Est-ce que tous les problèmes ont une résolution par programme ?

Pierre Lescanne

13 septembre 2001

Quoi de plus naturel pour un scientifique que d'espérer une réponse par calcul à une question qu'il se pose ? Nous entendons par là une réponse obtenue mécaniquement en temps fini à l'issue d'un processus calculatoire rigoureux. L'homme de la rue dirait simplement que la réponse est «donnée par un ordinateur». Nous allons montrer que ce fol espoir de résoudre «in silico» tous les problèmes qui nous paraissent un peu fastidieux n'est pas fondé. Oui, il y a des problèmes qui résistent à l'ordinateur. Certains peuvent être programmés, mais ensuite requièrent tant de puissance de calculs en temps (des millions de siècles peut-être !) et en matériel (un ordinateur plus grand que dix univers !) que l'on peut les considérer comme infaisables. D'autres sont rebelles à toute programmation. C'est à cette deuxième catégorie que nous allons nous intéresser.

Avant d'envisager des problèmes très généraux nous allons nous intéresser à des problèmes dont la réponse est soit oui, soit non. On les appelle des *problèmes de décision*.

Quelques problèmes issus des mathématiques

Le domaine des mathématiques offre une foule de tels problèmes qui ont l'avantage de pouvoir être formalisés. Ils sont de natures très variées. En voici une petite liste pour se donner une idée.

- Le graphe G [1] est-il *planaire* ? Autrement dit peut-on le dessiner sur un plan sans qu'aucunes de ses arêtes ne se croisent ?
- Dans le graphe G, y a-t-il un chemin qui passe par tous ses sommets une seule fois ? On appelle cela le *problème du voyageur de commerce* et l'on dit que le graphe est *hamiltonien*.
- Supposons que je me donne des règles d'équivalence pour des expressions. Est-ce que je peux dire que deux expressions quelconques e_1 et e_2 sont équivalentes en me servant seulement de mes règles de base ? Par exemple si mes règles de bases contiennent $sin^2(x) + cos^2(x) = 1$ et $1 + 1 = 2$, je peux prouver que $1 + sin^2(x) + cos^2(x) = 2$ et si les règles se limitent aux deux règles ci-dessus, on peut prouver que $2 + sin^2(x) \neq 3$. Mais dans le cas général de règles sur l'addition, la multiplication et l'exponentiation

[1] Un *graphe* est une figure qui comportent des points appelés «sommets» reliés par des lignes appelées «arêtes».

d'expressions mélangées à des formules trigonométriques, le problème est nettement plus difficile.
- Étant donné un système d'axiomes desquels je sais déduire des théorèmes en me servant de règles de déduction, puis-je affirmer qu'une proposition donnée est un théorème ou au contraire puis-je affirmer qu'elle n'est pas un théorème ?
- L'informaticien pourra se poser la question de savoir si un programme donné se termine sur telle donnée qu'on lui fournit ?

Les problèmes de correspondance

Intéressons-nous pour un temps aux problèmes d'équivalence. Pour simplifier les choses, examinons un problème très particulier et très simple appelé *problème de correspondance*. Son étude est due à Emil Post l'un des fondateurs de la théorie de la calculabilité. Dans un problème de correspondance, on se donne deux listes de mots que l'on essaie de mettre en «vers holorimes». Rappelons que deux vers *holorimes* sont deux vers qui riment entièrement, le second reproduisant phonétiquement le premier. On dit aussi parfois tout simplement une holorime. Quoique le rapprochement de son avec celui d'Emil Post puisse surprendre, Alphonse Allais a certainement été le grand maître de l'holorime française et nous allons nous inspirer de lui pour commencer par la réflexion d'un musicien à son ami à propos de l'art :

L'art efface. Olé l'ami !
La Ré Fa Sol et La Mi.

Replaçons nous au niveau strictement phonétique ; les vers ci-dessus peuvent être issus du problème de correspondance suivant :

```
(ar, ami, é, fas, l, olé)
(ré, mi, fa, sol, la, é)
```

Pour que les résultats de leur concaténation correspondent : le premier mot du premier vers est pris dans la première liste à un certain rang et le premier mot du second vers est pris dans la deuxième liste au même rang. Dans notre cas, le premier mot du premier vers est le cinquième mot de la première liste soit `l`, ce qui signifie que le premier mot du deuxième vers doit être `la`. Ensuite le deuxième mot est pris au premier rang ce qui donne `lar` et `laré`, puis le mot suivant au troisième rang ce qui donne `laré` et `laréfa`, puis au quatrième rang (`laréfas` et `laréfasol`) puis au sixième rang (`laréfasolé` et `laréfasolé`). On a une première correspondance, mais on peut continuer avec `laréfasolél` et `laréfasoléla` (cinquième rang) et `laréfasolélami` (deuxième rang). Les deux correspondances que nous obtenons proviennent des listes de rang *(5, 1, 3, 4, 6)* et *(5, 1, 3, 4, 6, 5, 2)*. Il y a aussi une correspondance beaucoup plus simple !

