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
```

Réponse. *Cet algorithme calcule le produit de a par b (multiplication dite russe ou égyptienne), ce que l'on peut « deviner » en faisant tournant l'algorithme à la main sur un exemple (ici a = 21 et b = 6)..*

opération	valeur des variables		
	a	b	c
Entrer (a)	21		
Entrer (b)		6	
c ← 0			0
a ≠ 0 ? oui…			
(a mod 2) ≠ 0 ? oui…			
c ← c + b			6
a ← a div 2	10		
b ← b * 2		12	
a ≠ 0 ? oui…			
(a mod 2) ≠ 0 ? non…			
a ← a div 2	5		
b ← b * 2		24	
a ≠ 0 ? oui…			
(a mod 2) ≠ 0 ? oui…			
c ← c + b			30
a ← a div 2	2		

b ← b * 2		48	
(a mod 2) ≠ 0 ? non…			
a ← a div 2	1		
b ← b * 2		96	
a ≠ 0 ? oui…			
a ≠ 0 ? oui…			
(a mod 2) ≠ 0 ? oui…			
c ← c + b			126
a ← a div 2	0		
b ← b * 2		192	
a ≠ 0 ? non… (fin de boucle)			
Afficher (c)			**126**

Cet algorithme est basé sur la décomposition en base 2 du nombre a*. Pour* a = 21*, nous avons en effet* $21 = 16 + 4 + 1 = 2^4 + 2^2 + 2^0$ *(*21 *s'écrit donc* 10101 *en binaire).*

Ainsi, $21 * 6 = (2^4 + 2^2 + 2^0) * 6 = 6 + (4 * 6) + (16 * 6)$.

La variable b *permet de calculer les produits de* b *par les puissances de 2 :* b *prend successivement les valeurs* 6 ($6 * 2^0$), 12 ($6 * 2^1$), 24 ($6 * 2^2$), 48 ($6 * 2^3$)*, etc.*

La variable a *est divisée par 2 (division entière) à chaque tour de boucle. Lorsque sa valeur est impaire, celle correspond à un 1 dans l'écriture binaire de* a*, et il faut donc prendre en compte le produit correspondant de b par une puissance de 2. La variable* c *permet de sommer les produits ainsi pris en compte, elle contiendra donc le produit* a * b *recherché…*

Exercice 12. 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
```

Réponse. *Cet algorithme calcule la valeur de* a *élevé à la puissance* b *(exponentiation rapide). Cet algorithme utilise le même principe que le précédent : cette fois, c'est la décomposition binaire de* b *qui est utilisée. La variable* a*, elle, prend successivement les valeurs* a, a^2, a^4, a^8*, etc. Lorsqu'un tour de boucle correspond à un 1 dans la décomposition binaire de* b*, la variable* c *« cumule par produit » la puissance correspondante de* a*.*

Ainsi, pour $b = 21 = 2^4 + 2^2 + 2^0$*, la variable* c *vaudra finalement* $a * a^4 * a^{16} = a^{1+4+16} = a^{21} = a^b$*. Cet algorithme calcule donc la valeur de* a *à la puissance* b*, ce que l'on peut aisément vérifier en le faisant tourner sur un (petit) exemple…*

Exercice 13. 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)).

Réponse. *On utilise une variable* res*, initialisée à* 0*, pour calculer la somme totale. Une boucle* Pour *permet d'additionner* b *fois la valeur* a*.*

L'algorithme est le suivant :

```
Algorithme multiplicationParAdditionsSuccessives
# cet algorithme permet de calculer le produit de deux entiers naturels
# entrés au clavier en effectuant des additions successives
variables   a, b, i, res : entiers naturels
début
      # lecture des données
  Entrer ( a, b )
      # initialisation et boucle de calcul
  res ← 0
  pour i de 1 à b
  faire    res ← res + a
  fin_pour
      # affichage du résultat
  Afficher ( res )
fin
```

Remarque. *On pourrait échanger les valeurs de* a *et* b *lorsque* a *est inférieur à* b *pour effectuer un plus petit nombre de tours de boucle…*

Exercice 14. 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)).

Réponse. *Le principe est le même que pour l'exercice précédent si ce n'est qu'on utilise des produits (d'où l'initialisation à* 1 *de la variable* res*).*

L'algorithme est le suivant :

```
Algorithme exponentiationParMultiplicationsSuccessives
# cet algorithme permet de calculer la valeur de a puissance b
# en effectuant des multiplications successives
variables   a, b, i, res : entiers naturels
début
      # lecture des données
  Entrer ( a, b )
      # initialisation
  res ← 1
      # boucle de calcul
  pour i de 1 à b
  faire    res ← res * a
  fin_pour
      # affichage du résultat
  Afficher ( res )
fin
```

Exercice 15. Calcul de factorielle

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

Réponse. *Il suffit d'utiliser une boucle* Pour *avec une variable de boucle* i *pour obtenir les entiers de* 1 *à* n *et ranger au fur et à mesure leur produit dans la variable résultat.*

L'algorithme est le suivant :

```
Algorithme factorielle
# cet algorithme permet de calculer la factorielle d'un entier naturel
# entré au clavier
variables   n, i, fact : entiers naturels
début
      # lecture des données
  Entrer ( n )
      # initialisation
  fact ← 1
      # boucle de calcul
  pour i de 1 à n
  faire    fact ← fact * i
  fin_pour
      # affichage du résultat
  Afficher ( fact )
fin
```

Exercice 16. Somme des entiers de 1 à n

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

Réponse. *Là encore, la boucle* Pour *permet d'obtenir la séquence d'entiers désirée, dont la somme est cumulée dans la variable résultat.*

L'algorithme est le suivant :

```
Algorithme sommeEntiers
# cet algorithme permet de calculer la somme des entiers naturels
# compris entre 1 et n
variables   n, i, somme : entiers naturels
début
      # lecture des données
  Entrer ( n )
      # initialisation et boucle de calcul
  somme ← 0
      # boucle de calcul
  pour i de 1 à b
  faire    somme ← somme + i
  fin_pour
      # affichage du résultat
  Afficher ( somme )
fin
```

On peut naturellement donner une version plus rapide de cet algorithme, à partir de la formule donnant directement la valeur de cette somme :

```
Algorithme sommeEntiersVersion2
# cet algorithme permet de calculer la somme des entiers naturels
# compris entre 1 et n
variables   n : entiers naturels
début
      # lecture des données
  Entrer ( n )
```

```
        # affichage du résultat
   Afficher ( (n * (n + 1)) div 2 )
fin
```

Exercice 17. Afficher les diviseurs d'un entier

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

Réponse. *La boucle* `Pour` *vient encore à notre rescousse ici (notons qu'il est inutile cependant de parcourir l'intervalle* `[n/2 + 1, n]`*). Attention, si* `n = 0`*, tous les entiers divisent* `n` *!...*

L'algorithme est le suivant :

```
Algorithme diviseursOrdreCroissant
# cet algorithme permet d'afficher les diviseurs d'un entier naturel
# par ordre croissant
variables   n, diviseur : entiers naturels
début
        # lecture des données
   Entrer ( n )
        # cas où n est nul
   si ( n = 0 )
   alors Afficher ( "Tous les entiers sont diviseurs de 0" )
        # cas général
   sinon
             # boucle de parcours, si diviseur divise n, on l'affiche
        pour diviseur de 1 à n div 2
        faire    si ( n mod diviseur = 0 )
                 alors    Afficher ( diviseur )
                 fin_si
        fin_pour
        Afficher ( n )
   fin_si
fin
```

Exercice 18. 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.

Réponse. *Il suffit de calculer la somme des diviseurs de l'entier* `n` *(l'algorithme ressemble au précédent : on cumule les diviseurs au lieu de les afficher…).*

L'algorithme est le suivant :

```
Algorithme nombreParfait
# cet algorithme permet de déterminer si un nombre est parfait
variables   n, diviseur, somme : entiers naturels
début
        # lecture des données
   Entrer ( n )
        # cas où n est nul
   si ( n = 0 )
   alors Afficher ( "le nombre 0 n'est pas parfait" )
        # cas général
   sinon
             # initialisation de la somme des diviseurs
        somme ← 0
             # boucle de parcours
        pour diviseur de 1 à n div 2
```

```
        faire    si ( n mod diviseur = 0 )
                 alors    somme ← somme + diviseur
                 fin_si
        fin_pour

             # affichage du résultat
        si ( n = somme )
        alors    Afficher ( "le nombre ", n, " est parfait" )
        sinon    Afficher ( "le nombre ", n, " n'est pas parfait" )
        fin_si

   fin_si

fin
```

Exercice 19. Maximum d'une suite d'entiers

Écrire un algorithme permettant de saisir une suite d'entiers naturels non nuls 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.

Réponse. *Il suffit d'utiliser une variable* `maximum` *servant à mémoriser le maximum courant. Ce maximum peut être initialisé à* 0 *car nous considérons des entiers naturels. Si ce maximum vaut toujours* 0 *à la fin de l'algorithme, c'est que la suite était vide (l'utilisateur a entré directement la valeur* 0*).*

L'algorithme est le suivant :

```
Algorithme maximumSuite
# cet algorithme permet de saisir une suite d'entiers naturels terminée
# par 0 et d'afficher ensuite la valeur maximale de la suite
variables   n, maximum : entiers naturels
début
        # initialisation maximum
   maximum ← 0
        # boucle de lecture et calcul à la volée du maximum
   Entrer ( n )
   tantque ( n ≠ 0 ) faire
        si ( maximum < n )
        alors    maximum ← n
        fin_si
        Entrer ( n )
   fin_tantque
        # affichage résultat, avec test du cas « suite vide »
   si ( maximum = 0 )
   alors    Afficher ( "suite d'entiers vide." )
   sinon    Afficher ( maximum )
   fin_si
fin
```

Exercice 20. 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 !...)

Réponse. *Même principe que pour l'exercice précédent, en faisant attention au cas où la liste entrée est vide… L'algorithme est le suivant :*

```
Algorithme moyenneSuite
# cet algorithme permet de saisir une suite d'entiers naturels terminée
# par 0 et d'afficher ensuite la moyenne de la suite
variables   n, nb, somme : entiers naturels
début
```

```
        # initialisations
   nb ← 0
   somme ← 0
        # boucle de lecture et calcul à la volée du maximum
   Entrer ( n )
   tantque ( n ≠ 0 ) faire
        somme ← somme + n
        nb ← nb + 1
        Entrer ( n )
   fin_tantque
        # affichage résultat, avec test du cas « liste vide »
   si ( nb = 0 )
   alors    Afficher ( "liste vide, pas de moyenne." )
   sinon    Afficher ( somme / nb )
   fin_si
fin
```

Exercice 21. 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.

Réponse. *On utilise un booléen* `croissant`*, initialisé à* `vrai`*, qui basculera à* `faux` *si l'on trouve deux entiers consécutifs non ordonnés par ordre croissant (on mémorisera donc la valeur lue précédemment dans une variable* `précédent`*). Attention, ce test ne peut se faire que lorsqu'on a lu au moins deux valeurs…*

L'algorithme est le suivant :

```
Algorithme suiteCroissante
# cet algorithme permet de vérifier si une suite d'entiers naturels
# terminée par 0 est ou non croissante
variables   n, précédent : entiers naturels
            croissant : booléen
début
        # initialisations
   croissant ← vrai
        # lecture premier valeur
   Entrer ( n )
        # lecture deuxième valeur, si la suite n'est pas terminée
   si ( n ≠ 0 )
   alors    précédent ← n
            Entrer ( n )
   fin_si
        # boucle de lecture et vérification de la croissance
   tantque ( n ≠ 0 ) faire
        si ( précédent > n )
        alors croissant ← faux
        fin_si
            # on sauvegarde n et on passe à la valeur suivante
        précédent ← n
        Entrer ( n )
   fin_tantque
        # affichage résultat
   si ( précédent = vrai )
   alors    Afficher ( "suite croissante" )
   sinon    Afficher ( "suite non croissante" )
   fin_si
fin
```

Exercice 22. 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.

