Chapitre 6

Qu'est-ce qu'un algorithme ?

Le but de ce chapitre est de faire comprendre au lecteur ce qu'est un *algorithme* et de bien faire la différence entre ce concept et celui de *programme informatique* (ce dernier étant aussi appelé *logiciel*, *progiciel* ou *application informatique*). Le concept d'« algorithme » est fondamental et s'applique à des problèmes concernant immédiatement le citoyen (c'est à ce titre que nous l'étudions ici) ou non.

Cela fait longtemps que les informaticiens font la différence entre *programme* (sous-entendu d'ordinateur) et *algorithme*. La définition formelle de ce qu'est un algorithme est plus récente (1984) mais ne joue pas de rôle dans le thème abordé, pour lequel on n'a pas à considérer des propriétés des algorithmes dans leur ensemble mais seulement des algorithmes particuliers. Pour satisfaire la curiosité du lecteur, nous en dirons quand même un mot.

Puisque les algorithmes vont jouer de plus en plus un rôle sur la vie du citoyen, il faut en connaître, au-delà de la définition formelle dont nous venons de parler, la définition juridique. Nous verrons que celle-ci est des plus floues.

6.1 La notion informelle d'algorithme

Un *algorithme* est la façon de décrire de façon très précise une suite d'opérations *primitives* pour obtenir le résultat de ce qu'on peut appeler un *calcul*. Qu'importe comment telle ou telle opération primitive est implémentée, c'est la suite (d'opérations) qui est importante.

Les premiers algorithmes

Une fois la notion d'*algorithme* dégagée, on s'est aperçu rétrospectivement que les algorithmes sont utilisés depuis fort longtemps. Les algorithmes les plus anciennement connus datent des Mésopotamiens [Knu-72], au moins deux mille ans avant J.-C., sans qu'il y ait eu de réflexion à cette époque sur ce que peut être un algorithme de façon générale. Parmi ces algorithmes, on trouve la méthode de résolution des équations du second degré, enseignée de nos jours aux lycéens, quelquefois au grand dam de ceux-ci.

Un nom pour désigner un concept

Le nom même 'algorithme' date du XII^e siècle. Il s'agit du premier nom donné à ce concept fondamental, bien que pas encore perçu en tant que concept général à cette époque, comme nous allons le voir. Ce nom apparaît en effet à propos de l'introduction des *chiffres arabes* en Occident et ne désigne pas le concept en question dans toute sa généralité à l'origine.

Les premiers *systèmes de numération*, c'est-à-dire la façon d'énoncer ou d'écrire un nombre, sont *additifs* : on décompose un ensemble contenant un certain nombre d'objets en une famille d'ensembles, chaque ensemble de cette famille contenant un nombre standardisé (un, cinq, dix, cinquante, cent, mille,...) d'éléments. Par exemple, dans le système bien connu de la *numération romaine*, I, V, X, C, M désignent des ensembles à un, cinq, dix, cent et mille éléments. Un nombre s'écrit alors en plaçant bout à bout les symboles des divers ensembles : par exemple MCCXXVI représente un ensemble à 1 226 éléments, si on le traduit en numération de position, sur laquelle nous allons revenir, à laquelle nous sommes plus habitués. Remarquons au passage qu'il n'y a pas d'ordre imposé pour une numération purement additive et que MCCXXIV représente le même nombre pour les Romains ; c'est au Moyen-Âge qu'on y ajoute un *principe soustractif* et que MCCXXIV signifie désormais 1224.

Dans un *systèmes de numération de position*, on a des symboles, non plus pour les groupes, mais pour les petits nombres. Ces symboles sont appelés des *chiffres*. Les groupes de dénombrement, quant à eux, sont entièrement dénotés par la position du chiffre dans la chaîne de caractères dénotant le nombre. On utilise un chiffre nouveau, le chiffre *zéro*, pour dénoter un groupe vide.

On ne connaît pas exactement l'histoire du système de numération de position. Les Babyloniens utilisaient un embryon de système de numération de position de base soixante. Embryon seulement car, malheureusement, ils ne disposaient de l'analogue du signe zéro que pour une position : il n'était pas possible, avec leur système, de mettre deux ou plusieurs zéros de suite, ce qui rendait les nombres ambigus.

La première référence que nous connaissions à propos des *chiffres* dits *arabes* (en fait dus aux Indiens) est un passage de l'évêque syrien Severus Sebokht écrit en 650 (cité dans [Smi-25], vol. 2, p. 64) :

> *Je ne parlerai pas de la science des Hindous, un peuple qui n'est pas le même que les Syriens, ni de leurs découvertes subtiles en astronomie, découvertes qui sont plus ingénieuses que celles des Grecs et des Babyloniens, ni de leurs méthodes de calcul de grande valeur et de leurs calculs qui dépassent la description. Je désire seulement dire que leurs calculs sont faits au moyen de neuf signes.*

On remarquera que le zéro ne fait pas parti des chiffres. La première occurrence du zéro que nous connaissions sans ambiguïté apparaît sur une inscription datant de 876, sise à Gwalior, sur laquelle on voit '50' et '270' ([Dat-26]).

Le premier siècle de l'empire musulman est, dans une large part, consacré aux travaux scientifiques. AL-MAMUM crée une *maison de la sagesse* à Bagdad, comparable à l'ancien Musée d'Alexandrie. C'est dans ce contexte que Mohammed ibn-Musa AL-KHWARIZMI reconnaît la valeur du système indien en 825 et écrit un petit livre expliquant son utilisation [AlK-32]. Ce livre est traduit/adapté en latin au XII^{e} siècle, avec pour incipit *Liber Algorismi de numero Indorum* (le livre d'al-Khwarizmi sur les nombres indiens), ce qui conduira à utiliser le mot 'algorithme' à propos de la nouvelle méthode pour écrire les nombres entiers et pour effectuer les calculs.

Le premier occidental (entendez par là hors de l'empire musulman) à enseigner la nouvelle numération, vers 980, est GERBERT d'Aurillac (938-1003), devenu pape en 999 sous le nom de Sylvestre II. Il est allé étudier en Espagne et a certainement appris ce système à Barcelone, alors en contact étroit avec la civilisation arabe.

Mais la méthode se développe surtout après la parution du livre de FIBONACCI intitulé *Liber abaci*, paru en 1202 [Fib-02]. On remarquera que la méthode des abaques est alors tellement prédominante qu'elle désigne l'arithmétique, d'où le titre 'Le livre des abaques' alors qu'il n'est jamais fait référence à ceux-ci dans ce livre.

Les deux composants d'un algorithme

L'origine du mot 'algorithme' étant éclairci, revenons au concept. On s'est aperçu petit à petit, et ce n'est pas le lieu ici d'en faire l'historique (d'ailleurs très mal connu), qu'un algorithme travaille sur des *structures de données* grâce à des *structures de contrôle*, qui en constituent les deux composants fondamentaux.

Structures de données

Les *données élémentaires* appartiennent à quelques *types* bien repérés : les entiers naturels (0, 1, 2,..., 125,..., 2 534 657, ...), les entiers relatifs (les entiers naturels auxquels on ajoute les entiers négatifs tels que - 3 et - 345), les nombres réels (comme les nombres à virgule tel que 2,34 mais aussi $\sqrt{2}$ ou π), les caractères (de 'a' à 'z', 'A' à 'Z' mais aussi 'é', '_' ou 'β'), les chaînes de caractères (telles que 'Bonjour' ou 'Jacques Martin', constituées d'une suite de caractères, que l'on n'appelle pas *mot* en Informatique car on permet l'espace ' ', alors que ce caractère est considéré comme un séparateur de mots en langue vernaculaire) mais aussi quelquefois, comme pour *Excel*, les dates.

Une *structure de données* est une façon de regrouper des données élémentaires (ou d'autres structures de données) pour obtenir des données plus complexes, un peu comme une fiche de renseignements contenant plusieurs items (le nom et le prénom qui sont des chaîne de caractères, l'âge qui est un entier, le revenu qui est un réel et ainsi de suite). Les structures de données ne ressemblent pas toutes à des fiches ; on peut aussi avoir, par exemple, des listes de données toutes de même type (c'est bien le cas d'une liste de courses à effectuer).

Instructions primitives

Les algorithmes ne s'intéressent pas aux instructions primitives *per se*. Il est cependant important d'avoir une idée de ce à quoi elles ressemblent. Il y en a traditionnellement de trois sortes : les *entrées* permettent de saisir les données fournies par l'utilisateur, les *affectations* permettent des calculs élémentaires sur les données, enfin les *sorties* permettent de communiquer les résultats à l'utilisateur.