Illustrons maintenant cela par une construction inspirée de Victor Hugo. Considérons les deux listes :

```
(ala, aman, dela, gal, magnanime, rène, tour)
(tour, dela, rène, galaman, anime, ala, magn)
```

Il y a une holorime qui est `galamandelarènealatourmagnanime` dont la suite des rangs associée est *(4, 2, 3, 6, 1, 7, 5)* et qui nous rappelle les vers bien connus.

Gal, amant de la reine, alla, tour magnanime
Galamment de l'arène à la tour Magne à Nîmes.

Hélas, quand on formalise le problème on perd en poésie ce que l'on gagne en généralité. Les lettres des mots deviennent des chiffres ; en prenant 0 et 1 cela suffit dans la suite de notre présentation. Considérons donc des listes constituées de suites de 0 et de 1. On obtient des problèmes de correspondance dont voici quelques uns. Le lecteur est invité à essayer de les résoudre avant de passer à la suite de l'article.

Problème 1

```
010 00  101
101 000 01
```

Problème 2

```
101  11   011
10   011  101
```

Problème 3

```
011  1    0    10
1    0    011  001
```

Problème 4

```
000  0    11  10
0    111  0   100
```

Le problème 1 a pour holorime *(2, 1, 3)*. Le problème 2 n'a pas d'holorime. On le voit en cherchant la solution à partir de la fin. Une holorime doit se terminer par `11`, car seuls les deuxièmes mots du problème qui sont `11` et `011` se terminent par le même sous-mot à savoir `11`. Mais alors le vers se termine par `011`, il faut donc chercher un avant-dernier mot dans la première suite qui se termine par 0 et il n'y en a pas. Le problème 3 a une holorime de taille 11

```
0110001011011011110011
```

qui correspond à la suite de rangs *(3, 2, 4, 3, 3, 2, 1, 1, 1, 4, 1)*. Le problème 4 a une holorime de longueur 204. Si vous aimez ces problèmes, l'équipe *PCP at home* de l'université de Leipzig vous invite à vous attaquer à des problèmes vraiment difficiles et ouverts.

Comme le problème 4 le suggère, il faut s'aider d'un ordinateur et d'un programme spécifique pour trouver des holorimes très longues. C'est ce qu'a fait Richard J. Lorentz quand il a trouvé sa solution au problème 4. Restreignons-nous au problème de décision qui consiste à se poser la question de l'existence d'une holorime. Si l'on cherche une réponse par programme, il faut concevoir un programme qui prend comme donnée un problème de correspondance et décide si `oui` ou `non` il a une solution holorime. Quelle devra être la stratégie de ce programme ? Devra-t-il commencer par le début et/ou par la fin et pourquoi pas par le milieu ? Comment et quand devra-t-il s'arrêter de chercher des solutions ? En fait la recherche d'un tel programme est sans espoir puisqu'Emil Post a démontré en 1947 qu'un tel programme ne peut pas exister. On dit que le problème de correspondance est *indécidable* ou *incalculable.*

Décidabilité ou calculabilité

Réciproquement, un problème est *décidable* s'il existe un algorithme pour résoudre le problème de décision, autrement dit s'il existe un algorithme qui réponde oui quand le problème a une réponse positive et non quand le problème a une réponse négative. On dit aussi que le problème de décision est *calculable.* Notons bien qu'une réponse évasive du type «je ne sais pas» n'est pas admise dans un problème de décision, alors qu'on pourrait parfaitement l'admettre de certains programmes d'ordinateur sous certaines conditions. Ainsi si je demande une information à un moteur de recherche, j'admets parfaitement qu'il me dise «je n'ai rien trouvé» alors qu'en fait cette information existe quelque part bien cachée sur le Web.

