Algorithmes et logique au lycée

P.Bouttier, A.Crumière, F.Didier, J-M.Fillia, M.Quatrini, H.Roland

Octobre 2009
I.R.E.M Campus de Luminy
163 avenue de Luminy
Case 901 13288 Marseille Cedex 9

Table des matières

Présentation

Ce document a pour but d'apporter un complément d'information dans le domaine de l'algorithmique et de la logique aux professeurs de mathématiques. Ces chapitres sont actuellement enseignés, de manière transversale, dans la spécialité mathématique du cycle terminal de la série L. Ce document peut aussi intéresser les autres enseignants de lycée, notamment ceux de la classe de seconde, puisque les nouveaux programmes intégrent ces deux domaines.

La première partie aborde la partie algorithmique de ce programme. Après avoir introduit succintement et naïvement la notion d'algorithme, suit une description du langage dans lequel les algorithmes seront décrits. Les instructions de ce langage sont communes à la plupart des langages de programmation ; ainsi ils pourront être facilement transcriptibles dans n'importe lequel de ces langages en vue d'être testés. La notion de preuve de la validité d'un algorithme est abordée par la notion d'invariant de boucle. Elle donne un autre éclairage du raisonnement par récurrence, raisonnement que les élèves ont souvent du mal à maîtriser.
Les exemples présentés sont principalement issus du domaine des mathématiques et plus particulièrement de l'arithmétique. Deux algorithmes fondamentaux, qui ont la particularité d'être utilisés dans de nombreux domaines de par leur efficacité, sont présentés : la méthode de dichotomie et l'algorithme de Hörner.

La deuxième partie propose une introduction à la partie de la logique mathématique qui définit les formules mathématiques formelles. Cette présentation est à destination des professeurs souhaitant compléter leurs connaissances en ce domaine afin de mieux dominer les savoirs logiques implicites dans les mathématiques de la classe, tant dans les objets enseignés que dans la pratique des exercices.
Ces notions sont illustrées et déclinées sur des exercices du programme de spécialité mathématique en série L mais sont adaptables aux programmes à venir.

Chapitre 1

Algorithmique

Les algorithmes sont très souvent considérés comme du domaine des mathématiques et de l'informatique, leur champ d'application est en réalité beaucoup plus vaste. Les historiens s'accordent pour dire que le mot "algorithme" vient du nom du grand mathématicien persan Al Khow$\hat{a}$rizm$\hat{i}$ (an 825 environ) qui introduisit en Occident la numération décimale et les règles de calculs s'y rapportant. La notion d'algorithme est donc historiquement liée aux manipulations numériques, mais ces dernières décennies elle s'est progressivement développée sur des objets plus complexes, graphes, textes, images, son, formules logiques, robotique, etc... Le but de ce chapitre est de préciser ce qu'on entend par algorithme, de présenter un langage dans lequel on pourra les exprimer, d'évoquer et d'illustrer les problèmes qui se posent dans leur élaboration.

1.1 Quelques idées en vrac

Approche naïve : C'est une méthode i.e une façon systématique de procéder pour faire quelque chose : trier, rechercher, calculer,... Il répond à des questions du type : comment faire ceci ?, obtenir cela ?, trouver telle information ?, calculer tel nombre ?, ... C'est un concept pratique, qui traduit la notion intuitive de procédé systématique, applicable mécaniquement, sans réfléchir, en suivant simplement un mode d'emploi précis.

Tentative de définition : Suite d'ordres précis, compréhensibles et exécutables de manière non ambigüe par un automate, devant être exécutés suivant un ordre parfaitement déterminé en vue de résoudre une classe de problèmes. Par exemple : classer des suites de nombres, chercher des mots dans un dictionnaire sont des exemples de classe de problèmes que des algorithmes sont capables de résoudre.

On évitera de prendre des exemples de la vie courante pour illustrer le concept d'algorithme : une recette de cuisine est un mauvais exemple d'algorithme car elle est toujours trop imprécise : une pincée de sel (combien de grammes ?), faire bouillir dix minutes (la température d'ébulition dépend entre autre de l'altitude à laquelle on se trouve),...

En général l'automate est une machine, et très souvent un ordinateur. L'activité principale en algorithmique est de concevoir et d'exprimer, puis de coder dans un langage de programmation un algorithme. En général, on l'exprime dans un langage proche de la langue naturelle, en utilisant des expressions dont on sait qu'elles seront facilement transcrites dans un langage de programmation.

Algorithme en français
↓
Codage dans un langage de programmation
↓
Traduction automatique en langage machine (compilation)
↓
Exécution

Points importants de la conception d'un algorithme : Une des difficultés dans l'élaboration d'un algorithme est de contrôler son aspect dynamique (exécution) à partir de son écriture statique (suite finie d'instructions). Un algorithme de tri peut s'écrire en quelques lignes et peut trier aussi bien une suite de 10 éléments qu'une suite de 100 000 éléments. Les problèmes importants qui se posent dans la conception d'un algorithme sont :

- **La preuve :** prouver que l'algorithme résout la classe de problèmes pour laquelle il a été écrit.
- **L'arrêt :** prouver que quelles que soient les données fournies en entrée l'algorithme s'arrête toujours.
- **Le choix des structures des données :** il est fondamental que les données manipulées soient sous un format bien précis et structurées. Par exemple, dans un dictionnaire, mots français, écrits en lettres latines et surtout classés par ordre alphabétique...

L'essentiel, dans l'élaboration d'un algorithme, est de percevoir les éléments clés d'un processus quelconque, et d'imaginer la suite d'opérations logiques les plus astucieuses et les plus efficaces pour le mettre en œuvre de manière automatique et performante.

Un algorithme peut être vu comme le squelette abstrait du programme informatique, sa substantifique moelle, indépendant du codage particulier qui permettra sa mise en oeuvre avec un ordinateur ou une machine mécanique.

Domaines où interviennent les algorithmes : L'algorithmique intervient dans des domaines divers et variés :

- Numériques.
- Tri, recherche de mots, plus généralement recherche d'information.
- Algorithmes géométriques, images de synthèse.
- Recherche de formes, génomes, internet (moteur de recherche).
- Compression des données.
- Cryptographie.
- ...

Malgré les énormes progrès de la technologie, les ordinateurs seront toujours soumis à des limitations physiques : nombre d'opérations par seconde, taille mémoire,... Une part importante de la recherche en algorithmique consiste à élaborer des algorithmes de plus en plus efficaces. Par exemple, en 1950 on pouvait calculer quelques milliers de décimales de Π et en 2002 on en était à plus d'un trillion (10^{18}). C'est à peu près à part égale en raison des avancées technologiques et algorithmiques. La recherche systématique d'efficacité passe aussi par la recherche de nouveaux types de représentation et d'organisation des données :
classement des mots, organisation arborescente, ...
D.Knuth [1] considère les arbres comme la structure la plus fondamentale de toute l'informatique.

1.2 Présentation d'un langage d'écriture des algorithmes

Certaines des instructions présentées ci-dessous utilisent la notion d'expression booléenne : ces expressions ne peuvent avoir que deux valeurs possibles, **vrai** et **faux**.
On peut dire simplement qu'elles sont obtenues en comparant entre elles deux expressions, dont la comparaison est possible, à l'aide des opérateurs classiques de comparaison : $<, \leq, >, \geq, =, \neq$.
Des expressions plus complexes pouvant être construites en utilisant les opérateurs logiques classiques :

ou, et , non

qui dénotent respectivement la disjonction, la conjonction et la négation.

1. informaticien célèbre pour ses travaux en algorithmique, créateur de TEX

1.2.1 L'instruction d'affectation

La syntaxe de cette instruction varie suivant les langages utilisés. Elle a pour but **d'affecter** à une variable la valeur d'une expression. En général, dans les langages de programmation, sa forme est :

"expression partie gauche" "symbole" "expression partie droite"

L'expression en partie gauche doit impérativement désigner une **adresse** et une **valeur** est associée à l'expression de la partie droite. Elle a pour but de ranger la valeur de l'expression à l'adresse indiquée par la partie gauche. Comme il a été dit le symbole varie suivant les langages de programmation.

- **:=** en Algol, Pascal, Maple,...
- **=** en Basic , C, CAML,...
- $\leftarrow$ en LSE

Signalons que sur certaines calculatrices de poche TI (Texas Instrument) le symbole utilisée pour l'affectation est $\rightarrow$. Cela est très ennuyeux car les élèves qui ont l'habitude de manipuler ces matériels ont tendance à inverser le sens de l'affectation lorsqu'ils se mettent à programmer sur ordinateurs... Nous choisirons comme symbole $\leftarrow$ car il est cohérent avec les notions de "partie gauche" et "partie droite" qui se retrouvent dans tous les langages utilisés par les ordinateurs et il ne comporte pas le symbole = qui perturbe les élèves, car il a pour eux une autre signification (l'égalité). En résumé notre instruction d'affectation sera :

"expression partie gauche" $\leftarrow$ "expression partie droite"
et se lit
L'expression partie gauche reçoit la valeur de l'expression partie droite

Exemples :
k $\leftarrow$ 0, la variable k reçoit la valeur 0.
k $\leftarrow$ k+1, la variable k reçoit la valeur de l'expression k+1.
k $\leftarrow$ a*(10+b)/n, la variable k reçoit la valeur de l'expression a*(b+10)/n.

1.2.2 Les instructions conditionnelles

Ces instructions sont le *si alors* et le *si alors sinon*.

L'instruction *si alors*

Elle s'écrit :

si "expression booléenne" **alors** "instruction".

Si l'évaluation de l'expression donne la valeur **vrai** c'est l'instruction qui suit immédiatement le **alors** qui est exécutée, sinon c'est l'instruction suivante (celle qui suit le *si alors*) qui est exécutée.
Dans le cas où l'on souhaiterait exécuter après le **alors** non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical qui englobe toutes ces instructions.

si "expression booléenne"
alors
| $instruction1$
| $instruction2$
| ⋮
| $instruction\ n$

L'instruction *si alors sinon*

Elle s'écrit :

si "expression booléenne" **alors** "instruction 1" **sinon** "instruction 2".

Si l'évaluation de l'expression donne la valeur **vrai**, c'est l'instruction 1 qui suit immédiatement le **alors** qui est exécutée, sinon c'est l'instruction 2 qui suit le **sinon** qui est exécutée.
Dans le cas où l'on souhaiterait exécuter non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical.

si "expression booléenne"
alors
| $instruction1$
| $instruction2$
| ⋮
| $instruction\ n$
sinon
| $instruction1$
| $instruction2$
| ⋮
| $instruction\ p$

1.2.3 Les instructions itératives

La boucle *tant que*

Elle s'écrit :

tant que "expression booléenne " **faire** " instruction".

Tant que l'expression booléenne est vraie on exécute l'instruction qui suit **faire**, dès qu'elle devient fausse, on passe à l'instruction suivante. Dans le cas où l'on souhaiterait exécuter non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical.

tant que "expression booléenne "**faire**
| $instruction1$
| $instruction2$
| $\vdots$
| $instruction\ n$

La boucle *pour*

Elle s'écrit :

pour "variable" **variant de** "expr. init" **jusqu'à** "expr. fin" **faire** "instruction".

ou encore
pour "variable " **variant de** "expr. init" **jusqu'à** "expr. fin" **avec un pas de** "expr. pas" **faire** "instruction".
L'instruction qui suit **faire** est exécutée pour les différentes valeurs de la "variable". La variable est appelée variable de contrôle de la boucle. Si l'*expression pas* n'est pas précisée, la variable de contrôle est incrémentée de 1 après chaque tour de boucle. Dans le cas où l'on souhaiterait exécuter non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical.

pour variable **variant de** "expr. init" **jusqu'à** "expr. fin" **faire**
| $instruction1$
| $instruction2$
| $\vdots$
| $instruction\ n$

En général la variable de contrôle, l'expression initiale, l'expression finale et l'expression pas (si elle est présente) ont des valeurs entières. Ce sera toujours le cas dans les algorithmes étudiés. Par ailleurs, on interdit de modifier la variable de contrôle à l'intérieur de la boucle.

1.3 Illustration des notions de preuve et de terminaison d'un algorithme

Ces notions sont illustrées à travers cinq exemples. On présente tout d'abord trois algorithmes qui permettent de calculer le produit $a \times b$ de nombres entiers naturels a et b, puis le très connu algorithme d'Euclide pour calculer le PGCD de deux nombres entiers et enfin un algorithme un peu moins connu permettant de calculer le quotient et le reste de la division euclidienne de deux nombres entiers.

1.3.1 Algorithme de multiplication : Un premier algorithme

Algorithme 1. Ce premier algorithme calcule le produit de deux nombres entiers ($a \times b$) en n'effectuant que des additions.

$x \leftarrow a$
$y \leftarrow b$
$z \leftarrow 0$
tant que $y \neq 0$ **faire**
  $z \leftarrow z + x$
  $y \leftarrow y - 1$
résultat z

Terminaison : y étant décrémenté de 1 à chaque itération, il deviendra nécessairement nul au bout de b itérations.

Validité : Il suffit d'établir que $z + x \times y$ reste en permanence égal à $a \times b$. On dit que cette quantité est invariante. C'est l'invariant qui caractérise la boucle. Cet invariant joint à la condition d'arrêt ($y = 0$) prouve que la variable z vaut $a \times b$ à la sortie de la boucle.

- C'est trivialement vrai avant la boucle.
- Montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x', y' et z' les valeurs respectives de x, y et z à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x, y et z pour l'itération suivante. On a $x' = x$, $y' = y - 1$ et $z' = z + x$.

$$\begin{aligned} z' + x' \times y' &= z + x + x \times (y - 1) \\ &= z + x \times y \\ &= a \times b \text{ par hypothèse de récurrence .} \end{aligned}$$

1.3.2 Algorithme de multiplication : Un deuxième algorithme

Algorithme 2. Ce second algorithme calcule le produit de deux nombres entiers en n'effectuant que des multiplications et divisions par 2.

$x \leftarrow a$
$y \leftarrow b$
$z \leftarrow 0$
tant que $y \neq 0$ **faire**
  si y impair **alors** $z \leftarrow z + x$
  $x \leftarrow 2 \times x$
  $y \leftarrow y$ **div** 2
résultat z

Terminaison : y étant divisé par 2 à chaque itération, il deviendra nécessairement nul après un nombre fini d'itérations.

Validité : Il suffit d'établir que $z + x \times y$ reste en permanence égal à $a \times b$. Cet invariant joint à la condition d'arrêt ($y = 0$) prouve, ici encore, que la variable z vaut $a \times b$ à la sortie de la boucle.

- C'est trivialement vrai avant la boucle.
- Montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x', y' et z' les valeurs respectives de x, y et z à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x, y et z pour l'itération suivante. Deux cas sont à envisager :

  - y pair
    On a $x' = 2 \times x$, $y' = y/2$ et $z' = z$.
    $$\begin{aligned} z' + x' \times y' &= z + (2 \times x) \times (y/2) \\ &= z + x \times y \\ &= a \times b \text{ par hypothèse de récurrence .} \end{aligned}$$
  - y impair
    On a $x' = 2 \times x$, $y' = (y-1)/2$ et $z' = z + x$.
    $$\begin{aligned} z' + x' \times y' &= z + x + (2 \times x) \times ((y-1)/2) \\ &= z + x + x \times (y-1) \\ &= z + x \times y \\ &= a \times b \text{ par hypothèse de récurrence.} \end{aligned}$$

Cet algorithme est connu sous divers noms : multiplication russe, multiplication égyptienne,... Notons que son codage en informatique donne un algorithme très efficace car multiplications et divisions par 2 consistent en des décalages, et ces

opérations élémentaires dans tout langage machine sont effectuées très rapidement. On peut donner une justification de sa validité autre que celle utilisant la notion d'invariant. Il suffit de considérer l'écriture binaire de y,
$y = \sum_{i=0}^{k} y[i]2^i$ où les coefficients $y[i]$ valent 0 ou 1. Le produit $x \times y$ est donc égal à $\sum_{i=0}^{k} y[i] \times (x \times 2^i)$. Tous les termes intervenant dans la somme sont de la forme $x \times 2^i$ et correspondent à des coefficients $y[i]$ non nuls, i.e aux valeurs impaires que prend la variable y dans l'algorithme 2.
Dans la pratique on organise les calculs en faisant deux colonnes, une contenant respectivement les valeurs des quotients successifs de y par 2 et l'autre les valeurs de la forme $x \times 2^i$. Il suffit alors de sommer les éléments de la deuxième colonne (en gras dans l'exemple ci-dessous) qui sont en face d'éléments impairs de la première colonne.
Exemple, soit à calculer le produit de x= 12 par y= 22.

Calcul en décimal

y	x
22	12
11	**24**
5	**48**
2	96
1	**192**

Calcul en binaire

y	x
10110	1100
1011	**11000**
101	**110000**
10	1100000
1	**11000000**

On trouve que le résultat de 12×22 est égal à $24 + 48 + 192 = 264$
En binaire, $11000 + 110000 + 11000000 = 100001000$.