Une instruction d'entrée permet de saisir une valeur d'un certain type (entier, caractère, réel, etc.) et de la communiquer à une *variable*, qui conserve cette valeur pour pouvoir la manipuler plus tard. Elle ressemble à :

$$entrer(n)$$

où n est une variable dont le type a auparavant été déclaré.

Une instruction de sortie permet d'afficher la valeur d'une certaine expression, toujours d'un type donné. Elle ressemble à :

$$afficher(e)$$

où e est une expression, également d'un type donné, explicitement déclaré ou que l'on déduit de l'analyse de l'expression.

Chacun connaît les *expressions numériques* qu'il a rencontré au collège, telle que $(2+5\times18)/3$, ou les *expressions algébriques* telle que ax^2+bx+c. De façon générale, une *expression* est obtenue en combinant des constantes, des variables et quelques opérations, dites *primitives* ou *permises*.

Une *affectation* permet de donner une certaine valeur (on dit *affecter*) à une variable d'entrée (mais on change ainsi sa valeur) ou à une variable auxiliaire (c'est-à-dire qui n'est pas une variable d'entrée). Une telle affectation est, sous forme simple, de la forme :

$$var := expression,$$

où *var* est une variable d'un type déclaré et *expression* une expression de même type. Il existe aussi des formes plus complexes telles que :

$$f(t_1, t_2, \cdots, t_n) = expression,$$

où f est une *fonction*, $t_1, t_2, \cdots, t_n$ des termes et *expression* une expression.

Les structures de contrôle

Nous avons dit en commençant qu'un *algorithme* est la façon de décrire de façon très précise une suite d'opérations *primitives* pour obtenir un résultat. Sauf dans de rares cas, un algorithme s'applique à divers *jeux de données* : par exemple pour l'addition, on a deux données, représentées par a et b, et le but est d'obtenir la somme $a + b$. Pour un jeu de données donné, la suite d'opérations primitives est bien déterminée et il suffirait de la lister. Toute la finesse de la chose provient du fait que cette liste d'opérations primitives n'est en général pas la même pour tous les jeux de données, sauf pour des algorithmes simples dits *linéaires*. Les *structures de contrôle* permettent de déterminer la liste d'instructions primitives à appliquer à un jeu de données donné.

Les premiers algorithmes (comme les premiers programmes) comportaient de nombreuses structures de contrôle (dont certaines sont absolument bannies de nos jours). On s'est aperçu petit à petit qu'il suffit de trois types de structures de contrôle pour décrire quelqu'algorithme que ce soit : il s'agit d'une structure de contrôle par défaut, le plus souvent le *séquencement* (mais qui n'est pas nécessairement la bonne, comme nous y reviendrons à propos de la définition formelle de ce qu'un un algorithme), les *branchements* et les *répétitions*.

Lorsqu'on décrit une suite d'instructions primitives (ou non primitives, d'ailleurs), on est bien obligé de suivre un ordre pour le faire. La structure de contrôle de *séquencement* est celle qui dit, qu'alors, on *exécute* (c'est le terme consacré) la première instruction écrite, puis, une fois celle-ci effectuée, la seconde, puis la troisième et ainsi de suite. Il s'agit d'une structure de contrôle bien naturelle, et d'ailleurs la seule à être mise en place dans les *calculateurs*, aussi sophistiqués soient-ils par ailleurs, qui ne sont pas des *ordinateurs*.

L'algorithme pour demander un entier, multiplier celui-ci par deux et afficher le résultat peut être décrit de la façon suivante, en utilisant un séquencement de trois instructions primitives :

```
entrer(n)
e := 2 x n
afficher(e)
```

On a supposé ici que l'on dispose de l'opération de multiplication comme opération primitive. Si on n'avait à notre disposition que l'opération d'addition, par exemple, on aurait utilisé un autre algorithme :

```
entrer(n)
e := n + n
afficher(e)
```

Les *branchements* sont utilisés lorsqu'on a besoin de faire un choix, celui-ci dépendant de la valeur d'une certaine expression au moment où on arrive à ce branchement. Un branchement peut être simple (on parle d'un *test*), souvent écrit :

si *condition* alors *instruction*

qui se comprend de lui-même.

Prenons l'exemple d'un algorithme qui demande la civilité (sous forme abrégée), le nom et le prénom d'une personne et qui affiche 'Bonjour' suivi de la civilité (en toute lettres), du prénom et du nom de la personne. Un algorithme répondant au problème est :

```
afficher('Civilite [1. madame, 2. mademoiselle, 3. monsieur] : ')
lire(civ)
afficher('Nom : ')
lire(nom)
afficher('Prenom : ')
lire(prenom)
si civ = 1 alors afficher('Madame ')
si civ = 2 alors afficher('Mademoiselle ')
si civ = 3 alors afficher('Monsieur ')
afficher(prenom)
afficher(' ')
afficher(nom)
```

dans lequel nous avons supposé qu'une constante chaîne de caractères s'écrit en l'entourant d'apostrophes. On remarquera au passage le traitement des espaces, dont on ne parle pas explicitement dans l'énoncé du problème. Il faut par ailleurs connaître les règles orthotypographiques de la langue dans laquelle on veut afficher le message; ceci est (normalement) facile lorsqu'il s'agit de sa langue maternelle mais le programmeur doit également s'attendre à devoir afficher des messages dans d'autres langues.

On peut aussi utiliser une *alternative* :

si *condition* alors *instruction1* sinon *instruction2*

ou une *alternative multiple* :

dans le cas où *expression*
$= constante_1$ alors $instruction_1$
$= constante_2$ alors $instruction_2$
$\ldots$
$= constante_n$ alors $instruction_n$

Seul le test est vraiment indispensable, les autres branchements pouvant s'en dériver.

Les *répétitions* permettent de répéter (*sic*) un certain nombre de fois une instruction donnée. Chaque instance de cette instruction peut effectuer exactement la même chose (comme elle aurait été utile lors de ma punition « Écrire deux cents fois 'Je ne dois pas parler en classe' » !) mais également des choses différentes, celles-ci dépendant d'un certain nombre de paramètres, variant d'une instance à l'autre, parce qu'une instance change la valeur des paramètres.

On s'est aperçu que trois types de répétitions sont couramment utilisées, et qu'en fait une seule est vraiment nécessaire.

Le premier type est la *répétition contrôlée*, ainsi appelée car l'on sait dès le départ combien d'*itérations* seront effectuées. Sa forme générale est :

pour *index* = m a n faire *instruction*

où *index* est une variable de type entier, m et n des expressions, également de type entier. La sémantique d'une telle instruction est celle que l'on attend d'après sa description : soient m_0 et n_0 les valeurs respectives des expressions m et n avant d'effectuer cette répétition ; l'instruction *instruction* est exécutée $n_0 - m_0 + 1$ fois (s'il s'agit d'un entier positif, aucune fois sinon) ; lors de la première exécution, la valeur de *index* utilisée dans *instruction*, s'il y apparaît des occurrences, vaut m_0 ; lors de la seconde exécution, elle vaut $m_0 + 1$; et ainsi de suite ; lors de la dernière exécution, elle vaut n_0.

Reprenons le problème de duplication consistant à demander un entier (naturel), à multiplier celui-ci par deux et à afficher le résultat ainsi obtenu. Nous avons vu un premier algorithme pour faire cela lorsqu'on dispose de la multiplication et d'un second algorithme lorsqu'on ne dispose que de l'addition. Voyons maintenant un troisième algorithme lorsqu'on ne dispose que du passage au successeur (c'est-à-dire de l'opération qui consiste à ajouter 1 à un entier donné) :

```
entrer(n)
e := n
pour i = 1 a n faire
    e := n + 1
afficher(e)
```

Dans une répétition *non contrôlée*, non seulement on ne sait pas, avant de l'exécuter, combien de fois on va itérer, mais même pire, si cela se terminera un jour. Remarquons, au passage, qu'il peut être intéressant que le processus ne se termine jamais : c'est le cas, en première approximation, d'une horloge (qui doit toujours donner l'heure) ou du pilotage d'un processus industriel (une des hypothèses sur l'accident de Tchernobyl est qu'un ingénieur aurait débranché l'ordinateur pour piloter la centrale « à la main »). Il existe usuellement deux formes de répétitions contrôlées :

tant que *condition* faire *instruction*

faire *instruction* tant que *condition*

Donnons la sémantique de la première forme :

— 1^o) on évalue la condition *condition* ;
— 2^o) a) si elle est vraie, on exécute l'instruction *instruction* et on revient au 1^o) ;
— 2^o) b) si elle est fausse, on a terminé.

