L'infini et l'univers des algorithmes

GILLES DOWEK

On ne peut pas toujours prévoir le nombre d'opérations nécessaires à un calcul. Aussi, pour permettre l'expression de tous les algorithmes, il faut introduire la possibilité que les calculs durent infiniment longtemps.

Un algorithme décompose un calcul en des suites d'opérations élémentaires. Quand un algorithme prescrit un nombre infini d'opérations élémentaires, on n'obtient jamais le résultat. Aussi, faut-il tenter de maîtriser ou de chasser cet infini.

Pour multiplier 35 par 12, nous devons multiplier 5 par 2, poser le 0, retenir le 1, multiplier 3 par 2, ajouter la retenue… au bout de nos efforts, nous obtiendrons le résultat : 420. Cette recette qui décrit l'enchaînement des opérations élémentaires d'une multiplication est dénommée l'algorithme de la multiplication, ou plutôt un algorithme de la multiplication, car il existe d'autres méthodes de calcul possibles. Ainsi, 35 + 35 + 35 + 35 + 35 + 35 + 35 + 35 + 35 + 35 + 35 + 35 est une méthode, un algorithme envisageable pour multiplier 35 par 12, à l'aide d'additions successives. Il en existe bien d'autres.

Ces algorithmes sont des méthodes de calcul qui donnent un résultat après un nombre fini d'opérations et dans un temps fini. L'infini ne semble ainsi jouer aucun rôle dans l'univers des algorithmes, et pourtant la notion d'algorithme ne peut être définie sans utiliser l'infini. L'infini est indispensable, mais il faut le maîtriser.

La définition d'un algorithme

Si nous utilisons des algorithmes depuis l'Antiquité et que le mot algorithme vient du nom du mathématicien Al Khwarizmi qui vécut à Bagdad au IXe siècle, ce n'est qu'au XXe siècle que cette notion a été définie avec précision.

Dans les années 1920, Thoralf Skolem (1887-1963) a proposé une première définition de la notion d'algorithme liée à la récurrence. Ce procédé consiste à définir une fonction en donnant sa valeur en 0 et en indiquant comment passer de sa valeur en n à sa valeur en $n + 1$. Par exemple, la fonction e, qui à chaque nombre entier n associe le nombre 2^n : $e(0) = 1$, $e(1) = 2$, $e(2) = 4$, $e(3) = 8$, $e(4) = 16$, … est définie par récurrence, en indiquant que sa valeur en 0 est 1 et qu'on passe d'une valeur à la suivante en la doublant.

Les fonctions qui sont définies par leur valeur en 0, ou en une autre valeur initiale, et par une opération permettant de passer directement d'une valeur à la suivante sont appelées, selon un terme dû à Rosza Péter, récursives primitives. La fonction $e(n) = 2^n$ est récursive primitive. Elle est définie par $e(0) = 1$ et $e(n + 1) = 2e(n)$. Richard Dedekind, qui avait déjà étudié les définitions par récurrence en 1888, avait montré que l'addition, la multiplication, l'élévation à la puissance étaient des fonctions récursives primitives.

La définition d'une fonction par récurrence donne un algorithme pour le calcul des valeurs de cette fonction. Ainsi, pour calculer la valeur de la fonction e en 10, il suffit de calculer sa valeur en 0, puis en 1, en 2,…, en 9 et enfin en 10. On obtient ainsi $e(0) = 1$, $e(1) = 2$, $e(2) = 4$, …, $e(9) = 512$, et $e(10) = 1\,024$.

On programme aisément de telles fonctions récursives primitives en utilisant des boucles *For*. Chaque itération de la boucle exécute une opération élémentaire ; on peut déterminer le nombre d'itérations nécessaires au calcul en fonction de la valeur que l'on souhaite calculer. Si l'on demande au programme de calculer la fonction e pour n égal à 10, après 10 itérations, 1 024 s'affichera sur l'écran de l'ordinateur.

Avec cette notion de fonction récursive primitive, l'infini n'était pas convoqué, et les mathématiciens pensaient que ces fonctions suffisaient pour définir l'ensemble des algorithmes. Pourtant, en 1928, Wilhelm Ackermann a trouvé une fonction de deux variables m et n pour laquelle il y avait une recette de calcul, mais qui ne répondait pas au souhait de prévisibilité du nombre de pas de calcul. Cette fonction est définie par : $A(0, n) = n + 1$; $A(m + 1, 0) = A(m, 1)$ et $A(m + 1, n + 1) = A(m, A(m + 1, n))$.