La terminologie indécidable *peut être ambiguë. En effet, on appelle parfois indécidable, une proposition dans une théorie qui ne peut ni être prouvée ni être réfutée. La terminologie actuelle tend à qualifier d'*indépendante *une telle proposition.*
Plus précisément, étant donné un système d'axiomes, une proposition p *est indépendante du système d'axiomes si on ne peut ni prouver* p*, ni prouver* non p *dans ce système.*
*Ainsi, en géométrie élémentaire, l'*axiome des parallèles*, qui s'énonce «Étant donné un point pris hors d'une droite, il existe une unique parallèle à cette droite qui passe par ce point» est indépendant des autres axiomes de la géométrie.*
Pour ne créer aucune autre confusion, une certaine terminologie tend à éviter l'emploi de problème décidable *en théorie de la calculabilité et à lui préférer celui de* problème de décision calculable.

Fig. 1 – Indépendance

Qu'est-ce qu'un calcul ?

Pour prouver qu'aucun programme ou plus exactement qu'aucun algorithme ne peut exister pour un problème donné de décision, il faut caractériser avec précision ce qu'on entend par calcul, il faut en fait donner une définition mathématique de ce qu'est un algorithme en partant de la définition assez intuitive et consensuelle que nous en avons. C'est ce travail de formalisation qu'ont entrepris indépendemment en 1936 Emil Post et Alan Turing. Eux et d'autres à leur suite ont donné une base mathématique précise de ce qu'est un calcul. Par exemple, pour Alan Turing un calcul est réalisé par une machine abstraite idéale, c'est-à-dire par un objet mathématique que l'on caractérise formellement. Une machine de Turing comporte des configurations identifiés par la valeur de leurs deux composantes : l'*état* (un ensemble finie de valeurs possibles) et la *bande* (une liste arbitrairement grande de symboles ou caractères que l'on peut lire et écrire). La machine comporte des transitions qui permettent de passer d'une configuration à une autre. Ce qui est intéressant sur les machines de Turing c'est que leur structure complètement mathématique permet de démontrer des théorèmes à leur propos. Pour qu'une machine de Turing résolve un problème, il faut auparavant coder celui-ci ; cette opération nous est très familière, car nos ordinateurs n'acceptent pas les données telles quelles, mais seulement quand elles ont été codées pour eux. Les problèmes de décision sont alors résolus par les machines de Turing qui s'arrêtent sur toute donnée à condition que l'on s'entende sur un codage du oui et du non.

Les machines de Turing

Nous avons présenté sommairement les machines de Turing et nous avons affirmé qu'elles avaient une présentation rigoureuse. Nous allons maintenant essayer d'en donner une idée plus précise. Nous savons déjà qu'il faut considérer un ensemble d'états que nous appelons Etat et une fonction de transition Trans. Si nous comparons la machine de Turing à un ordinateur, nous pouvons comprendre l'Etat comme une mémoire RAM et la fonction de transition comme le langage d'assembleur de l'ordinateur, la bande est moins réaliste puisqu'elle représente une mémoire inscriptible (on peut y lire et y écrire) potentiellement infinie, mais nous avons besoin de cette infinitude si nous voulons pouvoir exécuter n'importe quel type de calcul. Pour nous représenter cette bande, nous pouvons nous imaginer que l'opérateur de la machine de Turing a à sa disposition un lecteur-graveur de CD et autant qu'il veut de CD où stocker des informations. Nous ajoutons à cela l'ensemble Alphabet des caractères que l'on peut lire et écrire sur la bande. Enfin il va bien falloir savoir dans quel état commencer et quand on peut considérer que l'on a fini de calculer. Pour cela on définit un état particulier commencement dit *état initial* et un ensemble Final d'*états finals*. Muni de tous ces constituants, nous pouvons déclarer qu'une machine de Turing est un quintuplet {Etat, Alphabet, Trans, commencement, Final} soumis à quelques conditions : Etat est un ensemble fini, commencement appartient à Etat et Final est un sous-ensemble d'Etat. Nous supposerons que l'Alphabet contient un symbole particulier ␣ que nous appellerons une *espace*[2] et un symbole particulier $\triangleright$ pour noter le début de la chaîne écrite sur la bande. À chaque étape la machine de Turing est dans un état donné et a sa tête de lecture-écriture en face d'un emplacement donné de la bande où se trouve un caractère

