Discussion préalable. On s'intéresse ici à la question de savoir ce qu'un **algorithme** peut ou ne peut pas faire. Pour procéder de façon rigoureuse, il faudrait formaliser la notion d'algorithme (par exemple à travers le concept de machine de Turing) : on a préféré rester informel sur cette définition — par exemple « un algorithme est une série d'instruction précises indiquant des calculs à effectuer étape par étape et qui ne manipulent, à tout moment, que des données finies » ou « un algorithme est quelque chose qu'on pourrait, en principe, implémenter sur un ordinateur » — étant entendu que cette notion est déjà bien connue et comprise, au moins dans la pratique. Les démonstrations du fait que tel ou tel problème est décidable par un algorithme ou que telle ou telle fonction est calculable par un algorithme deviennent beaucoup moins lisibles quand on les formalise avec une définition rigoureuse d'algorithme (notamment, programmer une machine de Turing est encore plus fastidieux que programmer en assembleur un ordinateur, donc s'il s'agit d'exhiber un algorithme, c'est probablement une mauvaise idée de l'écrire sous forme de machine de Turing).

Néanmoins, il est essentiel de savoir que ces formalisations existent : on peut par exemple évoquer le paradigme du λ-calcul de Church (la première formalisation rigoureuse de la calculabilité), les fonctions générales récursives (=μ-récursives) à la Herbrand-Gödel-Kleene, les machines de Turing (des machines à états finis capables de lire, d'écrire et de se déplacer sur un ruban infini contenant des symboles d'un alphabet fini dont à chaque instant tous sauf un nombre fini sont des blancs), les machines à registres, le langage « FlooP » de Hofstadter, etc. Toutes ces formalisations sont équivalentes (au sens où, par exemple, elles conduisent à la même notion de fonction calculable ou calculable partielle, définie ci-dessous). La **thèse de Church-Turing** affirme, au moins informellement, que tout ce qui est effectivement calculable par un algorithme [1] est calculable par n'importe laquelle de ces notions formelles d'algorithmes, qu'on peut rassembler sous le nom commun de **calculabilité au sens de Church-Turing**, ou « calculabilité » tout court.

Notamment, quasiment tous les langages de programmation informatique [2], au moins si on ignore les limites des implémentations et qu'on les suppose capables de manipuler des entiers, chaînes de caractère, tableaux, etc., de taille arbitraire (mais toujours finie) [3], sont « Turing-complets », c'est-à-dire équivalents dans leur pouvoir de calcul à la calculabilité de Church-Turing. Pour imaginer intuitivement la calculabilité, on peut donc choisir le langage qu'on préfère et imaginer qu'on programme dedans. Essentiellement, pour qu'un langage soit Turing-complet, il lui suffit d'être capable de manipuler des entiers de taille arbitraire, de les comparer et de calculer les opérations arithmétiques dessus, et d'effectuer des tests et des boucles.

1. Voire, dans certaines variantes, physiquement calculable dans notre Univers.

2. C, C++, Java, Python, JavaScript, Lisp, OCaml, Haskell, Prolog, etc. Certains langages se sont même révélés Turing-complets alors que ce n'était peut-être pas voulu : par exemple, HTML+CSS.

3. Autre condition : ne pas utiliser de générateur aléatoire matériel.

Il faut souligner qu'on s'intéresse uniquement à la question de savoir ce qu'un algorithme peut ou ne peut pas faire (calculabilité), pas au temps ou aux autres ressources qu'il peut prendre pour le faire (complexité), et on ne cherche donc pas à rendre les algorithmes efficaces en quelque sens que ce soit. Par exemple, pour arguër qu'il existe un algorithme qui décide si un entier naturel n est premier ou non, il suffit de dire qu'on peut calculer tous les produits pq avec $2 \leq p, q \leq n-1$ et tester si l'un d'eux est égal à n, peu importe que cet algorithme soit absurdement inefficace. De même, nos algorithmes sont capables de manipuler des entiers arbitrairement grands : ceci permet de dire, par exemple, que toute chaîne binaire peut être considérée comme un entier, peu importe le fait que cet entier ait peut-être des milliards de chiffres (dans les langages informatiques réels, on a rarement envie de considérer toute donnée comme un entier, mais en calculabilité on peut se permettre de le faire).