1.3.3 Algorithme de multiplication : Un troisième algorithme

Algorithme 3. Ce troisième algorithme est celui que l'on apprend à l'école primaire, il utilise des multiplications et divisions par 10 (système décimal) et l'algorithme de multiplication d'un nombre entier quelconque par un nombre ayant un seul chiffre (utilisation des tables de multiplication).

$x \leftarrow a$
$y \leftarrow b$
$z \leftarrow 0$
tant que $y \neq 0$ **faire**
  $z \leftarrow z + x \times y$ **mod** 10
  $x \leftarrow 10 \times x$
  $y \leftarrow y$ **div** 10
résultat z

Terminaison : y étant divisé par 10 à chaque itération, il deviendra nécessairement nul après un nombre fini d'itérations.

Validité : Il suffit d'établir que $z + x \times y$ reste en permanence égal à $a \times b$. Cet invariant joint à la condition d'arrêt ($y = 0$) prouve, ici encore, que la variable z vaut $a \times b$ à la sortie de la boucle.

- C'est trivialement vrai avant la boucle.
- Montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x', y' et z' les valeurs respectives de x, y et z à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x,y et z pour l'itération suivante. On a $x' = 10 \times x$, $y' = y$ **div** 10 et $z' = z + x \times (y$ **mod** $10)$.

$$\begin{aligned} z' + x' \times y' &= z + x \times (y \textbf{ mod } 10) + (10 \times x) \times (y \textbf{ div } 10) \\ &= z + x \times (y \textbf{ mod } 10 + 10 \times (y \textbf{ div } 10)) \\ &= z + x \times y \\ &= a \times b \text{ par hypothèse de récurrence} \end{aligned}$$

1.3.4 Algorithme d'Euclide pour le calcul du PGCD de nombres entiers

Algorithme 4 (Euclide)**.** Cet algorithme prend deux entiers naturels en entrée a et b et retourne leur *pgcd*.

$x \leftarrow a$
$y \leftarrow b$
tant que $y \neq 0$ **faire**
  $r \leftarrow x$ **mod** y
  $x \leftarrow y$
  $y \leftarrow r$
résultat x

Le principe de l'algorithme d'Euclide s'appuie sur les deux propriétés suivantes :

- Le $pgcd(a, 0)$ est a.
- Le $pgcd(a, b)$ est égal au $pgcd(b, r)$ où r est le reste de la division euclidienne de a par b.

Cette dernière propriété est très facile à établir. On démontre que l'ensemble des diviseurs de a et b est le même que l'ensemble des diviseurs de b et r. Il est évident que tout nombre divisant a et b divise r, car $r = a - b \times q$ et réciproquement, tout nombre divisant b et r divise aussi a, car $a = r + b \times q$.

Pour démontrer que cet algorithme calcule bien le $pgcd$ de a et b, il faut démontrer que cet algorithme fournit dans tous les cas une réponse (**terminaison** de

l'algorithme) et que cette réponse est bien le $pgcd$ de a et b (**validité** de l'algorithme).

Terminaison : À chaque itération y est affecté avec le reste de la division euclidienne de x par y. On sait, par définition de la division euclidienne, que ce reste r est strictement inférieur au diviseur y. Ainsi à chaque itération y, entier naturel, diminue strictement. Il existe donc une étape où y recevra la valeur 0, permettant ainsi à la boucle de se terminer.

Validité : Il suffit d'établir que le $pgcd(x, y)$ reste en permanence égal au $pgcd(a, b)$. On dit que cette quantité est invariante. C'est l'invariant qui caractérise la boucle. Plus précisément l'invariant est la proposition logique $pgcd(x, y) = pgcd(a, b)$.

- C'est trivialement vrai avant la boucle.
- Montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x' et y' les valeurs de x et y à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x et y pour l'itération suivante.
  On a x' qui est égal à y et y' qui est égal à r (r reste de la division euclidienne de x par y), ainsi le $pgcd(x', y')$ est égal au $pgcd(y, r)$ qui est égal au $pgcd(x, y)$(propriété 2 ci-dessus), or par hypothèse de récurrence on a $pgcd(x, y) = pgcd(a, b)$. Ainsi $pgcd(x', y') = pgcd(a, b)$.

L'invariant joint à la condition d'arrêt ($y = 0$) prouve qu'à la sortie de la boucle x est égal au $pgcd(a, b)$.

Cette façon d'établir la validité de l'algorithme d'Euclide paraît pour le moins aussi simple que la démonstration qui introduit des suites indexées. Peut-être, pour un public non initié, est-il préférable d'écrire l'algorithme sous la forme suivante :

Algorithme 5 (Algorithme d'Euclide, 2^{eme} version)**.**

$x \leftarrow a$
$y \leftarrow b$
tant que $y \neq 0$ **faire**
  $r \leftarrow x$ **mod** y
  $x' \leftarrow y$
  $y' \leftarrow r$
  $x \leftarrow x'$
  $y \leftarrow y'$
résultat x

1.3.5 Algorithme d'Euclide étendu

Algorithme 6 (Euclide étendu)**.** Etant donnés deux entiers naturels a et b, cet algorithme calcule leur $pgcd$, ainsi que deux entiers u et v tels que $a \times u + b \times v = pgcd(a, b)$. A la fin de cet algorithme, x est le $pgcd$ de a et b, car si l'on examine les lignes qui ne concernent que x et y (en gras) on retrouve exactement l'algorithme d'Euclide.

$\mathbf{x} \leftarrow \mathbf{a}$
$\mathbf{y} \leftarrow \mathbf{b}$
$u \leftarrow 1$
$v \leftarrow 0$
$u_1 \leftarrow 0$
$v_1 \leftarrow 1$
tant que $y \neq 0$ **faire**
  $q \leftarrow x$ **div** y
  $\mathbf{r} \leftarrow \mathbf{x}$ **mod** $\mathbf{y}$
  $\mathbf{x} \leftarrow \mathbf{y}$
  $\mathbf{y} \leftarrow \mathbf{r}$
  $aux \leftarrow u; u \leftarrow u_1; u_1 \leftarrow aux - q \times u_1$
  $aux \leftarrow v; v \leftarrow v_1; v_1 \leftarrow aux - q \times v_1$
résultat x**,**u**,**v

L'invariant de la boucle est la conjonction des deux égalités :

$$a \times u + b \times v = x \text{ et } a \times u_1 + b \times v_1 = y$$

Ceci est trivialement vrai avant la boucle.
Notons x', y', u', v', u'_1, v'_1 les valeurs respectives de x, y, u, v, u_1, v_1 à la fin d'une itération. Les valeurs ayant un prime devenant les nouvelles valeurs au début de l'itération suivante. Ainsi par hypothèse de récurrence on a $a \times u + b \times v = x$ et $a \times u_1 + b \times v_1 = y$. Montrons qu'à la fin d'une itération les égalités

$$a \times u' + b \times v' = x' \tag{1.1}$$

et

$$a \times u'_1 + b \times v'_1 = y' \tag{1.2}$$

sont encore satisfaites :

1. $a \times u' + b \times v' = x'$.
   Dans la boucle, on a les affectations :
   $u' \leftarrow u_1$
   $v' \leftarrow v_1$
   $x' \leftarrow y$
   Ainsi $a \times u' + b \times v' = a \times u_1 + b \times v_1$ qui vaut y par hypothèse, donc x'.

2. $a \times u'_1 + b \times v'_1 = y'$.
   Dans la boucle, on a les affectations :
   $u'_1 \leftarrow u - q \times u_1$
   $v'_1 \leftarrow v - q \times v_1$
   $y' \leftarrow r$, où q et r sont respectivement le quotient et le reste de la division euclidienne de x par y.

$$\begin{aligned} a \times u'_1 + b \times v'_1 &= a \times (u - q \times u_1) + b \times (v - q \times v_1) \\ &= a \times u + b \times v - q \times (a \times u_1 + b \times v_1) \\ &= x - q \times y \text{ par hypothèse de récurrence} \\ &= r \text{ par définition du reste} \\ &= y' \end{aligned}$$

Ainsi, lorsque l'algorithme se termine on a bien : $a \times u + b \times v = x = pgcd(a, b)$.

1.3.6 Division euclidienne

Algorithme 7 (Division euclidienne)**.** Cet algorithme effectue la division euclidienne de deux nombres entiers a et b en calculant le quotient q et le reste r. Les opérations utilisées sont la soustraction, la multiplication par 2 et la division par 2.

$q \leftarrow 0$
$w \leftarrow b$
$r \leftarrow a$
tant que $w \leq r$ **faire**
  $w \leftarrow 2 \times w$
tant que $w \neq b$ **faire**
  $q \leftarrow 2 \times q$
  $w \leftarrow w$ **div** 2
  si $w \leq r$ **alors**
    $r \leftarrow r - w$
    $q \leftarrow q + 1$
résultat $[q, r]$

Cet algorithme est très performant lorsque la base est une puissance de 2, car multiplications et divisions consistent alors en des décalages. L'invariant de la seconde boucle est la conjonction des deux propriétés $q \times w + r = a$ et $0 \leq r < w$.

Terminaison : w est de la forme $2^p b$ au départ et est divisé par 2 à chaque itération. Il deviendra nécessairement égal à b après p itérations.

Validité : Il suffit d'établir que $q \times w + r = a$ et $0 \leq r < w$ reste vrai à chaque étape.
Cet invariant joint à la condition d'arrêt ($w = b$) prouve, ici encore, que l'on a bien calculé le quotient et le reste de la division euclidienne de a par b.

- C'est trivialement vrai avant la boucle.
- Montrons que cette propriété reste vraie à la fin de chaque itération. On commence par modifier q et w :
  $q' = 2 \times q$ et $w' = \frac{w}{2}$ (w étant divible par 2).
  Deux cas sont à envisager :
  - $r < w'$
    On a $q' = 2 \times q$, $w' = \frac{w}{2}$ et $r' = r$.
    $$\begin{aligned} q' \times w' + r' &= 2 \times q \times \tfrac{w}{2} + r \\ &= q \times w + r \\ &= a \text{ par hypothèse de récurrence .} \end{aligned}$$
    Par ailleurs on a $0 \leq r' < w'$
  - $w' \leq r$
    On a $q' = 2 \times q + 1$, $w' = \frac{w}{2}$ et $r' = r - w'$.
    $$\begin{aligned} q' \times w' + r' &= (2 \times q + 1) \times \tfrac{w}{2} + r - \tfrac{w}{2} \\ &= q \times w + \tfrac{w}{2} + r - \tfrac{w}{2} \\ &= q \times w + r \\ &= a \text{ par hypothèse de récurrence.} \end{aligned}$$
    On est dans le cas où $w' \leq r$ et $r' = r - w'$, ainsi on a $0 \leq r'$. D'autre part par hypothèse de récurrence $r < w$, donc $r - \frac{w}{2} < w - \frac{w}{2} = \frac{w}{2}$, ainsi $r' < w'$.
    Ainsi à la fin d'une étape on a encore les inégalités : $0 \leq r' < w'$.

1.4 Un principe fondamental : la dichotomie

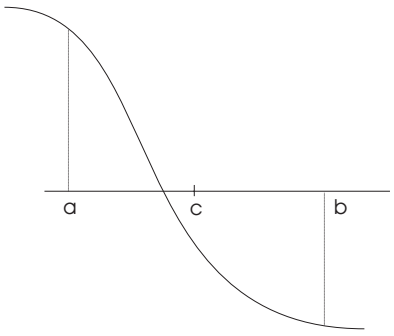
Le mot dichotomie provient du grec di (deux) et tomein (couper), ce qui signifie littéralement " couper en deux ". Ce principe très efficace et relativement facile à mettre en œuvre intervient dans de nombreux algorithmes :

- Calcul des zéros d'une fonction
- Algorithmes de recherche
- Algorithmes de classement
- ...

Sa mise en œuvre permet d'aborder et d'illustrer les problèmes fondamentaux que nous avons évoqués précédemment.

1.4.1 Méthode de dichotomie pour le calcul d'un zéro d'une fonction

Soit f une fonction continue sur un intervalle $[a, b]$, qui change de signe entre a et b. Le principe de dichotomie permet en divisant à chaque étape l'intervalle de moitié, en considérant le milieu c, de trouver une valeur approchée d'une racine de f à ϵ près. Pour atteindre la précision ϵ , arbitrairement petite, il suffit d'itérer ce processus n fois, où n est le plus petit entier tel $\frac{|a-b|}{2^n} \leq \epsilon$. On peut aussi dire que n est le plus petit entier supérieur ou égal à $log_2(\frac{|a-b|}{\epsilon})$.



À chaque étape, il faut veiller à ce que la fonction f ait une racine sur le nouvel intervalle. Cette condition, indispensable pour la validité de l'algorithme n'est pas toujours respectée, et bon nombre de versions que l'on peut rencontrer dans la littérature peuvent dans certains cas donner des résultats erronés. On rencontre aussi, très souvent, une erreur dans la condition d'arrêt : le test n'est pas fait sur la comparaison de la longueur de l'intervalle avec ϵ, mais sur la comparaison $|f(c)| \leq \epsilon$, où c est un point de l'intervalle qui contient une racine exacte. Clairement, $|f(c)| \leq \epsilon$ n'implique pas $|x_0 - c| \leq \epsilon$.

Un premier algorithme

Algorithme 8 (Recherche d'une racine par dichotomie)**.**
Entrées : a réel, b réel, $a < b$, f fonction continue sur $[a, b]$ telle que $f(a) * f(b) < 0$, ϵ un nombre réel arbitrairement petit.
Sorties : un nombre réel qui approche une racine x_0 de f à ϵ près.
Invariant : $(f(a) * f(b) < 0)$ ou ($a = b$ et $f(a) = 0$).

tant que $b - a > \epsilon$ **faire**
  $c \leftarrow \dfrac{a+b}{2}$
  si $f(c) = 0$ **alors**
    $a \leftarrow c$
    $b \leftarrow c$
  sinon si $f(a) * f(c) < 0$ **alors** $b \leftarrow c$ **sinon** $a \leftarrow c$
résultat a

Dans la pratique, il est peu probable que la variable c prenne pour valeur une des racines de la fonction f, ainsi le test $f(c) = 0$ devient inutile et nuit à l'efficacité de ce premier algorithme. En effet les valeurs données aux bornes de l'intervalle de départ sont très souvent des nombres entiers alors que les racines de f sont généralement des nombres irrationnels.

Un deuxième algorithme

Algorithme 9 (Recherche d'une racine par dichotomie, $2^{\text{ème}}$ version)**.**
Entrées : a réel, b réel, $a < b$, f fonction continue sur $[a, b]$ telle que $f(a) * f(b) \leq 0$, ϵ un nombre réel arbitrairement petit.
Sortie : un nombre réel qui approche une racine de f à ϵ près.
Invariant : $f(a) * f(b) \leq 0$
tant que $b - a > \epsilon$ **faire**
  $c \leftarrow \dfrac{a+b}{2}$
  si $f(a) * f(c) \leq 0$ **alors** $b \leftarrow c$ **sinon** $a \leftarrow c$
résultat a

On aurait pu aussi écrire autrement le corps de la boucle en gardant le même invariant :

Algorithme 10. Autre version de l'algorithme 9.
tant que $b - a > \epsilon$ **faire**
  $c \leftarrow \dfrac{a+b}{2}$
  si $f(a) * f(c) > 0$ **alors** $a \leftarrow c$ **sinon** $b \leftarrow c$
résultat a

Mais la version qui suit , bien que très proche, peut s'avérer erronée :

Algorithme 11. Version erronée de l'algorithme 9.
tant que $b - a > \epsilon$ **faire**
  $c \leftarrow \dfrac{a+b}{2}$
  si $f(a) * f(c) < 0$ **alors** $b \leftarrow c$ **sinon** $a \leftarrow c$
résultat a

En effet, l'invariant n'est plus satisfait dans le cas où le milieu de l'intervalle c est par "malchance" une racine de la fonction f. Ce cas, peu fréquent, comme on l'a déjà dit, peut néanmoins se produire, et l'algorithme ci-dessus fournit alors un résultat faux ! Dans cette configuration, le produit $f(a)*f(c)$ est nul, a prend donc la valeur c, et par la suite le produit $f(a) * f(c)$ n'étant jamais négatif, on trouve comme valeur approchée de la racine un point proche de b à ϵ près ! À chaque itération les algorithmes précédents évaluent deux fois la fonction f et effectuent un produit, on peut être plus efficace en écrivant les choses différemment.

Un troisième algorithme

Algorithme 12 (Recherche d'une racine par dichotomie, $3^{\text{ème}}$ version)**.**
Entrées : a réel, b réel, f fonction continue sur $[a, b]$ telle que $f(a) \leq 0$ et $f(b) > 0$, ϵ un nombre réel arbitrairement petit.
Sortie : un nombre réel qui approche une racine de f à ϵ près.
Invariant : $(f(a) \leq 0$ et $f(b) > 0$
tant que $b - a > \epsilon$ **faire**
  $c \leftarrow \dfrac{a+b}{2}$
  si $f(c) \leq 0$ **alors** $a \leftarrow c$ **sinon** $b \leftarrow c$
résultat a

1.5 Quelques algorithmes élémentaires en arithmétique

Nous allons présenter ici quelques algorithmes élémentaires en arithmétique comme création de la liste des nombres premiers, décomposition d'un nombre entier en facteurs premiers, liste des diviseurs d'un nombre entier, changement de base, ...