Remarquons qu'on a en général intérêt à ce que *condition* contienne une variable (dite *variable de contrôle*) dont la valeur est changée par *instruction* ; sinon il n'y a aucune itération (si la condition est fausse au départ) ou on « boucle indéfiniment » (si la condition est vraie au départ).

Une telle répétition non contrôlée peut faire tout ce qu'une répétition contrôlée peut faire. Donnons, par exemple, un algorithme utilisant une répétition non contrôlée pour le problème de duplication :

```
entrer(n)
e := n
i := 0
tant que i ≠ n faire
    e := n + 1
    i := i + 1
afficher(e)
```

Mais il est plus lisible d'utiliser une répétition contrôlée lorsqu'on le peut. L'intérêt des répétitions non contrôlées est ailleurs. L'indécision sur le nombre d'itérations à effectuer peut provenir de l'interaction avec l'utilisateur. Par exemple un utilisateur veut connaître le cube d'un certain nombre d'entiers naturels mais il ne sait pas de combien de tels entiers il veut connaître le cube dès l'abors. On va donc lui demander un entier, afficher son cube, un autre entier, afficher son cube, et ainsi de suite. Comment l'utilisateur va-t-il faire savoir à quel moment il veut s'arrêter. On peut décider d'une *valeur signal*, par exemple -1 dont le cube n'a pas vraiment besoin d'être calculé (on sait bien que c'est -1), et l'utilisateur fait savoir qu'il veut arrêter l'algorithme en entrant cette valeur signal. Ceci donne lieu à l'algorithme suivant :

```
entrer(n)
tant que n ≠ -1 faire
    e = n × n × n
    afficher(e)
    entrer(n)
```

La raison liée à l'interaction avec l'utilisateur est déjà largement suffisante pour introduire les répétitions non contrôlées mais les algorithméticiens y voient une raison encore plus profonde : il arrive qu'ils ne savent pas donner une expression simple du nombre d'itérations dont on aura besoin ou, pire, si la répétition se terminera un jour. Un petit problème *amusant* occupe les algorithméticiens depuis la fin des années 1920.

En 1928, Lothar Collatz invente le « problème 3x+1 », qu'il présente souvent ensuite dans ses séminaires. En 1952, lors d'une visite à Hambourg, il explique son problème à Helmut Hasse. Ce dernier le diffuse en Amérique à l'université de Syracuse : la *suite de Collatz* prend alors le nom de « suite de Syracuse ». De quoi s'agit-il ? Prenez un entier naturel (quelconque) n : s'il est pair, divisez-le par 2, s'il est impair considérez $2n + 1$; recommencez l'opération un certain nombre de fois. On s'aperçoit qu'on finit par obtenir 1. Par exemple, pour $n = 3$, on obtient :

$$3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

Essayez avec le nombre que vous voulez ! Cependant on ne sait pas établir qu'on trouvera toujours 1 [Lag-85, Del-98, Del-99, Con-13]. Pour nous aider à étudier cette suite de Syracuse, on peut utiliser un algorithme qui demande l'entier de départ et qui affiche les valeurs intermédiaires en s'arrêtant sur 1 (s'il y parvient) :

```
lire(n)
tant que n ≠ 1 faire
    si n mod 2 = 0 alors n = n/2
    sinon n = 3 × n + 1
    afficher(n)
```

où `n mod 2` signifie le reste dans la division euclidienne de n par 2, c'est-à-dire 0 si n est pair et 1 si n est impair.

6.2 Algorithmes et programmes informatiques

Programme informatique comme algorithme et comme directive

Écrire un programme informatique est une façon de décrire un algorithme mais ce n'est pas la seule façon des les décrire : rappelons d'ailleurs que les algorithmes les plus anciennement connus datent des Mésopotamiens alors que les ordinateurs ne datent que de la seconde moitié du XX^e siècle.

Un programme informatique a un autre intérêt que la description des algorithmes. C'est également une façon de l'implémenter pour le rendre opérationnel sur une machine (en l'occurrence un ordinateur), c'est-à-dire qu'on décrit l'algorithme, non pas dans un langage quelconque (compréhensible par un être humain), mais dans un langage également compréhensible par l'ordinateur.

Une dérive

La constation du fait qu'un programme informatique est à la fois la description d'un algorithme et la directive permettant de dire à la machine ce qu'elle doit effectuer (de façon concrète et non pas seulement abstraite) a petit à petit conduit les programmeurs à mal interprêter le rôle d'un « algorithme ». Il est devenu, pour eux, la première esquisse de leur programme, donnant une idée de ce que celui-ci va effectuer mais sans les détails techniques (on dit *syntaxiques*), indispensables pour qu'un tel programme puisse être confié opérationnellement à un ordinateur.

De concept fondamental décrivant l'un des composants de la pensée, allant bien au-delà de l'informatique, il est apparu aux yeux de certains comme un concept secondaire, utile pour l'une des phases seulement de la conception d'un programme.

Retour sur la dérive

Même en restant dans le cadre strict de l'informatique (considérée comme la science des ordinateurs), certains informaticiens se sont rebellés contre ce dernier point de vue et ont tenté de bien dégager la différence entre « algorithme » et « programme informatique ». C'est le cas de Donald Knuth, l'un des *algorithméticiens* les plus célèbres (qui a reçu le Prix Turing en 1974, considéré comme le prix Nobel d'Informatique théorique), dans un article de 1966 [Knu-66] :

> Lettre à l'éditeur
>
> *La lettre du Dr. Hubber définit « algorithme » en terme de langage de programmation. J'aimerais exprimer un point de vue différent, selon lequel les algorithmes sont des concepts ayant une existence propre, indépendamment de tout langage de programmation. Pour moi, le mot* algorithme *désigne une méthode abstraite de calcul de sorties à partir d'entrées alors qu'un* programme *est l'incarnation d'une telle méthode de calcul dans un langage donné. Je peux écrire plusieurs programmes différents pour le même algorithme (par exemple en* Algol 60 *et en* PL/I*, en supposant que ces langages en donnent une interprétation non ambiguë).*
>
> *Bien sûr si on m'oblige à expliquer plus précisément ce que j'entends par ces remarques, je suis forcé d'admettre que je ne connais pas de façon de définir un algorithme particulier autrement qu'à travers un langage de programmation. Peut-être l'ensemble de tous les concepts doit-il être regardé comme un langage formel d'une certaine sorte. Mais je crois que les algorithmes étaient présents bien avant que Turing* at al. *n'en formulent le concept, de même que le nombre « deux » existait bien avant que les auteurs des premiers manuels et autres logiciens n'en donnent une définition précise.*

6.3 Définition formelle des algorithmes

6.3.1 Historique

Le lecteur a certainement remarqué dans le texte de KNUTH cité ci-dessus que celui-ci avoue ne pas savoir comment définir ce qu'est un algorithme ni comment le décrire autrement qu'à travers un langage de programmation. Il est un peu plus explicite en 1973 dans le premier volume de sa célèbre étude des algorithmes :

> *La signification moderne d'algorithme est assez similaire à celle de recette, processus, méthode, technique, procédure, routine, à part que le mot « algorithme » connote quelque chose de légèrement différent. Étant simplement un ensemble fini de règles qui donne une suite d'opérations pour résoudre un type spécifique de problèmes, un algorithme a cinq caractéristiques importantes :*
>
> 1. Finitude. *Un algorithme doit toujours se terminer après un nombre fini d'étapes.* [...]
> 2. Défini. *Chaque étape d'un algorithme doit être définie précisément ; les actions à effectuer doivent être rigoureusement spécifiées de façon non ambiguë pour chaque cas.* [...]
> 3. Entrées. *Un algorithme a zéro entrée ou plus, c'est-à-dire des quantités qui lui sont données initialement avant que l'algorithme commence. Ces entrées sont prises dans des ensembles spécifiés d'objets.* [...]
> 4. Sorties. *Un algorithme a zéro sortie ou plus, c'est-à-dire des quantités qui ont une relation spécifiées avec les entrées.* [...]
> 5. Effectivité. *On s'attend en général à ce qu'un algorithme soit effectif. Ceci signifie que toutes les opérations qui doivent être effectuées dans l'algorithme doivent être suffisamment basiques pour qu'elles puissent l'être en principe exactement et en un temps fini par un homme utilisant du papier et un crayon.* [...]
>
> [Knu-73], p. 4, traduction de Patrick CÉGIELSKI.