Notamment, plutôt que de considérer des « mots » (éléments de Σ^* avec Σ un alphabet fini) et « langages » (parties de Σ^*), il sera plus pratique de remplacer l'ensemble Σ^* des mots par l'ensemble des entiers naturels, quitte à choisir un codage (calculable !) des mots par des entiers. (À titre d'exemple, on obtient une bijection de l'ensemble $\{0,1\}^*$ des mots sur l'alphabet à deux lettres avec $\mathbb{N}$ de la façon suivante : ajouter un 1 au début du mot, lire celui-ci comme un nombre binaire, et soustraire 1. Plus généralement, une fois choisi un ordre total sur l'alphabet fini Σ, on peut trier les mots par ordre de taille, et, à taille donnée, par ordre lexicographique, et leur associer les entiers naturels dans le même ordre : il n'est pas difficile de montrer que cela donne bien une bijection calculable entre Σ^* et $\mathbb{N}$.)

Terminaison des algorithmes. Un algorithme qui effectue un calcul utile doit certainement terminer en temps fini. Néanmoins, même si on voudrait ne s'intéresser qu'à ceux-ci, il n'est pas possible d'ignorer le « problème » des algorithmes qui ne terminent jamais (et ne fournissent donc aucun résultat). C'est le point central de la calculabilité (et du théorème de Turing ci-dessous) qu'on ne peut pas se débarrasser des algorithmes qui ne terminent pas : on ne peut pas, par exemple, formaliser une notion suffisante[4] de calculabilité dans laquelle tout algorithme termine toujours ; ni développer un langage de programmation suffisamment général dans lequel il est impossible qu'un programme « plante » indéfiniment ou parte en boucle infinie. (Cette subtilité est d'ailleurs sans doute en partie responsable de la difficulté historique à dégager la bonne notion d'« algorithme » : on a commencé par développer des notions d'algorithmes terminant forcément, comme les fonctions primitives récursives, et on se rendait bien compte que ces notions étaient forcément toujours incomplètes.)

4. Tout dépend, évidemment, de ce qu'on appelle « suffisant » : il existe bien des notions de calculabilité, plus faibles que celle de Church-Turing, où tout calcul termine, voir par exemple la notion de fonction « primitive récursive » ou le langage « BlooP » de Hofstadter ; mais de telles notions ne peuvent pas disposer d'une machine universelle comme expliqué plus loin (en raison d'un argument diagonal), donc elles sont nécessairement incomplètes en un certain sens.

Définition. On dit qu'une fonction $f : \mathbb{N} \to \mathbb{N}$ est **calculable** (ou « récursive ») lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine toujours en temps fini, et calcule (renvoie) $f(n)$.

On dit qu'un ensemble $A \subseteq \mathbb{N}$ (« langage ») est **décidable** (ou « calculable » ou « récursif ») lorsque sa fonction indicatrice $\mathbf{1}_A : \mathbb{N} \to \mathbb{N}$ (valant 1 sur A et 0 sur son complémentaire) est calculable. Autrement dit : lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine toujours en temps fini, et renvoie « oui » (1) si $n \in A$, « non » (0) si $n \notin A$ (on dira que l'algorithme « décide » A).

On dit qu'une fonction partielle $f : \mathbb{N} \dashrightarrow \mathbb{N}$ (c'est-à-dire une fonction définie sur une partie de $\mathbb{N}$, appelé ensemble de définition de f) est **calculable partielle** (ou « récursive partielle ») lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine en temps fini ssi $f(n)$ est définie, et dans ce cas calcule (renvoie) $f(n)$. (Une fonction calculable est donc simplement une fonction calculable partielle qui est toujours définie : on dira parfois « calculable totale » pour souligner ce fait.)