1.5.1 Construction de la liste des nombres premiers par la méthode du crible d'Erathostène

Cette méthode permet d'établir la liste de nombres premiers inférieurs à un nombre entier n donné. Son principe est très simple. On écrit la séquence de tous les nombres entiers de 1 jusqu'à n, on barre 1 qui n'est pas premier, on garde 2 qui est premier et on barre tous les nombres multiples de 2, on garde 3 et on barre tous ses multiples, puis on recherche à partir de 3 le premier nombre non barré, on le garde et on élimine, en les barrant, tous les multiples de ce nombre, et on continue ainsi jusqu'à épuiser toute la liste. Les nombres non barrés constituent la liste

des nombres premiers inférieurs ou égaux à n. Avant d'écrire plus précisément cet algorithme, il nous faut choisir une représentation informatique qui traduit le fait qu'un nombre soit barré ou non. Pour cela on peut utiliser un tableau de n éléments indexés de 1 jusqu'à n. Les composantes de ce tableau pouvant avoir soit une valeur booléenne (**vrai**, **faux**), soit une des deux valeurs 0 ou 1. Ainsi la valeur de la composante de rang i nous indiquera si le nombre i est premier ou non : la valeur **vrai** ou 1 signifie oui, la valeur **faux** ou 0 signifiant non.

Algorithme 13 (Crible d'Erathostène)**.**
Entrée : un nombre entier n.
Sortie : la liste des nombres premiers inférieurs ou égaux à n.

pour i **variant de** 1 **jusqu'à** n **faire** $t[i] \leftarrow$ **vrai**
$t[1] \leftarrow$ **faux**
pour i **variant de** 2 **jusqu'à** n
si $t[i]$ **alors** **** On barre les multiples de i ****
  $k \leftarrow 2$
  tant que $k * i \leq n$ **faire**
    $t[k * i] \leftarrow$ **faux**
    $k \leftarrow k + 1$
$L \leftarrow [\,]$ **** On initialise la liste L avec la liste vide ****
pour i **variant de** 2 **jusqu'à** n
si $t[i]$ **alors** $L \leftarrow L, i$ **** i est un nombre premier on l'ajoute à la liste L ****
résultat L

On peut écrire à partir de cet algorithme une version un peu plus performante en tenant compte des remarques suivantes :

- On peut passer d'un multiple m de i au multiple suivant en y ajoutant i, une addition est plus rapide qu'une multiplication.
- Il est inutile d'examiner les multiples m de i inférieurs à i^2, car ils ont été déjà barrés.
- Il est inutile de chercher à barrer des nombres plus grands que $\sqrt{n}$, car tous ceux qui ne sont pas premiers l'ont déjà été. En effet un nombre plus grand que $\sqrt{n}$ qui n'est pas premier a forcément un facteur premier plus petit que $\sqrt{n}$, donc il aura été barré lorsque les multiples de ce facteur l'ont été.

Ce qui conduit à écrire l'algorithme suivant :

Algorithme 14 (Crible d'Erathostène, version optimisée)**.**

pour i **variant de** 1 **jusqu'à** n **faire** $t[i] \leftarrow$ **vrai**
$t[1] \leftarrow$ **faux**
pour i **variant de** 2 **jusqu'à** $\sqrt{n}$
si $t[i]$ **alors** **** On barre les multiples de i ****
  $m \leftarrow i^2$
  tant que $m \leq n$ **faire**
    $t[m] \leftarrow$ **faux**
    $m \leftarrow m + i$
$L \leftarrow [\,]$ **** On initialise la liste L avec la liste vide ****
pour i **variant de** 2 **jusqu'à** n
si $t[i]$ **alors** $L \leftarrow L, i$ **** i est un nombre premier on l'ajoute à la liste L ****
résultat L

1.5.2 Décomposition d'un entier n en produit de facteurs premiers

On sait qu'un nombre entier se décompose de manière unique en un produit de facteurs premiers. Le principe de la décomposition est très simple. Il consiste à essayer de diviser n par les nombres premiers successifs. Lorsqu'un nombre premier p divise n, il faut continuer à diviser n par ce nombre tant que cela est possible afin de connaître l'exposant de p.
Voici l'algorithme :

Algorithme 15 (Décomposition d'un entier n en produits de facteurs premiers)**.**
Entrée : un nombre entier naturel n.
Sortie : la liste des nombres premiers $p_1, p_2, \ldots, p_r$ et de leur exposant respectif $e_1, e_2, \ldots, e_r$ tel que n soit égal à $p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$.

$P \leftarrow$ la liste des nombres premiers inférieurs ou égaux à n
$F \leftarrow [\,]$ **** On initialise la liste des facteurs F avec la liste vide ****
$E \leftarrow [\,]$ **** On initialise la liste des exposants E avec la liste vide ****
$i \leftarrow 1$
tant que $n \neq 1$ **faire**
  si p_i ne divise pas n **alors** $i \leftarrow i + 1$
  sinon
    $k \leftarrow 0$
    tant que p_i divise n **faire**
      $n \leftarrow n/p_i$
      $k \leftarrow k + 1$
    $F \leftarrow F, p_i$ **** p_i est un facteur premier on l'ajoute à la liste F ****
    $E \leftarrow E, k$ **** On ajoute son exposant k à la liste E des exposants ****
résultat F, E

1.5.3 Décomposition en base b d'un entier a

De façon générale, l'écriture d'un nombre a en base b est définie par la formule

$$(a_k a_{k-1} \dots\ a_1 a_0)_b = a_k b^k + a_{k-1} b^{k-1} + \dots\ + a_1 b + a_0 \tag{1.3}$$

où les coefficients a_i sont des nombres entiers vérifiant $0 \leq a_i < b$. Le système décimal étant le cas particulier où b est égal à 10, les coefficients a_i prenant pour valeur un des dix chiffres $0, 1, 2, \dots\ , 9$.

Décomposer un nombre a en base b revient à trouver les $k+1$ coefficients a_i, $0 \leq a_i < b$, tels que $a = a_k b^k + a_{k-1} b^{k-1} + \dots\ + a_1 b + a_0$. Pour cela il suffit de savoir faire la division euclidienne de a par b dans le système décimal. En effet, on obtient facilement le coefficient de poids le plus faible a_0 en calculant le reste de la division de a par b, puis on considère le nombre entier a', quotient de a par b,

$$a' = a \textbf{ div } b = a_k b^{k-1} + a_{k-1} b^{k-2} + \dots\ + a_2 b + a_1 \tag{1.4}$$

Le reste de la division euclidienne de a' par b nous donne a_1, et ainsi de suite pour obtenir les autres coefficients de l'écriture binaire de a. On peut choisir de ranger les coefficients soit dans une liste, soit dans une table.

Algorithme 16 (Décomposition d'un entier n en base b avec une liste)**.**
Entrée : Un nombre n écrit en base 10.
Sortie : La liste L des coefficients a_i de l'écriture de n en base b.

$a \leftarrow n$
$L \leftarrow [\,]$
tant que $a \neq 0$ **faire**
  $r \leftarrow a$ **mod** b
  $L \leftarrow r, L$
  $a \leftarrow a$ **div** b
résultat $L = [\, a_k, a_{k-1}, \dots\ a_1, a_0\,]$

Algorithme 17 (Conversion d'un entier n en base b avec une table)**.**
Entrée : Un nombre n écrit en base 10.
Sortie : La table T des coefficients a_i de l'écriture de n en base b.

$a \leftarrow n$
$i \leftarrow 0$
tant que $a \neq 0$ **faire**
  $r \leftarrow a$ **mod** b
  $T_i \leftarrow r$
  $i \leftarrow i + 1$
  $a \leftarrow a$ **div** b
résultat $T = [\, a_0, a_1, \dots\ a_{k-1}, a_k\,]$

1.5.4 Conversion d'un nombre écrit en base b en décimal

Soit $(a_k a_{k-1} \dots\ a_1 a_0)_b$ l'écriture en base b du nombre que l'on veut convertir en écriture décimale. Pour obtenir cette conversion, il suffit d'évaluer $a_k b^k + a_{k-1} b^{k-1} + \dots\ + a_1 b + a_0$ en effectuant les calculs en base 10.
On suppose dans les algorithmes présentés ci-dessous que les coefficients de l'écriture en base b sont rangés dans une table.

Algorithme 18 (Conversion en décimal d'un nombre écrit en base b)**.**
Entrée : Un nombre $(a_k a_{k-1} \dots\ a_1 a_0)_b$ écrit en base b.
Sortie : Le nombre n qui correspond à la conversion de $(a_k a_{k-1} \dots\ a_1 a_0)_b$ en base 10.

$n \leftarrow 0$
pour i **variant de** 0 **jusqu à** k **faire**
  $n \leftarrow n + a_i \times b^i$
résultat n

L'algorithme précédent n'est pas performant car il effectue un trop grand nombre de multiplications. En effet pour calculer b^i, il est nécessaire de faire $i-1$ multiplications, ainsi à l'étape i on effectue i multiplications, soit au total $1+2+\dots+k = \frac{k\times(k+1)}{2}$ multiplications. C'est beaucoup trop !
Voici un algorithme qui n'en effectue plus que $2(k+1)$ pour calculer la même expression.

Algorithme 19 (Conversion en décimal d'un nombre écrit en base b)**.**
Entrée : Un nombre $(a_k a_{k-1} \dots\ a_1 a_0)_b$ écrit en base b.
Sortie : Le nombre n qui correspond à la conversion de $(a_k a_{k-1} \dots\ a_1 a_0)_b$ en base 10.

$f \leftarrow 1$
$n \leftarrow 0$
pour i **variant de** 0 **jusqu à** k **faire**
  $n \leftarrow n + a_i \times f$
  $f \leftarrow f \times b$
résultat n

On peut encore diminuer le nombre de multiplications en utilisant l'algorithme de Hörner, présenté dans la section suivante, pour évaluer l'expression polynomiale $a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b + a_0$.

Algorithme 20 (Conversion en décimal d'un nombre écrit en base b)**.**
Entrée : Un nombre $(a_k a_{k-1} \dots a_1 a_0)_b$ écrit en base b.
Sortie : Le nombre n qui correspond à sa conversion en base 10.

$n \leftarrow 0$
pour i **variant de** k **jusqu à** 0 **avec un pas de** -1 **faire**
  $n \leftarrow n \times b + a_i$
résultat n

Cet algorithme n'effectue plus que $k+1$ multiplications pour évaluer l'expression, on ne peut pas faire mieux. Les trois algorithmes précédents effectuent le même nombre k d'additions.

1.5.5 L'algorithme de Hörner

L'algorithme de Hörner est très connu et très facile à mettre en oeuvre. Soit x un nombre réel et $p(x) = a_k x^k + a_{k-1}x^{k-1} + \ldots + a_1 x + a_0$ un polynôme de degré k dont les coefficients sont aussi des réels, a_k distinct de 0. En remarquant qu'on peut écrire le polynôme sous la forme suivante :
$p(x) = (a_k x^{k-1} + a_{k-1}x^{k-2} + \ldots + a_2 x + a_1)x + a_0$
$p(x) = ((a_k x^{k-2} + a_{k-1}x^{k-3} + \ldots + a_2)x + a_1)x + a_0$
$p(x) = (((a_k x^{k-3} + a_{k-1}x^{k-4} + \ldots + a_3)x + a_2)x + a_1)x + a_0$
.....
$p(x) = ((\ldots((a_k)x + a_{k-1})x + \ldots + a_2)x + a_1)x + a_0$
Ainsi pour calculer $p(x)$, on organise les calculs de manière à calculer successivement les valeurs $b_k, b_{k-1}, \ldots, b_0$ définies par :
$b_k = a_k$
$b_{k-1} = b_k * x + a_{k-1}$
...
$b_i = b_{i+1}x + a_i$
...
$b_0 = b_1 x + a_0$
On a donc $b_0 = p(x)$, c'est ce procédé qui est parfois appelé "schéma de Hörner".

Algorithme 21 (Schéma de Hörner)**.**
Entrées : Un tableau a de $k+1$ coefficients indexés de 0 à k et un nombre réel x.
Sortie : Le nombre $s = \sum_{i=0}^{k} a_i x^i$

$s \leftarrow 0$
pour i **variant de** k **jusqu à** 0 **avec un pas de** -1 **faire**
  $s \leftarrow s \times x + a_i$
résultat s

1.5.6 Construction d'un nombre décimal n inférieur à 1 à partir de la liste a de ses chiffres

Cet algorithme est encore une application directe de l'algorithme de Hörner. En effet si la liste a est [a_1,a_2,...,a_{k-1},a_k], le nombre n corrrespondant est :
$n = a_k 10^{-k} + a_{k-1} 10^{-(k-1)} + \ldots + a_1 10^{-1}$
Ce qui conduit à écrire l'algorithme suivant :

Algorithme 22 (Schéma de Hörner appliqué à la construction d'un nombre décimal plus petit que 1)**.**
Entrée : un tableau de chiffres a indexé de 1 à k
Sortie : un nombre décimal $n = \sum_{i=1}^{k} a_i 10^{-i}$ inférieur à 1
$n \leftarrow 0$
pour i **variant de** k **jusqu à** 1 **avec un pas de** -1 **faire**
  $n \leftarrow \frac{n}{10} + a_i$
résultat n

1.5.7 Génération d'un nombre entier aléatoire n dont la décomposition binaire comporte exactement k bits

Cet algorithme est encore une application directe de l'algorithme de Hörner. En effet il suffit que le bit de poids fort soit égal à 1 et de choisir aléatoirement entre 0 et 1 pour les $k-1$ autres bits. On évaluera au fur et à mesure l'entier décimal correspondant, exactement comme dans l'algorithme "Hörner" dans le cas où la base est 2. Voici l'algorithme correspondant :

Algorithme 23 (Génération d'un nombre entier aléatoire n dont la décomposition binaire comporte exactement k bits)**.**
Entrée : k un nombre entier
Sortie : un nombre entier aléatoire n écrit en décimal dont la décomposition binaire comporte exactement k bits
$n \leftarrow 1$
pour i **variant de** 1 **jusqu à** k **faire**
  $n \leftarrow 2 \times n + hasard(2)$
résultat n

où $hasard$ est la fonction qui renvoie pour résultat un nombre aléatoire supérieur ou égal à 0 et strictement plus petit que son argument.

1.5.8 Développements décimaux illimités

Présentation

Un nombre rationnel peut être représenté soit par son écriture fractionnaire $Q = \dfrac{N}{D}$ où N est un entier relatif et D un entier naturel non nul ou par un développement décimal illimité périodique de la forme :

$Q = 0, \underbrace{r_1 r_2 \cdots r_i}_{\text{Partie régulière}} \underbrace{\overline{p_1 p_2 \cdots p_j}}_{\text{Partie périodique}} \times 10^n$ où les r_k et les p_k sont des chiffres du système décimal et n un entier naturel.

Obtention de la fraction à partir du développement

Si on pose $a = 0, r_1 r_2 \cdots r_i$ et $b = 0, \overline{p_1 p_2 \cdots p_j}$ on a alors :

$Q = (a + b \times 10^{-i}) \times 10^n$

avec $a = \dfrac{r_1 r_2 \cdots r_i}{10^i}$ et $10^j \times b = p_1 p_2 \cdots p_j, \overline{p_1 p_2 \cdots p_j} = p_1 p_2 \cdots p_j + b$

d'où $b = \dfrac{p_1 p_2 \cdots p_j}{10^j - 1}$

donc $Q = \left(\dfrac{r_1 r_2 \cdots r_i}{10^i} + p_1 p_2 \cdots p_j \times \dfrac{10^{-i}}{10^j - 1} \right) \times 10^n$

$Q = \dfrac{(r_1 r_2 \cdots r_i) \times (10^j - 1) + (p_1 p_2 \cdots p_j)}{10^i \times (10^j - 1)} \times 10^n$

Il suffit maintenant de rendre cette fraction irréductible.

Il n'y a donc pas d'algorithme de calcul de Q sous forme de fraction mais simplement une méthode mathématique.