Réponse. *On calcule le PGCD en utilisant le célèbre algorithme d'Euclide. Le PPCM s'en déduit aisément ensuite (à condition de ne pas avoir écrasé les valeurs des deux entiers… On les sauvegardera donc dans deux variables auxiliaires* `aa` *et* `bb`*).*

L'algorithme est le suivant :

```
Algorithme pgcdPpcm
# cet algorithme permet de calculer le PGCD et le PPCM de deux entiers
# naturels non nuls entrés au clavier
variables   a, b, aa, bb, pgcd, ppcm : entiers naturels
début
      # lecture des données
  Entrer ( a, b )
      # initialisations
  aa ← a
  bb ← b
  reste ← aa mod bb
      # boucle de calcul
  tantque ( reste ≠ 0 ) faire
      aa ← bb
      bb ← reste
      reste ← aa mod bb
  fin_tantque
      # calcul du pgcd et du ppcm
  pgcd ← bb
  ppcm ← ( a * b ) div pgcd
      # affichage résultat
  Afficher ( pgcd, ppcm )
fin
```

Exercice 23. Nombre premier

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

Réponse. *Il suffit de chercher un diviseur de* `n` *dans l'intervalle* `[2,RacineCarrée(n)]`*. Dès qu'un tel diviseur est trouvé, le nombre* `n` *n'est pas premier. On utilisera donc la structure* `tantque` *qui permet d'arrêter la recherche dès qu'un diviseur est découvert.*

L'algorithme est le suivant :

```
Algorithme nombrePremier
# cet algorithme permet de déterminer si un entier naturel entré au clavier
# est premier
variables   n, diviseur : entiers naturels
            rac : réel
début
      # lecture des données
  Entrer ( n )
      # initialisations
  rac ← RacineCarrée ( n )            # fonction racine carrée
  diviseur ← 2
      # boucle de recherche d'un diviseur
  tantque ( ( n mod diviseur ≠ 0 ) et ( diviseur ≤ rac ) ) faire
      diviseur ← diviseur + 1
  fin_tantque
```

```
        # affichage résultat
   si ( diviseur > rac )
   alors    Afficher ( "Le nombre est premier" )
   sinon    Afficher ( "Le nombre n'est pas premier" )
   fin_si
fin
```

Exercice 24. Nombres premiers strictement inférieurs à 100

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

Réponse. *Il suffit « d'inclure » l'algorithme précédent dans une boucle de parcours. On obtient alors :*

```
Algorithme nombresPremiersStrictementInférieursA100
# cet algorithme permet d'afficher la liste de tous les nombres premiers
# strictement inférieurs à 100
variables   i, n, diviseur : entiers naturels
            rac : réel
début
        # boucle principale
   pour i de 2 à 99
   faire
            # initialisations
        rac ← RacineCarrée ( n )            # fonction racine carrée
        diviseur ← 2
            # boucle de recherche d'un diviseur
        tantque ( ( n mod diviseur ≠ 0 ) et ( diviseur ≤ rac ) ) faire
            diviseur ← diviseur + 1
        fin_tantque
            # affichage résultat
        si ( diviseur > rac )
        alors    Afficher ( n )
        fin_si
   fin_pour
fin
```

Exercice 25. 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.

Réponse. *Il suffit de modifier l'algorithme précédent, en mémorisant le dernier nombre premier trouvé. On obtient alors :*

```
Algorithme nombresPremiersJumeaux
# cet algorithme permet d'afficher la liste de tous les nombres premiers
# inférieurs à 100
variables   i, n, nprec, diviseur : entiers naturels
            rac : réel
début
        # initialisation
   nprec ← 2
        # boucle principale
   pour i de 3 à 999
   faire
            # initialisations
        rac ← RacineCarrée ( n )            # fonction racine carrée
```

```
        diviseur ← 2
             # boucle de recherche d'un diviseur
        tantque ( ( n mod diviseur ≠ 0 ) et ( diviseur ≤ rac ) ) faire
             diviseur ← diviseur + 1
        fin_tantque
             # si n premier
        si ( diviseur > rac )
        alors          # jumeau avec nprec ?
                  si ( n - nprec = 2 )
                  alors    Afficher ( nprec, n )
                  fin_si
                       # on met à jour nprec
                  nprec ← n
        fin_si
   fin_pour
fin
```

Exercice 26. 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).

Réponse. *Il suffit d'utiliser une boucle* pour *calculant un terme en fonction du terme précédent.*

L'algorithme est le suivant :

```
Algorithme niemeNombreSuite
# cet algorithme permet de calculer la n-ième valeur d'une suite de la
# forme u_n = au_n-1 + b, u_0 = c
variables   a, b, c, n, un, i : entiers naturels
début
        # lecture des données
   Entrer ( a, b, c, n )
        # initialisations
   un ← c
        # boucle de calcul
   pour i de 1 à n
   faire    un ← a * un + b
   fin_pour
        # affichage résultat
   Afficher ( un )
fin
```

Exercice 27. 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).

Réponse. *On applique le principe vu dans l'exercice précédent, si ce n'est que nous avons dans ce cas deux cas de base (*n = 0 *ou* n = 1*).*

L'algorithme est le suivant :

```
Algorithme nombreFibonacci
# cet algorithme permet de calculer le nombre de Fibonacci F(n)
variables   n, i, fibo, moins1 : entiers naturels
début
        # lecture des données
   Entrer ( n )
```

```
        # initialisations
   fibo ← 1
   moins1 ← 0

        # test si cas simple ( n = 0 )
   si ( n = 0 )
   alors     Afficher ( 0 )

   sinon          # boucle de calcul pour le cas général
             pour i de 1 à n
             faire     fibo ← fibo + moins1
                       moins1 ← fibo - moins1
             fin_pour

                  # affichage résultat
             Afficher ( fibo )

   fin_si
fin
```

Remarquons ici les opérations du corps de boucle `pour` *: nous nous sommes permis d'utiliser un artifice évitant l'utilisation d'une variable supplémentaire. Si nous avons effectué par exemple 3 tours de boucles, nous avons* `moins1 = 2` *et* `fibo = 3` *(on a calculé* `F(4)`*).*

Au 4ème tour, nous faisons `fibo ← 3 + 2 = 5`*, puis* `moins1 ← 5 - 2 = 3`*, ce qui est bien correct.*

Exercice 28. 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.

Réponse. *Un premier algorithme « naturel » (?) consiste à générer tous les nombres à trois chiffres à l'aide de trois boucles imbriquées et de vérifier pour chacun si la somme de ses chiffres vaut 5.*

On obtient alors l'algorithme suivant :

```
Algorithme nombresTroisChiffresMultiples5
# cet algorithme permet d'afficher par ordre croissant tous les nombres
# à 3 chiffres dont la somme des chiffres est multiple de 5
variables   ch1, ch2, ch3, nombre : entiers naturels
début
        # 3 boucles imbriquées, une par chiffre…
   pour ch1 de 1 à 9 faire
        pour ch2 de 0 à 9 faire
             pour ch3 de 0 à 9 faire
                       # test somme des chiffres
                  si ( ( ch1 + ch2 + ch3 ) mod 5 = 0 )
                  alors
                            # on affiche le nombre
                       nombre ← ( 100 * ch1 ) + ( 10 * ch2 ) + ch3
                       Afficher ( nombre )
                  fin_si
             fin_pour
        fin_pour
   fin_pour
fin
```

*On note que cet algorithme effectue 9 * 10 * 10 = 900 tours de la boucle la plus interne (qui teste si le nombre est acceptable). Or, il n'existe que 180 nombres à trois chiffres vérifiant notre propriété… Nous faisons donc 720 tours « inutiles », ou en tout cas qui ne génèrent pas de nombre acceptable. Peut-on éviter de passer par ces nombres inutiles ?*

*La réponse est oui. Il suffit d'observer que lorsque les 2 premiers chiffres sont connus, il existe exactement 2 valeurs du troisième qui conduisent à un nombre acceptable (d'où le 180 = 9 * 10 * 2), et ces 2 valeurs sont aisément calculables… d'où la modification suivante de notre algorithme :*

```
        # 2 boucles imbriquées, sur les 2 premiers chiffres…
   pour ch1 de 1 à 9 faire
        pour ch2 de 0 à 9 faire
             si (ch1 + ch2 ≤ 5)
             alors
                  nombre ← (100 * ch1) + (10 * ch2) + 5 - ch1 - ch2
                  Afficher ( nombre )
                  nombre ← (100 * ch1) + (10 * ch2) + 10 - ch1 - ch2
                  Afficher ( nombre )
             sinon
                  si (ch1 + ch2 ≤ 10)
                  alors
                       nombre ← (100*ch1) + (10 * ch2) + 10 - ch1 - ch2
                       Afficher ( nombre )
                       nombre ← (100 * ch1) + (10 * ch2) + 15 - ch1 - ch2
                       Afficher ( nombre )
                  sinon
                       si (ch1 + ch2 ≤ 15)
                       alors
                            nombre ← (100*ch1) + (10*ch2) + 15 - ch1 - ch2
                            Afficher ( nombre )
                            nombre ← (100*ch1) + (10*ch2) + 20 - ch1 - ch2
                            Afficher ( nombre )
                       sinon
                                 # cas où ch1 + ch2 > 15
                            nombre ← (100*ch1) + (10*ch2) + 20 - ch1 - ch2
                            Afficher ( nombre )
                            nombre ← (100*ch1) + (10*ch2) + 25 - ch1 - ch2
                            Afficher ( nombre )
                       fin_si
                  fin_si
             fin_si
        fin_pour
   fin_pour
```

Chapitre 5. 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. Chaque exercice est accompagné d'un corrigé et d'une traduction de celui-ci sous forme d'un script Python.

5.1. Fonctions

5.1.1. Images, antécédents

Exercice 29. 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).

Réponse. *L'algorithme est le suivant :*

```
Algorithme imageFonction

# Cet algorithme calcule l'image d'un nombre x par une fonction du type
# f(x) = ax² + bx + c (a, b et c donnés)

variables   a, b, c, x, res : réels

début

      # lecture données
  Entrer ( a, b, c, x )

      # calcul
  res ← a * x * x + b * x + c

      # affichage résultat
  Afficher ( res )

fin
```

Le script Python correspondant est le suivant :

```
# imageFonction

# Ce script calcule l'image d'un nombre x par une fonction du type
# f(x) = ax2 + bx + c (a, b et c donnés)

# lecture données
a = int(input("a = "))
b = int(input("b = "))
c = int(input("c = "))
x = int(input("x = "))

# calcul
res = a * x * x + b * x + c

# affichage résultat
print( res )
```

Exercice 30. 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).

Réponse. *L'algorithme est le suivant :*

```
Algorithme antécédentFonction1
# Cet algorithme calcule l'antécédent d'un nombre y par une fonction affine
# f(x) = ax + b (a et b donnés)
variables   a, b, y, x : réels
début
       # lecture données
  Entrer ( a, b, y )
       # a est-il nul ?
  si ( a = 0 )
  alors         # cas ou a est nul
       si ( y = b )
       alors Afficher ( "Tous les réels sont antécédents" )
       sinon Afficher ( "Aucun antécédent" )
       fin_si
  sinon     # cas où a est non nul
       x ← ( y - b ) / a
       Afficher ( x )
fin_si
fin
```

Le script Python correspondant est le suivant :

```
# antécédentFonction1
# Ce script calcule l'antécédent d'un nombre y par une fonction affine
# f(x) = ax + b (a et b donnés)
# lecture données
a = int(input("a = "))
b = int(input("b = "))
y = int(input("y = "))
# a est-il nul ?
if (a==0):
    # cas ou a est nul
    if (y==b):
        print("Tous les réels sont antécédents")
    else:
        print("Aucun antécédent")
else:
    # cas où a est non nul
    x = (y - b) / a
    print(x)
```

Exercice 31. 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).