On dit qu'un ensemble $A \subseteq \mathbb{N}$ est **semi-décidable** (ou « semi-calculable » ou « semi-récursif ») lorsque la fonction partielle $\mathbb{N} \dashrightarrow \mathbb{N}$ définie exactement sur A et y valant 1, est calculable partielle. Autrement dit : lorsqu'il existe un algorithme qui prend en entrée $n \in \mathbb{N}$, termine en temps fini ssi $n \in A$, et renvoie « oui » (1) dans ce cas [5] (on dira que l'algorithme « semi-décide » A).

On s'est limité ici à des fonctions d'une seule variable (entière), mais il n'y a pas de difficulté à étendre ces notions à plusieurs variables, et de parler de fonction calculable $\mathbb{N}^k \to \mathbb{N}$ (voire $\mathbb{N}^* \to \mathbb{N}$ avec $\mathbb{N}^*$ l'ensemble des suites finies d'entiers naturels) ou de fonction calculable partielle de même type : de toute manière, on peut « coder » un couple d'entiers naturels comme un seul entier naturel (par exemple par $(m, n) \mapsto 2^m(2n + 1)$, qui définit une bijection calculable $\mathbb{N}^2 \to \mathbb{N}$), ou bien sûr un nombre fini quelconque (même variable), ce qui permet de faire « comme si » on avait toujours affaire à un seul paramètre entier.

Complément : Comme on n'a pas défini formellement la notion d'algorithme, il peut être utile de signaler explicitement les faits suivants (qui devraient être évidents sur toute notion raisonnable d'algorithme) : les fonctions constantes sont calculables ; les opérations arithmétiques usuelles sont calculables ; les projections $(n_1, \ldots, n_k) \mapsto n_i$ sont calculables, ainsi que la fonction qui à (m, n, p, q) associe p si $m = n$ et q sinon ; toute composée de fonctions calculables (partielle ou totale) est calculable idem ; si $\underline{m} \mapsto g(\underline{m})$ est calculable (partielle ou totale) et que $(\underline{m}, n, v) \mapsto h(\underline{m}, n, v)$ l'est, alors la fonction f définie par récurrence par $f(\underline{m}, 0) = g(\underline{m})$ et $f(\underline{m}, n + 1) = h(\underline{m}, n, f(\underline{m}, n))$ est encore calculable idem (algorithmiquement, il s'agit juste de boucler n fois) ; et enfin, si $(\underline{m}, n) \mapsto g(\underline{m}, n)$ est calculable partielle, alors la fonction f (aussi notée $\mu_n g$) définie par $f(\underline{m}) = \min\{n : g(\underline{m}, n) = 0 \wedge \forall n' < n(g(\underline{m}, n') \downarrow)\}$ (et non définie si ce $\min$ n'existe pas) est calculable partielle (algorithmiquement, on teste $g(\underline{m}, 0), g(\underline{m}, 1), g(\underline{m}, 2) \ldots$ jusqu'à tomber sur 0). Ces propriétés peuvent d'ailleurs servir à *définir* rigoureusement la notion de fonction calculable, c'est le modèle des fonctions « générales récursives ». (Dans ce qui précède, la notation $\underline{m}$ signifie $m_1, \ldots, m_k$.)

5. En fait, la valeur renvoyée n'a pas d'importance ; on peut aussi définir un ensemble semi-décidable comme l'ensemble de définition d'une fonction calculable partielle.

Exemples : L'ensemble des nombres pairs, des carrés parfaits, des nombres premiers, sont décidables, c'est-à-dire qu'il est algorithmique de savoir si un nombre est pair, parfait, ou premier. Quitte éventuellement à coder les mots d'un alphabet fini comme des entiers naturels (cf. plus haut), tout langage rationnel, et même tout langage défini par une grammaire hors-contexte, est décidable. On verra plus bas des exemples d'ensembles qui ne le sont pas, et qui sont ou ne sont pas semi-décidables.

Les deux propositions suivantes, outre leur intérêt intrinsèque, servent à donner des exemples du genre de manipulation qu'on peut faire avec la notion de calculabilité et d'algorithme :

Proposition. Un ensemble $A \subseteq \mathbb{N}$ est décidable ssi A et $\mathbb{N} \setminus A$ sont tous les deux semi-décidables.