La bonne question est la suivante :

> *Existe-t-il un langage de présentation des algorithmes qui permette de présenter tout algorithme écrit en n'importe quel langage avec le même nombre d'étapes d'exécution (on dit* pas à pas*) que dans le langage originel ?*

Un langage est dit **algorithmiquement complet** *s'il permet d'écrire pas à pas tout algorithme.*

KOLMOGOROV et USPENKY publient en 1958 un article dans lequel est présenté ce qui peut être considéré comme un essai de définition de la notion de machine algorithmiquement complète [KU-58]. Cependant il n'y apparaît aucune réflexion globale, aucun aspect philosophique. À la mort de KOLMOGOROV, Yuri GUREVICH écrit un article [Gur-88] dans lequel il indique que KOLMOGOROV aurait pu poser ce problème des langages algorithmiquement complets : « *chaque calcul, utilisant seulement une action locale restreinte à chaque instant, peut être vu comme le calcul (pas seulement une simulation, mais vraiment un calcul) d'une machine KU appropriée* ». Un peu plus tard, USPENSKY répond : « *Je pense que Yuri Gurevich avait une bonne idée, un nouvel éclairage sur notre article.* » [Usp-92].

La notion est dégagée mais le modèle n'est pas satisfaisant. Yuri GUREVICH finit par critiquer ce modèle et par proposer son propre modèle, sous des noms variés (allant de *evolving algebras* à *ASM* pour *Abstract State Machine*). Il défend ce qu'on peut appeler la **thèse de Gurevich** (*la notion d'algorithme est entièrement appréhendée par le modèle des* ASM) à la fin des années 1990. Les ASM apparaissent dans [Gur-84] (sous le nom de « *dynamic structures* »). Un an plus tard, il reprend dans une note :

> *Premièrement, nous adaptons la thèse de Turing au cas où seules des machines avec des ressources bornées sont considérées. Deuxièmement, nous définisson un type plus général d'outils de calculs abstraits, appelés structures dynamiques, et nous posons la nouvelle thèse suivante : chaque outil de calcul peut être simulé par une structure dynamique appropriée – d'à peu près la même taille – en temps réel ; une famille uniforme d'outils de calculs peut être simulée uniformément par une famille appropriée de structures dynamiques en temps réel. En particulier, tout outil de calcul séquentiel peut être simulé par une structure dynamique séquentielle appropriée.*
>
> [Gur-85]

6.3.2 Définition des ASM

Structure du premier ordre

Yuri GUREVICH résout le problème des structures de données permises en considérant que l'on peut utiliser toute « structure mathématique du premier ordre », et même si l'on préfère que les *algèbres*. Une structure du premier ordre est en ensemble A muni d'un certain nombre d'opérations, c'est-à-dire, en termes techniques, de fonctions (à plusieurs variables) de A^n dans A, n étant appelé l'*arité* de la fonction. Pour $n = 0$, on parle plutôt de *constante* et, pour $n \geq 2$, d'*opération*. La structure primitive pour les calculs est $(\mathbb{N}, 0, S)$, où $\mathbb{N}$ est l'ensemble des entiers naturels (0, 1, 2, 3, ...) et S la fonction successeur qui, à un entier x associe son *successeur* $x + 1$ (donc $S0 = 1$, $S123 = 124$).

Termes clos

Les constantes permettent de désigner directement des éléments de l'*ensemble de base* A : zéro est le seul élément ainsi distingué par la structure $(\mathbb{N}, 0, S)$. Les constantes et les opérations permettent de distinguer, indirectement, d'autres éléments : ainsi, dans la même structure, deux est représenté par $SS0$; tout entier naturel n est d'ailleurs représenté par S^n0 dans cette structure avec n occurrences du symbole S.

Si on veut formaliser cela, on parle de *termes clos* définis par induction de la façon suivante :

— toute constante est un terme clos ;
— si $t_1, \cdots, t_n$ sont des termes clos et f une fonction n-aire de base alors $f(t_1, \cdots, t_n)$ est un terme clos ;
— tous les termes clos sont définis de cette façon.

Mise à jour

Parmi les fonctions de la structure du premier ordre, certaines sont « constantes » (on dit *statiques*) lorsqu'on passe d'une algèbre à une autre au cours de l'exécution du programme, d'autres varient (on dit qu'elles sont *dynamiques*). Un pas de calcul ne permet pas cependant de changer du tout au tout une fonction (dynamique), mais uniquement sa valeur en un seul point,

grâce à l'instruction primitive d'*affectation* (renommée *mise à jour* dans le cadre des ASM) :

$$f(t_1, \cdots, t_n) := t_0$$

Simultanéïté

Nous avons vu que la structure de contrôle par défaut est, dans la plupart des langages de programmation, le *séquencement*, qui paraît naturel. Yuri GUREVICH a découvert qu'il vaut mieux remplacer celle-ci par la *simultanéïté*, c'est-à-dire que lorsqu'on a un *bloc* d'instructions :

`par`
$instruction_1$
$instruction_2$
$\cdots$
$instruction_n$
`endpar`

(où `par` était le début de *parallel* à l'origine, le début de *parenthesis* maintenant), on n'exécute pas la première instruction, puis la seconde et ainsi de suite. On les exécute simultanément ou, si on préfère, on les exécute (dans l'ordre que l'on veut) mais on ne tient compte des résultats, c'est-à-dire des mises à jour qu'une fois toutes les instructions exécutées.

Par exemple, si avant l'exécution, les termes dynamiques n et m valent respectivement 2 et 3 alors, après exécution du bloc :

`par`
$n := 5$
$m := 2 \times n + 3$
`endpar`

avec une interprétation séquentielle, on obtient 5 et 13 ($= 2 \times 5 + 3$) pour n et m respectivement alors, qu'avec une interprétation simultanéiste, on obtient toujours 5 pour n mais 9 ($= 2 \times 2 + 3$) pour m.

Le lecteur n'aura pas été sans remarquer que l'on peut tomber sur une *incohérence*, c'est-à-dire que l'on peut tenter ainsi essayer de donner *a priori* deux valeurs différentes à un même terme dynamique. Par exemple, l'exécution du bloc :

`par`
$n := 5$
$n := 2 \times m + 3$
`endpar`

essaie d'attribuer les valeurs 5 et 9 ($= 2 \times 3 + 3$) à n. Comme on ne sait pas laquelle choisir, dans un tel cas, on arrête purement et simplement le déroulement de l'algorithme lorsqu'on rencontre une telle incohérence.

Structures de contrôle

Les structures de contrôle (explicites) sont uniquement, pour les ASM, la structure de contrôle par défaut, à savoir la simultanéïté, et le test. Un *programme* ASM est donc de la forme :

`par`
`if` $condition_1$ `then` $instruction_1$
`if` $condition_2$ `then` $instruction_2$
$\cdots$
`if` $condition_n$ `then` $instruction_n$

`endpar`

où les *conditions* sont des termes *booléens*, c'est-à-dire prenant l'une des valeurs de vérité `vrai` ou `faux`, et les *instructions* des blocs de mises à jour [1].

On convient de ne pas écrire explicitement les marqueurs `par` et `endpar` du bloc initial.

Aucune répétition n'apparaît explicitement. Ceci est dû au fait qu'une application d'un tel programme constitue un pas de programme. On recommence ensuite pour un deuxième pas, puis une troisième fois et ainsi de suite.

Deux cas peuvent se produire à terme. Si on parvient à un pas de programme tel que tous les termes (dynamiques) gardent la même valeur que dans le pas précédent alors on peut alors appliquer autant de fois que l'on veut le programme, les valeurs seront toujours inchangées ; on dit que l'on a atteint un *point fixe*. Dans ce cas, le résultat est donné par les valeurs des termes (dynamiques) que l'on a choisi comme sortie. Sinon le programme *boucle* (sous-entendu indéfiniment) et il n'y a pas de résultat.

Algèbre initiale

Le fonctionnement d'une ASM doit être désormais suffisamment claire. On considère une *algèbre initiale* constituée d'un ensemble de base, d'un certain nombre de fonctions statiques, qui en constituent le contexte, et d'un certain nombre de fonctions dynamiques.

Après un pas de calcul, on obtient une nouvelle algèbre, de même ensemble de base, de même *signature*, c'est-à-dire que la suite des arités des fonctions est la même. Le contexte ne change pas mais les fonctions dynamiques peuvent avoir changé, en un nombre fini de points.

Un exemple

Prenons l'exemple du calcul du n-ième terme de la suite de Fibonacci.