Réponse. *L'algorithme est le suivant :*

```
Algorithme antécédentFonction2
# Cet algorithme calcule l'antécédent d'un nombre y par une fonction du
# type f(x) = ax² + b (a et b donnés)
variables   a, b, y, x : réels
début
       # lecture données
  Entrer ( a, b, y )
       # a est-il nul ?
  si ( a = 0 )
  alors         # cas ou a est nul
```

```
        si ( y = b )
        alors Afficher ( "Tous les réels sont antécédents" )
        sinon Afficher ( "Aucun antécédent" )
        fin_si
   sinon     # cas où a est non nul
        x ← RacineCarrée ( ( y - b ) / a )
        Afficher ( x )
fin_si
fin
```

Le script Python correspondant est le suivant :

```
# antécédentFonction2
# Ce script calcule l'antécédent d'un nombre y par une fonction affine
# f(x) = ax^2 + b (a et b donnés)
import math
# lecture données
a = int(input("a = "))
b = int(input("b = "))
y = int(input("y = "))
# a est-il nul ?
if (a==0):
    # cas ou a est nul
    if (y==b):
        print("Tous les réels sont antécédents")
    else:
        print("Aucun antécédent")
else:
    # cas où a est non nul
    x = math.sqrt((y - b) / a)
    print(x)
```

5.1.2. Résolution d'équations

Exercice 32. 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).

Réponse. *L'algorithme est le suivant :*

```
Algorithme résolutionEquationDegré1
# Cet algorithme résout une équation du premier degré, ax + b = c
# (a, b et c donnés)
variables   a, b, c, x : réels
début
        # lecture données
   Entrer ( a, b, c )
        # a est-il nul ?
   si ( a = 0 )
   alors          # cas ou a est nul
        si ( b = c )
        alors Afficher ( "Tous les réels sont solution" )
        sinon Afficher ( "Aucune solution" )
        fin_si
   sinon     # cas où a est non nul
        x ← ( c - b ) / a
        Afficher ( x )
fin_si
fin
```

Le script Python correspondant est le suivant :

```
# résolutionEquationDegré1

# Ce script résout une équation du premier degré, ax + b = c
# (a, b et c donnés)

# lecture données
a = int(input("a = "))
b = int(input("b = "))
c = int(input("c = "))

# a est-il nul ?
if (a==0):
    # cas ou a est nul
    if (b==c):
        print("Tous les réels sont solution")
    else:
        print("Aucune solution")

else:
    # cas où a est non nul
    x = (c - b) / a
    print(x)
```

Exercice 33. 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).

Réponse. *Le principe est d'obtenir itérativement des encadrements de plus en plus serrés, en partant de l'encadrement initial [0,n]. À chaque étape, il suffit de vérifier si le carré du milieu de l'intervalle est inférieur ou non à x pour obtenir un encadrement deux fois plus précis (le demi-intervalle gauche ou droit de l'intervalle précédent).*

On obtient ainsi l'algorithme suivant :

```
Algorithme encadrementRacine

# Cet algorithme donne un encadrement de racine(x), x donné, en procédant
# par dichotomie, la précision souhaitée étant également donnée.

variables   x, inf, sup, milieu, precision : réels

début
       # lecture données
  Entrer ( x, precision )
       # initialisations
  inf ← 0
  sup ← x
       # boucle de traitement, tant que la précision souhaitée
       # n'est pas atteinte
  tantque ( sup - inf > precision ) faire
            # milieu de l'intervalle
       milieu ← ( sup + inf ) / 2
            # on garde le demi-intervalle gauche ou droit ?
       si ( milieu * milieu > x )
       alors sup ← milieu
       sinon inf ← milieu
  fin_tantque
       # affichage résultat
  Afficher (inf, sup)
fin
```

Le script Python correspondant est le suivant :

```
# encadrementRacine

# Ce script donne un encadrement de racine(x), x donné, en procédant
```

```
# par dichotomie, la précision souhaitée étant également donnée.

# lecture données
x = float(input("x = "))
precision = float(input("precision = "))

# initialisations
inf = 0
sup = x

# boucle de traitement, tant que la précision souhaitée
# n'est pas atteinte
while (sup - inf > precision):

    # milieu de l'intervalle
    milieu = (sup + inf)/2

    # on garde le demi-intervalle gauche ou droit ?
    if (milieu * milieu > x):
        sup = milieu
    else:
        inf = milieu

# affichage résultat
print(inf," < racine(", x,") < ", sup)
```

5.1.3. Fonctions de référence

Exercice 34. Tracé de courbe

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

Réponse. *On prend pour exemple la fonction carré, les autres s'en déduisent facilement. La courbe est dessinée point par point sur l'intervalle `[alpha, beta]`, en ayant choisi une valeur de pas.*

L'algorithme est le suivant :

```
Algorithme tracéCourbeFonctionCarré

# Cet algorithme trace la courbe de la fonction « carré » sur l'intervalle
# [ alpha, beta ], alpha et beta donnés

constante   pas = 0.05    # par exemple…

variables   alpha, beta, X, Y : réels

début

       # lecture données
  Entrer ( alpha, beta )

       # initialisation
  X ← alpha

       # boucle de parcours de l'intervalle [ alpha, beta ]
  tantque ( X <= beta ) faire

            # calcul de Y - c'est ici que l'on peut
            # adapter l'algorithme pour dessiner d'autres courbes…
       Y ← X * X

            # dessin du point (X, Y)
       DessinerPoint ( X, Y )

            # on passe à l'abscisse suivante
       X ← X + pas

  fin_tantque

fin
```

Remarque. *On pourra ici utiliser avantageusement la possibilité offerte par Algobox de définir une fonction paramètre F1, qui permet de modifier la fonction considérée sans modifier l'algorithme.*

Le script Python correspondant est le suivant :

```
# traceCourbeFonctionCarre

# Ce script trace la courbe de la fonction « carré » sur l'intervalle
# [ alpha, beta ], alpha et beta donnés

import turtle

turtle.reset()
turtle.title("Trace courbe")
turtle.color('red')
turtle.width(10)

# constante pas
pas = 0.05

# lecture données
alpha = float(input("alpha = "))
beta = float(input("beta = "))

# initialisation
x = alpha

# boucle de parcours de l'intervalle [ alpha, beta ]
while (x <= beta):

    # calcul de Y - c'est ici que l'on peut
    # adapter l'algorithme pour dessiner d'autres courbes…
    y = x * x

    # dessin du point (X, Y)
    turtle.penup()
    turtle.goto(20*x,20*y)
    turtle.pendown()
    turtle.dot(3,'blue')

    # on passe à l'abscisse suivante
    x = x + pas
```

5.1.4. Polynômes de degré 2

Exercice 35. 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), sur un intervalle de la forme `[alpha, beta]` (`alpha` et `beta` donnés).

Réponse. *Il suffit d'adapter en conséquence l'algorithme* `tracéCourbeFonctionCarré`.

5.1.5. Fonctions homographiques

Exercice 36. Tracé de courbe d'une fonction homographique

Écrire un algorithme permettant de tracer la courbe d'une fonction homographique $f(x) = (ax + b)/(cx + d)$, sur un intervalle de la forme `[alpha, beta]` (`alpha` et `beta` donnés).

Réponse. *Il suffit d'adapter en conséquence l'algorithme* `tracéCourbeFonctionCarré`. *Il est cependant nécessaire de faire attention à la division par zéro…*

5.1.6. Inéquations

Exercice 37. Résolution graphique d'inéquation 1

Écrire un algorithme permettant de résoudre graphiquement l'inéquation $f(x) < k$, sur un intervalle de la forme `[alpha, beta]` (`alpha` et `beta` donnés). (On pourra par exemple tracer la courbe de la fonction `f` en noir, et en rouge pour la partie satisfaisant l'inéquation…).

Réponse. *On adapte l'algorithme* `tracéCourbeFonctionCarré`*, en choisissant la couleur d'affichage de façon adéquate. Pour la fonction* `carré`*, on obtient :*

```
Algorithme RésolutionGraphiqueInéquationFonctionCarré
# Cet algorithme trace la courbe de la fonction « carré » sur l'intervalle
```

```
# [ alpha, beta ], alpha et beta donnés, en affichant en rouge les valeurs
# supérieures ou égales à une valeur k donnée

constante   pas = 0.05    # par exemple…

variables   alpha, beta, k, X, Y : réels

début

      # lecture données
  Entrer ( alpha, beta, k )

      # initialisation
  X ← alpha

      # boucle de parcours de l'intervalle [ alpha, beta ]
  tantque ( X <= beta ) faire

          # calcul de Y - c'est ici que l'on peut
          # adapter l'algorithme pour dessiner d'autres courbes…
      Y ← X * X

          # couleur du point
      si (Y < k)
      alors CouleurTrait ( "bleu" )
      sinon CouleurTrait ( "rouge" )

          # dessin du point (X, Y)
      DessinerPoint ( X, Y )

          # on passe à l'abscisse suivante
      X ← X + pas

  fin_tantque

fin
```

Le script Python correspondant est le suivant :

```
# traceCourbeFonctionCarre

# Ce script trace la courbe de la fonction « carré » sur l'intervalle
# [ alpha, beta ], alpha et beta donnés

import turtle

turtle.reset()
turtle.title("Trace courbe")
turtle.color('red')
turtle.width(10)

# constante pas
pas = 0.05

# lecture données
alpha = float(input("alpha = "))
beta = float(input("beta = "))
k = float(input("k = "))

# initialisation
x = alpha

# boucle de parcours de l'intervalle [ alpha, beta ]
while (x <= beta):

    # calcul de Y - c'est ici que l'on peut
    # adapter l'algorithme pour dessiner d'autres courbes…
    y = x * x

    # dessin du point (X, Y)
    turtle.penup()
    turtle.goto(20*x,20*y)
    turtle.pendown()
    if (y < k):
        turtle.dot(3,'red')
    else:
        turtle.dot(3,'black')

    # on passe à l'abscisse suivante
    x = x + pas
```

Exercice 38. Résolution graphique d'inéquation 2

Écrire un algorithme permettant de résoudre graphiquement l'inéquation `f(x) < g(x)`, sur un intervalle de la forme `[alpha, beta]` (`alpha` et `beta` donnés). (On pourra par exemple tracer les courbes des fonctions `f` et `g` en rouge, et en bleu pour la partie satisfaisant l'inéquation…).

Réponse. *Pour les fonctions* `f(x) = x * x` *et* `g(x) = 2x + 5` *par exemple, on obtient :*

```
Algorithme RésolutionGraphiqueInéquationFonctionsFetG

# Cet algorithme résout graphiquement l'inéquation f(x) < g(x) sur
# l'intervalle [ alpha, beta ], alpha et beta donnés, en affichant en bleu
# les images des points satisfaisant l'inéquation

constante   pas = 0.05    # par exemple…

variables   alpha, beta, X, fX, gX : réels

début

      # lecture données
  Entrer ( alpha, beta )

      # initialisation
  X ← alpha

      # boucle de parcours de l'intervalle [ alpha, beta ]
  tantque ( X <= beta ) faire

          # calcul de fX et gX - c'est ici que l'on peut
          # adapter l'algorithme pour dessiner d'autres courbes…
      fX ← X * X
      gX ← 2 * X + 5

          # couleur du point
      si (fX < gX)
      alors CouleurTrait ( "bleu" )
      sinon CouleurTrait ( "rouge" )

          # dessin des deux points (X, f(X)) et (X, g(X))
      DessinerPoint ( X, fX )
      DessinerPoint ( X, gX )

          # on passe à l'abscisse suivante
      X ← X + pas

  fin_tantque

fin
```