Démonstration. Il est évident qu'un ensemble décidable est semi-décidable (si un algorithme décide A, on peut l'exécuter puis effectuer une boucle infinie si la réponse est « non » pour obtenir un algorithme qui semi-décide A) ; il est également évident que le complémentaire d'un ensemble décidable est décidable (quitte à échanger les réponses « oui » et « non » dans un algorithme qui le décide). Ceci montre qu'un ensemble décidable est semi-décidable de complémentaire semi-décidable, i.e., la partie « seulement si ». Montrons maintenant le « si » : si on dispose d'algorithmes T_1 et T_2 qui semi-décident respectivement A et son complémentaire, on peut lancer leur exécution en parallèle sur $n \in \mathbb{N}$ (c'est-à-dire exécuter une étape de T_1 puis une étape de T_2, puis de T_1, et ainsi de suite jusqu'à ce que l'un des deux termine) : comme il y en a toujours (exactement) un qui termine, selon lequel c'est, ceci permet de décider algorithmiquement si $n \in A$ ou $n \notin A$. ☺

Proposition. Un ensemble $A \subseteq \mathbb{N}$ non vide est semi-décidable ssi il existe une fonction calculable $f\colon \mathbb{N} \to \mathbb{N}$ dont l'image ($f(\mathbb{N})$) vaut A (on dit aussi que A est « calculablement énumérable » ou « récursivement énumérable »).

Démonstration. Montrons qu'un ensemble semi-décidable non vide est calculablement énumérable. Fixons $n_0 \in A$ une fois pour toutes. Soit T un algorithme qui semi-décide A. On définit une fonction $f\colon \mathbb{N}^2 \to \mathbb{N}$ de la façon suivante : $f(m,n) = n$ lorsque l'algorithme T, exécuté sur l'entrée n, termine au plus m étapes ; sinon, $f(m,n) = n_0$. On a bien sûr $f(m,n) \in A$ dans tous les cas ; par ailleurs, si $n \in A$, comme l'algorithme T appliqué à n doit terminer, on voit que pour m assez grand on a $f(m,n) = n$, donc n est bien dans l'image de f. Ceci montre que $f(\mathbb{N}^2) = A$. Passer à $f\colon \mathbb{N} \to \mathbb{N}$ est alors facile en composant par une bijection calculable $\mathbb{N} \to \mathbb{N}^2$ (par exemple la réciproque de $(m,n) \mapsto 2^m(2n+1)$).

Réciproquement, si A est calculablement énumérable, disons $A = f(\mathbb{N})$ avec f calculable, on obtient un algorithme qui semi-décide A en calculant successivement $f(0)$, $f(1)$, $f(2)$, etc., jusqu'à trouver un k tel que $f(k) = n$ (où n est l'entrée proposée),

auquel cas l'algorithme renvoie « oui » (et sinon, il ne termine jamais puisqu'il effectue une boucle infinie à la recherche d'un tel k). ☺

Clarification : Les deux démonstrations ci-dessus font appel à la notion intuitive d'« étape » de l'exécution d'un algorithme. Un peu plus précisément, pour chaque entier m et chaque algorithme T, il est possible d'« exécuter au plus m étapes » de l'algorithme T, c'est-à-dire commencer l'exécution de celui-ci, et si elle n'est pas finie au bout de m étapes, s'arrêter (on n'aura pas le résultat de l'exécution de T, juste l'information « ce n'est pas encore fini » et d'éventuels résultats intermédiaires, mais on peut décider de faire autre chose, y compris reprendre l'exécution plus tard). La longueur d'une « étape » n'est pas spécifiée et n'a pas d'importance, les choses qui importent sont que (A) le fait d'exécuter les m premières étapes de T termine toujours (c'est bien l'intérêt), et (B) si l'algorithme T termine effectivement, alors pour m suffisamment grand, exécuter au plus m étapes donne bien le résultat final de T résultat.