La définition de la transition est certainement le corps de la définition d'une machine de Turing. Que peut faire une machine de Turing quand elle est dans un état donné qui n'est pas un état final ? Elle peut en fait faire deux choses : soit lire et écrire un caractère sur la bande, soit se déplacer sur la bande à droite ou à gauche. Si l'état est q et si la tête de lecture-écriture est en face du caractère a, on peut soit écrire un caractère, disons b et se mettre dans l'état q', soit se déplacer vers la droite $\rightarrow$ soit se déplacer vers la gauche $\leftarrow$ et se mettre dans l'état q''. Dans le premier cas la valeur $\mathsf{Tans}(q,a)$ de la fonction de transition est (q',b), tandis que les deux autres cas elle est respectivement $(q'',\rightarrow)$ et $(q'',\leftarrow)$. On impose aussi d'autres propriétés qui expriment que la machine se comporte correctement ; ainsi quand elle

[2]Rappelons qu'espace est féminin en typographie.

lit le caractère de début $\triangleright$, elle se déplace vers la droite ; elle s'interdit d'écrire un caractère de début. On initialise une machine de Turing en écrivant une suite finie de caractères sur la bande à droite du caractère $\triangleright$ qui y figure déjà. On suppose que le reste de la bande est rempli de caractères $\sqcup$. A l'initialisation la machine est dans l'état initial.

La machine effaceuse

Définissons une machine de Turing $M = \{\mathsf{Etat}_M, \mathsf{Alphabet}_M, \delta, \mathsf{commencement}_M, \{h\}\}$ par

$$\begin{array}{rcl} \mathsf{Etat}_M & = & \{q_0, q_1, h\} \\ \mathsf{Alphabet}_M & = & \{a, \sqcup, \triangleright\} \\ \mathsf{commencement}_M & = & q_0 \end{array}$$

la transition δ est donnée par le tableau :

q	η	$\delta(q, \eta)$
q_0	a	$(q_1, \sqcup)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	(q_0, a)
q_1	$\sqcup$	$(q_0, \rightarrow)$
q_1	$\triangleright$	$(q_1, \rightarrow)$

La machine de Turing M lit chaque caractère a à partir de la position initiale de sa tête de lecture-écriture, puis le remplace par une espace avant de se déplacer vers la droite. En effet, au commencement, la machine est dans l'état q_0 et se trouve en face d'un caractère, si c'est le caractère $\triangleright$ elle se déplace vers la droite en restant dans l'état q_0. Plus tard, chaque fois que la machine se trouvera dans l'état q_0 elle renouvellera la même suite de transitions ; si la tête est en face de $\sqcup$, elle se met dans l'état h puis s'arrête, si la tête est en face de a, elle remplace cet a par $\sqcup$, puis passe dans l'état q_1, puis revient dans l'état q_0. On voit que la machine effectue bien la suite d'effacements décrite. On voit aussi que la machine ne parviendra jamais dans l'état q_1 avec sa tête de lecture-écriture en face du caractère $\triangleright$ et ne parviendra jamais dans l'état q_0 avec sa tête de lecture-écriture en face du caractère a. M n'effectuera donc jamais les transitions correspondant au quatrième et sixième lignes du tableau. Dès qu'elle rencontre une espace sur la bande la machine s'arrête. On a donc décrit une machine *effaceuse*. On constate aisément que la tête de lecture de M ne se déplace que vers la droite et cela régulièrement, on peut donc démontrer que M s'arrête sur toute donnée. On remarque aussi que toutes les configurations de la machine M peuvent être décrite comme un objet mathématique sur lequel on peut faire toute sorte de raisonnement dont des raisonnements par récurrence.

A condition de se donner une convention pour coder les entiers, on peut définir des machines de Turing pour faire des additions, des multiplications et toutes les opérations de base sur les entiers. Ce qui est plus original c'est qu'on peut définir une machine de Turing universelle, c'est-à-dire une machine de Turing qui prend en entrée une machine de Turing (il serait plus exact de dire son codage) et une donnée et rend le résultat du calcul de cette machine sur la donnée. Cette machine universelle est l'image de nos ordinateurs (la mémoire infinie en moins) qui peuvent exécuter un peu n'importe quel programme à condition que l'on leur code convenablement.