Le script Python correspondant est le suivant :

```
# traceCourbeFonctionCarre

# Ce script trace la courbe de la fonction « carré » sur l'intervalle
# [ alpha, beta ], alpha et beta donnés

import turtle

turtle.reset()
turtle.title("Trace courbe")
turtle.color('red')
turtle.width(10)

# constante pas
pas = 0.05

# lecture données
alpha = float(input("alpha = "))
beta = float(input("beta = "))

# initialisation
x = alpha

# boucle de parcours de l'intervalle [ alpha, beta ]
while (x <= beta):

    # calcul de fX et gX - c'est ici que l'on peut
    # adapter l'algorithme pour dessiner d'autres courbes…
    fx = x * x
```

```
    gx = 2 * x + 5

    # couleur du point
    if (fx < gx):
        col = 'blue'
    else:
        col = 'red'

    # dessin des points (X, fX) et (X, gX)
    turtle.penup()
    turtle.goto(20*x,20*fx)
    turtle.pendown()
    turtle.dot(3,col)
    turtle.penup()
    turtle.goto(20*x,20*gx)
    turtle.pendown()
    turtle.dot(3,col)

    # on passe à l'abscisse suivante
    x = x + pas
```

Exercice 39. Résolution d'inéquation

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

Réponse. *L'algorithme est le suivant :*

```
Algorithme résolutionInéquation

# Cet algorithme résout une inéquation de la forme ax + b < cx + d
# (a, b, c et d donnés)

variables   a, b, c, d, x : réels

début

       # lecture données
  Entrer ( a, b, c, d )

       # test a = c ?
  si ( a = c )
  alors

            # cas où a = c, 2 cas à considérer…
       si ( b < d )
       alors Afficher ( "Tous les réels sont solution" )
       sinon Afficher ( "Pas de solution" )
       fin_si

  sinon
            # cas général, résolution
       x ← ( d - b ) / ( a - c )

            # affichage résultat
       Afficher ( "Solution : x < ", x )
  fin_si

fin
```

Le script Python correspondant est le suivant :

```
# résolutionInéquation

# Ce script résout une inéquation de la forme ax + b < cx + d
# (a, b, c et d donnés)

# lecture données
a = int(input("a = "))
b = int(input("b = "))
c = int(input("c = "))
d = int(input("d = "))

# test a = c ?
if (a==c):

    # cas où a = c, 2 cas à considérer…
```

```
    if (b<d):
        print( "Tous les réels sont solution" )
    else:
        print( "Pas de solution" )

else:
    # cas général, résolution
    x = (d - b) / (a - c)

    # affichage résultat
    print("Solution : x < ", x )
```

5.1.7. Trigonométrie

Exercice 40. 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>.

Réponse. *L'algorithme est le suivant :*

```
Algorithme calculSinusCosinus

# Cet algorithme calcule les sinus et cosinus des angles <OAB> et <OBA>
# pour un triangle rectangle de la forme OAB avec O = (0,0), A = (a,0)
# et B = (0,b) (a et b donnés)

variables   a, b, diagonale : réels

début

      # lecture données
  Entrer ( a, b )

      # calcul de la diagonale
  diagonale ← RacineCarrée ( a * a + b * b )

      # affichage résultats
  Afficher ( "sinus(<OAB>) = ", b / diagonale )
  Afficher ( "cosinus(<OAB>) = ", a / diagonale )
  Afficher ( "sinus(<OBA>) = ", a / diagonale )
  Afficher ( "cosinus(<OBA>) = ", b / diagonale )

fin
```

Le script Python correspondant est le suivant :

```
# calculSinusCosinus

# Ce script calcule les sinus et cosinus des angles <OAB> et <OBA>
# pour un triangle rectangle de la forme OAB avec O = (0,0), A = (a,0)
# et B = (0,b) (a et b donnés)

import math

# lecture données
a = float(input("a = "))
b = float(input("b = "))

# calcul de la diagonale
diagonale = math.sqrt(a*a + b*b)

# affichage résultats
print("sinus(<OAB>) = ", b / diagonale)
print("cosinus(<OAB>) = ", a / diagonale)
print("sinus(<OBA>) = ", a / diagonale)
print("cosinus(<OBA>) = ", b / diagonale)
```

5.2. Géométrie

5.2.1. Coordonnées d'un point du plan

Exercice 41. Longueur d'un segment

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

Réponse. *L'algorithme est le suivant :*

```
Algorithme longueurSegment
# Cet algorithme calcule la longueur d'un segment AB
# (XA, YA, XB, YB donnés)
variables   XA, XB, YA, YB, longueur : réels
début
      # lecture données
  Entrer ( XA, YA, XB, YB )
      # calcul de la longueur
  longueur ← RacineCarrée ( ((XB-XA) * (XB-XA)) + ((YB-YA) * (YB-YA)) )
      # affichage résultat
  Afficher ( longueur )
fin
```

Le script Python correspondant est le suivant :

```
# longueurSegment
# Ce script calcule la longueur d'un segment AB
# (XA, YA, XB, YB donnés)
import math
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
# calcul de la longueur
longueur = math.sqrt ( ((XB-XA) * (XB-XA)) + ((YB-YA) * (YB-YA)) )
# affichage résultat
print("longueur du segment : ", longueur)
```

Exercice 42. 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.

Réponse. *L'algorithme est le suivant :*

```
Algorithme coordonnéesMilieuSegment
# Cet algorithme calcule les coordonnées du milieu d'un segment AB
# (XA, YA, XB, YB donnés)
variables   XA, XB, YA, YB, Xmilieu, Ymilieu : réels
début
      # lecture données
  Entrer ( XA, YA, XB, YB )
      # calcul des coordonnées du milieu
  Xmilieu ← ( XB + XA ) / 2
  Ymilieu ← ( YB + YA ) / 2
      # affichage résultat
  Afficher ( Xmilieu, Ymilieu )
```

```
fin
```

Le script Python correspondant est le suivant :

```
# coordonnéesMilieuSegment

# Ce script calcule les coordonnées du milieu d'un segment AB
# (XA, YA, XB, YB donnés)

# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))

# calcul des coordonnées du milieu
Xmilieu = ( XB + XA ) / 2
Ymilieu = ( YB + YA ) / 2

# affichage résultat
print("coordonnées du milieu : ", Xmilieu, ",", Ymilieu)
```

5.2.2. Configurations du plan

Exercice 43. 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.

Réponse. *L'algorithme est le suivant :*

```
Algorithme périmètreAireRectangle

# Cet algorithme calcule le périmètre et l'aire d'un rectangle donné
# par ses longueur et largeur

variables   longueur, largeur, périmètre, aire : réels

début

      # lecture données
  Entrer ( longueur, largeur )

      # calcul périmètre et aire
  périmètre ← 2 * ( longueur + largeur )
  aire ← longueur * largeur

      # affichage résultat
  Afficher ( périmètre, aire )

fin
```

Le script Python correspondant est le suivant :

```
# périmètreAireRectangle

# Ce script calcule le périmètre et l'aire d'un rectangle donné
# par ses longueur et largeur

# lecture données
longueur = int(input("Longueur : "))
largeur = int(input("Largeur : "))

# calcul périmètre et aire
perimetre = 2 * ( longueur + largeur )
aire = longueur * largeur

# affichage résultat
print ("périmètre : ", perimetre, " - aire : ", aire )
```

Exercice 44. Périmètre et aire d'autres figures

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

Réponse. *Il suffit d'adapter en conséquence l'algorithme* `périmètreAireRectangle`*.*

Exercice 45. Est-ce un triangle rectangle ?

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

Réponse. *Grâce au théorème de Pythagore, on vérifie si un triangle ABC est rectangle en A, B ou C.*

L'algorithme est le suivant :

```
Algorithme estUnTriangleRectangle
# Cet algorithme vérifie si un triangle ABC est ou non un
# triangle rectangle
variables   XA, XB, XC, YA, YB, YC, longAB2, longAC2, longBC2 : réels
début
      # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC )
      # calcul des carrés des longueurs des côtés
  longAB2 ← (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
  longAC2 ← (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
  longBC2 ← (XB-XC)*(XB-XC) + (YB-YC)*(YB-YC)
      # rectangle en A ?
  si ( longBC2 = longAB2 + longAC2 )
  alors
      Afficher ( "ABC est rectangle en A" )
  sinon
          # rectangle en B ?
      si ( longAC2 = longAB2 + longBC2 )
      alors
          Afficher ( "ABC est rectangle en B" )
      sinon
              # rectangle en C ?
          si ( longAB2 = longBC2 + longAC2 )
          alors
              Afficher ( "ABC est rectangle en C" )
          sinon
              Afficher ( "ABC n'est pas un triangle rectangle" )
          fin_si
      fin_si
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# estUnTriangleRectangle
# Ce script vérifie si un triangle ABC est ou non un
# triangle rectangle
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
# calcul des carrés des longueurs des côtés
longAB2 = (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
longAC2 = (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
longBC2 = (XB-XC)*(XB-XC) + (YB-YC)*(YB-YC)
# rectangle en A ?
if ( longBC2 == longAB2 + longAC2 ):
    print( "ABC est rectangle en A" )
    # rectangle en B ?
elif ( longAC2 == longAB2 + longBC2 ):
    print( "ABC est rectangle en B" )
```

```
    # rectangle en C ?
elif ( longAB2 == longBC2 + longAC2 ):
    print( "ABC est rectangle en C" )
else:
    print( "ABC n'est pas un triangle rectangle" )
```

Exercice 46. Est-ce un triangle équilatéral ?

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

Réponse. *On vérifie que les trois côtés sont de même longueur.*

L'algorithme est le suivant :

```
Algorithme estUnTriangleEquilatéral
# Cet algorithme vérifie si un triangle ABC est ou non un
# triangle équilatéral
variables   XA, XB, XC, YA, YB, YC, longAB2, longAC2, longBC2 : réels
début
      # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC )
      # calcul des carrés des longueurs des côtés
  longAB2 ← ( (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
  longAC2 ← (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
  longBC2 ← ( (XB-XC)*(XB-XC) + (YB-YC)*(YB-YC)
      # trois côtés de même longueur ?
  si ( ( longAC2 = longAB2 ) ET ( longAC2 = longBC2 ) )
  alors
      Afficher ( "ABC est un triangle équilatéral" )
  sinon
      Afficher ( "ABC n'est pas un triangle équilatéral" )
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# estUnTriangleEquilatéral
# Ce script vérifie si un triangle ABC est ou non un
# triangle équilatéral
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
# calcul des carrés des longueurs des côtés
longAB2 = (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
longAC2 = (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
longBC2 = (XB-XC)*(XB-XC) + (YB-YC)*(YB-YC)
# trois côtés de même longueur ?
if ( ( longAC2 == longAB2 ) and ( longAC2 == longBC2 ) ):
    print( "ABC est un triangle équilatéral" )
else:
    print( "ABC n'est pas un triangle équilatéral" )
```

Exercice 47. Est-ce un triangle isocèle ?

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

Réponse. *On vérifie que ABC possède deux côtés qui sont de même longueur.*

L'algorithme est le suivant :

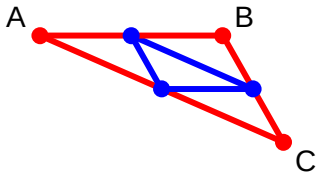
```
Algorithme estUnTriangleIsocèle
# Cet algorithme vérifie si un triangle ABC est ou non un
# triangle isocèle
variables   XA, XB, XC, YA, YB, YC, longAB2, longAC2, longBC2 : réels
début
      # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC )
      # calcul des carrés des longueurs des côtés
  longAB2 ← ( (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
  longAC2 ← (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
  longBC2 ← ( (XB-XC)*(XB-XC) + (YB-YC)*(YB-YC)
      # deux côtés de même longueur ?
  si ( ( longAC2 = longAB2 ) OU ( longAC2 = longBC2 )
          OU ( longAB2 = longBC2 ) )
  alors
      Afficher ( "ABC est un triangle isocèle" )
  sinon
      Afficher ( "ABC n'est pas un triangle isocèle" )
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# estUnTriangleIsocèle
# Ce script vérifie si un triangle ABC est ou non un
# triangle isocèle
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
# calcul des carrés des longueurs des côtés
longAB2 = (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
longAC2 = (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
longBC2 = (XB-XC)*(XB-XC) + (YB-YC)*(YB-YC)
# deux côtés de même longueur ?
if ( ( longAC2 == longAB2 ) or ( longAC2 == longBC2 )
     or ( longAB2 == longBC2 ) ):
    print( "ABC est un triangle isocèle" )
else:
    print( "ABC n'est pas un triangle isocèle" )
```

Exercice 48. 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.