Complément/exercice : Un ensemble $A \subseteq \mathbb{N}$ infini est décidable ssi il existe une fonction calculable $f : \mathbb{N} \to \mathbb{N}$ strictement croissante dont l'image vaut A. (Esquisse : si A est décidable, on peut trouver son n-ième élément par ordre croissant en testant l'appartenance à A de tous les entiers naturels dans l'ordre jusqu'à trouver le n-ième qui appartienne ; réciproquement, si on a une telle fonction, on peut tester l'appartenance à A en calculant les valeurs de la fonction jusqu'à tomber sur l'entier à tester ou le dépasser.) En mettant ensemble ce fait et la proposition, on peut en déduire le fait suivant : tout ensemble semi-décidable infini a un sous-ensemble décidable infini (indication : prendre une fonction qui énumère l'ensemble et jeter toute valeur qui n'est pas strictement plus grande que toutes les précédentes).

Codage et machine universelle. Les algorithmes sont eux-mêmes représentables par des mots sur un alphabet fini donc, si on préfère, par des entiers naturels : on parle aussi de **codage de Gödel** des algorithmes/programmes par des entiers. On obtient donc une énumération $\varphi_0, \varphi_1, \varphi_2, \varphi_3 \ldots$ de toutes les fonctions calculables partielles (la fonction φ_e étant la fonction que calcule l'algorithme [codé par l'entier] e, avec la convention que si cet algorithme est syntaxiquement invalide ou erroné pour une raison quelconque, la fonction φ_e est simplement non-définie partout). Les détails de cette énumération dépendent de la formalisation utilisée pour la calculabilité.

Un point crucial dans cette numérotation des algorithmes est l'existence d'une **machine universelle**, c'est-à-dire d'un algorithme U qui prend en entrée un entier e (codant un algorithme T) et un entier n, et effectue la même chose que T sur l'entrée n (i.e., U termine sur les entrées e et n ssi T termine sur l'entrée n, et, dans ce cas, renvoie la même valeur).

Informatiquement, ceci représente le fait que les programmes informatiques sont eux-mêmes représentables informatiquement : dans un langage de programmation Turing-complet, on peut écrire un *interpréteur* pour le langage lui-même (ou pour un autre langage Turing-complet), c'est-à-dire un programme qui prend en entrée la représentation e d'un autre programme et qui exécute ce programme (sur une entrée n).

Mathématiquement, on peut le formuler comme le fait que la fonction (partielle) $(e, n) \mapsto \varphi_e(n)$ (= résultat du e-ième algorithme appliqué sur l'entrée n) est elle-même calculable partielle.

Philosophiquement, cela signifie que la notion d'exécution d'un algorithme est elle-même algorithmique : on peut écrire un algorithme qui, donnée une description

(formelle !) d'un algorithme et une entrée à laquelle l'appliquer, effectue l'exécution de l'algorithme fourni sur l'entrée fournie.

On ne peut pas démontrer ce résultat ici faute d'une description rigoureuse d'un modèle de calcul précis, mais il n'a rien de conceptuellement difficile (même s'il peut être fastidieux à écrire dans les détails : écrire un interpréteur d'un langage de programmation demande un minimum d'efforts).

Compléments : Les deux résultats classiques suivants sont pertinents en lien avec la numérotation des fonctions calculables partielles. • Le *théorème de la forme normale de Kleene* assure qu'il existe un ensemble décidable $\mathscr{T} \subseteq \mathbb{N}^4$ tel que $\varphi_e(n)$ soit défini ssi il existe m, v tels que $(e, n, m, v) \in \mathscr{T}$, et dans ce cas $\varphi_e(n) = v$ (pour s'en convaincre, il suffit de définir $\mathscr{T}$ comme l'ensemble des (e, n, m, v) tels que le e-ième algorithme exécuté sur l'entrée n termine en au plus m étapes et renvoie le résultat v : le fait qu'on dispose d'une machine universelle et qu'on puisse exécuter m étapes d'un algorithme assure que cet ensemble est bien décidable — il est même « primitif récursif »). • Le *théorème s-m-n* assure qu'il existe une fonction calculable s telle que $\varphi_{s(e,\underline{m})}(\underline{n}) = \varphi_e(\underline{m}, \underline{n})$ (intuitivement, donné un algorithme qui prend plusieurs entrées et des valeurs $\underline{m}$ de certaines de ces entrées, on peut fabriquer un nouvel algorithme dans lequel ces valeurs ont été fixées — c'est à peu près trivial — mais de plus, cette transformation est *elle-même algorithmique*, i.e., on peut algorithmiquement substituer des valeurs $\underline{m}$ dans un programme [codé par l'entier] e : c'est intuitivement clair, mais cela ne peut pas se démontrer avec les seules explications données ci-dessus sur l'énumération des fonctions calculables partielles, il faut regarder précisément comment le codage standard est fait pour une formalisation de la calculabilité).