Obtention du développement à partir de la fraction

$Q = \dfrac{N}{D} = E + \dfrac{N'}{D}$ où E est la partie entière de Q et $\dfrac{N'}{D}$ sa partie décimale (illimitée)

On aura donc :

$\dfrac{N'}{D} = 0, \underbrace{r_1 r_2 \cdots r_i}_{\text{Partie régulière}} \underbrace{\overline{p_1 p_2 \cdots p_j}}_{\text{Partie périodique}}$

Pour déterminer les suites de décimales r_k et p_k ainsi que i et j on construit les suites (N_k), (Q_k) et (R_k) de la façon suivante :

- $R_k = N_k$ *mod* D (reste euclidien de N_k par D)
- $Q_k = \dfrac{N_k - R_k}{D}$ (quotient euclidien de N_k par D)
- $N_{k+1} = 10 \times R_k$ avec $R_0 = N'$

Remarques

a) R_k peut prendre D valeurs entières comprises entre 0 et $D - 1$

b) Si $R_k = 0$ alors tous les restes suivants sont nuls

c) Il en découle que $i + j < D$

Calcul des décimales

Il faut vérifier que les quotients calculés sont bien les décimales du développement cherché

Procédons par récurrence sur k

On a bien $N_1 = 10 \times N'$ et $Q_1 = \text{partie entière}\left(\frac{N_1}{D}\right) = r_1$ puisque $\frac{10 \times N'}{D} = r_1, r_2 \cdots$

Si on suppose qu'au rang k, $\frac{N_k}{D} = r_k, r_{k+1} r_{k+2} \cdots$ alors

$Q_k = \text{partie entière}\left(\frac{N_k}{D}\right) = r_k$ et $\frac{N_{k+1}}{D} = \frac{10 \times R_k}{D} = \frac{10 \times (N_k - D \times Q_k)}{D} =$

$10\left(\frac{N_k}{D} - Q_k\right) = 10(r_k, r_{k+1} r_{k+2} \cdots - r_k) = 10 \times (0, r_{k+1} r_{k+2} \cdots)) = r_{k+1}, r_{k+2} \cdots$

donc $Q_{k+1} = \text{partie entière}\left(\frac{N_{k+1}}{D}\right) = r_{k+1}$

Conclusion : les Q_k sont les chiffres du développement décimal illimité associé à $\frac{N'}{D}$.

Calcul de i et j

D'après la remarque a) à partir d'un certain rang on va retrouver un des restes précédents.

Soit i et j les plus petits entiers tels que $R_{i+j} = R_i$. On a donc $Q_{i+j+1} = Q_{i+1}$ et $p_{j+1} = r_{i+j+1} = Q_{i+j+1} = Q_{i+1} = r_{i+1} = p_1$ ce qui prouve que la période du développement décimal est j.

Algorithme 24 (Développement décimal d'un nombre rationnel $\frac{N}{D}$)**.**

Entrée : Un nombre rationnel sous la forme $\frac{N}{D}$.

Sortie : Le quotient entier e de $\frac{N}{D}$, la liste $NonPeriodique$ des chiffres de la partie non périodique, la liste $Periodique$ des chiffres de la partie périodique.

$num \leftarrow N$
$den \leftarrow D$
**** Initialisation, $TabReste[i]$ indique la position où le reste i a été obtenu****
pour i **variant de** 0 **jusqu'à** $den - 1$ **faire** $TabReste[i] \leftarrow 0$
$e \leftarrow num$ **div** den
$r \leftarrow num$ **mod** den
**** Initialisation de la liste des quotients avec la liste vide****
$ListeQuotient \leftarrow [\,]$
$i \leftarrow 0$
tant que $TabReste[r] = 0$ **faire**
    $i \leftarrow i + 1$
    $TabReste[r] \leftarrow i$
    $num \leftarrow 10 \times r$
    $q \leftarrow num$ **div** den
    $ListeQuotient \leftarrow ListeQuotient, q$
    $r \leftarrow num$ **mod** den
$k \leftarrow TabReste[r]$
$NonPeriodique \leftarrow ListeQuotient[0..k-1]$
$Periodique \leftarrow ListeQuotient[k..i]$
résultat $e, Nonperiodique, Periodique$

1.6 Les algorithmes de la brochure avec Excel et VBA

VBA (Visual Basic for Applications) fait partie intégrante d'Excel.

En tapant ALT+F11 dans Excel vous accédez à VBA et il suffit d'ajouter un module pour pouvoir écrire des programmes et les tester dans Excel.

1.6.1 Les formules avec Excel

Le =

Une formule Excel commence toujours par le symbole =

Les références

- A3 fait référence à la cellule située à la première colonne et troisième ligne du tableau.
- A3 ;B7 fait référence à deux cellules du tableau (énumération).
- A3 :B7 fait référence à une région rectangulaire du tableau allant de la cellule A3 à la cellule B7.
- Par exemple :
  A8 = somme(A3 ; B7) calcule dans A8 la somme de deux cellules
  A8 = somme(A3 : B7) calcule dans A8 la somme de $2 \times 5 = 10$ cellules.

La recopie des formules

Lorsqu'une formule est recopiée d'une cellule vers une autre et que la recopie correspond à une translation de l lignes et c colonnes dans la grille, toutes les références à des cellules sont décalées de l lignes et c colonnes.

Par exemple si on a C3 = C2 + B3 et que l'on recopie la cellule C3 dans la cellule D5 ce qui correspond à un décalage de deux lignes et d'une colonne, on pourra lire dans la cellule D5 = D4 + C5.

Cette fonctionnalité sera très utile pour recopier une formule dans une ligne ou une colonne.

Le $

Lors d'une recopie, il est parfois nécessaire qu'une référence ne soit pas "décalée". Pour cela il suffit de mettre un "$" devant la référence de ligne ou de colonne pour "figer" cette référence.

Par exemple si on a C3 = $C2 + B$3 et que l'on recopie la cellule C3 dans la cellule D5, on pourra lire dans la cellule D5 = $C4 + C$3.

Dans la première référence, on a figé le numéro de ligne mais pas l'indice de colonne et dans la seconde, c'est le contraire.

Le SI$\cdots$ALORS$\cdots$SINON

Cette formule correspond à une affectation conditionnelle.

Par exemple, la formule suivante C3 = SI (C2 $>$ 0 ; C2 + B3 ; B3) correspond à l'instruction suivante :

$$\textit{Si}\ \ \text{C2} > 0\ \ \textit{alors}\ \ \text{C3} \longleftarrow C2 + B3\ \ \textit{sinon}\ \ \text{C3} \longleftarrow B3$$

la syntaxe générale est
SI (<condition> ; <valeur-si-vrai>) ou
SI (<condition> ; <valeur-si-vrai> ; <valeur-si-faux>)

Les fonctions Excel

Excel dispose de nombreuses fonctions mathématiques, statistiques etc.

Pour en voir la liste il suffit de cliquer sur l'icône $\boxed{f_x}$

1.6.2 Le langage Visual Basic for Applications

Présentation

C'est un langage de programmation avec des sous-programmes (Sub) et des fonctions (Function) et qui permet aussi la programmation objet.

Les variables sont réparties dans de nombreux types.

Les fonctions crées dans VBA sont utilisables dans Excel de la même façon que les fonctions Excel de base qui sont elles aussi écrites en VBA.

Les types de variables

Les variables utilisées dans le cadre des mathématiques seront de type :

- **Integer** ou **long** si elles doivent contenir des entiers
- **Single** ou **Double** si elles doivent contenir des nombres à virgule
- **String** si elles correspondent à des chaînes de caractères
- **Variant** pour des variables pouvant contenir n'importe quel type de données
- **Array** si ce sont des tableaux.

Les instructions

- **=** désigne aussi bien l'affectation que le test d'égalité (c'est le contexte qui décide)
- **If**$\cdots$**Then**$\cdots$**Else**$\cdots$**Endif** ou **If**$\cdots$**Then**$\cdots$**Endif** est l'instruction conditionnelle
- **Do**$\cdots$**Loop** indique le début et la fin d'une boucle
- **While** $\cdots$ est synonyme de Tant que $\cdots$
- **For** $\cdots$**to**$\cdots$ $\cdots$**Next**$\cdots$ indique un groupe d'instructions à répéter un certain nombre de fois.

Les fonctions

- **Function**$\cdots$**End** Function indique le début et la fin d'une déclaration de fonction.
- Le mot Function est suivi sur la même ligne du nom de la fonction et des paramètres de la fonction entre parenthèses. Ces paramètres servent à fournir à la fonction les données dont elle a besoin pour fonctionner.
- Les variables déclarées dans la fonction après l'instruction **Dim** ne seront visibles et utilisables qu'à l'intérieur de la fonction (variables locales).
- La dernière instruction de la fonction doit être de la forme :
  nom-de-la-fonction = valeur-à-retourner
  Cette instruction détermine quel sera le résultat de la fonction lorsqu'elle sera utilisée.

1.6.3 La multiplication simple

Avec Excel

	A	B	C
1	**A**	**B**	**R**
2	12	5	0
3	12	4	12
4	12	3	24
5	12	2	36
6	12	1	48
7	12	0	60

Formules A3 = A2 B3 = SI (B2 <> 0 ; B2 - 1 ; 0) correspond à $y \leftarrow y-1$
C3 = SI (B2 = 0 ; C2 ; C2 + A2) correspond à $z \leftarrow z + x$
et ces formules contiennent le test d'arrêt **tant que** $y \neq 0$ **faire**.

Ces formules sont ensuite recopiées vers le bas dans les trois colonnes.

Programme en VBA

```
Public Function MultSimple(a As Integer, b As Integer) As Integer
Dim r As Integer
r = 0
Do While b <> 0
 r = r + a
 b = b - 1
Loop
MultSimple = r
End Function
```

1.6.4 La multiplication égyptienne

Avec Excel

	A	B	C
1	**A**	**B**	**R**
2	40	27	0
3	80	13	40
4	160	6	120
5	320	3	120
6	640	1	440
7	1280	0	1080

Formules

A3 = 2 * A2 correspond à $a \leftarrow 2 \times a$

B3 = quotient (B2 ; 2) correspond à $b \leftarrow b$ **div** 2

C3 = SI (reste (B2 ; 2) = 0 ; C2 ; C2 + A2) correspond à **si** b impair **alors** $r \leftarrow r + a$

L'algorithme se termine lorsque 0 apparaît dans la colonne B.

Ces formules sont recopiées vers le bas dans les trois colonnes.

Remarque : quotient et reste sont des fonctions écrites en VBA qui correspondent à la division euclidienne.

Programme en VBA

```
Public Function Reste(a As Integer, b As Integer) As Integer
Reste = a Mod b
```

```
End Function

Public Function Quotient(a As Integer, b As Integer) As Integer
Quotient = (a - a Mod b) / b
End Function

Public Function MultEgypt(a As Integer, b As Integer) As Integer
Dim r As Integer

r = 0
Do While b <> 0
    If Reste(b, 2) = 1 Then
        r = r + a
    End If
    a = 2 * a
    b = Quotient(b, 2)
Loop
MultEgypt = r
End Function
```

1.6.5 La multiplication décimale

Avec Excel

	A	B	C
1	**A**	**B**	**R**
2	40	27	0
3	400	2	280
4	4000	0	1080

Formules

A3 = 10 * A2 correspond à $a \leftarrow 10 \times a$

B3 = quotient (B2 ; 10) correspond à $b \leftarrow b$ **div** 10

C3 = C2 + A2 * reste (B2 ; 10) correspond à $r \leftarrow r + a \times (b$ **mod** $10)$

L'algorithme se termine lorsque 0 apparaît dans la colonne B.

Ces formules sont recopiées vers le bas dans les trois colonnes.

Programme en VBA

```
Public Function MultDecimale(a As Integer, b As Integer) As Integer
Dim r As Integer
r = 0
Do While b <> 0
```

```
        r = r + a * Reste(b, 10)
        a = 10 * a
        b = Quotient(b, 10)
    Loop
    MultDecimale = r
    End Function
```

1.6.6 Algorithme d'euclide

Avec Excel

	A	B	C
1	A	B	
2	92	100	
3			
4	A	B	R
5	92	100	
6	100	92	92
7	92	8	8
8	8	4	4
9	4	0	0

Formules

A5 = A2 B5 = B2

C6 = reste (A5 ; B5) correspond à $r \leftarrow a$ **mod** b

A6 = B5 correspond à $a \leftarrow b$

B6 = D5 correspond à $b \leftarrow r$

L'algorithme se termine lorsque 0 apparaît dans la colonne B.

Ces formules sont recopiées vers le bas dans les trois colonnes.

Programme en VBA

```
Public Function PGCD(a As Integer, b As Integer)
Dim r As Integer, q As Integer
Do While b <> 0
    q = Quotient(a, b)
    r = a - b * q
    a = b
    b = r
```

```
Loop
PGCD = a
End Function
```

1.6.7 Méthode de dichotomie

Avec Excel

	A	B	C	D	E
1	Calcul approché de la racine cubique de 2 par dichotomie				
2					
3	$f(x)=x^3-2$				
4					
5	**a**	**b**	**c**	**f(c)**	**b-a**
6	1	2			
7	1	1,5	1,5	1,375	0,5
8	1,25	1,5	1,25	-0,046875	0,25
9	1,25	1,375	1,375	0,59960938	0,125
10	1,25	1,3125	1,3125	0,26098633	0,0625
11	1,25	1,28125	1,28125	0,103302	0,03125
12	1,25	1,265625	1,265625	0,02728653	0,015625
13	1,2578125	1,265625	1,2578125	-0,01002455	0,0078125
14	1,2578125	1,26171875	1,26171875	0,00857323	0,00390625

Formules

C7 = (A6 + B6) / 2 correspond à $c \leftarrow \dfrac{a+b}{2}$

D7 = C7 ^ 3 - 2 correspond à $y \leftarrow f(c)$

E7 = B7 - A7 correspond à $e \leftarrow b - a$

A7 = SI (D7 <= 0 ; C7 ; A6) correspond à **si** $f(c) \leq 0$ **alors** $a \leftarrow c$

B7 = SI (D7 <= 0 ; B6 ; C7) correspond à **si** $f(c) > 0$ **alors** $b \leftarrow c$

et ces formules sont recopiées vers le bas.

Programme en VBA

```
Public Function f(x As Double) As Double
 = x * x * x - 2
End Function

Public Function Dichotomie(a As Double, b As Double, epsi As Double)
As Double
```

```
Dim c As Double, ya As Double
ya = f(a)
Do While b - a > epsi
    c = (a + b) / 2
    If ya * f(c) <= 0 Then
        b = c
    Else
        a = c
    End If
Loop
Dichotomie = a
End Function
```

1.6.8 Crible d'Eratosthène

Avec Excel

La présence d'une double boucle fait que cet algorithme ne peut pas être simulé directement avec Excel.

Programme en VBA

```
Public Function Crible(nmax As Integer) As String
Dim i As Integer, j As Integer, Liste As String
Dim T(1 To 2000) As Boolean

For i = 1 To nmax
    T(i) = True
Next i

T(1) = False
i = 2
Liste = ""
Do While i * i <= nmax
    If T(i) = True Then
        j = i + i
        Liste = Liste & "-" & Str(i)
        Do While j <= nmax
            T(j) = False
            j = j + i
        Loop
    End If
```

```
        i = i + 1
    Loop
    ' il faut maintenant ajouter à la liste
    'les nombres premiers restants
    For j = i To nmax
        If T(j) Then
            Liste = Liste & "-" & Str(j)
        End If
    Next j
    Crible = Liste
    End Function
```

1.6.9 Décomposition en produit de facteurs premiers

Avec Excel

	A	B	C	D	E	F	G
1	nombre	N° diviseur	Diviseur	Reste	facteurs premiers	Nombres premiers →	2
2	175	1	2				3
3	175	2	3	1	1		5
4	175	3	5	1	1		7
5	35	3	5	0	5		11
6	7	3	5	0	5		13
7	7	4	7	2	1		17
8	1	4	7	0	7		19
9	1	5	11	1	1		23

Nous avons vu que nous ne pouvions pas simuler dans Excel la double boucle de la recherche des nombres premiers, donc pour réaliser avec Excel la décomposition en produit de facteurs premiers, nous devons disposer d'une liste de nombres premiers. Cette liste est écrite dans la colonne G.

Formules

B2 = 1 correspond à l'initialisation $i \leftarrow 1$

C2 = INDEX (G :G ; B2 ; 1) correspond à $c \leftarrow p_i$ car INDEX renvoie la valeur située dans la colonne G à la ligne dont l'indice est donné par B2 (ici il renvoie donc la valeur de la cellule G1)

D3 = RESTE(A2 ; C2) correspond à $r \leftarrow n$ **mod** p_i

A3 = SI (D3 = 0 ; A2 / C2 ; A2) correspond à **si** $r = 0$ **alors** $n \leftarrow n/p_i$

B3 = SI (D3 <> 0 ; B2 + 1 ; B2) correspond à **si** $r \neq 0$ **alors** $i \leftarrow i + 1$

C3 = INDEX (G :G ; B3 ; 1) Calcule la valeur de p_i
E3 = SI (D3 = 0 ; C2 ; 1) **si** $r = 0$ **alors** on ajoute p_i à la liste des diviseurs
Ces formules sont ensuite recopiées vers le bas.

Programme en VBA

```
Public Function decomposition(n As Integer) As String
Dim T, Nombre As Integer, NumDiv As Integer, Diviseur As Integer, Liste As String
Const nmax = 2500
If n > nmax Then Exit Function
T = Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47)
Nombre = n : NumDiv = 0
Diviseur = T(NumDiv) : Liste = ""
Do While Nombre <> 1
    If Reste(Nombre, Diviseur) <> 0 Then
        NumDiv = NumDiv + 1
        Diviseur = T(NumDiv)
    Else
        Liste = Liste & Diviseur & "*"
        Nombre = Nombre / Diviseur
    End If
Loop
decomposition = Liste & 1
End Function
```

1.6.10 Décomposition en base b

Avec Excel

	A	B	C	D	E	F
1	Base<=10	B=	12		Dividende	Chiffres
2					971	
3	En base 10	N=	971		80	11
4					6	8
5					0	6
6	En base	12	68B		0	0

Formules

$L \leftarrow r, L$

E2 = C3 correspond à l'initialisation de a
F3 = reste (E2 ; C1) correspond à $r \leftarrow a$ **mod** b
E3 = quotient (E2 ; C1) correspond à $n \leftarrow a$ **div** b
Ces formules sont ensuite recopiées vers le bas.
L'algorithme s'arrête lorsque $n = 0$.
La liste des chiffres peut être lue dans la colonne F.