Réponse. *L'algorithme est le suivant :*

```
Algorithme triangleMilieux

# Cet algorithme dessine en rouge un triangle ABC et en bleu le triangle
# joignant les milieux des trois côtés

variables   XA, XB, XC, YA, YB, YC : réels
            XmilAB, XmilBC, XmilAC, YmilAB, YmilBC, YmilBC : réels

début

      # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC )

      # calcul des coordonnées des milieux des côtés
  XmilAB ← ( XA + XB ) / 2
  YmilAB ← ( YA + YB ) / 2
  XmilAC ← ( XA + XC ) / 2
  YmilAC ← ( YA + YC ) / 2
  XmilBC ← ( XB + XC ) / 2
  YmilBC ← ( YB + YC ) / 2

      # dessin du triangle ABC
  Couleur ( "rouge" )
  TracerSegment ( XA, YA, XB, YB )
  TracerSegment ( XB, YB, XC, YC )
  TracerSegment ( XC, YC, XA, YA )

      # dessin du triangle des milieux
  Couleur ( "bleu" )
  TracerSegment ( XmilAB, YmilAB, XmilBC, YmilBC )
  TracerSegment ( XmilBC, YmilBC, XmilAC, YmilAC )
  TracerSegment ( XmilAC, YmilAC, XmilAB, YmilAB )

fin
```

Le script Python correspondant est le suivant :

```
# triangleMilieux

# Ce script dessine en rouge un triangle ABC et en bleu le triangle
# joignant les milieux des trois côtés

import turtle
turtle.reset()
turtle.title("Triangle des milieux")
turtle.width(10)

# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))

# calcul des coordonnées des milieux des côtés
XmilAB = ( XA + XB ) / 2
YmilAB = ( YA + YB ) / 2
XmilAC = ( XA + XC ) / 2
YmilAC = ( YA + YC ) / 2
XmilBC = ( XB + XC ) / 2
YmilBC = ( YB + YC ) / 2

# dessin du triangle ABC
turtle.color('red')
turtle.penup()
turtle.goto(XA,YA)
turtle.pendown()
turtle.goto(XB,YB)
turtle.goto(XC,YC)
turtle.goto(XA,YA)

# dessin du triangle des milieux
turtle.color('blue')
turtle.penup()
```

```
turtle.goto(XmilAB,YmilAB)
turtle.pendown()
turtle.goto(XmilBC,YmilBC)
turtle.goto(XmilAC,YmilAC)
turtle.goto(XmilAB,YmilAB)
```

Exercice 49. Est-ce un rectangle ?

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

Réponse. *On vérifie que les quatre angles sont des angles droits, grâce au théorème de Pythagore, en remarquant que dès que trois angles le sont, le quatrième l'est nécessairement…*

L'algorithme est le suivant :

```
Algorithme estUnRectangle
# Cet algorithme vérifie si un quadrilatère ABCD est ou non un rectangle
variables   XA, XB, XC, XD, YA, YB, YC, YD : réels
            longAB2, longBC2, longCD2, longAD2, longAC2, longBD2 : réels
début
       # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC, XD, YD )
       # calcul des carrés deslongueurs des quatre côtés
       # et des deux diagonales
  longAB2 ← (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
  longBC2 ← (XC-XB)*(XC-XB) + (YC-YB)*(YC-YB)
  longCD2 ← (XC-XD)*(XC-XD) + (YC-YD)*(YC-YD)
  longAD2 ← (XA-XD)*(XA-XD) + (YA-YD)*(YA-YD)
  longAC2 ← (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
  longBD2 ← (XB-XD)*(XB-XD) + (YB-YD)*(YB-YD)
            # non rectangle en A ?
  si (longBD2 ≠ longAB2 + longAD2 )
  alors
            Afficher ("ABCD n'est pas un rectangle - ",
                      " l'angle DAB n'est pas un angle droit" )
  sinon
            # non rectangle en B ?
       si (longAC2 ≠ longAB2 + longBC2 )
       alors
            Afficher ("ABCD n'est pas un rectangle - ",
                      " l'angle ABC n'est pas un angle droit" )
       sinon
                 # non rectangle en C ?
            si (longBD2 ≠ longBC2 + longCD2 )
            alors
                 Afficher ("ABCD n'est pas un rectangle - ",
                           " l'angle BCD n'est pas un angle droit" )
            sinon
                 Afficher ( "ABCD est bien un rectangle" )
            fin_si
       fin_si
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# estUnRectangle
# Ce script vérifie si un quadrilatère ABCD est ou non un rectangle
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
```

```
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
XD = float(input("XD : "))
YD = float(input("YD : "))

# calcul des carrés deslongueurs des quatre côtés
# et des deux diagonales
longAB2 = (XA-XB)*(XA-XB) + (YA-YB)*(YA-YB)
longBC2 = (XC-XB)*(XC-XB) + (YC-YB)*(YC-YB)
longCD2 = (XC-XD)*(XC-XD) + (YC-YD)*(YC-YD)
longAD2 = (XA-XD)*(XA-XD) + (YA-YD)*(YA-YD)
longAC2 = (XA-XC)*(XA-XC) + (YA-YC)*(YA-YC)
longBD2 = (XB-XD)*(XB-XD) + (YB-YD)*(YB-YD)

# non rectangle en A ?
if (longBD2 != longAB2 + longAD2 ):
    print("ABCD n'est pas un rectangle - ",
            " l'angle DAB n'est pas un angle droit" )

    # non rectangle en B ?
elif (longAC2 != longAB2 + longBC2 ):
    print("ABCD n'est pas un rectangle - ",
            " l'angle ABC n'est pas un angle droit" )

    # non rectangle en C ?
elif (longBD2 != longBC2 + longCD2 ):
    print("ABCD n'est pas un rectangle - ",
            " l'angle BCD n'est pas un angle droit" )

else:
    print( "ABCD est bien un rectangle" )
```

5.2.3. Droites

Exercice 50. Équation de droite donnée par deux points

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

Réponse. *On détermine le coefficient directeur et l'ordonnée à l'origine. Un cas particulier, lorsque XA = XB.*

L'algorithme est le suivant :

```
Algorithme équationDroite
# Cet algorithme détermine l'équation d'une droite donnée par deux de
# ses points
variables   XA, YA, XB, YB, coef, cste : réels
début
       # lecture données
  Entrer ( XA, YA, XB, YB )
       # cas particulier : XA = XB
  si ( XA = XB )
  alors Afficher ( "équation de la droite : x = ", XA )
       # cas général
  sinon
            # calcul du coefficient directeur
       coef ← ( YA - YB ) / ( XA - XB )
            # calcul de l'ordonnée à l'origine
       cste ← YA - coef * XA
            # affichage résultat
       Afficher ( "équation de la droite : y = ", coef, "x + ", cste )
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# équationDroite

# Ce script détermine l'équation d'une droite donnée par deux de
# ses points

# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))

# cas particulier : XA = XB
if ( XA == XB ):
    print( "équation de la droite : x = ", XA )

# cas général
else:
    # calcul du coefficient directeur
    coef = ( YA - YB ) / ( XA - XB )

    # calcul de l'ordonnée à l'origine
    cste = YA - coef * XA

    # affichage résultat
    print ( "équation de la droite : y = ", coef, "x + ", cste )
```

Exercice 51. 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.

Réponse. *On détermine le coefficient directeur et l'ordonnée à l'origine. Deux cas particuliers, lorsque XA = XB et lorsque YA = YB.*

L'algorithme est le suivant :

```
Algorithme équationDroitePerpendiculaire

# Cet algorithme détermine l'équation d'une droite passant par A et
# perpendiculaire à un segment AB

variables   XA, YA, XB, YB, coef, cste : réels

début

      # lecture données
  Entrer ( XA, YA, XB, YB )

      # cas particulier 1 : XA = XB
  si ( XA = XB )
  alors Afficher ( "équation de la droite : y = ", YA )

  sinon

           # cas particulier 2 : YA = YB
      si ( YA = YB )
      alors Afficher ( "équation de la droite : x = ", XA )

                # cas général
      sinon
                # calcul du coefficient directeur
           coef ← - 1 / ( ( YA - YB ) / ( XA - XB ) )

                # calcul de l'ordonnée à l'origine
           cste ← YA - coef * XA

                # affichage résultat
           Afficher ( "équation de la droite : y = ", coef, "x + ", cste )
      fin_si
  fin_si

fin
```

Le script Python correspondant est le suivant :

```
# équationDroitePerpendiculaire

# Ce script détermine l'équation d'une droite passant par A et
```

```
# perpendiculaire à un segment AB

# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))

# cas particulier 1 : XA = XB
if ( XA == XB ):
    print( "équation de la droite : y = ", YA )

# cas particulier 2 : YA = YB
elif ( YA == YB ):
    print( "équation de la droite : x = ", XA )

# cas général
else:
    # calcul du coefficient directeur
    coef = - 1 / ( ( YA - YB ) / ( XA - XB ) )

    # calcul de l'ordonnée à l'origine
    cste = YA - coef * XA

    # affichage résultat
    print ( "équation de la droite : y = ", coef, "x + ", cste )
```

Exercice 52. 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.

Réponse. *On détermine le coefficient directeur et l'ordonnée à l'origine. Deux cas particuliers, lorsque XA = XB et lorsque YA = YB.*

L'algorithme est le suivant :

```
Algorithme équationDroiteParallèle

# Cet algorithme détermine l'équation d'une droite passant par A et
# parallèle à un segment BC

variables   XA, YA, XB, YB, XC, YC, coef, cste : réels

début

      # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC )

      # cas particulier 1 : XB = XC
  si ( XB = XC )
  alors Afficher ( "équation de la droite : x = ", XA )

  sinon

          # cas particulier 2 : YB = YC
      si ( YA = YB )
      alors Afficher ( "équation de la droite : y = ", YA )

              # cas général
      sinon
              # calcul du coefficient directeur
          coef ← ( YB - YC ) / ( XB - XC )

              # calcul de l'ordonnée à l'origine
          cste ← YB - coef * XB

              # affichage résultat
          Afficher ( "équation de la droite : y = ", coef, "x + ", cste )
      fin_si
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# équationDroiteParallèle
```

```
# Ce script détermine l'équation d'une droite passant par A et
# parallèle à un segment BC

# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))

# cas particulier 1 : XB = XC
if ( XB == XC ):
    print( "équation de la droite : x = ", XA )

# cas particulier 2 : YB = YC
elif ( YA == YB ):
    print( "équation de la droite : y = ", YA )

# cas général
else:
    # calcul du coefficient directeur
    coef = ( YB - YC ) / ( XB - XC )

    # calcul de l'ordonnée à l'origine
    cste = YB - coef * XB

    # affichage résultat
    print ( "équation de la droite : y = ", coef, "x + ", cste )
```

Exercice 53. Droites parallèles ou sécantes ?

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

Réponse. *Il suffit de vérifier si les deux droites ont même coefficient directeur ou pas, en traitant à part le cas où l'une au moins des deux droites est verticale… L'algorithme est le suivant :*

```
Algorithme droitesParallèles

# Cet algorithme détermine si deux droites AB et CD données par deux de
# leurs points sont parallèles ou sécantes

variables   XA, YA, XB, YB, XC, YC, XD, YD, coefAB, coefCD : réels

début

      # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC, XD, YD )

      # une droite verticale ?
  si ( ( XA = XB ) ou ( XC = XD ) )
  alors    si ( ( XA = XB ) et ( XC = XD ) )
           alors Afficher ( "droites parallèles" )
           sinon Afficher ( "droites sécantes" )
           fin_si

  sinon
      # calcul des coefficients directeurs
      coefAB ← ( YA - YB ) / ( XA - XB )
      coefCD ← ( YC - YD ) / ( XC - XD )

          # affichage du résultat
      si ( coefAB = coefCD )
      alors Afficher ( "droites parallèles" )
      sinon Afficher ( "droites sécantes" )
      fin_si

  fin_si

fin
```

Le script Python correspondant est le suivant :

```
# droitesParallèles

# Ce script détermine si deux droites AB et CD données par deux de
```

```
# leurs points sont parallèles ou sécantes

# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
XD = float(input("XD : "))
YD = float(input("YD : "))

# une droite verticale ?
if ( ( XA == XB ) or ( XC == XD ) ):
    if ( ( XA == XB ) and ( XC == XD ) ):
        print( "droites parallèles" )
    else:
        print( "droites sécantes" )

else:
    # calcul des coefficients directeurs
    coefAB = ( YA - YB ) / ( XA - XB )
    coefCD = ( YC - YD ) / ( XC - XD )

    # affichage du résultat
    if ( coefAB == coefCD ):
        print( "droites parallèles" )
    else:
        print( "droites sécantes" )
```

Exercice 54. Droites perpendiculaires ?