Les machines à registres *forment un autre modèle de calcul dont on peut montrer l'équivalence avec celui des machines de Turing.*
*La mémoire d'une telle machine consiste d'un nombre fini de registres. Chaque registre contient un nombre naturel arbitrairement grand. Le comportement de la machine est spécifié par un programme consistant d'un nombre fini d'*instructions *étiquetées. Les instructions sont de quatre types.*

Addition $L_j : R_i \longleftarrow R_i + 1$

Soustraction $L_j : R_i \longleftarrow R_i - 1$

Branchement L_j : ***si*** $R_j = 0$ ***alors aller à*** L_k

Arrêt L_j : ***stop***

*Une instruction d'**Addition** accroît le registre de 1, tandis qu'une instruction de **Soustraction** décroît le registre de* 1*. Un **Branchement** interrompt l'ordre naturel des instructions pour le continuer à l'instruction étiqueté par* L_k *et un **Arrêt** interrompt définitivement le calcul. Le programme est écrit de telle façon que les **Soustractions** ne soient exécutées que quand le registre concerné ne contient pas* 0*.*
On peut aussi définir un autre modèle de calcul lui aussi équivalent aux deux autres, celui des fonctions récursives *qui consiste grosso modo à définir des fonctions sur les naturels à partir de trois schémas de récurrence.*

FIG. 2 – Machine à registres

La thèse de Church-Turing

Alan Turing, Emil Post et les autres ont défini des modèles de calcul mais qu'est-ce qui nous prouve que ces modèles capturent bien le concept de calcul dans sa forme la plus générale telle que nous la connaissons ? En fait, on a pu montrer que toutes les définitions d'algorithmes, c'est-à-dire tous les modèles de calcul qui ont été donnés sont équivalents : ce qui est un algorithme pour un modèle peut être codé en un algorithme pour un autre et réciproquement. Alonzo Church et Alan Turing ont énoncé la proposition suivante que l'on appelle la *thèse de Church-Turing* : «Les machines de Turing qui s'arrêtent pour toute donnée capturent la notion d'algorithme». Il ne s'agit pas d'un théorème car on met côte à côte un concept formel et une notion intuitive. D'autre part, aucun modèle n'est privilégié plutôt qu'un autre puisque tous les modèles de calcul sont équivalents. La thèse de Church-Turing affirme que quoi qu'on fasse on ne trouvera pas de modèle de calcul qui outrepasse ceux que nous connaissons actuellement. De fait soixante ans de développement d'informatique n'ont pas réussi à remettre en cause cette proposition.

L'incalculabilité de l'arrêt

Pour montrer qu'un problème P est incalculable il suffit de montrer que l'affirmation de la calculabilité de ce problème conduit à une contradiction. La plupart du temps, on utilise une *réduction*, c'est-à-dire que l'on part d'un problème de décision Q connu comme incalculable et l'on montre que si l'on sait décider (calculer) le problème P alors l'on sait décider (calculer) le problème Q, ce qui est contradictoire. Mais pour démarrer ce processus il faut exhiber un problème incalculable de départ. Celui qui est à la source de tout est *le problème de l'arrêt des machines de Turing* ; il s'agit du problème de décider si étant donné une machine de Turing et une donnée le calcul se termine pour cette donnée. Par définition, si l'arrêt est calculable alors il existe une machine A qui prend en entrée le codage d'une machine et le codage d'une donnée et rend en sortie oui si la machine s'arrête sur cette donnée et non si la machine ne s'arrête pas sur cette donnée. On construit alors une machine B comme suit : B prend en entrée un codage d'une machine de Turing M et un codage d'une donnée et s'arrête si la machine M ne s'arrête pas sur cette donnée, et B boucle (ne s'arrête pas) si M s'arrête sur cette donnée. B fait appel à A dans sa construction, ainsi qu'à une machine qui ne s'arrête jamais ; on peut, en effet, montrer que l'on sait inclure le code d'une machine déjà connue dans celui d'une autre, ce sont les équivalents des sous-programmes ou des modules en programmation. Que se passe-t-il si on applique B au codage de B et d'une donnée ? B s'arrête si B ne s'arrête pas sur cette donnée et B ne s'arrête pas sur cette donnée si B s'arrête sur cette donnée. On a donc une contradiction. Donc il n'y a pas de machine A qui décide de l'arrêt. Ce procédé est dit «diagonal» . C'est, en fait, un procédé assez courant en mathématique pour conduire à une contradiction.

Reprenons nos problèmes du début. Le deux problèmes de graphes (planarité et chemin du voyageur de commerce) sont décidables, c'est-à-dire incalculables. Le problème de l'équivalence d'expressions est en général indécidable (incalculable). De même est incalculable le problème de savoir si une proposition est un théorème pour un système d'axiomes. Le problème de l'arrêt est aussi incalculable comme nous l'avons vu.