Programme en VBA

```
Public Function VersBase(b As Integer, N As Integer) As String
Dim Mot As String, Chiffre As String

Dim T
T = Array("0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D",
"E", "F")
Mot = ""

Do While N <> 0
    Chiffre = Reste(N, b)
    Mot = T(Chiffre) & Mot
    N = Quotient(N, b)
Loop
VersBase = Mot
End Function
```

1.6.11 Conversion en décimal

Avec Excel

	A	B	C	D	E	F	G	H	I	J
1	Base	B=	7							
2										
3	En base B	N=	1	0	5	6	4	0	1	
4										
5			1	7	54	384	2692	18844	131909	
6										
7	En base 10	N=	131909							

Le nombre en base b est écrit "à l'envers" dans la ligne 3
Le résultat est le plus grand des nombres écrits dans la ligne 5
Le cas des bases supérieures à 10 n'est pas traité avec Excel

Formules

C5 = C3

D5 = SI (D3 = "" ; "" ; C5 * C1 + D3) correspond à l'instruction $s \leftarrow s \times b + a_i$ où a_i correspond au chiffre de la cellule D3. Le test permet l'arrêt de l'algorithme lorsqu'il n'y a plus de chiffres à traiter.

Cette formule est ensuite recopiée vers la droite.

Programme en VBA

```
Public Function EnDecimal(N As String, b As Integer) As Long
Dim i As Integer, c As Integer, Chiffre As String
Dim Resultat As Long
Dim T
T = Array("0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D",
"E", "F")

Resultat = 0
For i = 1 To Len(N)
    Chiffre = Mid(N, i, 1) 'on extrait le chiffre N° i
 ' on recherche le chiffre dans la table
    For j = 0 To 15
        If T(j) = Chiffre Then
            c = j
        End If
    Next j
    Resultat = Resultat * b + c  '    algorithme de Hörner
```

```
    Next i
    EnDecimal = Resultat
    End Function
```

Chapitre 2

Introduction à la logique mathématique

Je confesse bien comme vous,
Que tous les poètes sont fous.
Mais puisque poètes vous n'êtes,
Tous les fous ne sont pas poètes.

Scévole de Sainte-Marthe
(1536 - 1623)

2.1 Bref survol historique

On fait remonter traditionnellement, l'histoire de la logique à l'antiquité grecque ; cette histoire se confond, au moins jusqu'à une période récente, avec l'histoire des mathématiques et de la philosophie.

La logique contemporaine : *mathématique* et *symbolique* apparaît au cours de la seconde moitié du $19^{\text{ème}}$ siècle. On peut repérer cette naissance autour de trois moments :

- G. Boole réalise une algébrisation de la logique en développant le *calcul propositionnel* comme un calcul dépendant uniquement de la combinaison de symboles et non de l'interprétation de ceux-ci. Les travaux contemporains de De Morgan qui initient la logique des relations développée ensuite par Pierce, Schröder et Russel, contribuent au développement de cette nouvelle forme de logique.

- La publication du *Begriffsschrift* de Frege en 1879 marque le début de la

formalisation de la logique et du formalisme comme philosophie programmatique des mathématiques. Dans les travaux de Frege la logique s'applique d'abord et essentiellement à déterminer les lois mêmes de la déduction.

- la logique joue un rôle décisif dans le vaste mouvement d'axiomatisation des mathématiques. L'analyse sera axiomatiquement ramenée à l'arithmétique grâce à Bolzano, Weierstrass, Dedekind, Méray, Cantor (avec donc des grandes questions sur la nature des entiers, des ensembles d'entiers, des ensembles...) ; Hilbert mènera à bien l'axiomatisation complète de la géométrie euclidienne.

Les principaux acquis de cette période ne vont pas tarder à être remis en cause par une profonde crise au tournant du $20^{\text{ème}}$ siècle (connue sous le nom de crise des fondements). La théorie des ensembles a provoqué beaucoup de résistance chez de nombreux mathématiciens. Elle s'est élaborée comme tentative de fonder les mathématiques à partir d'ingrédients de base bien définis : les ensembles, et de tout reconstruire à partir de là. Mais vite sont apparus de nombreux paradoxes. Par exemple le paradoxe de Russel : *soit a l'ensemble des ensembles ne se contenant pas eux-mêmes ; il est facile de montrer que $a \in a$ ssi $a \notin a$*, ce qui est manifestement contradictoire. Il était alors nécessaire de mettre de l'ordre dans tout cela. Il s'agissait de préciser quels étaient les raisonnements fiables, qui mettraient à l'abri de toute contradiction.

À cette crise on distingue trois réponses :

- le projet logistique d'une réduction des mathématiques à une logique cohérente et suffisamment développée. Projet illustré par la *"Théorie des Types"* de Russel qui cesse vite d'avoir une influence directe en logique mathématique, mais qui tient aujourd'hui une place importante dans certaines recherches en informatique théorique.

- le projet formaliste énoncé par Hilbert au début du $20^{\text{ème}}$ siècle est celui d'une solution définitive à tous les problèmes de fondement par l'application des mathématiques à leur propre langage pris dans sa dimension formelle. Résumé grossièrement il s'agissait de re-traduire toutes les mathématiques (même celles concernant l'infini) à des énoncés élémentaires : d'une part on réduit les démonstrations à des suites finies de symboles et on construit un algorithme qui permet de transformer une démonstration de la théorie des ensembles en une démonstration finitaire. Ce projet sera définitivement mis en échec par le théorème d'incomplétude de Gödel. Mais la tentative de Hilbert a quand même permis un travail de formalisation des mathématiques qui s'est révélé fructueux et a donné naissance à une branche importante de la logique : **la théorie de la démonstration** dont les applications, notamment en informatique se sont révélées très importantes.

- le projet intuitionniste d'une profonde réforme des mathématiques considérées comme saisies d'égarement, qui a donné naissance à certaines branches marginales des mathématiques (le constructivisme) et qui trouve également des applications importantes en logique avec le développement de l'informatique.

Avec le théorème d'incomplétude en 1931 ("une théorie axiomatisable, suffisamment développée ne peut contenir la démonstration de sa propre cohérence"), Gödel ouvre deux voies : le développement de la théorie de la démonstration et **la théorie des fonctions récursives**.

À la même époque Tarsky crée la "sémantique scientifique". Ce qui importe ce n'est pas la notion d'interprétation d'un langage formel dans un domaine d'objets (notion parfaitement claire depuis le $19^{\text{ème}}$ siècle) mais un traitement mathématique de la question. Il ouvre la voie à **la théorie des modèles**.

La logique (notamment les branches que nous avons soulignées) se développe alors, gagnant peu à peu le statut de discipline mathématique à part entière. Un échange fructueux s'est établi avec d'autres branches des mathématiques, l'algèbre mais aussi diverses parties de l'analyse, de la topologie, de la théorie des nombres, de la théorie des catégories. De plus l'informatique théorique a repris à son compte de vastes fragments de la logique fournissant en échange la matière de nouvelles investigations.

Soulignons, pour conclure ce bref survol historique, l'importance du rôle joué par l'irruption et l'explosion de l'informatique dans tous les domaines de la vie économique et scientifique sur les développement récents de la logique. En effet les bases théoriques de cette nouvelle science sont justement la logique mathématique. Tout d'abord, les modèles théoriques des premiers ordinateurs sont les machines de Türing. Rappelons ensuite quelques unes des interactions entre la logique et l'informatique : le calcul booléen pour la conception et l'étude des circuits ; la récursivité qui est la théorie des fonctions calculables sur machines. L'approche logique s'avère également pertinente pour le développement des langages de programmation. Par exemple la programmation logique, née en 1973 à Luminy (PROLOG, équipe d'A. Colmerauer) est un langage informatique dans lequel on peut considérer un programme comme une recherche de preuve basée sur des méthodes de démonstration automatique. Autre exemple : les langages de programmation fonctionnelle. Le paradigme connu sous le nom d'isomorphisme de Curry-Howard établit une parfaite correspondance entre programmes et démonstrations formelles : on identifie une tâche avec une formule mathématique et on identifie un programme satisfaisant cette tâche avec une preuve de l'énoncé. Notons que ces méthodes permettent d'obtenir des programmes "sûrs" (i.e. qui font bien ce que l'on attend d'eux). Enfin, le cadre théorique pour étudier certains aspects de la complexité des programmes [1] se situe dans le champ de la Logique.

1. Il n'est pas suffisant de savoir que l'on dispose d'un algorithme (d'un programme) pour résoudre un problème, encore faut-il que la réponse rendue par l'ordinateur arrive en temps "raisonnable".

2.2 Un langage mathématiquement défini

Un des premiers objets de la logique mathématique est la formalisation du langage mathématique. Il s'agit de définir les énoncés que l'on manipule et ce de telle façon que l'on puisse les analyser relativement à une notion de vérité. Traditionnellement, cette formalisation s'effectue en deux étapes : on s'intéresse dans un premier temps à l'articulation logique des énoncés (calcul propositionnel) puis aux contenus des énoncés mathématiques (calcul des prédicats). L'approche syntaxique permet de construire des formules correctes (du point de vue du langage et de la prouvabilité), l'approche sémantique permet de vérifier la validité des formules en calculant leur valeur de vérité ou en se plaçant dans des modèles.

On utilise dans ce qui suit les notions de *mots* et d'*arbres*, qui permettent l'agencement et la manipulation des symboles, et sur lesquelles reposent les définitions syntaxiques.

2.2.1 Le calcul propositionnel

Cette partie de la logique mathématique ne s'intéresse pas au contenu des propositions mais seulement à la manière dont elles sont logiquement articulées. L'étude du calcul propositionnel est une première étape pour aborder l'étude complète des énoncés mathématiques.

Les formules propositionnelles

Dans un premier temps il s'agit de définir précisément les objets d'étude : **les propositions** ou **formules propositionnelles**. Ainsi nous allons manipuler des symboles de **variables propositionnelles** ou **atomes** A, B, C et des symboles de combinateurs ou opérations logiques : les **connecteurs**. On en retient traditionnellement cinq : un connecteur unaire pour la négation, noté $\neg$ et lu "non", quatre connecteurs binaires, celui de conjonction, noté $\wedge$ et lu "et", de disjonction, noté $\vee$ et lu "ou", celui de l'implication, noté $\Rightarrow$ et lu "implique" et celui d'équivalence, noté $\Leftrightarrow$ et lu "si et seulement si" .

On se donne un ensemble P fini ou dénombrable de variables propositionnelles, $P = \{A, B, \ldots, X, Y, Z \ldots\}$. On considère l'alphabet $\mathcal{A} = P \cup \{\Rightarrow, \vee, \wedge, \neg, \Leftrightarrow\} \cup \{(,)\}$. Les formules vont être certains mots sur $\mathcal{A}$.

Définition 1. L'ensemble $\mathcal{F}$ des formules propositionnelles est le plus petit ensemble de mots sur $\mathcal{A}$ tels que :

- $\mathcal{F}$ contient P ;
- si $\mathcal{F}$ contient F alors $\mathcal{F}$ contient $\neg F$;

- si $\mathcal{F}$ contient F et G alors $\mathcal{F}$ contient $(F \vee G)$, $(F \wedge G)$, $(F \Rightarrow G)$ et $(F \Leftrightarrow G)$;

C'est à dire que si F et G sont des formules alors $\neg F$, $(F \vee G)$, $(F \wedge G)$, $(F \Rightarrow G)$, $(F \Leftrightarrow G)$ sont des formules.

Une formule contient par définition un assez grand nombre de parenthèses qui parfois semblent inutiles. Aussi pour ne pas alourdir l'écriture, on convient de se donner la possibilité de supprimer certaines parenthèses. La règle étant que les suppressions faites n'apportent pas d'ambiguïtés, c'est à dire que l'on peut toujours reconstituer la "vraie" écriture (avec toutes les parenthèses). Ainsi, on supprime les parenthèses les plus extérieures encadrant une formule ; on supprime les parenthèses séparant des connecteurs successifs identiques s'il s'agit du $\vee$, du $\wedge$ ou de $\Leftrightarrow$; on supprime les parenthèses séparant des $\Rightarrow$ successifs groupés par la droite : $A \Rightarrow B \Rightarrow C$ à la place de $A \Rightarrow (B \Rightarrow C)$,

On peut donner de l'ensemble $\mathcal{F}$ une description plus explicite : on définit par récurrence une suite $(\mathcal{F}_n)_n$ d'ensembles de mots sur $\mathcal{A}$. On pose :

-$\mathcal{F}_0 = P$

-$\mathcal{F}_{n+1} = \mathcal{F}_n \cup \{\neg F; F \in \mathcal{F}_n\} \cup \{(F \alpha G); F, G \in \mathcal{F}_n, \alpha \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}\}$.

Théorème 1. $\mathcal{F} = \bigcup_{n \in \mathbb{N}} \mathcal{F}_n$.

Définition 2. Soit $F \in \mathcal{F}$ une formule propositionnelle. La hauteur de F notée $h(F)$ est le plus petit entier n tel que $F \in \mathcal{F}_n$.

Lorsqu'il s'agit de démontrer une propriété sur toutes les formules on peut soit faire une démonstration par récurrence sur la hauteur des formules, soit revenir à la définition précédente des formules et montrer que la propriété considérée est vraie pour les formules atomiques et qu'elle se préserve par les opérations de négation et par les connecteurs binaires. Dans ce dernier cas on dit qu'on fait une **démonstration par induction** sur l'ensemble des formules.

De la même façon on peut faire des **définitions par induction** sur l'ensemble des formules, par exemple :

- On peut définir, par induction, l'ensemble des sous-formules d'une formule.
- On peut associer à une formule F son arbre de décomposition F^*. On procède de la façon suivante, par induction sur F :
  - si F est une variable propositionnelle, F^* est l'arbre réduit à la racine F ;
  - si $F = \neg G$ alors F^* est l'arbre de racine $\neg$ et de fils G^* ;
  - si $F = G \alpha H$ (où α est un connecteur binaire, alors F^* est l'arbre de racine α avec comme fils gauche G^* et comme fils droit H^*.

Distribution de valeurs de vérité : valuation

Définition 3. Une distribution de valeurs de vérité ou valuation est une fonction de l'ensemble P des variables propositionnelles dans $\Omega = \{0, 1\}$.

Lemme 1. Une valuation v se prolonge de manière unique en une fonction v^* de $\mathcal{F}$ dans Ω satisfaisant :
- $v^*(A) = v(A)$ lorsque $A \in P$;
- $v^*(\neg F) = 1$ ssi $(v^*(F) = 0$;
- $v^*(F \wedge G) = 1$ ssi $v^*(F) = v^*(G) = 1$;
- $v^*(F \vee G) = 0$ ssi $v^*(F) = v^*(G) = 0$;
- $v^*(F \Rightarrow G) = 0$ ssi $v^*(F) = 1$ et $v^*(G) = 0$;
- $v^*(F \Leftrightarrow G) = 1$ ssi $v^*(F) = v^*(G)$;

La démonstration se fait par induction sur F. Par abus de notation, on note v pour v^*.

Soit F une formule ; la définition de v sur $\mathcal{F}$ nous donne une méthode pour calculer $v(F)$, on commence par calculer les valeurs prises par v sur les sous-formules de F de hauteur 1 et on applique autant de fois que nécessaire les fonctions asssociées aux connecteurs.

Lemme 2. La valeur de vérité d'une formule ne dépend que des atomes figurant dans cette formule.

Ce lemme permet de représenter par un tableau la valeur de vérité d'une formule F en fonction de toutes les distributions de vérité possibles : **la table de vérité d'une formule**.
EXEMPLE : La table de vérité du $\Rightarrow$:

A	B	$A \Rightarrow B$
1	1	1
1	0	0
0	1	1
0	0	1

Tautologie, formules logiquement équivalentes

Définition 4. Soit F une formule, v une valuation et Σ un ensemble de formules :
- v satisfait F ssi $v(F) = 1$.
- v satisfait Σ ssi v satisfait toutes les formules de Σ.
- F est satisfaisable ssi il existe une valuation v qui satisfait F (i.e. telle que $v(F) = 1$).
- F est une tautologie ssi elle est satisfaite par toutes les valuations.
- F est une antilogie ou une contradiction ssi elle n'est satisfaite par aucune valuation.
- Σ est satisfaisable ssi il existe une valuation qui satifait Σ.
- Σ est insatisfaisable ou inconsistante ou contradictoire ssi il n'existe pas de valuation qui satisfasse simultanément toutes les formules de Σ.

