Calculabilité
Cours 1 : Introduction et cardinalité

Kévin Perrot – L3 Info Aix Marseille Université – printemps 2023-24

0 Divertissement

```
collatz(n:int)
    print n;
    si n == 1 alors stop, sinon
        si n%2 == 0 alors collatz(n/2), sinon collatz(3*n+1), finsi
    finsi
```

Est-ce que le programme `collatz` termine sur toute entrée `n` ? Comment le savoir ?
- Tester pour tout $n \leq 10^{20}$?
- Tester si $\exists$ `n,x` tel que `collatz(n)=collatz`$^{\texttt{x}}$`(n)` ?
- Vous avez un an avec 10 programmeurs surdoués. Comment faites-vous ?

Suite de Collatz (1937) : $n \mapsto \begin{cases} n/2 & \text{si } n \mod 2 == 0 \\ n*3+1 & \text{sinon} \end{cases}$

Conjectures réfutées par de grands contre-exemples :
- Conjecture d'Euler (1772). $\forall n > 2 : \forall x_1, ..., x_k, z \in \mathbb{N} : \sum_{i=1}^{k} x_i^n \neq z^n$.
  $n = 5$ (1966) : $27^5 + 84^5 + 110^5 + 133^5 = 144^5$
  $n = 4$ (1988) : $2682440^4 + 15365639^4 + 18796769^4 = 20615673^4$
  $n > 5$: inconnu.
- Conjecture de Pólya (1919). Plus de la moitié des entiers naturels inférieurs à un entier donné ont un nombre impair de facteurs premiers.
  (1980) : le plus petit contre exemple est 906 150 257.
- Conjecture de Mertens (1885) : prouvée fausse mais aucun contre exemple explicite connu.
  2006 : plus petit contre exemple compris entre 10^{14} et 1.59×10^{40}
- Nombre de Skewes (1914).
  1955 : il existe un tel nombre inférieur à $10^{10^{10^{963}}}$.
  1987 : il existe un tel nombre inférieur à 7×10^{370}.

La morale de cet example est que les ordinateurs n'ont pas réponse à tout !

Table des matières

Ce cours est basé sur le livre de Sipser [2] et le cours de Kari [1].

1 Introduction

Une machine de Turing est un objet mathématique, défini en 1936 par *Alan Turing*, qui a pour but de décrire ce qu'est un *calcul*. Tout le monde sent ce qu'est un calcul : si vous avez deux nombre en base 10, disons `123` et `456`, et vous voulez calculer leur somme, vous le faites chiffre par chiffre de droite à gauche en propageant d'éventuelles retenues pour obtenir `579`. Si vous voulez calculer leur produit, vous le décomposez en multiplications plus simples pour lesquelles vous avez appris un nombre fini de tables, et vous sommez les résultats des sous-problèmes :

```
123x456  =(1x100+2x10+3)x(4x100+5x10+6)
         =(1x4)x(100x100)+(1x5)x(100x10)+(1x6)x(100)
         +(2x4)x(10x100)+(2x5)x(10x10)+(2x6)x(10)
         +(3x4)x(100)+(3x5)x(10)+(3x6)
         =56088.
```

Lorsque vous apprenez à quelqu'un une méthode pour effectuer une multiplication, vous décrivez un *algorithme* : vous donnez une description finie (par exemple en 10 minutes de parole, ou en 2 pages) de la procédure à suivre pour obtenir le résultat. **Les machines de Turing sont des objets mathématiques pour décrire les algorithmes**. Une machine de Turing pour l'addition de nombres naturels en base 10 donne l'ensemble des instructions qu'il faut effectuer pour calculer la somme de deux nombres. Une remarque importante est que la description de la procédure est finie, mais elle permet de calculer la somme de n'importe quel couple de nombres : `123+456` ou `1234567890+2345678901` ou des nombres plus grands !

Bon, soyons sérieux. Une machine de Turing décrit comment calculer quelque chose. Mais quel est ce *quelque chose* ? C'est une *fonction*. Par exemple, une fonction peut-être l'addition, ou la multiplication : étant donnée une *entrée* finie (dans notre exemple `123` et `456`), une fonction associe une unique *sortie*. L'entrée et la sortie peuvent être un ou plusieurs entiers naturels, des nombres négatifs, une phrase écrite en alphabet Latin ou n'importe quel autre. Le point important est qu'elle soit de taille *finie*. Une fonction associe alors une unique sortie (le résultat) à chaque entrée.