Les ensembles diophantiens

Un problème de décision définit clairement un ensemble, à savoir l'ensemble des données pour lesquelles l'algorithme répond oui. Un tel ensemble est dit *calculablement énumérable*. L'ensemble des données pour lesquelles l'algorithme répond non est aussi parfaitement défini et si le problème de décision est calculable c'est le complémentaire du précédent. Un ensemble qui peut-être défini en même temps que son complémentaire par un algorithme associé à un problème de décision est appelé un *ensemble calculable*. Tout ensemble calculablement énumérable n'est pas forcément calculable, il faudrait pour cela que l'on puisse prouver l'arrêt de l'algorithme, pour pouvoir garantir que pour toute donnée la réponse est soit oui, soit non, ce qui revient à dire que le problème de décision associé est calculable.

Examinons maintenant une catégorie d'ensemble sur les p-uplets d'entiers relatifs qui n'a a priori rien à voir et que nous appelons les ensembles *diophantiens*, du nom de Diophante d'Alexandrie 325-409 qui étudia le premier ce type de problèmes. Tout d'abord définissons ce que nous appelons une équation diophantienne, c'est une équation à p variables $X_1 ... X_p$

de la forme
$$c_1 X_1^{n_1} X_2^{n_2} ... X_p^{n_p} + ... + c_n X_1^{n_1} X_2^{n_2} ... X_p^{n_p} = 0.$$
Par exemple,
$$2X_1^2 X_2 - X_1 X_2^2 + X_2 X_3 = 0$$
mais aussi pour tout entier k
$$X_1^k + X_2^k - X_3^k = 0$$
sont des équations diophantiennes à trois variables. Un ensemble est diophantien s'il est l'ensemble des solutions d'une équation diophantienne ; un ensemble diophantien est donc l'ensemble des p-uplets $(a_1, ..., a_p)$ qui annulent une équation diophantienne, c'est-à-dire tels que
$$c_1 a_1^{n_1} a_2^{n_2} ... a_p^{n_p} + ... + c_n a_1^{n_1} a_2^{n_2} ... a_p^{n_p} = 0.$$
Quel rapport ont les ensembles diophantiens avec les ensembles calculablement énumérables ? En fait, il s'agit des mêmes ensembles comme l'a montré Youri Matijasévitch. Il réglait du même coup une très vieille question connue comme le *dixième problème de Hilbert*. En effet, dans une conférence désormais célèbre donnée au deuxième congrès international des mathématiciens en 1900, David Hilbert avait lancé 23 défis qu'il considérait comme des témoins des grandes questions ouvertes de son temps en mathématiques. Le dixième problème s'énonce ainsi :

> *Étant donnée une équation diophantienne avec un nombre quelconque de quantités inconnues et avec des coefficients entiers rationnels : trouver un procédé par lequel on peut déterminer par un nombre fini d'opérations si l'équation est soluble sur les entiers rationnels.*

Il a fallu attendre soixante dix ans pour que Youri Matijasévitch lui apporte une solution négative en montrant l'équivalence entre les calculablement énumérables et les diophantiens. En effet, on prouve facilement que le résultat de Youri Matijasévitch offre un procédé mécanique d'énumération des solutions, mais on ne peut pas déterminer par un procédé mécanique universel si une équation diophantienne quelconque a ou n'a pas de solution. En résolvant le dixième problème de Hilbert, Youri Matijasévitch a non seulement scellé un problème ouvert, il a aussi et surtout donné une nouvelle et très élégante caractérisation de la calculabilité en reliant de façon surprenante la théorie des nombres et celle des algorithmes.

Les problèmes infaisables

La division que nous venons d'opérer entre problèmes calculables et incalculables est-elle suffisante pour les informaticiens ? Oui si l'on considère qu'elle exclut d'emblée des problèmes sans solutions algorithmiques, mais elle est insuffisante, car étant donné un problème il faut encore savoir si l'on peut trouver sa solution en un temps raisonnable. Or il existe des problèmes calculables dont la solution ne peut-être trouvée que dans un temps rédhibitoire (plus que l'âge de l'univers par exemple). Ainsi alors que le problème de la planarité d'un graphe est «facile», le problème du voyageur de commerce est extrêmement complexe au point qu'on peut le considérer comme infaisable. Une classification des problèmes calculables s'impose, mais ceci est un autre affaire, c'est celle de la *complexité*.