Cette fonction n'est pas récursive primitive : le nombre de pas de calcul nécessaire pour passer d'une valeur de m ou de n à la suivante, $m + 1$ ou $n + 1$ n'est pas connu. Cette impossibilité n'est pas due à une mauvaise formulation de la fonction, mais à une impossibilité théorique démontrée par Ackermann : même si la fonction d'Ackermann peut toujours se calculer en un nombre fini d'étapes, le nombre d'itérations nécessaires au calcul est imprévisible. En informatique, le calcul d'une telle fonction ne peut être réalisé en utilisant des boucles *For*.

Les fonctions récursives primitives peuvent toujours se calculer par un algorithme ; en revanche, les fonctions calculables par un algorithme ne sont pas toujours récursives primitives, comme le prouve la fonction d'Ackermann. La définition de Skolem était donc trop restrictive, et le défi lancé aux mathématiciens, au début des années 1930, était de dépasser cette définition pour en proposer une qui englobe l'ensemble des algorithmes.

Le schéma *mu*

Plusieurs solutions ont été proposées par Jacques Herbrand, Kurt Gödel, Alonzo Church, Stephen Kleene, Alan Turing, Emil Post… Dans la définition la plus simple, avancée par Kleene en 1936, on ajoute aux définitions par récurrence un second procédé de calcul, portant le joli nom de schéma *mu*. Ce schéma nous oblige à continuer le calcul jusqu'à ce qu'une condition soit atteinte. Ainsi, à partir d'une fonction g, Kleene définit un nombre a qui est le plus petit nombre x pour lequel $g(x)$ est égal à 0 (condition recherchée ici). Par exemple, si g est la fonction qui à x associe le nombre $|x^2 - 9|$, 3 est le plus petit nombre x tel que $|x^2 - 9|$ soit égal à 0.

On sait calculer une fonction définie par un schéma *mu* au moyen d'un algorithme, c'est-à-dire que l'on peut calculer la valeur de a, bien que ce calcul soit moins simple que celui par récurrence : pour calculer a, nous devons rechercher le plus petit nombre x tel que $|x^2 - 9|$ soit égal à 0, c'est-à-dire calculer successivement $g(0)$, $g(1)$, $g(2)$, $g(3)$, … jusqu'à une valeur d'annulation. Contrairement au calcul par récurrence classique des fonctions récursives primitives, le nombre d'itérations est imprévisible. Nous pouvons trouver la solution après quelques itérations, ou alors tester un grand nombre de valeurs avant de trouver celle qui convient.

Dans les langages de programmation, cette recherche de la solution s'effectue à l'aide des boucles *While*, qui de la même façon que les boucles *For* exécutent une opération plusieurs fois. Cependant, la boucle *For* n'est adaptée qu'aux fonctions récursives primitives, elle contient l'instruction d'arrêter le calcul lorsque le nombre d'itérations souhaité est atteint. La boucle *While* contient l'instruction d'arrêter le calcul lorsque la condition recherchée est atteinte, ici $g(x) = 0$, et ce quel que soit le nombre d'itérations. Le programme recherche la solution en itérant tant qu'elle (*While*) n'est pas trouvée.

Avec le schéma *mu*, on calcule toutes les fonctions où le nombre d'étapes n'est pas connu *a priori* et où l'on découvre au fur et à mesure la structure du calcul. Le schéma *mu* permet d'élaborer progressivement la structure du calcul de la fonction d'Ackermann *(voir l'encadré)*. Cette notion de découverte progressive était absente de la définition de Skolem.

Les fonctions définies avec ces deux procédés – définitions par récurrence et schéma *mu* – sont calculables par algorithme, et l'on a de bonnes raisons de penser que, réciproquement, toutes les fonctions qui peuvent se calculer par un algorithme peuvent se définir ainsi. Le défi lancé aux mathématiciens par l'introduction de la fonction d'Ackermann était ainsi relevé. Les fonctions récursives primitives et le schéma *mu* englobent l'ensemble des algorithmes.

L'arrivée de l'infini

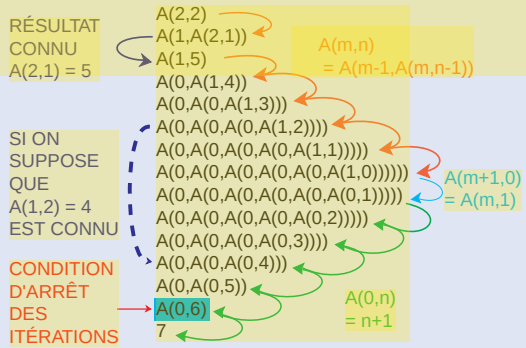
Avec le schéma *mu*, contrairement aux définitions par récurrence, il n'est pas possible de prévoir le nombre d'itérations nécessaires pour trouver le résultat. Il se peut même que l'on ne trouve pas de solution, et, dans ce cas, le calcul se poursuivra éternellement.