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

Réponse. *Il suffit de calculer le produit des deux coefficients directeurs, en traitant à part le cas où l'une au moins des deux droites est verticale… L'algorithme est le suivant :*

```
Algorithme droitesPerpendiculaires

# Cet algorithme détermine si deux droites AB et CD données par deux de
# leurs points sont ou non perpendiculaires

variables   XA, YA, XB, YB, XC, YC, XD, YD, coefAB, coefCD : réels

début

       # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC, XD, YD )

       # une droite verticale ?
  si ( ( XA = XB ) ou ( XC = XD ) )
  alors     si ( ( ( XA = XB ) et ( YC = YD ) )
                    ou ( ( YA = YB ) et ( XC = XD ) ) )
            alors Afficher ( "droites perpendiculaires" )
            sinon Afficher ( "droites non perpendiculaires" )
            fin_si

  sinon
            # calcul des coefficients directeurs
       coefAB ← ( YA - YB ) / ( XA - XB )
       coefCD ← ( YC - YD ) / ( XC - XD )

            # affichage du résultat
       si ( coefAB * coefCD = -1 )
       alors Afficher ( "droites perpendiculaires" )
       sinon Afficher ( "droites non perpendiculaires" )
       fin_si

  fin_si

fin
```

Le script Python correspondant est le suivant :

```
# droitesPerpendiculaires
```

```
# Ce script détermine si deux droites AB et CD données par deux de
# leurs points sont ou non perpendiculaires

# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
XD = float(input("XD : "))
YD = float(input("YD : "))

# une droite verticale ?
if ( ( XA == XB ) or ( XC == XD ) ):
    if ( ( ( XA == XB ) and ( YC == YD ) )
  or ( ( YA == YB ) and ( XC == XD ) ) ):
        print( "droites perpendiculaires" )

    else:
        print( "droites non perpendiculaires" )

else:
    # calcul des coefficients directeurs
    coefAB = ( YA - YB ) / ( XA - XB )
    coefCD = ( YC - YD ) / ( XC - XD )

    # affichage du résultat
    if ( coefAB * coefCD == -1 ):
        print( "droites perpendiculaires" )
    else:
        print( "droites non perpendiculaires" )
```

Exercice 55. 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.

Réponse. *Il suffit de vérifier si les droites AB et AC ont même coefficient directeur ou pas, en traitant à part le cas où l'une au moins des deux droites est verticale… L'algorithme est le suivant :*

```
Algorithme troisPointsAlignés

# Cet algorithme détermine si trois points A, B et C sont ou non alignés

variables   XA, YA, XB, YB, XC, YC, coefAB, coefAC : réels

début

       # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC )

       # une droite verticale ?
  si ( ( XA = XB ) ou ( XA = XC ) )
  alors     si ( ( XA = XB ) et ( XA = XC ) )
            alors Afficher ( "les trois points sont alignés" )
            sinon Afficher ( "les trois points ne sont pas alignés" )
            fin_si

  sinon
            # calcul des coefficients directeurs
       coefAB ← ( YA - YB ) / ( XA - XB )
       coefAC ← ( YA - YC ) / ( XA - XC )

            # affichage du résultat
       si ( coefAB = coefAC )
       alors Afficher ( "les trois points sont alignés" )
       sinon Afficher ( "les trois points ne sont pas alignés" )
       fin_si

  fin_si

fin
```

Le script Python correspondant est le suivant :

```
# troisPointsAlignés
# Ce script détermine si trois points A, B et C sont ou non alignés
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
# une droite verticale ?
if ( ( XA == XB ) or ( XA == XC ) ):
    if ( ( XA == XB ) and ( XA == XC ) ):
        print( "les trois points sont alignés" )
    else:
        print( "les trois points ne sont pas alignés" )
else:
    # calcul des coefficients directeurs
    coefAB = ( YA - YB ) / ( XA - XB )
    coefAC = ( YA - YC ) / ( XA - XC )
    # affichage du résultat
    if ( coefAB == coefAC ):
        print( "les trois points sont alignés" )
    else:
        print( "les trois points ne sont pas alignés" )
```

5.2.4. Vecteurs

Exercice 56. Coordonnées d'un vecteur

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

Réponse. *L'algorithme est le suivant :*

```
Algorithme coordonnéesVecteur
# Cet algorithme détermine les coordonnées d'un vecteur AB
variables   XA, YA, XB, YB, Xvecteur, Yvecteur : réels
début
      # lecture données
  Entrer ( XA, YA, XB, YB )
      # calcul des coordonnées
  Xvecteur ← XB - XA
  Yvecteur ← YB - YA
      # affichage du résultat
  Afficher ( Xvecteur, Yvecteur )
fin
```

Le script Python correspondant est le suivant :

```
# coordonnéesVecteur
# Ce script détermine les coordonnées d'un vecteur AB
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
# calcul des coordonnées
Xvecteur = XB - XA
Yvecteur = YB - YA
# affichage du résultat
```

```
print( Xvecteur, ",", Yvecteur )
```

Exercice 57. 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.

Réponse. *Il suffit de vérifier s'ils ont ou pas mêmes coordonnées. L'algorithme est le suivant :*

```
Algorithme vecteursEgaux
# Cet algorithme détermine si deux vecteurs AB et CD sont ou non égaux
variables   XA, YA, XB, YB, XvecteurAB, YvecteurAB : réels
            XC, YC, XD, YD, XvecteurCD, YvecteurCD : réels
début
      # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC, XD, YD )
      # calcul des coordonnées
  XvecteurAB ← XB - XA
  YvecteurAB ← YB - YA
  XvecteurCD ← XD - XC
  YvecteurCD ← YD - YC
      # affichage du résultat
  si ( ( XvecteurAB = XvecteurCD ) et ( YvecteurAB = YvecteurCD ) )
  alors Afficher ( "les vecteurs sont égaux" )
  sinon Afficher ( "les vecteurs ne sont pas égaux" )
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# vecteursEgaux
# Ce script détermine si deux vecteurs AB et CD sont ou non égaux
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
XD = float(input("XD : "))
YD = float(input("YD : "))
# calcul des coordonnées
XvecteurAB = XB - XA
YvecteurAB = YB - YA
XvecteurCD = XD - XC
YvecteurCD = YD - YC
# affichage du résultat
if ( ( XvecteurAB == XvecteurCD ) and ( YvecteurAB == YvecteurCD ) ):
    print( "les vecteurs sont égaux" )
else:
    print( "les vecteurs ne sont pas égaux" )
```

Exercice 58. Vecteurs colinéaires

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.

Réponse. *L'algorithme est le suivant :*

```
Algorithme vecteursColinéaires
# Cet algorithme détermine si deux vecteurs AB et CD sont ou non
```

```
# colinéaires
variables   XA, YA, XB, YB, XvecteurAB, YvecteurAB : réels
            XC, YC, XD, YD, XvecteurCD, YvecteurCD : réels
début
       # lecture données
  Entrer ( XA, YA, XB, YB, XC, YC, XD, YD )
       # calcul des coordonnées
  XvecteurAB ← XB - XA
  YvecteurAB ← YB - YA
  XvecteurCD ← XD - XC
  YvecteurCD ← YD - YC
       # affichage du résultat
  si ( ( XvecteurAB * YvecteurCD ) - ( YvecteurAB * XvecteurCD ) = 0 )
  alors Afficher ( "les vecteurs sont colinéaires" )
  sinon Afficher ( "les vecteurs ne sont pas colinéaires" )
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# vecteursColinéaires
# Ce script détermine si deux vecteurs AB et CD sont ou non
# colinéaires
# lecture données
XA = float(input("XA : "))
YA = float(input("YA : "))
XB = float(input("XB : "))
YB = float(input("YB : "))
XC = float(input("XC : "))
YC = float(input("YC : "))
XD = float(input("XD : "))
YD = float(input("YD : "))
# calcul des coordonnées
XvecteurAB = XB - XA
YvecteurAB = YB - YA
XvecteurCD = XD - XC
YvecteurCD = YD - YC
# affichage du résultat
if ( ( XvecteurAB * YvecteurCD ) - ( YvecteurAB * XvecteurCD ) == 0 ):
    print( "les vecteurs sont colinéaires" )
else:
    print( "les vecteurs ne sont pas colinéaires" )
```

5.3. Statistiques et probabilités

Exercice 59. 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 ».

Réponse. *On utilise la primitive* `Hasard(n)`*, qui renvoie un entier compris entre 0 et n-1. L'algorithme est alors le suivant :*

```
Algorithme lancerPièces_1
# Cet algorithme simule n lancers de pièces, n donné, et affiche la
# fréquence de l'événement « la pièce tombe sur Pile ».
variables   n, i, piece, nbPiles : entiers naturels
début
       # lecture données
```

```
   Entrer ( n )
        # initialisations
   nbPiles ← 0
        # boucle de traitement
   pour i de 1 à n faire
             # tirage aléatoire
        piece = Hasard(2)
        si (piece = 0)          # on choisit pile = 0, face = 1
        alors nbPiles ← nbPiles + 1
        fin_si
   fin_pour
        # affichage de la fréquence
   Afficher ( nbPiles / n )
fin
```

Le script Python correspondant est le suivant :

```
# lancerPièces_1
# Ce script simule n lancers de pièces, n donné, et affiche la
# fréquence de l'événement « la pièce tombe sur Pile ».
import random
# lecture données
n = int(input("Entrer n :"))
# initialisations
nbPiles = 0
# boucle de traitement
for i in range (n):
    # tirage aléatoire
    piece = random.randint(0,1)
    if (piece == 0): # on choisit pile = 0, face = 1
        nbPiles = nbPiles + 1
# affichage de la fréquence
print (nbPiles/n)
```

Exercice 60. 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 ».

Réponse. *On utilise la primitive* `Hasard(n)`*, qui renvoie un entier compris entre 0 et n-1. On utilise cette fois deux boucles* `pour` *imbriquées :*

```
Algorithme lancerPièces_2
# Cet algorithme simule n expériences de 100 lancers de pièces, n donné, et
# affiche la fréquence de l'événement « au moins 60 Piles sur 100
# lancers ».
variables   n, i, j, piece, nbPiles, nbAuMoins60 : entiers naturels
début
        # lecture données
   Entrer ( n )
        # initialisations
   nbAuMoins60 ← 0
        # boucle de traitement, n expériences
   pour i de 1 à n faire
             # initialisation compteur des « pile »
        nbPiles ← 0
```

```
            # boucle des 100 tirages
       pour j de 1 à 100 faire
                # tirage
            piece = Hasard(2)
            si (piece = 0)          # on choisit pile = 0, face = 1
            alors nbPiles ← nbPiles + 1
            fin_si
       fin_pour
            # au moins 60 ?
       si (nbPiles >= 60)
       alors nbAuMoins60 ← nbAuMoins60 + 1
       fin_si
  fin_pour
       # affichage de la fréquence
  Afficher ( nbAuMoins60 / n )
fin
```

Le script Python correspondant est le suivant :

```
# lancerPièces_2
# Ce script simule n expériences de 100 lancers de pièces, n donné, et
# affiche la fréquence de l'événement « au moins 60 Piles sur 100
# lancers ».
import random
# lecture données
n = int(input("Entrer n :"))
# initialisations
nbAuMoins60 = 0
# boucle de traitement, n expériences
for i in range (n):
    # initialisation compteur des « pile »
    nbPiles = 0
    # boucle des 100 tirages
    for j in range (100):
        # tirage
        piece = random.randint(0,1)
        if (piece == 0):     # on choisit pile = 0, face = 1
            nbPiles = nbPiles+1
    # au moins 60 ?
    if (nbPiles >= 60):
        nbAuMoins60 = nbAuMoins60+1
# affichage de la fréquence
print (nbAuMoins60/n)
```

Exercice 61. 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 ».