La **suite de Fibonacci** est la suite de nombres entiers :

$$1, 1, 2, 3, 5, 8, 13, \cdots$$

Elle paraît mystérieuse à première vue mais une règle la gouverne : les deux premiers éléments de cette suite, 1 et 1, semblent choisis au hasard. Par contre, les éléments suivants sont le résultat de la somme des deux éléments qui le précèdent :

$$\begin{aligned} 2 &= 1 + 1, \\ 3 &= 1 + 2, \\ 5 &= 2 + 3, \\ 8 &= 3 + 5. \end{aligned}$$

Cette suite a toute une histoire et une revue (*The Fibonacci Quaterly*) lui est même entièrement consacrée.

Elle doit son nom à Leonardo Fibonacci qui, dans un problème récréatif posé dans l'ouvrage *Liber abaci* publié en 1202 [Fib-02], décrit la croissance d'une population de lapins : « *Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence*? »

Qui n'a pas été ému par le pauvre héros éponyme du roman *Jean de Florette* de Marcel Pagnol, publié en 1963, interprété par Gérard Depardieu dans le film de 1986 de Claude Berri,

1. Cela peut être plus complexe *a priori*, avec des inbrications de test et de blocs, mais on peut se ramener à la *forme normale* indiquée.

qui pense pouvoir appliquer à la lettre la solution du problème de FIBONACCI et commence un élevage de lapins.

Une algèbre adéquate est $(\mathbb{N}, +, 0, 1, \leq, n, f_p, f_a, f, step)$, c'est-à-dire la structure des entiers naturels munie de l'addition, de l'inégalité au sens large, des deux constantes 0 et 1 ainsi que des constantes dynamiques n, f_p, f_a, f et $step$. L'entrée est n, initialisée dans l'algèbre initiale, la sortie est f, la valeur de F_n. Les constantes dynamiques f_p (valeur pénultième de f), f_a (valeur de f dans l'étape précédente) et $step$ servent de « variables » auxiliaires. La constante $step$, indiquant le numéro d'étape, est initialisée à zéro dans l'algèbre initiale. Le programme est :

```
if step = 0 ou step = 1 then
   par
   f_p = 1
   f_a = 1
   f = 1
   step = step + 1
   endpar
if step ≠ 0 et step ≠ 1 et step ≤ n then
   par
   f_p = f_a
   f_a = f
   f = f_p + f_a
   step = step + 1
   endpar
```

Pour calculer F_4, par exemple, l'agèbre initiale est $\mathcal{A}_0 = (\mathbb{N}, +, 0, 1, \leq, 4, \bot, \bot, \bot, 0)$, où $\bot$ désigne une valeur indéfinie car la valeur n'a pas d'importance. En appliquant le programme ASM ci-dessus, la suite des algèbres est :

$\mathcal{A}_0 = (\mathbb{N}, +, 0, 1, \leq, 3, \bot, \bot, \bot, 0)$
$\mathcal{A}_1 = (\mathbb{N}, +, 0, 1, \leq, 3, 1, 1, 1, 1)$
$\mathcal{A}_2 = (\mathbb{N}, +, 0, 1, \leq, 3, 1, 1, 1, 2)$
$\mathcal{A}_3 = (\mathbb{N}, +, 0, 1, \leq, 3, 1, 1, 2, 3)$
$\mathcal{A}_4 = (\mathbb{N}, +, 0, 1, \leq, 3, 1, 2, 3, 4)$
$\mathcal{A}_5 = (\mathbb{N}, +, 0, 1, \leq, 3, 2, 3, 5, 5) = \mathcal{A}_6 = \mathcal{A}_7 = \cdots$

On obtient donc un point fixe et le résultat est 5.

6.3.3 Propriété fondamentale des ASM

L'intérêt des ASM est que, pour tout algorithme, au sens intuitif, décrit par quelque manière que ce soit, on peut :

— lui associer une algèbre spécifiant son contexte et les variables qui varient au cours de l'exécution de cet algorithme ;
— un programme ASM tel que les fonctions dynamiques correspondant à ces variables prennent les mêmes valeurs qu'avec l'algorithme originel à chaque étape, une étape de l'ASM correspondant à une étape de l'algorithme originel.

Lorsqu'on rencontrait Yuri GUREVICH dans les années 90, il vous demandait toujours : « *Quel est ton algorithme préféré* ? », puis il écrivait un programme d'ASM permettant de simuler celui-ci, pas à pas. Il mettait ainsi en œuvre ce qui est maintenant appelé la *thèse de Gurevich*.

En 2000, il a donné trois conditions que doit vérifier un algorithme (séquentiel), acceptables par tout le monde, et a démontré un théorème montrant que tout tel algorithme peut être simulé pas à pas par une ASM.

6.4 Les algorithmes d'un point de vue juridique

Alors qu'il existe une définition formelle de ce qu'est un algorithme depuis 1984, le droit français, comme le droit de l'union européenne ou le droit d'autres pays, le considère encore comme une notion annexe fort imprécise.

Contexte

Le droit s'intéresse aux algorithmes [2] dans des domaines variés allant de la responsabilité civile au commerce électronique [3]. Un texte de droit, l'arrêté du 27 juin 1989 relatif à l'enrichissement du vocabulaire de l'informatique, définit l'algorithmique comme « *l'étude de la résolution de problèmes par la mise en œuvre de suites d'opérations élémentaires selon un processus défini aboutissant à une solution* ». Ce texte propose une définition très générale de l'algorithmique ; il n'a pas de portée juridique spécifique, car il ne contient pas de règles applicables aux algorithmes. Ce sont dans des textes, dans des décisions de justice et dans des travaux de doctrine portant sur la propriété intellectuelle que la notion même d'algorithme est analysée. Plus précisément, le terme « algorithme » existe dans des sources qui concernent la protection du logiciel par le droit d'auteur ou par le droit des brevets [4]. En droit d'auteur, la question est de savoir si l'algorithme est protégé en tant que tel par le droit d'auteur des logiciels ; en droit des brevets, il s'agit d'apprécier si le logiciel est davantage qu'un algorithme abstrait. Néanmoins, malgré cet intérêt, le droit français, comme le droit communautaire ou le droit d'autres pays, considère encore l'algorithme comme une notion annexe fort imprécise.

Textes juridiques

Un texte, propre à la propriété intellectuelle traite, sans le définir, de l'algorithme. Ce n'est pas un texte de droit des brevets, car ce droit ne s'applique pas aux logiciels en tant que tels : ils ne sont pas considérés comme des inventions [5]. En effet, le droit des brevets confère « *un droit exclusif d'exploitation* » sur « *les inventions nouvelles impliquant une activité inventive et susceptibles d'application industrielle* » (CPI, art. L. 611-1 et L. 611-10). Or les logiciels ont été considérés comme dépourvus d'application industrielle [6]. Le rejet de la brevetabilité des logiciels, au profit de la protection par le droit d'auteur, est en réalité surtout un choix légal [7], qui est

2. V. L. Costes, *2017, l'année de la régulation des algorithmes*, Revue Lamy Droit de l'Immatériel, janvier 2017, p. 3.

3. V. M. Lamoureux, *La causalité juridique à l'épreuve des algorithmes*, JCP G 2016, doctr. 731 et L. D. Godefroy, *Pour un droit du traitement des données par les algorithmes prédictifs dans le commerce électronique*, D. 2016, p. 438.

4. En droit des brevets, « *la propriété porte(rait) sur ce que fait le programme, c'est-à-dire sur sa fonction. Le droit d'auteur des logiciels concerne la forme du code source du programme et son architecture. Autrement dit, le droit de brevet protège le fond de l'idée tandis que le droit d'auteur en protège la forme* », M. Dhenne, Technique et droit des brevets, L'invention en droit des brevets, préface de J.-Ch. Galloux, BDE n^o 89, LexisNexis, 2016, n^o 628.

5. Code de la propriété intellectuelle (CPI), art. L. 611-10, 2, c et Convention sur le brevet européen (CBE), art. 52, 2, C.

6. V. N. Binctin, Droit de la propriété intellectuelle, 4ème éd., LGDJ, 2016, n^o 462 : l'application industrielle « *est recherchée en identifiant la présence d'un effet technique. L'effet technique est parfois nié pour les logiciels* ». V. égal. A. Lucas, La protection des créations industrielles abstraites, préface de E. du Pontavice, Libr. Tech., 1975, n^o 137 : la loi considère que les programmes d'ordinateur sont « *des systèmes abstraits non brevetables* ».