La machine universelle n'a rien de « magique » : elle se contente de suivre les instructions de l'algorithme T qu'on lui fournit, et termine ssi T termine. Peut-on savoir à l'avance si T terminera ? C'est le fameux « problème de l'arrêt ».

Intuitivement, le « problème de l'arrêt » est la question « l'algorithme suivant termine-t-il sur l'entrée suivante » ?

Définition. On appelle **problème de l'arrêt** l'ensemble des couples (e, n) tels que le e-ième algorithme termine sur l'entrée n, i.e., $\{(e, n) \in \mathbb{N}^2 : \varphi_e(n) \downarrow\}$ (où la notation « $\varphi_e(n) \downarrow$ » signifie que $\varphi_e(n)$ est défini, i.e., l'algorithme termine). Quitte à coder les couples d'entiers naturels par des entiers naturels (par exemple par $(e, n) \mapsto 2^e(2n+1)$), on peut voir le problème de l'arrêt comme une partie de $\mathbb{N}$. On peut aussi préférer[6] définir le problème de l'arrêt comme $\{e \in \mathbb{N} : \varphi_e(e) \downarrow\}$, on va voir dans la démonstration ci-dessous que c'est cet ensemble-là qui la fait fonctionner.

(On pourrait aussi définir le problème de l'arrêt comme $\{e \in \mathbb{N} : \varphi_e(0) \downarrow\}$ si on voulait, ce serait moins pratique pour la démonstration, mais cela ne changerait rien au résultat comme on peut le voir en appliquant le théorème s-m-n.)

Théorème (Turing)**.** Le problème de l'arrêt est semi-décidable mais non décidable.

Démonstration. Le problème de l'arrêt est semi-décidable en vertu de l'existence d'une machine universelle : donnés e et n, on exécute le e-ième algorithme sur l'entrée n (c'est

6. Même si au final c'est équivalent, c'est *a priori* plus fort de dire que $\{e \in \mathbb{N} : \varphi_e(e) \downarrow\}$ n'est pas décidable que de dire que $\{(e, n) \in \mathbb{N}^2 : \varphi_e(n) \downarrow\}$ ne l'est pas.

ce que fait la machine universelle), et s'il termine on renvoie « oui » (et s'il ne termine pas, bien sûr, on n'a pas de choix que de ne pas terminer).

Montrons par l'absurde que le problème de l'arrêt n'est pas décidable. S'il l'était, on pourrait définir un algorithme qui, donné un entier e, effectue les calculs suivants : (1°) utiliser le problème de l'arrêt (supposé décidable !) pour savoir, algorithmiquement en temps fini, si le e-ième algorithme termine quand on lui passe son propre numéro e en entrée, i.e., si $\varphi_e(e) \downarrow$, (2°) si oui, effectuer une boucle infinie, et si non, terminer, en renvoyant, disons, 42. L'algorithme qui vient d'être décrit aurait un certain numéro, disons, p, et la description de l'algorithme fait que, quelque soit e, la valeur $\varphi_p(e)$ est indéfinie si $\varphi_e(e)$ est définie tandis que $\varphi_p(e)$ est définie (de valeur 42) si $\varphi_e(e)$ est indéfinie. En particulier, en prenant $e = p$, on voit que $\varphi_p(p)$ devrait être défini si et seulement si $\varphi_p(p)$ n'est pas défini, ce qui est une contradiction. ☺