Définition 5. Soit F et G deux formules sur P. F et G sont logiquement équivalentes ssi pour toute valuation v, $v(F) = v(G)$. On notera alors $F \equiv G$.

Lemme 3 (Quelques équivalences bien connues)**.** Soit F, G et H des formules propositionnelles, on a :

1. $F \Rightarrow G \equiv \neg F \vee G$;
2. $(F \vee G) \vee H \equiv F \vee (G \vee H)$ (associativité du $\vee$) ;
3. $(F \wedge G) \wedge H \equiv F \wedge (G \wedge H)$ (associativité du $\wedge$) ;
4. $F \vee G \equiv G \vee F$ (commutativité du $\vee$) ;
5. $F \wedge G \equiv G \wedge F$ (commutativité du $\wedge$) ;
6. $\neg(F \wedge G) \equiv \neg F \vee \neg G$ (négation d'une conjonction) ;
7. $\neg(F \vee G) \equiv \neg F \wedge \neg G$ (négation d'une disjonction) ;
8. $\neg\neg F \equiv F$ (double négation) ;
9. $F \wedge F \equiv F$ (idempotence du $\wedge$) ;
10. $F \vee F \equiv F$ (idempotence du $\vee$) ;
11. $F \Rightarrow G \equiv \neg G \Rightarrow \neg F$ (contraposée).

Déduction sémantique

Définition 6. Soit $\mathcal{E}$ un ensemble de formules et ϕ une formule. On dit que $\mathcal{E}$ valide ϕ si toute valuation qui satisfait $\mathcal{E}$ satisfait ϕ. On note $\mathcal{E} \models \phi$.

REMARQUE : Si ϕ est une tautologie on a $\emptyset \models \phi$, ce que l'on notera $\models \phi$.

Lemme 4 (Quelques tautologies bien connues)**.** Soit F et G des formules propositionnelles, les formules suivantes sont des tautologies :

1. $F \Rightarrow (G \Rightarrow F)$
2. $F \Rightarrow F \vee G$
3. $G \Rightarrow F \vee G$
4. $F \wedge G \Rightarrow F$
5. $F \wedge G \Rightarrow G$
6. $\neg F \Rightarrow (F \Rightarrow G)$
7. $F \vee \neg F$

2.2.2 Le calcul des prédicats

Le calcul des prédicats comporte deux volets. Tout d'abord on se donne les outils adéquats pour nommer les objets (les termes) et décrire leurs propriétés (les formules). On étudie ensuite la satisfaction des formules dans telle ou telle structure.

Le langage du premier ordre

L'objectif est de définir les mots que nous appellerons formules. Il s'agit de trouver un alphabet et une grammaire permettant d'exprimer les énoncés mathématiques tels que :

- $\forall \epsilon \, \exists N \, \forall n \, (n > N \Rightarrow |u_n - a| < \epsilon)$
- $\forall n \exists m (m \geq n)$
- $\forall n \, P(n \times (n+1))$ est pair.
- $\forall n \quad (\exists k \quad k < n) \vee (n = 0)$
- $\forall y (\exists x (sin(x+2) = y))$.

Dans ces énoncés on distingue : des connecteurs ($\vee$, $\Rightarrow$) ; des quantificateurs ($\forall, \exists$) ; les parenthèses ouvrantes et fermantes ; des fonctions (unaires comme $\sin$, valeur absolue, binaires comme $\times$ ou $+$, ...) ; des relations ou prédicats ($\leq$, $=$, "être pair", ...) ; des variables (x, y, n, m, ...) ; des constantes (0, ...)

Ainsi, comme pour le calcul propositionnel, les formules seront des suites de symboles prises dans un alphabet. Toutefois, il n'y aura pas un alphabet unique, mais un alphabet approprié (appelé souvent langage) pour chaque type de structure envisagée. Certains symboles sont communs à tous les alphabets : les connecteurs, les parenthèses les quantificateurs et les variables. Les autres symboles dépendent du type de structure que l'on a en vue. Par exemple si on veut étudier des structures algébriques et plus précisément les groupes, on a besoin d'un symbole de constante (pour l'élément neutre), d'un symbole de fonction binaire (pour l'opération interne) et d'un symbole de relation binaire (pour l'égalité). Pour étudier les ensembles ordonnés on a seulement besoin de deux symboles de relation binaire pour la relation d'ordre et pour l'égalité.

Les formules que l'on va étudier ici s'appellent les formules du premier ordre. Ceci est justifié par le fait que l'on quantifie sur les éléments de la structure. Mais il y a de nombreuses propriétés que l'on ne peut pas exprimer ainsi (par exemple la notion de bon ordre), il faut alors se permettre de quantifier sur d'autres objets que les variables du premier ordre (les relations, les fonctions...)

Définition 7 (Langage du premier ordre)**.** Un langage $\mathcal{L}$ du premier ordre est un ensemble de symboles qui se compose de deux parties :

- La première commune à tous les langages du premier ordre est constituée **des symboles logiques** : les variables, les connecteurs, les quantificateurs et les parenthèses, soit :
  - Un ensemble $\mathcal{V}$ dénombrable dont les éléments sont appelés les **variables** notées x, y, z, n, ...
  - L'ensemble des connecteurs $\mathcal{C} = \{\neg\ ,\ \vee\ ,\ \wedge\ ,\ \Rightarrow\ ,\ \Leftrightarrow\}$
  - Les quantificateurs : $\forall$ et $\exists$
  - Les parenthèses ouvrantes et fermantes.
- La deuxième partie contient **des symboles non logiques** et varie d'un langage à l'autre
  - Un ensemble de symboles fonctionnels. Un symbole fonctionnel est constitué par un couple $(f,\ n)$ où n est appelé l'arité [2] de f.
  - Un ensemble de constantes $\{a, b, c, \ldots\}$ (que l'on peut voir aussi comme des symboles fonctionnels d'arité 0).
  - Un ensemble de symboles relationnels. Un symbole relationnel est constitué par un couple (R, n) où n est appelé l'arité de R.

Remarque La donnée d'un langage du calcul des prédicats se fera par la donnée des symboles non logiques.
EXEMPLE : $\mathcal{L} = \{(f, 2); (g, 2); (c, 0)\} \cup \{(R, 2); (S, 2)\}$ où f et g sont des symboles de fonctions binaires, c est un symbole de fonction 0-aire (une constante), R et S sont des symboles de relations binaires.

Nous allons construire des mots bien formés appelés des termes. Les symboles qui servent d'ingrédients pour la fabrication des termes sont les variables, les constantes et les symboles de fonctions.
EXEMPLE : Dans le langage $\mathcal{L}$, $(xfy)gc$ sera un terme. (On écrira aussi ce terme en notation préfixée : $g(f(x, y), c)$.

Définition 8 (Termes d'un langage $\mathcal{L}$)**.** Un **terme** t est un mot fini sur $\mathcal{L}$ défini par induction par :
- les variables et les constantes sont des termes ;
- si $t_1, \ldots, t_n$ sont des termes, si f est un symbole fonctionnel d'arité n, alors $f(t_1, \ldots, t_n)$ est un terme.

REMARQUE : On peut définir un terme comme étant un arbre fini dont les feuilles sont des variables ou des constantes et dont chaque nœud qui n'est pas une feuille vérifie la règle de construction suivante : le nœud est étiqueté par un symbole de fonction n-aire et possède n fils qui sont des termes.

2. le nombre d'arguments de la fonction ou de la relation

Définition 9.

1. Un **sous-terme** u **d'un terme** t est un mot situé à un nœud de l'arbre de construction de t.
   EXEMPLE : $h(x,\ y,\ a)$ est un sous terme de $f(g(a),\ h(x,\ y,\ a))$.
2. Les **variables de** t sont les variables situées aux feuilles de l'arbre de construction de t. On note $\mathcal{V}(t)$ l'ensemble des variables de t.
3. t est un **terme clos** si $\mathcal{V}(t)$ est $\emptyset$.

REMARQUE : Un langage sans symbole de constante n'a pas de terme clos.

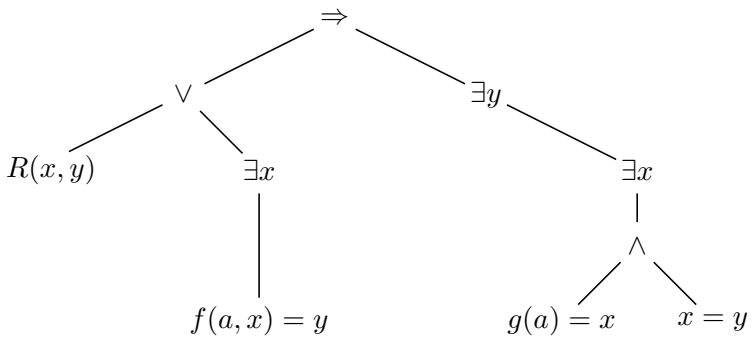
Définition 10 (Formules du premier ordre sur $\mathcal{L}$)**.** Soit $\mathcal{L}$ un langage du premier ordre. On définit par induction l'ensemble des formules sur $\mathcal{L}$:
- Si R est un symbole relationnel d'arité n et si $t_1, \ldots, t_n$ sont n termes alors $\phi = R(t_1, \ldots, t_n)$ est une formule atomique.
- Si ϕ et ψ sont des formules alors $\neg\phi$, $(\phi\alpha\psi)$ où $\alpha \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$ sont des formules ;
- Si ϕ est une formule et x est une variable alors $\forall x(\phi)$ et $\exists x(\phi)$ sont des formules.

On peut définir comme précédemment l'arbre de décomposition d'une formule sur $\mathcal{L}$; les feuilles sont toujours les formules atomiques.

EXEMPLE : Considérons la formule

$$[R(x,\ y) \vee (\exists x(f(a,\ x) = y))] \Rightarrow [\exists y(\exists x((g(a) = x) \wedge (x = y)))]$$

son arbre de décomposition est :



On s'attarde ensuite sur la notion d'occurrence et surtout de variable libre qui est importante pour l'interprétation des formules quantifiées.

Définition 11 (occurrence)**.**

- Une occurrence de x dans ϕ désigne une place particulière de cet x dans ϕ. On peut repérer cette place dans la feuille de l'arbre de construction de ϕ dans laquelle x est produit. Il peut y avoir plusieurs occurrences de x dans une même feuille (ex : dans $R(x, x, y)$) ou plus généralement dans une même formule ϕ.
- Une occurrence de x dans ϕ est **liée** ssi la feuille dans laquelle on considère l'occurence de x appartient à une branche où la règle de formation $\exists x$ ou $\forall x$ a été appliquée. Autrement dit cette occurence de x dans ϕ est dans une sous formule de ϕ qui est de la forme $\forall x\psi$ ou $\exists x\psi$.
- Une occurrence de x dans ϕ est **libre** ssi la feuille dans laquelle on considère l'occurence de x n'appartient pas à une branche où la règle de formation $\forall x$ ou $\exists x$ a été appliquée.

EXEMPLE : Dans la formule $(f(\mathbf{x},\ y) = a \Rightarrow \forall y\ R(y,\ \mathbf{x}))$, les occurrences de x sont libres ; y a une occurrence libre et une occurrence liée.

Définition 12 (Variables libres)**.** Une variable x est une **variable libre** de ϕ ssi x possède au moins une occurrence libre dans ϕ. On note $\mathcal{V}_l(\phi)$, l'ensemble des variables libres de ϕ.

EXEMPLE : Dans l'exemple ci-dessus, x et y sont des variables libres.

Définition 13 (Formule close)**.** Une formule ϕ est une formule close lorsque $\mathcal{V}_l(\phi) = \emptyset$, ouverte si elle possède des variables libres.

EXEMPLE :
La formule $\forall x((\exists y R(x,\ y)) \Rightarrow\ f(a,\ x) = a)$ est close.
La formule $\forall x\ (R(x,\ y) \Rightarrow\ f(a,\ x) = z)$ est ouverte.

Interprétation

Il s'agit de donner un sens aux symboles de prédicats et de fonctions, aux variables et aux termes et une valeur de vérité aux formules. Pour cela on va d'abord fixer l'univers du discours, c'est à dire l'ensemble des objets que l'on étudie ; les termes seront interprétés par des éléments de cet ensemble. On peut alors associer à chaque symbole fonctionnel une fonction sur cet ensemble (fonction au sens mathématique habituel) ; de même les symboles relationnels seront interprétés par des relations sur cet ensemble. Ensuite on donnera une valeur de vérité aux formules en tenant compte de leurs variables libres, par le biais des assignations.

Définition 14 (Réalisation d'un langage)**.** Soit $\mathcal{L} = \{(f_i, n_i)/i \in I\} \cup \{(R_j, n_j)/j \in J\}$. Une réalisation $\mathcal{M}$ du langage $\mathcal{L}$ est la donnée :

- d'un ensemble non vide E appelé le domaine de la réalisation (noté parfois $\mid \mathcal{M} \mid$).
- pour chaque symbole fonctionnel (f, n) de $\mathcal{L}$, d'une fonction $f^* : E^n \longrightarrow E$
- pour chaque symbole relationnel (R, n) de $\mathcal{L}$, d'une relation n-aire R^* , $R^* \subset E^n$

EXEMPLE : Avec $\mathcal{L} = \{(f, 2)\ ;\ (g, 2)\ ;\ (h, 1)\ ;\ (c, 0)\} \cup \{(R, 2)\}$
1) $\mathcal{M}_1 = \{\mathbb{N}; +, \times, succ, 0, \leq\}$
2) $\mathcal{M}_2 = \{$Le plan euclidien E d'origine O,
$f^* : E^2 \longrightarrow E$ définie par $f^*(A, B) = C$ tel que $\overrightarrow{OA} + \overrightarrow{OB} = \overrightarrow{OC}$
$g^* : E^2 \longrightarrow E$ définie par : $g^*(A, B) = C$ milieu de $[AB]$
$h^* : E \longrightarrow E$ définie par : $h^*(A) = C$ symétrique de A par rapport à O
c^* est l'origine O
$R^* = \{(A, B)/A, B, O$ sont alignés $\}$

Une assignation de variables s est une application $s : \mathcal{V} \longrightarrow E$. Lorsque $\mathcal{V}$ est fini (i.e. $\mathcal{V} = \{x_1, \ldots, x_n\}$), s sera plutôt noté : $\{x_1 = a_1, \ldots, x_n = a_n\}$.

Définition 15 (Réalisation valuée)**.** Une réalisation valuée (une interprétation) d'un langage $\mathcal{L}$ est la donnée d'un couple $(\mathcal{M}, s)$ où $\mathcal{M}$ est une réalisation du langage et s est une assignation de variables.

Une réalisation valuée $(\mathcal{M}, s)$ permet de définir l'interprétation d'un terme t par un élément du domaine de la réalisation E, noté $s(t)$, puis ensuite de déterminer la satisfaction des formules dans cette réalisation valuée.

Lemme 5 (Interprétation des formules)**.** Soit $(\mathcal{M}, s)$ une réalisation valuée du langage $\mathcal{L}$. Il existe une et une seule fonction booléenne $s^{\mathcal{M}}$ (i.e. définie sur l'ensemble des formules du premier ordre de $\mathcal{L}$, à valeur dans $\{0, 1\}$), qui vérifie les conditions ci-dessous, en notant $\mathcal{M} \models_s \phi$ pour $s^{\mathcal{M}}(\phi) = 1$:

- Si ϕ est une formule atomique $R(t_1, \ldots, t_n)$ alors $\mathcal{M} \models_s R$ ssi $(s(t_1), \cdots,\ s(t_n)) \in R^*$
- Si ϕ est $\neg\psi$ alors $\mathcal{M} \models_s \neg\psi$ ssi $s^{\mathcal{M}}(\psi) = 0$
- Si ϕ est $\psi \wedge \theta$ alors $s^{\mathcal{M}}(\phi) = 1$ ssi $s^{\mathcal{M}}(\psi) = 1$ et $s^{\mathcal{M}}(\theta) = 1$
  De même pour $\vee,\ \Rightarrow,\ \Leftrightarrow$, on revient aux tables de vérité vues en calcul propositionnel (lemme 1).
- Si ϕ est $\exists x\ \psi$ alors $\mathcal{M} \models_s \exists x\ \psi$ ssi il existe $a \in |\mathcal{M}|$ tel que $s^{\mathcal{M}}_{x:=a}(\psi) = 1$ où $s_{x:=a}$ coïncide avec s sauf en x pour laquelle $s_{x:=a}(x) = a$.
- Si ϕ est $\forall x\ \psi$ alors $s^{\mathcal{M}}(\psi) = 1$ ssi quelque soit $a \in |\mathcal{M}|$ $s^{\mathcal{M}}_{x:=a}(\psi) = 1$

Définition 16 (Formules synonymes)**.** F et G sont synonymes ssi quel que soit $(\mathcal{M}, s), [\mathcal{M} \models_s F$ ssi $\mathcal{M} \models_s G]$. On note alors $F \equiv G$.

EXEMPLES :

1. $\forall x F \equiv \neg\exists x \neg F$
2. $\exists x F \equiv \neg\forall x \neg F$
3. $\forall x \forall y F \equiv \forall y \forall x F$

Définition 17 (Clôture universelle)**.** Soit F une formule ayant ses variables libres parmi $\{x_1, \ldots, x_n\}$. On appelle clôture universelle de F la formule $\forall x_1 \cdots \forall x_n F$.