7. V. J. Raynard, E. Py et P. Tréfigny, Droit de la propriété intellectuelle, LexisNexis, 2016, n^o 61 : « *historiquement,* (...)*, l'exclusion n'est pas étrangère à des considérations d'ordre pratique, tenant à la difficulté d'accéder à l'état de la technique dans ce domaine, ainsi qu'à des considérations d'opportunité liées à la prédominance américaine dans l'industrie du logiciel* ». Sur le choix politique de la protection du logiciel par le droit d'auteur, v. Ph. Gaudrat et F. Sardain, Traité de droit civil du numérique, tome 1. Droit des biens, Larcier, 2015, n^o 219s et 230s.

par ailleurs contesté[8]. Le droit américain[9] et la jurisprudence de l'Office Européen des Brevets (OEB)[10] tendent d'ailleurs à évoluer sur cette question.

Le texte en question est la directive européenne concernant la protection juridique des programmes d'ordinateur[11]. Cette directive, transposée en France dans le Code de la propriété intellectuelle[12], protège les logiciels par le droit d'auteur (avec un régime adapté). Le droit d'auteur accorde à l'auteur d'une « *œuvre de l'esprit* » (création de forme originale) « *un droit de propriété incorporelle exclusif et opposable à tous* »[13]. L'œuvre de l'esprit est une création qui doit être à la fois formalisée (le droit d'auteur ne protège pas les idées) et originale (elle doit porter l'empreinte de la personnalité de son auteur : cette exigence a été adaptée pour les logiciels[14]).

La directive, qui ne propose pas non plus de définition du logiciel[15], dispose que « *seule l'expression d'un programme d'ordinateur est protégée* ». La Cour de Justice de l'Union Européenne précise que la protection porte sur le code source et sur le code objet du logiciel[16]. Elle porte aussi sur le matériel de conception préparatoire[17]. La protection est en revanche exclue pour les fonctionnalités, les langages de programmation[18] ainsi que pour les interfaces graphiques[19].

Le droit décompose donc le logiciel en plusieurs éléments et distingue entre ceux qui doivent être protégés par le droit d'auteur du logiciel et les autres[20]. *Quid* de l'algorithme ? Selon la

8. V. A. Lucas, La protection des créations industrielles abstraites, *op. cit.*, n^o 254 et Ph. Gaudrat et F. Sardain, *op. cit.*, n^o 251s.

9. Décision In re Lowry (32 F3d 1579, 1583, 1584, 32 USPQ2d 1031, 1035 (Fed. Cir. 1994). Sur cette décision, v. M. Dhenne, *op. cit.*, n^o 614 (« *recevabilité des revendications programmes-produits* » : « *la demande portait sur une méthode d'organisation des données stockées dans la mémoire d'un ordinateur. La revendication litigieuse visait une mémoire lisible par un ordinateur. Un programme d'ordinateur était inséré dans la mémoire pour que la machine fonctionne d'une manière particulière* »).

10. D'après l'étude de la jurisprudence de l'OEB, est admise la brevetabilité des programmes d'ordinateur « *dont l'exécution est capable d'engendrer un effet technique supplémentaire* » (M. Dhenne, *op. cit.*, n^o 616).

11. Directive 2009/24/CE du 23 avril 2009, laquelle remplace la directive 91/250/CEE du même nom du 14 mai 1991. Aux États-Unis, v. le *Copyright Software Act* de 1980.

12. CPI, art. L. 112-2, 13^o.

13. CPI, art. L. 111-1.

14. Cass., Ass. plén., 7 mars 1986, « Pachot », n^o 83-10.477, Bull. n^o 3 : en matière de logiciel, l'originalité est un « *effort personnalisé allant au-delà de la simple mise en œuvre d'une logique automatique et contraignante* ».

15. « *Aux fins de la présente directive, les termes 'programme d'ordinateur' visent les programmes sous quelque forme que ce soit, y compris ceux qui sont incorporés au matériel. Ces termes comprennent également les travaux préparatoires de conception aboutissant au développement d'un programme, à condition qu'ils soient de nature à permettre la réalisation d'un programme d'ordinateur à un stade ultérieur* », considérant n^o 7 de la directive.

16. CJUE 22 déc. 2010, aff. C-393/09 ; v. égal. en ce sens l'Accord sur les aspects des droits de propriété intellectuelle qui touchent au commerce, « ADPIC », 15 avr. 1994, art. 10 : « *les programmes d'ordinateur, qu'ils soient exprimés en code source ou en code objet, seront protégés en tant qu'œuvres littéraires en vertu de la Convention de Berne* ». C'est conforme à l'exception de la décompilation pour des raisons d'interopérabilité qui permet d'accéder au code source, v. CPI, art. L. 122-6-1, IV : « *La reproduction du code du logiciel ou la traduction de la forme de ce code n'est pas soumise à l'autorisation de l'auteur lorsque la reproduction ou la traduction au sens du 1^o ou du 2^o de l'article L. 122-6 est indispensable pour obtenir les informations nécessaires à l'interopérabilité d'un logiciel créé de façon indépendante avec d'autres logiciels* (...) ».

17. Pour une définition, v. les conclusions de l'avocat général M. Yves Bot présentées le 14 octobre 2010 (affaire C-393/09) : « *62. D'ailleurs, c'est la raison pour laquelle le matériel de conception préparatoire, lorsqu'il permet d'aboutir à la création d'un tel programme, est également protégé par le droit d'auteur applicable au programme d'ordinateur. 63. Ce matériel peut comprendre, par exemple, une structure ou un organigramme mis au point par le programmeur et qui seraient susceptibles d'être retranscrits en code source et en code objet, permettant ainsi à la machine d'exécuter le programme d'ordinateur. Cet organigramme élaboré par le programmeur pourrait être comparé au scénario d'un film.* ».

18. CJUE 2 mai 2012, n^o C-406/10, aff. SAS Institute Inc./World Programming Ltd.

19. CJUE 22 déc. 2010 préc. La protection reste possible par le droit commun du droit d'auteur en présence d'une création de forme originale.

20. V. Ph. Gaudrat, « Forme numérique et propriété intellectuelle », RTD com. 2000, p. 910 : l'auteur qualifie le logiciel d'« *universalité* ». V. égal. M. Dhenne, *op. cit.*, n^o 282 : « *la propriété littéraire ne concerne donc en principe que la forme du programme d'ordinateur. Cette délimitation de l'objet du droit s'avère difficile à réaliser* ».

directive, « **les idées et les principes qui sont à la base des différents éléments d'un programme, y compris ceux qui sont à la base de ses interfaces, ne sont pas protégés par le droit d'auteur en vertu de la présente directive**. *En accord avec ce principe du droit d'auteur,* **les idées et principes qui sont à la base de la logique, des algorithmes et des langages de programmation ne sont pas protégés en vertu de la présente directive**. *Conformément à la législation et à la jurisprudence des États membres ainsi qu'aux conventions internationales sur le droit d'auteur, l'expression de ces idées et principes doit être protégée par le droit d'auteur* » (considérant n° 11) [21]. Deux interprétations peuvent être données de cette disposition. Dans une première lecture, elle pourrait signifier que les algorithmes sont des idées et principes non protégés. Mais une lecture plus attentive du texte pourrait conduire à une seconde interprétation selon laquelle les algorithmes seraient protégés par le droit d'auteur du logiciel : seuls les idées et principes à la base de ces algorithmes seraient exclus de la protection [22]. Contrairement à la directive européenne, le Code de la propriété intellectuelle ne recourt pas au terme « algorithme » et ne permet pas de faire un choix entre les deux interprétations.

Jurisprudence

Il ressort pourtant de l'analyse de décisions de justice françaises rendues en droit d'auteur une certaine exclusion de l'algorithme de la protection du logiciel. Il s'agit d'espèces où les juges devaient se prononcer sur la contrefaçon d'un logiciel [23]. Tout d'abord, la Cour de cassation elle-même dénie tout rôle à l'algorithme dans l'appréciation de l'originalité du logiciel, condition de sa protection, au contraire de l'organigramme et du code [24]. Ensuite, selon la Cour d'appel de Paris, « *il importe peu que les algorithmes et les fonctionnalités des deux programmes soient identiques, ceux-ci n'étant pas protégeables au titre du droit d'auteur* » [25]. Il y a vingt ans, un arrêt de la même cour contenait une sorte de définition de l'algorithme dans une affaire où il s'agissait d'apprécier si le logiciel développé par une société contrefaisait l'algorithme d'un automate de paiement créé par un inventeur [26]. La cour rejette la contrefaçon, car l'algorithme « *qui n'est, selon les experts, qu'une succession d'opérations et ne traduit qu'un énoncé logique de fonctionnalité, dénué de toutes les spécifications fonctionnelles du produit recherché, n'est pas une œuvre de l'esprit originale allant au-delà d'une simple logique automatique et contraignante, et dès lors, il ne peut être considéré comme un logiciel* » [27]. Cette disposition ne doit peut-être pas être comprise comme une définition générale, mais comme une simple analyse, en l'espèce, de

21. V. égal. l'article 1.2 de la directive sur l'objet de la protection (« *la protection prévue par la présente directive s'applique à toute forme d'expression d'un programme d'ordinateur. Les idées et principes qui sont à la base de quelque élément que ce soit d'un programme d'ordinateur, y compris ceux qui sont à la base de ses interfaces, ne sont pas protégés par le droit d'auteur en vertu de la présente directive* »), la directive de 1991 susmentionnée et, en jurisprudence, CJUE, 2 mai 2012, préc.