La démonstration ci-dessus est une instance de l'« argument diagonal » de Cantor, qui apparaît souvent en mathématiques. (La « diagonale » en question étant le fait qu'on considère $\varphi_e(e)$, i.e., on passe le numéro e d'un algorithme en argument à cet algorithme lui-même, donc on regarde la diagonale de la fonction de deux variables $(e, n) \mapsto \varphi_e(n)$; en modifiant les valeurs sur cette diagonale, on produit une fonction qui ne peut pas se trouver dans une ligne φ_p.) Une variante facile du même argument permet de fabriquer des ensembles non semi-décidables (voir le « bonus » ci-dessous), ou bien on peut appliquer ce qui précède :

Corollaire. Le complémentaire du problème de l'arrêt n'est pas semi-décidable.

Démonstration. On a vu que le problème de l'arrêt n'est pas décidable, et qu'un ensemble est décidable ssi il est semi-décidable et que son complémentaire l'est aussi : comme le problème de l'arrêt est bien semi-décidable, son complémentaire ne l'est pas. ☺

Complément : L'argument diagonal est aussi au cœur du (voire, équivalent au) *théorème de récursion de Kleene*, qui affirme que pour toute fonction calculable partielle $h\colon \mathbb{N}^2 \dashrightarrow \mathbb{N}$, il existe p tel que $\varphi_p(n) = h(p, n)$ pour tout n (la signification intuitive de ce résultat est qu'on peut supposer qu'un programme a accès à son propre code source p, i.e., on peut programmer comme s'il recevait en entrée un entier p codant ce code source ; ceci permet par exemple — de façon anecdotique mais amusante — d'écrire des programmes, parfois appelés « quines », qui affichent leur propre code source sans aller le chercher sur disque ou autre tricherie). *Démonstration :* donné $e \in \mathbb{N}$, on considère $s(e, m)$ tel que $\varphi_{s(e,m)}(n) = \varphi_e(m, n)$: le théorème s-m-n (cf. ci-dessus) assure qu'une telle fonction calculable $(e, m) \mapsto s(e, m)$ existe, et $(e, n) \mapsto h(s(e, e), n)$ est alors aussi calculable partielle ; il existe donc q tel que $\varphi_q(e, n) = h(s(e, e), n)$: on pose $p = s(q, q)$, et on a $\varphi_p(n) = \varphi_q(q, n) = h(s(q, q), n) = h(p, n)$, comme annoncé. ☺ La non-décidabilité du problème de l'arrêt s'obtient en appliquant (de nouveau par l'absurde) ce résultat à $h(e, n)$ la fonction qui n'est pas définie si $\varphi_e(n)$ l'est et qui vaut 42 si $\varphi_e(n)$ n'est pas définie.

La non-décidabilité du problème de l'arrêt est un résultat fondamental, car très souvent les résultats de non-décidabilité soit sont démontrés sur un modèle

semblable, soit s'y ramènent directement : pour montrer qu'un certain ensemble A (un « problème ») n'est pas décidable, on cherche souvent à montrer que si un algorithme décidant A existait, on pourrait s'en servir pour construire un algorithme résolvant le problème de l'arrêt.