Remplaçons dans l'exemple précédent le chiffre 9 par 10, et tentons de calculer le nombre a, qui soit le plus petit nombre x tel que $|x^2 - 10|$ soit égal à 0. Comme aucun nombre ne convient, on essaiera 0, 1, 2, 3, 4, 5… sans jamais trouver la valeur d'annulation. Techniquement, on dit que ce

La fonction d'Ackermann

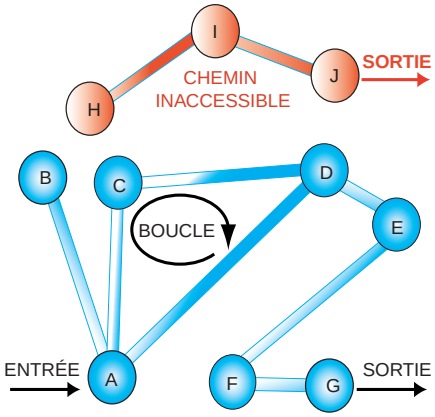
La fonction d'Ackermann est une fonction parfaitement définie et calculable, mais non récursive primitive. Prenons l'exemple du calcul de la fonction A(2,2) en décomposant le calcul. Si cette fonction était récursive primitive, il nous suffirait de partir de la valeur de A(0,0), puis de calculer A(1,0), A(2,0), A(2,1) et enfin A(2,2). Quatre étapes seraient suffisantes. Le problème principal de la fonction d'Ackermann est que les résultats obtenus pour des valeurs de m et de n inférieures à celles que l'on souhaite calculer ne sont pas suffisants. La décomposition du calcul de la fonction A(2,2), nécessite de calculer des fonctions où les valeurs de n sont supérieures à la valeur initiale. Seul un mécanisme de recherche par schéma mu, où la condition d'arrêt des itérations est l'obtention d'une seule valeur de m qui soit égale à 0, donne le résultat.

D'après la définition d'Ackermann, A(2,2) se décompose en A(1, A(2,1)). D'après l'hypothèse, A(2,1) est connu et égal à 5, nous obtenons alors la fonction A(1,5), dont la valeur est inconnue. On décompose cette fonction jusqu'aux A(0, n) dont les valeurs sont égales à $n + 1$. Ainsi A(2,2) est égal à 7.



nombre n'est pas défini. Concrètement, le calcul se poursuit à l'infini. C'est ce que Kleene appelle le mouvement perpétuel. Ce mécanisme de recherche à l'infini offert par le schéma *mu* se retrouve dans les boucles *While* des programmes informatiques ; le programme «boucle» tant que $|x^2 - 10|$ est différent de 0.

Traditionnellement, un algorithme était une recette de calcul qui fournissait toujours un résultat après un nombre fini d'opérations ; avec le schéma *mu*, l'univers des algorithmes s'élargit et, dans certains cas, le calcul se prolonge à l'infini et ne donne jamais de résultat. On dénomme parfois de tels algorithmes, des semi-algorithmes.



LA RECHERCHE DE LA SORTIE dans un labyrinthe peut être infinie. Il faut donc borner la recherche en excluant, par exemple, les boucles telle que ACDA.

Maîtriser l'infini

Le schéma *mu* permet de définir des semi-algorithmes qui recherchent un certain objet dans un ensemble infini, sans même savoir si un tel objet existe. Un programme informatique qui recherche la sortie dans un labyrinthe est une illustration du schéma *mu*. Par exemple, on peut chercher un chemin qui va de A à G dans le labyrinthe de l'illustration ci-contre, et l'on trouvera peut-être le chemin A, D, E, F, G. Si l'on cherche un chemin allant de A à J, on n'en trouvera pas.

Comment procède-t-on pour trouver le chemin qui va de A à G?