REMARQUE : Une formule F et sa clôture universelle ne sont pas nécessairement synonymes.
EXEMPLE : Les formules $x = y$ et $\forall x \forall y \ x = y$ ne sont pas synonymes. Dans une réalisation dont le domaine contient plus d'un élément la formule $x = y$ sera satisfaite pour certaines valuations alors que la formule $\forall x \forall y \ x = y$ ne le sera jamais.

Définition 18.

- Une formule close est dite **universellement valide** (ou **valide**) si elle est satisfaite dans toute réalisation de $\mathcal{L}$.
- Une formule comportant des variables libres est dite **universellement valide** si sa clôture universelle l'est.
- Etant donné deux formules F et G, on dit que ces formules sont **universellement équivalentes** ou **logiquement équivalentes** ou **équivalentes** si la formule $F \Leftrightarrow G$ est universellement valide.
- On appelle **théorie** de $\mathcal{L}$ tout ensemble de formules closes de $\mathcal{L}$.
- Soit une théorie T de $\mathcal{L}$ et une réalisation $\mathcal{M}$ de $\mathcal{L}$. On dit que $\mathcal{M}$ est **un modèle** de T si $\mathcal{M}$ satisfait toutes les formules de T. On note $\mathcal{M} \models T$.
- Une théorie est **consistante** si elle possède au moins un modèle. Une théorie qui n'est pas consistante est dite contradictoire.
- Etant donné une théorie T et une formule close F de $\mathcal{L}$, on dit que F est une **conséquence sémantique** (ou **conséquence**) de T si tout modèle de T est aussi modèle de F.

2.3 La formalisation des démonstrations

Dans la partie précédente nous nous sommes appliqués d'une part à définir précisément les formules, d'autre part à vérifier parmi les formules bien définies lesquelles étaient vraies (universellement valides). Nous allons maintenant approfondir cette démarche :

1. Nous étendons notre effort de formalisation, réalisé jusqu'ici sur les formules, aux démonstrations elle-mêmes ;

2. Nous ne nous contentons plus de vérifier si une formule est vraie ou pas mais nous allons construire des formules vraies.

Nous allons donc donner un sens précis à l'expression "la formule ϕ est démontrable". Dans la mesure où est établie, par les théorèmes de complétude la correspondance entre formules vraies (universellement valides) et formules démontrables, nous pourrons alors considérer que nous savons, en faisant des démonstrations formelles, construire des formules vraies.

Il existe plusieurs systèmes dans lesquels on peut définir les preuves formelles. De tels systèmes s'appellent des systèmes de déduction ou des systèmes formels. Historiquement les premiers systèmes formels, par exemple les systèmes à la Hilbert, étaient basés sur des schémas d'axiomes (les tautologies du calcul propositionnel ou du calcul des prédicats) ; une démonstration formelle est une suite de propositions qui appartiennent à cette suite soit parce qu'elles sont des axiomes, soit qu'elles peuvent se déduire de propositions précédentes par l'application d'une règle. Par exemple pour le calcul propositionnel on utilise une seule règle appelée *Modus ponens* :

Si A et $A \Rightarrow B$ sont deux propositions qui appartiennent à une démonstration formelle $\mathcal{D}$ alors la suite de propositions constituée de $\mathcal{D}$ suivie de B est une démonstration formelle.

Gerhard Gentzen est le premier à avoir développé des formalismes qui redonnent à la logique le caractère d'un cheminement naturel. La principale idée de départ de Gentzen était simple : pas d'axiomes logiques, que des règles de déduction et autant qu'il en faut pour reproduire toutes les formes élémentaires et naturelles de raisonnement. Pour réaliser cette idée, Gentzen a développé un formalisme où les déductions ne sont pas des suites d'énoncés mais des arbres, faits de colonnes d'énoncés qui se rejoignent via des règles de déduction, jusqu'à la conclusion.

G. Gentzen a proposé en 1934 plusieurs systèmes d'inférence, dont les systèmes de déduction naturelle NK pour la logique classique et NJ pour la logique intuitionniste. Le choix de graduer les règles de déduction provient d'un souci philosophique que nous n'abordons pas ici, bien qu'il soit du plus grand intérêt. Le système utilisé dans les cours de mathématique qu'ils soient de l'enseignement secondaire ou de l'enseignement universitaire, est le système NK ; c'est celui qui correspond à la sémantique présentée ci-dessus [3].

3. Nous illustrerons les règles de déduction présentées ci-dessous dans la section suivante.

2.3.1 Un peu de vocabulaire

En Déduction Naturelle (ND), on distingue deux types de règles de déduction : les règles d'introduction et les règles d'élimination des connecteurs et quantificateurs.

- le $\neg$ n'est pas primitif, mais défini à partir du connecteur $\Rightarrow$ et de la constante $\bot$ (symbole de l'antilogie) : $\neg A = A \Rightarrow \bot$.
- De même pour $\Leftrightarrow$. En effet : $A \Leftrightarrow B = (A \Rightarrow B) \wedge (B \Rightarrow A)$.

Définition 19. Une **démonstration formelle** dans un système de déduction naturelle est un arbre fini. Ses noeuds sont des expressions $\Gamma \vdash \phi$ où Γ est un ensemble de formules et ϕ est une formule ; ces expressions sont appelées **séquents** (on peut intuitivement comprendre un tel séquent par "la formule ϕ est déduite de l'ensemble d'hypothèses Γ"). La racine de l'arbre est la conclusion de la démonstration, les feuilles sont soit des axiomes d'une théorie soit des axiomes logiques, c'est à dire des séquents $\Gamma \vdash A$ où $A \in \Gamma$.

Les nœuds-pères se déduisent des nœuds fils par l'application d'une des règles de déduction du système. Ces règles sont qualifiées de règles d'introduction ou règles d'élimination selon qu'elles permettent d'introduire un connecteur ou bien d'éliminer un connecteur.

2.3.2 Règles de la Déduction Naturelle en calcul propositionnel

Les règles de la logique intuitionniste NJ

1. Axiomes logiques

   $\dfrac{}{\Gamma \vdash A}$ Lorsque A appartient à l'ensemble de formules Γ.

2. Règles relatives à l'implication :

   `Introduction de` $\Rightarrow$

   $$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow i$$

   `Elimination de` $\Rightarrow$

   $$\frac{\Delta \vdash A \Rightarrow B \quad \Omega \vdash A}{\Delta, \Omega \vdash B} \Rightarrow e$$

   REMARQUE : Considérons la règle que l'on a appelée "élimination de $\Rightarrow$" pour commenter la lecture des règles de déduction. Dans une telle règle, on distingue un ou deux séquents prémisses, ceux qui sont au dessus (ici $\Delta \vdash A \Rightarrow B$ et $\Omega \vdash A$) et un séquent conclusion, celui qui est au dessous (ici $\Delta, \Omega \vdash B$).

   Et la règle peut être lue de la façon suivante : on sait que sous les hypothèses Δ on peut déduire la formule $A \Rightarrow B$ et que sous les hypothèses Ω on peut déduire la formule A ; on peut alors affirmer que la réunion des hypothèses Δ et Ω permet de déduire la formule B.

Ainsi une règle de déduction permet de formuler une nouvelle conclusion (ici la formule B, la formule conclusion du séquent conclusion) en contrôlant la gestion des hypothèses.

3. Règles relatives au connecteur $\wedge$:

$$\texttt{Introduction de } \wedge$$
$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \wedge i$$
$$\texttt{Elimination de } \wedge$$
$$\frac{\Delta \vdash A \wedge B}{\Delta \vdash A} \wedge e1 \qquad \frac{\Delta \vdash A \wedge B}{\Delta \vdash B} \wedge e2$$

4. Règles relatives au connecteur $\vee$:

$$\texttt{Introduction de } \vee$$
$$\frac{\Delta \vdash A}{\Delta \vdash A \vee B} \vee i1 \qquad \frac{\Delta \vdash B}{\Delta \vdash A \vee B} \vee i2$$
$$\texttt{Elimination de } \vee$$
$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C \quad \Gamma \vdash A \vee B}{\Gamma \vdash C} \vee e$$

REMARQUE : On peut noter que la règle d'élimination du $\vee$ permet de faire des raisonnements par disjonction de cas.

5. Règle relative au $\perp$:

$$\texttt{Elimination du } \perp$$
$$\frac{\Gamma \vdash \perp}{\Gamma \vdash A}$$
Pour n'importe quelle formule A

Si de l'ensemble de formules Γ, on déduit l'absurdité alors on peut en déduire n'importe quelle formule.

Définition 20. Le système de déduction naturelle de la logique intuitionniste est l'ensemble des arbres de dérivation construits à partir des axiomes logiques ($\Gamma \vdash A$ lorsque $A \in \Gamma$) et des règles relatives aux connecteurs $\vee$, $\wedge$ et $\Rightarrow$.

Les règles classiques de NK

– L'**Axiome du tiers exclus** (TE) est le fait que l'on rajoute comme axiome logique le séquent suivant :

$$\frac{}{\vdash \phi \vee \neg\phi} te$$

– La **Règle de l'absurde** (A) est la règle suivante :

$$\frac{\Gamma, \neg\phi \vdash \bot}{\Gamma \vdash \phi}\, a$$

– La **Règle "de la double négation"** $(\neg\neg)$ est la règle suivante :

$$\frac{\Gamma \vdash \neg\neg\phi}{\Gamma \vdash \phi}\, \neg\neg$$

Les trois règles (TE), (A) et $(\neg\neg)$ sont équivalentes. Elles permettent de faire des raisonnements par l'absurde.

Définition 21 (Le Système de déduction naturelle **NK**)**.** C'est le système NJ auquel on rajoute la règle TE.

REMARQUE : Comme on le fait usuellement en mathématiques, on a envie de réutiliser ce qu'on a démontré, sans pour autant refaire la démonstration à chaque fois. On pourra considérer une règle dérivée comme une boîte noire dont on ne connaît que les entrées/sorties.
EXEMPLES :

1. On peut dériver la règle suivante, qui permet de faire des raisonnements par contraposée :

$$\frac{\vdash A \Rightarrow B}{\vdash \neg B \Rightarrow \neg A}$$

En effet, en prenant $\vdash A \Rightarrow B$ comme hypothèse on peut faire la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\vdash A \Rightarrow B \qquad \dfrac{}{A \vdash A}}{A \vdash B}\Rightarrow e \qquad \dfrac{}{B \Rightarrow\bot\vdash B \Rightarrow\bot}}{A, B \Rightarrow\bot\vdash\bot}\Rightarrow e}{B \Rightarrow\bot\vdash A \Rightarrow\bot}\Rightarrow i}{\vdash (B \Rightarrow\bot) \Rightarrow (A \Rightarrow\bot)}\Rightarrow i$$

2. En logique classique NK les formules $\neg A \vee \neg B$ et $A \Rightarrow \neg B$ sont équivalentes. Nous établissons, ici, comme exemple, la formule $(\neg A \vee \neg B) \Rightarrow (A \Rightarrow \neg B)$, qui ne nécessite pas l'utilisation de la règle TE, et qui est, par conséquent, démontrable dans NJ

On fait d'abord les deux dérivations suivantes, où Γ désigne l'ensemble de formules $\{\neg A \vee \neg B, A, B\}$:

$$\frac{\Gamma, \neg A \vdash A \quad \Gamma, \neg A \vdash \neg A (= A \Rightarrow \bot)}{\Gamma, \neg A \vdash \bot}\Rightarrow_e \qquad \frac{\Gamma, \neg B \vdash B \quad \Gamma, \neg B \vdash \neg B (= B \Rightarrow \bot)}{\Gamma, \neg B \vdash \bot}\Rightarrow_e$$

puis la dérivation :

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \neg A \vee \neg B \qquad \Gamma, \neg A \vdash \bot \qquad \Gamma, \neg B \vdash \bot}{\Gamma \vdash \bot}\vee_e}{\neg A \vee \neg B, A \vdash \neg B}\Rightarrow_i}{\neg A \vee \neg B \vdash A \Rightarrow \neg B}\Rightarrow_i}{\vdash (\neg A \vee \neg B) \Rightarrow (A \Rightarrow \neg B)}\Rightarrow_i$$

2.3.3 Déduction Naturelle en calcul des prédicats

Le système de Déduction Naturelle NK pour le calcul des prédicats est constitué par les axiomes et règles du calcul propositionnel auxquels on rajoute des règles d'introduction et d'élimination pour les quantificateurs $\forall$ et $\exists$.

Règles pour $\forall$

- Règle d'introduction de $\forall$

$$\dfrac{\Gamma \vdash \phi(y)}{\Gamma \vdash \forall x\ \phi(x)}\forall i$$

y n'est pas libre dans Γ

- Règle d'élimination de $\forall$

$$\dfrac{\Gamma \vdash \forall x\ \phi(x)}{\Gamma \vdash \phi(t/x)}\forall e$$

$\phi(t/x)$ désigne la formule ϕ dans laquelle toutes les occurences libres de x ont été remplacées par un terme quelconque t.

Règles pour $\exists$

Règle d'introduction de $\exists$

$$\dfrac{\Gamma \vdash \phi(t/x)}{\Gamma \vdash \exists x\ \phi(x)}\exists i$$

Règle d'élimination de $\exists$

$$\dfrac{\Gamma, \phi(y) \vdash \theta \qquad \Delta \vdash \exists x\ \phi(x)}{\Gamma, \Delta \vdash \theta}\exists e$$

y n'est libre ni dans Γ, ni dans θ

EXEMPLE : démonstration formelle de la distributivité du $\forall$ par rapport au $\wedge$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x(\phi(x) \wedge \theta(x)) \vdash \forall x(\phi(x) \wedge \theta(x))}\forall e}{\forall x(\phi(x) \wedge \theta(x)) \vdash \phi(y) \wedge \theta(y)}\wedge e1}{\forall x(\phi(x) \wedge \theta(x)) \vdash \phi(y)}\forall i}{\forall x(\phi(x) \wedge \theta(x)) \vdash \forall x\phi(x)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\forall x(\phi(x) \wedge \theta(x)) \vdash \forall x(\phi(x) \wedge \theta(x))}\forall e}{\forall x(\phi(x) \wedge \theta(x)) \vdash \phi(z) \wedge \theta(z)}\wedge e2}{\forall x(\phi(x) \wedge \theta(x)) \vdash \theta(z)}\forall i}{\forall x(\phi(x) \wedge \theta(x)) \vdash \forall x\theta(x)}$$

$$\dfrac{\forall x(\phi(x) \wedge \theta(x)) \vdash \forall x\phi(x) \qquad \forall x(\phi(x) \wedge \theta(x)) \vdash \forall x\theta(x)}{\forall x(\phi(x) \wedge \theta(x)) \vdash \forall x\phi(x) \wedge \forall x\theta(x)}\wedge i$$

2.3.4 Complétude de la logique du premier ordre

Pour le calcul propositionnel ce théorème de complétude s'énonce aisément :

Théorème 2 (Complétude de la logique propositionnelle)**.** Soit F une formule propositionnelle,

$$F \text{ est une tautologie ssi le séquent } \vdash F \text{ est démontrable.}$$

Ce théorème peut être vu comme un cas particulier du théorème de complétude de la logique du premier ordre, dont la démonstration est plus complexe. Nous ne ferons ici que l'énoncer et nous renvoyons pour sa démonstration aux ouvrages de référence, par exemple l'*Introduction à la logique* de David, Nour et Raffali.

Théorème 3 (Complétude de la logique du premier ordre)**.** Soit T une théorie (les formules closes appartenant à T sont appelés axiomes de la théorie) et F une formule close.

$$T \vdash F \text{ ssi } F \text{ est une conséquence sémantique de } T$$

où la notation $T \vdash F$ signifie qu'il existe un sous ensemble fini T' de formules de T tel que $T' \vdash F$.

Ce théorème signifie intuitivement qu'une formule est "vraie" (quelque soit le sens que l'on donne aux symboles propres au langage du premier ordre) si et seulement si elle est démontrable.

Le second sens de l'équivalence "si une formule est démontrable alors elle est vraie" assure que les règles de démonstration que l'on s'est donné sont correctes. L'autre sens "si une formule est vraie alors elle est démontrable " assure que l'on s'est donné suffisament de règles (que le système de règles est *complet* pour démontrer tout ce qui est vrai).

2.4 Illustration de quelques règles

1. Introduction de $\Rightarrow$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow i$$

Γ : n est un nombre entier naturel
A : l'entier n est divisible par 6
B : l'entier n est divisible par 3

La règle d'introduction du $\Rightarrow$ formalise la démonstration d'un théorème de la forme « Si… alors… ».
La formalisation ci-dessus peut être lue comme :
À partir des hypothèses "*n est un entier*" (Γ), et "*n est divisible par 6*" (A),

on démontre "*n est divisible par 3*" (B).
On déduit que, sous l'hypothèse "*n est un entier*" (Γ), la formule "si *n est divisible par 6* alors *n est divisible par 3*", ($A \Rightarrow B$), est vraie.