Bonus / exemple(s) : L'ensemble des $e \in \mathbb{N}$ tels que la fonction calculable partielle φ_e soit totale (i.e., définie sur tout $\mathbb{N}$) n'est pas semi-décidable. En effet, s'il l'était, d'après ce qu'on a vu, il serait « calculablement énumérable », c'est-à-dire qu'il existerait une fonction calculable $f\colon \mathbb{N} \to \mathbb{N}$ dont l'image soit exactement l'ensemble des e pour lesquels φ_e est totale, i.e., toute fonction calculable totale s'écrirait sous la forme $\varphi_{f(k)}$ pour un certain k. Mais la fonction $n \mapsto \varphi_{f(n)}(n) + 1$ est calculable totale, donc il devrait exister un m tel que cette fonction s'écrive $\varphi_{f(m)}$, c'est-à-dire $\varphi_{f(m)}(n) = \varphi_{f(n)}(n) + 1$, et on aurait alors en particulier $\varphi_{f(m)}(m) = \varphi_{f(m)}(m) + 1$, une contradiction. • Son complémentaire, c'est-à-dire l'ensemble des $e \in \mathbb{N}$ tels que la fonction calculable partielle φ_e ne soit pas totale, n'est pas non plus semi-décidable. En effet, supposons qu'il existe un algorithme qui, donné e, termine ssi φ_e n'est pas totale. Donnés e et m, considérons l'algorithme qui prend une entrée n, *ignore* celle-ci, et effectue le calcul $\varphi_e(m)$: ceci définit une fonction calculable partielle (soit totale et constante, soit définie nulle part !) $\varphi_{s(e,m)}$ où s est calculable (on applique ici le théorème s-m-n) — en appliquant à $s(e, m)$ l'algorithme supposé semi-décider si une fonction récursive partielle est non-totale, on voit qu'ici il semi-décide si $\varphi_e(m)$ est non-défini, autrement dit on semi-décide le complémentaire du problème de l'arrêt, et on a vu que ce n'était pas possible !

Exercice : Considérons une fonction h qui à e associe un nombre au moins égal au nombre d'étapes (cf. ci-dessus) du calcul de $\varphi_e(e)$, si celui-ci termine, et une valeur quelconque si $\varphi_e(e)$ n'est pas défini. Alors h n'est pas calculable. (Indication : si elle l'était, on pourrait décider si $\varphi_e(e)$ est défini en exécutant son calcul pendant $h(e)$ étapes.) On peut même montrer que $H(n) := \max\{h(i) : i \leq n\}$ domine asymptotiquement n'importe quelle fonction calculable mais c'est un peu plus difficile.

Application à la logique : Sans rentrer dans les détails de ce que signifie un « système formel », on peut esquisser, au moins informellement, les arguments suivants. Imaginons qu'on ait formalisé la notion de démonstration mathématique (c'est-à-dire qu'on les écrit comme des mots dans un alphabet indiquant quels axiomes et quelles règles logiques sont utilisées) : même sans savoir quelle est exactement la logique formelle, le fait de *vérifier* qu'une démonstration est correcte doit certainement être algorithmique (il s'agit simplement de vérifier que chaque règle a été correctement appliquée), autrement dit, l'ensemble des démonstrations est décidable. L'ensemble des théorèmes, lui, est semi-décidable (on a un algorithme qui semi-décide si un certain énoncé est un théorème en énumérant toutes les chaînes de caractères possibles et en cherchant s'il s'agit d'une démonstration valable dont la conclusion est l'énoncé recherché). Or l'ensemble des théorèmes n'est pas décidable : en effet, si on avait un algorithme qui permet de décider si un énoncé mathématique est un théorème, on pourrait appliquer cet algorithme à l'énoncé formel (*)« le e-ième algorithme termine sur l'entrée e », en observant qu'un tel énoncé, s'il est vrai, est forcément démontrable (i.e., si l'algorithme termine, on peut *démontrer* ce fait en écrivant étape par étape l'exécution de l'algorithme pour constituer une démonstration qu'il a bien été appliqué jusqu'au bout et a terminé), et en espérant que s'il est démontrable alors il est vrai : on aurait alors une façon de décider le problème de l'arrêt, une contradiction. Mais du coup, l'ensemble des non-théorèmes ne peut pas être semi-décidable ; or comme l'ensemble des énoncés P tels que $\neg P$ (« non-P », la négation logique de P) soit un théorème est semi-décidable (puisque l'ensemble des théorèmes l'est), ils ne peuvent pas coïncider. Ceci montre qu'il existe un énoncé tel que ni P ni $\neg P$ ne sont des théorèmes : c'est une forme du *théorème de Gödel* que Turing cherchait à démontrer ; mieux : en appliquant aux énoncés du type (*), on montre ainsi qu'il existe un algorithme qui *ne termine pas* mais dont la non-terminaison *n'est pas démontrable*. (Modulo quelques hypothèses qui n'ont pas été explicitées sur le système formel dans lequel on travaille.)