Le nombre de chemins dans un labyrinthe est infini : on peut très bien se perdre en parcourant une boucle, en passant de A en D, de D en C et de C en A, sans jamais prendre le couloir qui va de D vers E. Pour trouver la sortie d'un labyrinthe, il ne suffit pas de faire la liste des chemins en allant toujours droit devant soi ; il faut énumérer les chemins de manière systématique afin de n'en oublier aucun, par exemple en prenant tous les chemins de longueur 1 (qui sont en nombre fini), puis tous ceux de longueur 2 (qui sont également en nombre fini), puis ceux de longueur 3… L'énumération de tous les chemins de tailles finies du labyrinthe permet de trouver la sortie, si celle-ci existe et correspond à un chemin de taille finie, mais énumérera les chemins éternellement s'il n'en existe pas. Ce risque de recherche à l'infini, dans le cas où le problème n'a pas de solution, est inévitable dans un ensemble infini.

Dans le cas du labyrinthe, il est possible d'éviter cette recherche dans un ensemble infini. Pour traverser un labyrinthe, il n'est jamais nécessaire de passer deux fois par le même carrefour ; s'il y a un chemin qui comporte une boucle, on la supprime et l'on obtient un autre chemin qui mène directement au même endroit.

On se restreint ainsi à chercher une solution parmi les chemins sans boucle. Ces chemins sont alors en nombre fini, et s'il n'y a pas de solution (comme quand on essaie d'aller de A à J), après avoir énuméré tous les chemins et constaté qu'aucun d'eux ne convient, on saura que les deux points ne sont pas reliés. Qu'il y ait une solution ou non, la recherche arrivera à son terme.

Restreindre ainsi l'espace de recherche de manière à le rendre fini, sans pour autant perdre la solution, est le problème principal quand on cherche à concevoir certains algorithmes, par exemple des algorithmes de sortie de labyrinthe : cela s'appelle borner le schéma *mu*. Parfois – comme dans ce cas – cela est possible, mais dans d'autres cas, en particulier quand le problème qu'on cherche à résoudre est indécidable, cela ne l'est pas. Lorsque le problème est indécidable, on ne peut pas toujours, après avoir effectué un nombre fini d'étapes, prédire si la solution existe ou non. Le problème demande une recherche dans un ensemble infini, et donc une recherche à l'infini quand il n'y a pas de solution.

Dans certaines situations, on ne peut éviter la recherche à l'infini, et le schéma *mu* est le bon outil. Mais dans d'autres, par exemple quand on cherche à définir la fonction d'Ackermann, dont les étapes sont certes imprévisibles, mais sont toujours finies, le schéma *mu* semble un outil mal adapté, car trop puissant, puisqu'il convoque l'infini dans sa recherche. En introduisant le schéma *mu*, Kleene a fait entrer le loup dans la bergerie : l'infini dans la théorie des algorithmes qui était *a priori* le cadre du fini.

Cela est-il dû à une maladresse de Kleene ou cette intrusion de l'infini dans l'univers des algorithmes était-elle nécessaire?

Chasser l'infini?

Church et Kleene ont montré qu'il ne s'agissait pas d'une maladresse et que cette intrusion était inévitable. En effet, tous les procédés de définition d'algorithmes qui se limitent à définir des fonctions qui donnent toujours un résultat après un nombre fini d'opérations (c'est-à-dire qui excluent ce mécanisme de recherche à l'infini) sont incomplets : on pourra toujours construire une fonction, calculable par un algorithme en un nombre fini d'étapes, mais qui n'est pas définissable par ce procédé. Cette fonction se construit en utilisant la méthode de la diagonale introduite par Georg Cantor pour étudier les cardinaux infinis (voir l'article de Jean-Paul Delahaye, *L'infini est-il paradoxal en mathématiques?*, page 30).

Il n'y a donc pas de définition qui englobe toutes les fonctions donnant un résultat après un nombre fini d'étapes et rien qu'elles. Certains procédés, comme le schéma *mu*, définissent toutes ces fonctions, mais aussi certaines fonctions qui ne donnent pas de résultat après un nombre fini d'étapes. D'autres procédés comme ceux – plus complexes – proposés par Kurt Gödel en 1941 ou Jean-Yves Girard en 1970 ne permettent de définir que des fonctions qui donnent un résultat en un nombre fini d'étapes. Cependant, ils ne les définissent pas toutes.

Chassez l'infini, il reviendra au galop...

Gilles DOWEK est chargé de recherches à l'Institut national de recherche en informatique et en automatique.

Jean-Luc CHABERT et al., *Histoire des algorithmes*, Belin, 1995.

Richard LASSAIGNE et Michel DE ROUGEMONT, *Logique et fondement de l'informatique*, Hermès, 1993.

René CORI, *Logique mathématique*, Masson, 1993.