22. Rappr. de l'exception d'analyse, CPI, art. L. 122-6-1, III : « *la personne ayant le droit d'utiliser le logiciel peut sans l'autorisation de l'auteur observer, étudier ou tester le fonctionnement ou la sécurité de ce logiciel afin de déterminer les idées et principes qui sont à la base de n'importe quel élément du logiciel lorsqu'elle effectue toute opération de chargement, d'affichage, d'exécution, de transmission ou de stockage du logiciel qu'elle est en droit d'effectuer* ».

23. Reproduction ou représentation illicite du logiciel.

24. Cass., civ. 1ère, 14 nov. 2013, n° 12-20.687, « Microsoft » : « *mais attendu que l'arrêt, après avoir relevé que le rapport d'expertise qui se bornait à étudier les langages de programmation mis en œuvre, et évoquait les algorithmes et les fonctionnalités du programme, non protégés par le droit d'auteur, constate que les intéressés n'avaient fourni aucun élément de nature à justifier de l'originalité des composantes du logiciel, telles que les lignes de programmation, les codes ou l'organigramme, ou du matériel de conception préparatoire* (...) ».

25. CA Paris, pôle 5, 1ère ch., 24 nov. 2015 (absence de contrefaçon pour ce motif).

26. Ce dernier soutient que « *la programmation de cet appareil (fabriqué par la société) reprend l'enchaînement des fonctions ou opérations matérialisant son idée et décrites par lui dans les algorithmes, communément appelés organigrammes* ».

27. CA Paris, 1ère ch. d'accusation, 23 janvier 1995, RD propr. intell., avr. 1995, p. 52 ; PIBD 1995, III, p. 278.

l'algorithme [28]. Enfin, pour la Cour d'appel de Caen, seule est une contrefaçon la décompilation, sans but d'interopérabilité, de simples algorithmes du code source d'un logiciel : pas celle de simples algorithmes non protégés [29].

À la lecture de ces décisions, l'algorithme apparaît comme une simple idée, non protégée, à distinguer du logiciel qui est la seule création de forme protégeable. Une explication à cette jurisprudence serait une transposition au logiciel de la distinction, classique en droit d'auteur, entre l'idée (non protégeable par le droit d'auteur) et la forme (objet de la protection). Le problème est que cette distinction est née à propos d'œuvres d'art non logicielles. De manière générale, le droit d'auteur a été conçu pour des œuvres communicables au public, ce qui n'est pas le cas du code source d'un logiciel. Les concepts classiques du droit d'auteur sont donc difficiles à transposer en matière de logiciel (comme en témoigne également l'adaptation de la notion d'originalité mentionnée plus haut).

On retrouve une analyse voisine en droit des brevets : l'Office Européen des Brevets a déjà défini l'algorithme comme n'étant, à l'instar de la méthode mathématique, qu'« *un concept abstrait prescrivant la façon de traiter les nombres* » [30], tandis que « *la Cour Suprême des États-Unis affirma à plusieurs reprises qu'un programme d'ordinateur, qu'elle assimilait à un algorithme, constituait un procédé purement mental, une création abstraite* » [31].

Doctrine

Des articles et des ouvrages juridiques ont été écrits sur la protection des algorithmes et des logiciels par le droit d'auteur et par le droit des brevets.

Tout d'abord, des auteurs tentent de définir ce qu'est un algorithme et un logiciel. M. Dhenne donne de l'algorithme la définition suivante : « *un algorithme classique (...) constitue une formule mathématique abstraite* » [32]. De la même manière, pour le professeur André Lucas, l'algorithme « *constitue une méthode abstraite de traitement de l'information* » [33]. M. Dhenne estime que le logiciel « *est pour l'essentiel un algorithme ou une suite d'algorithmes dont la réalisation permet l'accomplissement d'une fonction par une machine* » [34]. Pour MM. Gaudrat et Sardain, « *à la base du programme se trouve l'algorithme qui exprime le processus logique* » [35]. L'algorithme est donc vu, au sens de la dérive mentionnée dans la partie théorique ci-dessus, comme une partie, abstraite, du logiciel.

Ensuite, pour la doctrine, le logiciel ne se réduit pas à l'algorithme : le logiciel est, lui, davantage qu'une formule mathématique abstraite, et donc que l'algorithme classique susmentionné [36]. Pour M. Dhenne, « *on doit (...) dissocier le programme de l'algorithme. Un algorithme constitue une formule mathématique dont la réalisation dépend uniquement de l'intelligence de son exécutant. C'est la forme écrite d'un procédé de calcul composé d'une série d'étapes. (...). D'un point de vue informatique, l'algorithme correspond à une fonction du programme. Cette fonction est déterminée durant la phase de conception de l'analyse organique, c'est-à-dire au stade de la détermination de la fonction. L'algorithme n'acquiert un sens informatique que quand il est traduit par un langage informatique. Il doit, autrement dit, être programmé. Un programme*

28. La définition a été néanmoins reprise récemment par CA Caen, ch. appels corr., 18 mars 2015, « Skype ».
29. CA Caen, « Skype » préc.
30. OEB, chambre de recours, 15 juillet 1986, n^o T0208/84 (invention concernant un calculateur), point 5.
31. F. Toubol, Le logiciel. Analyse juridique, avant-propos de P. Catala, préf. de C. Lucas de Leyssac, LGDJ, 1986, p 192.
32. M. Dhenne, *op. cit.*, n^o 266.
33. A. Lucas La protection des créations industrielles abstraites, *op. cit.*, n^o 137. C'est à rapprocher de l'algorithme en tant qu'idée en droit d'auteur (V. ibid n^o 255).
34. M. Dhenne, *op. cit.*, n^o 276.
35. Ph. Gaudrat et F. Sardain, *op. cit.*, n^o 197s et A. Lucas, Le droit de l'informatique, PUF, 1987.
36. M. Dhenne, *op. cit.*, n^o 266.

est en définitive la traduction par programmation d'un ensemble d'algorithmes dans un langage informatique particulier. Cette proximité entre programme d'ordinateur et algorithme explique qu'ils soient souvent confondus, en particulier par les juges américains »[37].

M. André Lucas distingue également entre algorithme et logiciel, afin de déterminer à partir de quel moment la brevetabilité pourrait s'appliquer : « *en tant qu'il constitue une méthode abstraite de traitement de l'information, l'algorithme ne doit pas pouvoir être monopolisé ; mais il en va autrement lorsque, appliqué au problème donné, il se concrétise dans l'organigramme. C'est donc dans l'organigramme que se révèle, le cas échéant, l'invention et c'est à ce niveau que devrait intervenir la protection* »[38]. Pour l'auteur, « *il est excessif (...) de résumer le programme à l'algorithme car cela revient à ne considérer que le principe de l'invention sans se préoccuper de l'application qui en est faite* »[39]. L'algorithme n'est ainsi considéré que comme une première étape dans l'élaboration du logiciel[40].

Enfin, la doctrine suit la jurisprudence susmentionnée qui refuse d'analyser l'algorithme lorsqu'elle apprécie l'originalité d'un logiciel[41].

D'après la doctrine juridique, il semble que l'algorithme est considéré comme une partie du logiciel, plus précisément comme sa partie abstraite non protégée ou que le logiciel est un algorithme qui a accédé à une certaine complexité[42].

Conclusion

En conclusion, la frontière est délicate à tracer, en droit de la propriété intellectuelle, entre le logiciel et l'algorithme. Or le droit doit s'appliquer à des concepts définis[43]. Avant d'apporter un traitement juridique à ces notions « techniques », le législateur et les juges doivent saisir le phénomène informatique[44]. La question du mode de protection juridique adéquat pour le logiciel

37. *Ibid*, n° 269.

38. A. Lucas La protection des créations industrielles abstraites, *op. cit.*, n° 137. C'est à rapprocher de l'algorithme en tant qu'idée en droit d'auteur (V. *ibid* n° 255).

39. *Ibid*, n° 251.