2. Elimination de $\wedge$

$$\frac{\Delta \vdash A \wedge B}{\Delta \vdash A} \wedge e$$

Δ : n est un nombre entier divisible par 6
A : l'entier n est divisible par 2
B : l'entier n est divisible par 3

3. Elimination de $\vee$

$$\frac{\Gamma, A \vdash D \quad \Gamma, B \vdash D \quad \Gamma, C \vdash D \quad \Gamma \vdash A \vee B \vee C}{\Gamma \vdash D} \vee e$$

Γ : n est un nombre entier naturel
A : $n \equiv 0$ mod 3
B : $n \equiv 1$ mod 3
C : $n \equiv 2$ mod 3
D : $n^3 - n$ est divisible par 3

Cette règle formalise la méthode de démonstration par disjonction des cas.
Ainsi on peut lire la formalisation ci-dessus comme :
À partir des hypothèses "*n est un entier*", (Γ), et " $n \equiv 0$ *modulo 3*", (A), on démontre que "$n^3 - n$ *est divisible par 3*", (D).
À partir des hypothèses "*n est un entier*", (Γ), et " $n \equiv 1$ *modulo 3*", (B), on démontre que "$n^3 - n$ *est divisible par 3*", (D).
À partir des hypothèses "*n est un entier*", (Γ), et " $n \equiv 2$ *modulo 3*", (C), on démontre que "$n^3 - n$ *est divisible par 3*", (D).
À partir de l'hypothèse "*n est un entier*", (Γ), on démontre que "$n \equiv 0$ *modulo 3*" ou "$n \equiv 1$ *modulo 3*" ou "$n \equiv 2$ *modulo 3*", ($\Gamma \vdash A \vee B \vee C$).
On déduit que, sous l'hypothèse "*n est un entier*", (Γ), la formule "$n^3 - n$ *est divisible par 3*", ($\Gamma \vdash D$), est vraie.

4. Règle d'élimination de $\forall$

$$\frac{\Gamma \vdash \forall x\ \phi(x)}{\Gamma \vdash \phi(t/x)} \forall e$$

$\phi(t/x)$ désigne la formule ϕ dans laquelle toutes les occurences libres de x ont été remplacées par un terme quelconque t.

Γ : x est un nombre réel
$\phi(x)$: $(x-1)^2 = x^2 - 2x + 1$
t : le nombre entier 100

Cette règle représente l'instanciation d'une variable universellement quantifiée. L'exemple ci-dessus montre son utilisation pour calculer mentalement 99^2.

2.5 La logique au quotidien en classe de mathématiques

2.5.1 Le modus ponens omniprésent : un exemple en géométrie

Considérons l'exercice suivant :

Le triangle ABC est rectangle en A et les côtés AB et AC mesurent respectivement 5 et 8 cm. Quelle est la longueur du côté BC ?

La rédaction classique de la solution de cet exercice est :
le triangle ABC est rectangle en A donc, d'après le théorème de Pythagore, on a

$$BC^2 = AB^2 + AC^2$$

d'où $BC^2 = 5^2 + 8^2$ soit $BC^2 = 89$. On en déduit $BC = \sqrt{89}$.

Ce type de démonstration, fréquent en mathématiques, utilise la règle appelée *modus ponens* (ou règle d'élimination du $\Rightarrow$).
Notons

- P la proposition « le triangle ABC est rectangle en A »
- Q la proposition « $BC^2 = AB^2 + AC^2$ »,

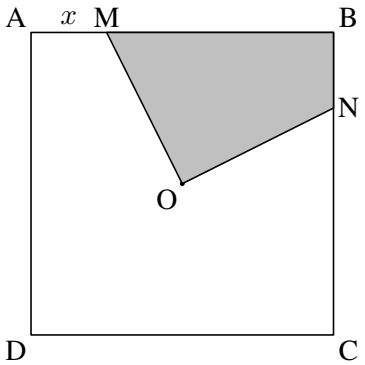
L'instanciation du théorème de Pythagore sur le triangle ABC peut être formalisée par : $P \Rightarrow Q$ et la démonstration ci-dessus par :

$$\frac{\vdash P \quad \vdash (P \Rightarrow Q)}{\vdash Q} \Rightarrow e$$

Le *donc* de la rédaction ci-dessus représente la « mise en acte » de la règle d'élimination du $\Rightarrow$.

2.5.2 La pratique du contre-exemple

On considère un carré $(ABCD)$ de 8 cm de côté et de centre O. Soit M un point du segment $[AB]$ et N le point du segment $[BC]$ tel que les droites (OM) et (ON) soient perpendiculaires.

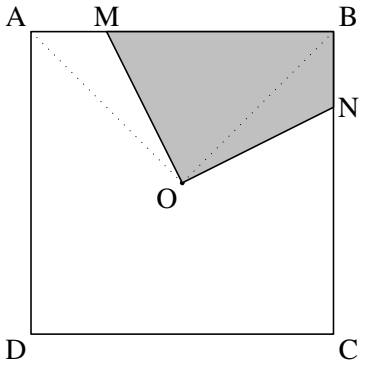


1. *On suppose que $AM = 1$cm. Calculer la longueur OM. En déduire le périmètre du quadrilatère $(MONB)$. Reprendre les calculs précédents en supposant que $AM = 6$cm. Le périmètre du quadrilatère $(MONB)$ est-il constant ?*
2. *On pose $AM = x$. Exprimer OM en fonction de x. En déduire l'expression $P(x)$ du périmètre du quadrilatère $(MONB)$ en fonction de x.*
3. *Étudier l'aire du quadrilatère $(MONB)$ pour différentes positions du point M sur $[AB]$. Quelle conclusion peut-on tirer de cette étude ?*

– Des calculs simples montrent que, si $AM = 1$, $OM = 5$ et le périmètre du quadrilatère $(MONB)$ mesure 18cm. Si $AM = 6$cm, $OM = 2\sqrt{5}$ et le périmetre mesure $8 + 4\sqrt{5}$. On peut donc en conclure que le périmètre du quadrilatère $(MONB)$ n'est pas constant. L'étude du 2. montrera que $P(x) = 8+2\sqrt{x^2 - 8x + 32}$ et que ce périmètre est minimal lorsque $AM = 4$.

– L'étude, pour plusieurs positions du point M sur $[AB]$ montre que l'aire correspondante du quadrilatère $(MONB)$ est de 16cm^2. Cependant ce constat ne suffit pas pour en conclure que l'aire est indépendante de la position du point M sur $[AB]$.

On montre, en utilisant les cas d'égalité des triangles, que OMA et ONB sont isométriques. On déduit, par découpage et comparaison des aires, que le quadrilatère $MONB$ a une aire égale à celle du triangle AOB et, donc, que celle-ci est indépendante de la position du point M sur le segment $[AB]$.



Si l'on note x, y des réels appartenant à [0 ;8] et $P(x)$ le périmètre du quadrilatère $(MONB)$ lorsque $AM = x$, la proposition « le périmètre du quadrilatère $(MONB)$ est constant » se formalise par

$$\forall x \, \forall y \; P(x) = P(y)$$

Montrer que cette proposition est fausse revient à vérifier que sa négation est vraie :

$$\exists x \, \exists y \; \neg(P(x) = P(y))$$

La vérification se fait en exhibant deux valeurs x, y pour lesquelles on a $P(x) \neq P(y)$.

Pour démontrer que l'aire du quadrilatère est constante, on est amené à comparer l'aire du quadrilatère $MONB$ à celle du triangle AOB pour **une** position quelconque du point M sur $[AB]$ et à constater que ces deux aires sont égales. La règle d'introduction du $\forall$ permet de conclure.

2.5.3 Le raisonnement par l'absurde et le raisonnement par récurrence

Si l'on admet comme axiome le principe de raisonnement par récurrence, on peut démontrer le résultat suivant :

Toute partie non vide de $\mathbb{N}$ *admet un plus petit élément.*

Soit Ω une partie non vide de $\mathbb{N}$. Raisonnons par l'absurde et supposons donc que Ω n'admet pas de plus petit élément. Soit $n \in \mathbb{N}$. Notons $P(n)$ la proposition :

$$P(n) \qquad \forall p \in \mathbb{N},\ 0 \leq p \leq n,\ p \notin \Omega$$

$P(0)$ est vraie car, dans le cas contraire, 0 serait le plus petit élément de Ω. Soit n un entier tel que $P(n)$ soit vraie ; alors $P(n+1)$ est vraie.
En effet, $\forall p \in \mathbb{N},\ 0 \leq p \leq n,\ p \notin \Omega$ par hypothèse de récurrence. De plus, $n+1 \notin \Omega$ sinon $n+1$ serait le plus petit élément de Ω. Finalement,

$$\forall p \in \mathbb{N},\ 0 \leq p \leq n+1,\ p \notin \Omega$$

ce qu'il fallait montrer.
D'après le principe de récurrence on en déduit que $\forall n \in \mathbb{N},\ P(n)$ est vraie et en particulier $\forall n \in \mathbb{N},\ n \notin \Omega$, ce qui contredit l'hypothèse « Ω est une partie non vide de $\mathbb{N}$ » et amène la contradiction cherchée.

La démonstration par récurrence de la proposition « $\forall n \in \mathbb{N},\ P(n)$ » peut être formalisée par :

$$\frac{\vdash P(0) \quad \vdash \forall n\, P(n) \Rightarrow P(n+1)}{\vdash \forall n\, P(n)}\ \text{axiome du principe de récurrence}$$

Si l'on note :
– B la proposition « Ω a un plus petit élément »,
– C la proposition « Ω est vide »,
la proposition « $\forall n\, P(n)$ » est équivalente à la proposition C et on peut formaliser le raisonnement par l'absurde par :

$$\dfrac{\dfrac{\neg B \vdash C \quad \vdash \neg C}{\neg B \vdash C \wedge \neg C}\wedge_i}{B}\,a$$

2.5.4 Utilisation de la contraposée à propos d'un exercice de perspective

La photographie ci-dessous montre la façade d'un bâtiment du XVIII^{ème} siècle.

La façade de ce bâtiment est-elle située dans un plan frontal[4] ?

On peut supposer que la façade du bâtiment est située dans un plan vertical et que les bords horizontaux des corniches, des chapiteaux, des balustrades, etc... sont parallèles entre eux. Si la façade du bâtiment était située dans un plan frontal, les représentations des bords horizontaux des différents éléments de la façade devraient être parallèles entre eux.

4. Dans une projection centrale, un plan frontal est un plan parallèle au plan de projection ne passant pas par le centre de la projection (dit aussi point de vue)

Or ils ne le sont pas (ce que montre une vérification graphique sur la photographie) donc la façade du bâtiment n'est pas située dans un plan frontal.

La démonstration repose sur le résultat de cours suivant :
Dans une projection centrale (ou dans une représentation en perspective centrale), si des droites coplanaires sont parallèles entre elles et si le plan qu'elles définissent est un plan frontal, alors leurs projections (leurs représentations) sont des droites parallèles.

Considérons les propositions suivantes où D_1 et D_2 sont des droites coplanaires :
- A : les droites D_1 et D_2 sont parallèles
- B : le plan défini par les droites D_1 et D_2 est un plan frontal
- C : les projections des droites D_1 et D_2 sont parallèles

Le résultat de cours peut se formaliser par : $(A \wedge B) \Rightarrow C$. Le raisonnement utilisé pour résoudre l'exercice s'appuie sur la contraposée de cet énoncé :

$$\neg C \Rightarrow \neg(A \wedge B) \qquad \text{ou encore} \qquad \neg C \Rightarrow (\neg A \vee \neg B).$$

Il se formalise par [5] :

$$\dfrac{\vdash A \qquad \dfrac{\dfrac{\vdash \neg C \quad \vdash \neg C \Rightarrow (\neg A \vee \neg B)}{\vdash \neg A \vee \neg B}\Rightarrow e \qquad \begin{array}{c}\pi\\ \vdots \\ \vdash (\neg A \vee \neg B) \Rightarrow (A \Rightarrow \neg B)\end{array}}{\vdash A \Rightarrow \neg B}\Rightarrow e}{\vdash \neg B}\Rightarrow e$$

2.5.5 La démonstration par disjonction des cas

Au III$^{\text{ème}}$ siècle avant JC, Euclide démontrait dans le livre IX de ses Eléments la proposition 20 suivante : *Les nombres premiers sont en plus grande quantité que toute quantité proposée de nombres premiers* que l'on énonce maintenant sous la forme : *Il existe une infinité de nombres premiers.*

La démonstration d'Euclide [6] :

5. π désigne la démonstration de l'implication $(\neg A \vee \neg B) \Rightarrow (A \Rightarrow \neg B)$. Cf Section 2.3.2 exemple 2 page 57

6. extraite de "LES QVINZE LIVRES DES ELEMENTS GEOMETRIQVES D'EVCLIDE traduicts du François par D.HENRION À PARIS MDCXXXII"

THEOR. 18. PROP. XX.

Quelque multitude de nombres premiers qu'on propoſe, il s'en trouuera encores d'autres.

Soit quelconque multitude de nombres premiers, A, B, C. Ie dis qu'il s'en trouuera encores d'autres.

Car ſi on trouue le nombre D E, le plus petit de tous ceux qui peuuent eſtre meſurez par les trois nombres A, B, C, par la 37. prop. 7. & à icelûy D E on adiouſte l'vnité E F, le tout D F ſera premier ou non: S'il eſt premier, on a vn nombre premier autre que pas vn des propoſez: Mais ſi D F n'eſt premier, il ſera meſuré par quelque nombre premier, par la 34. prop. 7. Soit donc meſuré par G; il eſt éuident que G ne peut pas eſtre vn des trois A, B, C: car s'il eſtoit quelqu'vn d'iceux, il meſureroit comme eux D E. Donc G meſurant le tout D F, & le retranché D E, par la 12. comm. ſent. il meſureroit auſſi le reſte D F, ſçauoir vn nombre l'vnité: Ce qui eſt abſurde. Donc G eſt autre nombre premier que pas vn des propoſez. Et en ceſte façon on en peut trouuer infiny autres. Parquoy quelque multitude de nombres premiers qu'on propoſe, &c. Ce qu'il falloit demonſtrer.

A..2 B...3 C.....5
D---------- 30 E.F
G-----

Soit p_1, p_2, ...,p_n, n nombres premiers distincts. Alors il existe un nombre premier p différent de p_1, p_2, ..., p_n.

Soit l'entier q défini par $q = p_1 \times p_2 \times ... \times p_n + 1$. Ou bien q est premier et il est, de façon évidente, différent de chacun des p_i, ce qu'il fallait montrer. Ou bien q n'est pas premier et il admet un diviseur premier p (proposition 34 du livre VII). Si ce nombre p est l'un des p_i alors il divise à la fois $p_1 \times p_2 \times ... \times p_n$ et $p_1 \times p_2 \times ... \times p_n + 1$ donc il divise leur différence, c'est à dire 1, ce qui est absurde. Donc il est différent de chacun des p_i.

On peut formaliser cette démonstration de la manière suivante :
p_1,..., p_n, q, p sont des constantes, Q désigne la proposition « les constantes p_1,..., p_n représentent des nombres premiers distincts », $P(x)$ désigne la proposition « le nombre x est premier », $R(x)$ désigne la proposition « le nombre x est différent de chacune des constantes p_i » et $D(x, y)$ désigne la proposition « le nombre x divise le nombre y ».

La première partie de la démonstration peut se formaliser par :

$$\dfrac{\dfrac{Q, P(q) \vdash \neg D(p_k, q)}{Q, P(q) \vdash \forall i \neg D(p_i, q)}\forall i}{Q, P(q) \vdash \exists x\, P(x) \wedge R(x)}\exists i$$

en remarquant que la formule $\forall i \, \neg D(p_i, q)$ est équivalente à la formule $R(q)$.

La formule $\neg P(q)$ est équivalente à $\exists p \, P(p) \wedge D(p, q)$ (c'est l'application de la proposition 34 du livre VII des Éléments au nombre q) ; la deuxième partie de la démonstration peut alors se formaliser par :

$$\dfrac{Q, \exists p \, P(p) \wedge D(p, q), \neg R(p) \vdash \bot}{Q, \exists p \, P(p) \wedge D(p, q) \vdash \exists x \, P(x) \wedge R(x)} \, a$$

En regroupant les conclusions de ces deux parties, on peut formaliser la fin de la démonstration par :

$$\dfrac{Q, P(q) \vdash \exists x \, P(x) \wedge R(x) \quad Q, \neg P(q) \vdash \exists x \, P(x) \wedge R(x) \quad Q \vdash P(q) \vee \neg P(q)}{Q \vdash \exists x \, P(x) \wedge R(x)} \, \vee e$$

2.6 Bibliographie

- Article " Logique mathématique" de l'Encyclopædia Universalis.
- Cori R., Lascar D. *Logique mathématique* Cours et exercices corrigés, Licence, Master, Tomes 1 et 2, Dunod
- Dowek G., La logique, Coll. Dominos, Flammarion (1995).
- David R., Nour K, Raffalli C. *Introduction à la logique, Théorie de la démonstration* Cours et exercices corrigés, 2nd cycle, Dunod (2001)