Réponse. *On utilise la primitive* `Hasard(n)`*, qui renvoie un entier compris entre 0 et n-1. Il est nécessaire ici de mémoriser la valeur du tirage précédent pour pouvoir détecter l'événement ; l'algorithme est alors le suivant :*

```
Algorithme lancerPièces_3
# Cet algorithme simule n lancers de pièces, n donné, et affiche la
# fréquence de l'événement « deux Piles successifs ».
variables   n, i, piece, pieceAvant, nbDeuxPiles : entiers naturels
début
```

```
        # lecture données
   Entrer ( n )
        # initialisations
   pieceAvant ← 1          # 1 signifie face, 0 signifie pile
   nbDeuxPiles ← 0
        # boucle de traitement
   pour i de 1 à n faire
             # tirage aléatoire
        piece = Hasard(2)
             # deux piles successifs ?
        si ((piece = 0) et (pieceAvant = 0))
        alors nbDeuxPiles ← nbDeuxPiles + 1
        fin_si
             # on mémorise le tirage courant
        pieceAvant ← piece
   fin_pour
        # affichage de la fréquence
   Afficher ( nbDeuxPiles / n )
fin
```

Le script Python correspondant est le suivant :

```
# lancerPièces_3
# Ce script simule n lancers de pièces, n donné, et affiche la
# fréquence de l'événement « deux Piles successifs ».
import random
# lecture données
n = int(input("Entrer n :"))
# initialisations
pieceAvant = 1       # 1 signifie face, 0 signifie pile
nbDeuxPiles = 0
# boucle de traitement
for i in range (n):
    # tirage aléatoire
    piece = random.randint(0,1)
    # deux piles successifs ?
    if ((piece == 0) and (pieceAvant == 0)):
        nbDeuxPiles = nbDeuxPiles + 1
    # on mémorise le tirage courant
    pieceAvant = piece
# affichage de la fréquence
print (nbDeuxPiles/n)
```

Exercice 62. 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 ».

Réponse. *On utilise la primitive* `Hasard(n)`*, qui renvoie un entier compris entre 0 et n-1. L'algorithme est le suivant :*

```
Algorithme lancerDés_1
# Cet algorithme simule n lancers de deux dés, n donné, et affiche la
# fréquence de l'événement « la somme de deux dés est paire ».
variables   n, i, de1, de2, nbSommePaire : entiers naturels
début
        # lecture données
   Entrer ( n )
```

```
        # initialisations
   nbSommePaire ← 0
        # boucle de traitement
   pour i de 1 à n faire
             # tirage des deux dés
        de1 = Hasard(6) + 1     # pour avoir 1 <= dé <= 6
        de2 = Hasard(6) + 1
             # somme paire ?
        si ((de1 + de2) mod 2 = 0)
        alors nbSommePaire ← nbSommePaire + 1
        fin_si
   fin_pour
        # affichage de la fréquence
   Afficher ( nbSommePaire / n )
fin
```

Le script Python correspondant est le suivant :

```
# lancerDés_1
# Ce script simule n lancers de deux dés, n donné, et affiche la
# fréquence de l'événement « la somme de deux dés est paire ».
import random
# lecture données
n = int(input("Entrer n :"))
# initialisations
nbSommePaire = 0
# boucle de traitement
for i in range (n):
    # tirage des deux dés
    de1 = random.randint(1,6)  # pour avoir 1 <= dé <= 6
    de2 = random.randint(1,6)
    # somme paire ?
    if ((de1 + de2) % 2 == 0):
        nbSommePaire = nbSommePaire + 1
# affichage de la fréquence
print (nbSommePaire/n)
```

Exercice 63. 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 ».

Réponse. *On utilise la primitive* `Hasard(n)`*, qui renvoie un entier compris entre 0 et n-1. Il est nécessaire cette fois de mémoriser la valeur de la somme du précédent tirage. L'algorithme est le suivant :*

```
Algorithme lancerDés_2
# Cet algorithme simule n lancers de deux dés, n donné, et affiche la
# fréquence de l'événement « les sommes de deux lancers consécutifs sont
# identiques ».
variables   n, i, de1, de2, sommeAvant, nbSommesEgales : entiers naturels
début
        # lecture données
   Entrer ( n )
        # initialisations
   nbSommesEgales ← 0
   sommeAvant ← 0
        # boucle de traitement
```

```
  pour i de 1 à n faire
          # tirage des deux dés
      de1 = Hasard(6) + 1     # pour avoir 1 <= dé <= 6
      de2 = Hasard(6) + 1
          # sommes égales ?
      si ((de1 + de2) = sommeAvant)
      alors nbSommesEgales ← nbSommesEgales + 1
      fin_si
          # on mémorise la somme
      sommeAvant ← de1 + de2
  fin_pour
      # affichage de la fréquence
  Afficher ( nbSommesEgales / n )
fin
```

Le script Python correspondant est le suivant :

```
# lancerDés_2
# Ce script simule n lancers de deux dés, n donné, et affiche la
# fréquence de l'événement « les sommes de deux lancers consécutifs sont
# identiques ».
import random
# lecture données
n = int(input("Entrer n :"))
# initialisations
nbSommesEgales = 0
sommeAvant = 0
# boucle de traitement
for i in range (n):
    # tirage des deux dés
    de1 = random.randint(1,6)       # pour avoir 1 <= dé <= 6
    de2 = random.randint(1,6)
    # sommes égales ?
    if((de1 + de2) == sommeAvant):
        nbSommesEgales = nbSommesEgales+1
    # on mémorise la somme
    sommeAvant = de1 + de2
# affichage de la fréquence
print (nbSommesEgales/n)
```

Exercice 64. 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 »).

Réponse. *On utilise la primitive* `Hasard(n)`*, qui renvoie un entier compris entre 0 et n-1. L'algorithme est le suivant :*

```
Algorithme tirageBoules_1
# Cet algorithme simule k tirages avec remise d'une boule dans une urne
# contenant R boules rouges et N boules noires, k, R et N donnés, et
# affiche la fréquence de l'événement « la boule tirée est rouge ».
variables   k, i, R, N, boule, nbRouges : entiers naturels
début
      # lecture données
  Entrer ( k, R, N )
      # initialisations
```

```
   nbRouges ← 0
        # boucle de traitement
   pour i de 1 à k faire
             # tirage d'une boule
        boule = Hasard(R+N)
             # boule rouge ? ( rouge si 0 <= boule <= R-1 )
        si (boule < R)
        alors nbRouges ← nbRouges + 1
        fin_si
   fin_pour
        # affichage de la fréquence
   Afficher ( nbRouges / k )
fin
```

Le script Python correspondant est le suivant :

```
# tirageBoules_1
# Ce script simule k tirages avec remise d'une boule dans une urne
# contenant R boules rouges et N boules noires, k, R et N donnés, et
# affiche la fréquence de l'événement « la boule tirée est rouge ».
import random
# lecture données
k = int(input("Entrer k le nombre de tirages :"))
R = int(input("Entrer le nombre de boules rouges :"))
N = int(input("Entrer le nombre de boules noires :"))
# initialisations
nbRouges = 0
# boucle de traitement
for i in range (k):
    # tirage d'une boule
    boule = random.randint(0,R1)
    # boule rouge ? ( rouge si 0 <= boule <= R-1 )
    if (boule < R):
        nbRouges = nbRouges
# affichage de la fréquence
print(nbRouges/k)
```

Exercice 65. Boules rouges et noires (2)

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

Réponse. *On utilise la primitive Hasard(n), qui renvoie un entier compris entre 0 et n-1. La différence avec l'algorithme précédent est que le nombre de boules, rouges ou noires, est modifié à chaque tirage. On obtient alors l'algorithme suivant :*

```
Algorithme tirageBoules_2
# Cet algorithme simule k tirages sans remise d'une boule dans une urne
# contenant R boules rouges et N boules noires, k, R et N donnés, et
# affiche la fréquence de l'événement « la boule tirée est rouge ».
variables   k, i, R, N, boule, nbRouges : entiers naturels
début
        # lecture données
   Entrer ( R, N, k )
        # vérification du nombre de tirages
   tantque (k > R + N) faire
        Afficher("Le nombre de tirages ne doit pas être supérieur au nombre
                        total de boules!")
        Entrer (k)
```

```
   fin_tantque
        # initialisations
   nbRouges ← 0
        # boucle de traitement
   pour i de 1 à k faire
            # tirage d'une boule
        boule = Hasard(R+N)
            # boule rouge ? ( rouge si 0 <= boule <= R-1 )
            # dans les deux cas, on met à jour le nombre de boules
        si (boule < R)
        alors
            nbRouges ← nbRouges + 1
            R ← R - 1
        sinon
            N ← N - 1
        fin_si
   fin_pour
        # affichage de la fréquence
   Afficher ( nbRouges / k )
fin
```

Le script Python correspondant est le suivant :

```
# tirageBoules_2
# Ce script simule k tirages sans remise d'une boule dans une urne
# contenant R boules rouges et N boules noires, k, R et N donnés, et
# affiche la fréquence de l'événement « la boule tirée est rouge ».
import random
# lecture données
R=int(input("Entrer le nombre de boules rouges :"))
N=int(input("Entrer le nombre de boules noires :"))
k=int(input("Entrer k le nombre de tirages :"))
# vérification du nombre de tirages
while (k > R + N):
    print("Le nombre de tirages ne doit pas être supérieur au nombre total
          de boules!")
    k=int(input("Entrer k le nombre de tirages :"))
# initialisations
nbRouges = 0
# boucle de traitement
for i in range (k):
    # tirage d'une boule
    boule = random.randint(0,R+N-1)
    # boule rouge ? ( rouge si 0 <= boule <= R-1 )
    # dans les deux cas, on met à jour le nombre de boules
    if (boule < R):
        nbRouges = nbRouges + 1
        R = R - 1
    else:
        N = N - 1
# affichage de la fréquence
print(nbRouges/k)
```

5.4. Divers

5.4.1. Intervalles

Exercice 66. Appartenance à un intervalle

Écrire un algorithme permettant de déterminer si un nombre `n` appartient ou non à un intervalle `[a, b]` donné.

Réponse. *L'algorithme (simple) est le suivant :*

```
Algorithme appartenanceIntervalle
# Cet algorithme détermine si un nombre n appartient ou non à un intervalle
# donné [a, b]
variables   n, a, b : réels
début
       # lecture données
  Entrer ( n, a, b )
       # affichage résultat
  si ( (n >= a) et (n <= b) )
  alors Afficher ("oui")
  sinon Afficher ("non")
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# appartenanceIntervalle
# Ce script détermine si un nombre n appartient ou non à un intervalle
# donné [a, b]
# lecture données
n = float(input("Entrer le nombre n : "))
a = float(input("Entrer la borne inf de l'intervalle : "))
b = float(input("Entrer la borne sup de l'intervalle : "))
# affichage résultat
if ((n >= a) and (n <= b)):
    print("oui")
else:
    print("non")
```

Exercice 67. Inclusion d'intervalles

Écrire un algorithme permettant de déterminer si un intervalle `[a, b]` est ou non inclus dans un intervalle `[c, d]`.