40. Sur la question de l'achèvement du logiciel, v. A. Lucas, Le droit de l'informatique, *op. cit.*, n° 196 : « *il n'est pas facile de déterminer à partir de quel stade un logiciel peut être considéré comme créé* ». Avec la rédaction des instructions ? Plutôt « *dès (l)a première phase (du logiciel)* » selon le professeur André Lucas (le droit d'auteur protège les œuvres inachevées).

41. V. F. Toubol, *op. cit.*, p. 67 (« *si l'on admet qu'un programme d'ordinateur puisse être original, c'est donc au niveau de son code source et éventuellement de l'organigramme qu'il peut y avoir protection* »), A. Lucas La protection des créations industrielles abstraites, *op. cit.*, n° 184 (l'auteur se demande dans quels éléments du logiciel réside l'originalité : selon lui, dans l'organigramme et les instructions) et M. Dhenne, *op. cit.*, n° 282 (« *le code source et l'architecture du programme d'ordinateur constituent respectivement une expression et une composition protégeables* ». Pour la composition, le logiciel « *constitue une suite de modules formés d'algorithmes. Ces modules sont organisés en fonction d'une architecture donnée (...) qui constitue une composition protégeable* »).

42. G. Bertin et I. de Lamberterie, La protection du logiciel. Enjeux juridiques et économiques, préf. J.-C. Combaldieu, LGDJ, 1995, p. 27 (« *l'originalité ainsi décrite n'apparaît que dans des logiciels quelque peu complexes. Si l'opération est relativement simple et l'algorithme prédéterminé par le résultat à atteindre, le logiciel pourra apparaître alors comme dépourvu du caractère d'originalité* »).

43. V. Ph. Gaudrat, art. préc. : « *à tous égards, le fichier s'impose comme le concept transversal et spécifique à la matière. Le choix d'une technique juridique adéquate de protection ne vient qu'en second lieu. On ne peut en discuter et l'évaluer correctement qu'à partir du moment où l'objet à protéger a été clairement identifié. Or, c'est une phase méthodologique qui, pour essentielle qu'elle soit, a été souverainement négligée en matière informatique. Le concept de fichier a été contourné ; la spécificité d'une forme numérique a été gommée. On s'est entièrement focalisé sur les fonctions actives ou passives des fichiers, dont on a voulu faire des objets en soi. Et, sans autre considération que la gratuité et la couverture internationale, on a élu pour cadre le droit d'auteur, alors qu'il est essentiellement indifférent à toute fonction de la forme. Il ne faut pas s'étonner si l'on constate un malaise dans la définition de l'objet du droit* ».

44. Dans le cadre d'APB, l'enjeu juridique de l'appropriation de l'algorithme serait l'obstacle qu'elle pourrait constituer à sa divulgation. En effet, l'article L. 311-4 du Code des Relations entre le Public et l'Administration (modifié par la loi n° 2016-1321 du 7 octobre 2016 pour une République numérique - art. 8 V) dispose que :

et celle de la place de l'algorithme dans cette protection illustrent particulièrement bien cette exigence [45]. Le fait que le choix du régime de protection du logiciel entre le droit des brevets et le droit d'auteur ait résulté de considérations d'opportunité, et pas seulement de la vérification de la conformité du logiciel à des critères juridiques, peut expliquer que le juriste n'a pas toujours tenu compte des concepts informatiques.

6.5 Bibliographie

[AlK-32] Al-Khwarizmi, Muhammad Ibn Musa, **Le calcul indien : versions latines du XII^{e} siècle**, édition de André Allard et traductions françaises, Albert Blanchard, 1992, LXXI + 270 p.

[La meilleure édition. La version arabe est perdue.]

[Con-13] Conway, John H., *On Unsettleable Arithmetical Problems*, **American Mathematical Monthly**, vol. 120, 2013, pp. 192–198.

[Dat-26] Datta, *Early literary evidence of the use of zero in India (Part 1)*, **American Mathematical Monthly**, vol. XXXIII, 1926, p. 449–454.

[Del-98] Delahaye, Jean-Paul, *La conjecture de Syracuse*, **Pour la Science** n^{o} 247, mai 1998.

[Del-99] Delahaye, Jean-Paul, **Jeux mathématiques et mathématiques des jeux**, Belin – Pour la Science, décembre 1999, ISBN 2-84245-010-8, chap. 18 (*La conjecture de Syracuse*), pp. 124–125.

[Fib-02] Fibonacci, **Liber Abaci**, 1202. English translation by Sigler, L. E., **Fibonacci's Liber Abaci**, Springer, 2002, viii + 636 p.

[Gur-84] Gurevich, Yuri, **Reconsidering Turing's Thesis : Toward More Realistic Semantics of Programs**, University of Michigan, Technical Report CRL–TR–38–84, EECS Department, 1984.

[Gur-85] Gurevich, Yuri, *A New Thesis*, **Abstracts, American Mathematical Society**, p. 317, August 1985.

[Gur-88] Gurevich, Yuri, *Kolmogorov machines and related issues*, **Bulletin of EATCS**, vol. 35, 1988, pp. 71–82.

« *les documents administratifs sont communiqués ou publiés sous réserve des droits de propriété littéraire et artistique* ». L'algorithme APB, créé par un salarié d'une école d'ingénieurs de Toulouse, appartiendrait à l'école employeur s'il a été créé dans le cadre de ses fonctions (CPI, art. L. 113-9 ; sinon une cession de droits doit être conclue entre ces deux parties, art. L. 122-7). L'État serait titulaire dérivé des droits après la conclusion d'une cession de droits avec cette école (CPI, art. L. 122-7).

45. « *La difficulté de donner une définition du logiciel vient du fait qu'il n'appartient pas réellement au droit de donner une définition d'une notion qui est de l'ordre de la technique (du logiciel pas plus que du transistor...). C'est ailleurs qu'il faut chercher la réponse, aux informaticiens de caractériser ce qui fait pour eux la spécificité du logiciel* », M. Vivant, Lamy Droit du numérique. V. égal. Ph. Gaudrat, art. préc. : « *le logiciel, par exemple, ne désigne pas une variété d'œuvre, identifiable par ses caractères formels ; il est le terme générique désignant le champ matériel de la réservation légale dans le domaine des outils de traitement numériques, champ d'application délimité par l'énumération d'éléments disparates. Autant dire que l'on n'est pas en présence d'un objet défini, prenant place à l'intérieur de la catégorie œuvre, mais d'une universalité érigée en catégorie nouvelle (construite sur une logique fonctionnelle) à l'intérieur d'une catégorie préexistante dont l'essence et la logique reposent sur la forme* ».

[Knu-66] KNUTH, Donald E., , **Communications of the Association for Computing Machinery**, vol. 9, 1966, p. 654. Réimprimé comme partie du chapitre 0 de **Selected Papers on Computer Science**. Traduction française par Patrick CÉGIELSKI comme chapitre 8 de **Éléments pour une histoire de l'informatique**, 2011, Lecture Notes 190, CSLI Publications, Stanford, Société Mathématique de France, xvi + 371 p., pp. 315–316.

[Knu-72] KNUTH, Donald E., *Ancient Babylonian Algorithms*, **Communications of the Association for Computing Machinery**, vol. 15, 1972, pp. 671–677 ; vol. 19, 1976, p. 108. Réimprimé comme chapitre 11 de **Selected Papers on Computer Science**. Traduction française par Patrick CÉGIELSKI comme chapitre 1 de **Éléments pour une histoire de l'informatique**, 2011, Lecture Notes 190, CSLI Publications, Stanford, Société Mathématique de France, xvi + 371 p., pp. 1–20.

[Knu-73] KNUTH, Donald E., **The art of computer programming, vol.1**, 1968, 2nd ed. 1973, 3rd ed. 1993.

[KU-58] KOLMOGOROV, A. N. et USPENSKII, V. A., *On the definition of an algorithm* (en russe), **Uspekhi Mat. Nauk**, vol. 13, pp. 3–28, 1958. English translation in **AMS translations, 2nd series**, vol. 29, 1963, pp. 217–245.

[Lag-85] LAGARIAS, Jeffrey C., *The 3x + 1 problem and its generalizations*, **American Mathematical Monthly**, vol. 92, 1985, pp. 3–23

[Smi-25] SMITH, David Eugene, **History of Mathematics**, Ginn and Company, 1925. Reed. Dover, 1958, vol. 1 xii + 596 p., vol. 2 xii + 725 p. Version électronique téléchargeable.

[Usp-92] USPENSKY, V. A., *Kolmogorov and mathematical logic*, **The Journal of Symbolic Logic**, vol. 57, 1992, pp. 385–412.