Réponse. *Un exercice à peine plus compliqué que le précédent :*

```
Algorithme inclusionIntervalles
# Cet algorithme détermine si un intervalle [a, b] est ou non inclus dans
# un intervalle [c, d]
variables   a, b, c, d : réels
début
       # lecture données
  Entrer ( a, b, c, d )
       # affichage résultat
  si ( (a >= c) et (b <= d) )
  alors Afficher ("inclus")
  sinon Afficher ("non inclus")
  fin_si
fin
```

Le script Python correspondant est le suivant :

```
# inclusionIntervalles

# Ce script détermine si un intervalle [a, b] est ou non inclus dans
# un intervalle [c, d]

# lecture données
a = float(input("Entrer a :"))
b = float(input("Entrer b :"))
c = float(input("Entrer c :"))
d = float(input("Entrer d :"))

# affichage résultat
if ((a >= c) and (b <= d)):
    print ("inclus")
else:
    print ("non inclus")
```

Exercice 68. Intersection de deux intervalles

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

Réponse. *On calcule le maximum* `max` *des bornes inférieures et le minimum* `min` *des bornes supérieures. Si* `max <= min`*, l'intersection est l'intervalle* `[max, min]`*, sinon l'intersection est vide.*

L'algorithme est le suivant :

```
Algorithme intersectionIntervalles

# Cet algorithme permet de calculer l'intersection de deux intervalles
# d'entiers de la forme [a, b] et [c, d].

variables   a, b, c, d, max, min : réels

début

       # lecture données
  Entrer ( a, b, c, d )

       # calcul du maximum des bornes inférieures
  si (a < c)
  alors     max ← c
  sinon     max ← a
  fin_si

       # calcul du minimum des bornes supérieures
  si (b < d)
  alors     min ← b
  sinon     min ← d
  fin_si

       # affichage du résultat
  si ( max <= min )
  alors     Afficher ("[", max, ",", min, "]" )
  sinon     Afficher ( "intersection vide" )

fin
```

Le script Python correspondant est le suivant :

```
# intersectionIntervalles

# Ce script permet de calculer l'intersection de deux intervalles
# d'entiers de la forme [a, b] et [c, d].

# lecture données
a = float(input("Entrer a :"))
b = float(input("Entrer b :"))
c = float(input("Entrer c :"))
d = float(input("Entrer d :"))

# calcul du maximum des bornes inférieures
if (a < c):
    max = c
else:
```

```
    max = a

# calcul du minimum des bornes supérieures
if (b < d):
    min = b
else:
    min = d

# affichage du résultat
if (max <= min):
    print ("[", max,";",min,"]")
else:
    print ("Intersection vide.")
```

Exercice 69. Réunion d'intervalles

Écrire un algorithme permettant de déterminer si la réunion de deux intervalles `[a, b]` et `[c, d]` donnés est ou non un intervalle.

Réponse. *La réponse est positive si et seulement si les deux intervalles ont une intersection non vide. On peut donc adapter l'algorithme précédent :*

```
Algorithme réunionIntervalles

# Cet algorithme permet de déterminer si la réunion de deux intervalles
# [a, b] et [c, d] donnés est ou non un intervalle

variables   a, b, c, d, max, min : réels

début

       # lecture données
  Entrer ( a, b, c, d )

       # calcul du maximum des bornes inférieures
  si (a < c)
  alors     max ← c
  sinon     max ← a
  fin_si

       # calcul du minimum des bornes supérieures
  si (b < d)
  alors     min ← b
  sinon     min ← d
  fin_si

       # affichage du résultat
  si ( max <= min )
  alors     Afficher ( "la réunion est un intervalle" )
  sinon     Afficher ( "la réunion n'est pas un intervalle" )

fin
```

Le script Python correspondant est le suivant :

```
# réunionIntervalles

# Ce script permet de déterminer si la réunion de deux intervalles
# [a, b] et [c, d] donnés est ou non un intervalle

# lecture données
a = float(input("Entrer a :"))
b = float(input("Entrer b :"))
c = float(input("Entrer c :"))
d = float(input("Entrer d :"))

# calcul du maximum des bornes inférieures
if (a < c):
    max = c
else:
    max = a

# calcul du minimum des bornes supérieures
if (b < d):
    min = b
else:
```

```
    min = d
# affichage du résultat
if (max <= min):
    print ("La réunion est un intervalle.")
else:
    print ("La réunion n'est pas un intervalle.")
```

5.4.2. Approximations de Pi

Exercice 70. 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) :

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

Réponse. *Une boucle* pour *permet de calculer le second facteur de la formule. On obtient l'algorithme suivant :*

```
Algorithme approximationDePI_1
# Cet algorithme calcule une approximation de PI - formule 1
variables   nbEtapes, i : entiers naturels
            signe : entier relatif          # signe vaudra +1 ou -1
            approxPi : réel
début
       # lecture données
  Entrer ( nbEtapes )
       # initialisations
  approxPi ← 1
  signe ← -1
       # boucle de calcul
  pour i de 1 à nbEtapes faire
            # on ajoute le i-ième terme
       approxPi ← approxPi + signe / ( 2*i + 1 )
            # on change de signe pour le tour suivant
       signe ← - signe
  fin_pour
       # affichage du résultat
  approxPi ← 4 * approxPi
  Afficher ( approxPi )
fin
```

Le script Python correspondant est le suivant :

```
# approximationDePI_1
# Ce script calcule une approximation de PI - formule 1
# lecture données
nbEtapes = int(input("Entrer le nombre d'étapes : "))
# initialisations
approxPi = 1
signe = -1
# boucle de calcul
for i in range (1,nbEtapes+1):
    # on ajoute le i-ième terme
    approxPi = approxPi + signe / (2*i+1)
    # on change de signe pour le tour suivant
    signe = -signe
```

```
# affichage du résultat
approxPi = 4 * approxPi
print (approxPi)
```

Exercice 71. 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) :

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

Réponse. *L'algorithme ressemble au précédent, si ce n'est que l'on doit être attentif à la valeur du dénominateur des termes successifs… En effet, le dénominateur du i-ième terme vaut* $(2i-1) * 3^{i-1}$*, qui est très rapidement un entier gigantesque ! Il est donc préférable ici de calculer le i-ième terme en fonction de la valeur du (i-1)-ième. On obtient l'algorithme suivant :*

```
Algorithme approximationDePI_2
# Cet algorithme calcule une approximation de PI - formule 2
variables   nbEtapes, i : entiers naturels
            terme : entier relatif
            approxPi : réel
début
      # lecture données
  Entrer ( nbEtapes )
      # initialisations
  approxPi ← 0
  terme ← 1
      # boucle de calcul
  pour i de 1 à nbEtapes faire
          # on ajoute le i-ième terme
      approxPi ← approxPi + terme
          # on prépare le prochain terme
      terme ← -1 * terme * (2*i-1) / (3*(2*i+1))
  fin_pour
      # affichage du résultat
  approxPi ← 6 * approxPi / RacineCarrée(3)
  Afficher ( approxPi )
fin
```

Le script Python correspondant est le suivant :

```
# approximationDePI_2
# Ce script calcule une approximation de PI - formule 2
import math
# lecture données
nbEtapes = int(input("Nombre d'etapes : "))
# initialisations
approxPi = 0
terme = 1
# boucle de calcul
for i in range (1,nbEtapes+1):
    # on ajoute le i-ième terme
    approxPi = approxPi + terme
    # on prépare le prochain terme
    terme = -1 * terme * (2*i-1) / (3*(2*i+1))
# affichage du résultat
approxPi = 6 * approxPi / math.sqrt(3)
```

```
print(approxPi)
```

Exercice 72. Approximation de Pi (3)

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

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

Réponse. *Toujours le même principe :*

```
Algorithme approximationDePI_3
# Cet algorithme calcule une approximation de PI - formule 3
variables   nbEtapes, i : entiers naturels
            approxPi : réel
début
       # lecture données
  Entrer ( nbEtapes )
       # initialisations
  approxPI ← 0
       # boucle de calcul
  pour i de 1 à nbEtapes faire
            # on ajoute le i-ième terme
       approxPi ← approxPi + 1 / ( i*i )
  fin_pour
       # affichage du résultat
  approxPi ← RacineCarrée(6 * approxPi)
  Afficher ( approxPi )
fin
```

Le script Python correspondant est le suivant :

```
# approximationDePI_3
# Ce script calcule une approximation de PI - formule 3
import math
# lecture données
nbEtapes = int(input("Entrer le nombre d'étapes : "))
# initialisations
approxPi = 0
# boucle de calcul
for i in range (1,nbEtapes+1):
    # on ajoute le i-ième terme
    approxPi = approxPi + 1/(i*i)
# affichage du résultat
approxPi = math.sqrt(6 * approxPi)
print (approxPi)
```

Exercice 73. Approximation de Pi (4)

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

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

Réponse. *Soit 2/1 le 1er facteur, 2/3 le 2e facteur, etc. On observe que pour i pair, le (i+1)-ième dénominateur sera identique et le (i+1)-ième numérateur aura augmenté de 2. Pour i impair, c'est le contraire… On obtient alors l'algorithme suivant :*

```
Algorithme approximationDePI_4
# Cet algorithme calcule une approximation de PI - formule 4
variables   nbEtapes, i, numerateur, denominateur : entiers naturels
            approxPi : réel
début
      # lecture données
  Entrer ( nbEtapes )
      # initialisations
  numerateur ← 2
  denominateur ← 1
  approxPI ← 1
      # boucle de calcul
  pour i de 1 à nbEtapes faire
          # on multiplie par le i-ième facteur
      approxPi ← approxPi * ( numerateur / denominateur )
          # on modifie selon la parité de i pour préparer le tour suivant
      si (i mod 2 = 0)
      alors    # i est pair
          numerateur ← numerateur + 2
      sinon    # i est impair
          denominateur ← denominateur + 2
      fin_si
  fin_pour
      # affichage du résultat
  approxPi ← 2 * approxPi
  Afficher ( approxPi )
fin
```

Le script Python correspondant est le suivant :

```
# approximationDePI_4
# Ce script calcule une approximation de PI - formule 4
import math
# lecture données
nbEtapes = int(input("Entrer le nombre d'étapes : "))
# initialisations
approxPi = 1
numerateur = 2
denominateur = 1
# boucle de calcul
for i in range (1,nbEtapes+1):
    # on multiplie par le i-ième facteur
    approxPi = approxPi * (numerateur / denominateur)
    # on modifie selon la parité de i pour préparer le tour suivant
    if (i%2 == 0):
        numerateur = numerateur + 2
    else:
        denominateur = denominateur + 2
# affichage du résultat
approxPi = 2 * approxPi
print (approxPi)
```

On peut également remarquer que le i-ième facteur a pour numérateur `2 * ( (i+1) div 2 )` *et pour dénominateur* `2 * ( (i-1) div 2 ) + 1`*. La boucle de calcul peut alors se simplifier ainsi :*

```
      # boucle de calcul
```

```
   pour i de 1 à nbEtapes faire
            # on multiplie par le i-ième facteur
        approxPi ← approxPi * ( 2*((i+1) div 2) / (2*((i-1) div 2)+1))
   fin_pour
```

L'extrait de script Python modifié correspondant est alors le suivant :

```
# boucle de calcul
for i in range (1,nbEtapes+1):
    # on multiplie par le i-ième facteur
    approxPi = approxPi * (2* ((i+1)//2) / (2*((i-1)//2)+1))
```

On peut aussi remarquer que le numérateur et le dénominateur de rang i sont les inverses de ceux de rang i-1, augmentés de 1. Pour inverser numérateur et dénominateur, on utilise une variable transfert. On obtient alors :

```
       # boucle de calcul
   pour i de 1 à nbEtapes faire
            # on multiplie par le i-ième facteur
        approxPi ← approxPi * ( numerateur / denominateur )
            # on prépare le prochain terme
        a ← numerateur
        numerateur ← denominateur + 1
        denominateur ← a + 1
   fin_pour
```

L'extrait de script Python modifié correspondant est alors le suivant :

```
# boucle de calcul
for i in range (1,nbEtapes+1):
    # on multiplie par le i-ième facteur
    approxPi=approxPi*(numerateur/denominateur)
    # on prépare le prochain terme
    a=numerateur
    numerateur=denominateur+1
    denominateur=a+1
```