Les fonctions peuvent être simples : l'addition ou la multiplication de deux entiers naturels ; ou plus complexes : étant donné un entier naturel, quelle est sa décomposition en produit de facteurs premiers (entiers naturels plus grand que 1 qui n'ont pas de diviseur autre que 1 et lui-même) ; ou même *non calculable* : étant donné un énoncé mathématique, décider s'il est vrai ou faux. Oui, certaines fonctions ne sont pas calculables : il n'existe pas d'algorithme qui les calcule. De plus, il y a infiniment plus de fonctions non calculables que de fonctions calculables ! Il existe une infinité de fonctions, et une infinité de machines de Turing (d'algorithmes), mais le nombre de fonctions est infiniment plus grand que le

nombre de machines de Turing.

Une question naturel est : si les machines de Turing peuvent calculer si peu de fonctions, pourquoi ne pas calculer avec un autre modèle mathématique ? En réalité, les machines de Turing ne sont pas le seul objet mathématique permettant de décrire des algorithmes. De tels modèles sont appelés *modèles de calcul effectifs*, où *effectif* signifie approximativement « en accord avec le monde réel ». Il est cependant magnifique de constater que tous les modèles de calcul proposés jusqu'à présent sont équivalents ! Deux modèles sont équivalents s'ils peuvent calculer exactement le même ensemble de fonctions. Rappelons nous bien : soit `F` l'ensemble de toutes les fonctions, les machines de Turing ne peuvent pas calculer toutes les fonctions de l'ensemble `F`, mais seulement un sous-ensemble `C`. **La croyance (répandue) selon laquelle tout autre modèle de calcul effectif que l'on pourrait imaginer sera également capable de calculer toutes les fonctions de l'ensemble `C` et aucune autre est appelée *thèse de Church-Turing*, et `C` est appelé l'ensemble des *fonctions calculables***. L'une des questions les plus fondamentales de la science informatique est la suivante : pourquoi une fonction est-elle calculable ou non calculable ?

Un point intéressant est l'existence de machines de Turing *universelles*. Une machine de Turing universelle `U` est une machine capable de *simuler* tout autre machine de Turing. Qu'est-ce que cela signifie ? Soit `M` une machine de Turing quelconque et `x` une entrée, la sortie de `M` sur l'entrée `x` est notée `M(x)`. Une machine `U` est universelle si l'on peut écrire une entrée `y` sur le ruban telle que le calcul de `U` sur `y` donne `M(x)` en sortie.

`U` et `y` ne sont pas très compliqués à construire : `M` a une description finie (principalement sa table d'actions), donc cette description peut être écrite sur le ruban, elle utilisera `n` cellules ; et sur d'autres cellules vides, on peut écrire `x` ; le tout donne l'entrée y. Maintenant, `U` a toute l'information qui définit `M(x)` sur le ruban, et il est possible [1] de construire une telle machine `U` qui *lit* l'entrée `x` sur le ruban, ensuite *lit* la table d'action de `M` sur les `n` cellules dédiées du ruban, et ensuite réalise sur `x` ce que la machine `M` aurait réalisé si elle avait été exécutée sur un ruban contenant `x`. Une telle machine `U` est un peu délicate à construire, mais pas excessivement.

De nos jours, un ruban est appelé *disque dur*, la table d'action de `M` écrite sur le ruban est un *programme*, et `U` est un *ordinateur* !

Une machine de Turing universelle entièrement mécanique a été réalisée en Lego.

`http://www.dailymotion.com/video/xrmfie/`

`http://rubens.ens-lyon.fr/`

Par conséquent, ce tas de Lego est capable de calculer l'ensemble des fonctions calculables `C` : il a exactement la même *puissance de calcul* que votre ordinateur ou votre téléphone portable ! En comparaison avec un ordinateur moderne qui réalise une instruction toutes les nano-secondes (0.000000001 secondes), cette machine en Lego réalise une instruction toutes les 100 secondes. **Elle peut faire ce qu'un ordinateur moderne peut faire, mais pour réaliser ce que ce dernier effectue en 1 seconde, il lui faut 3168 ans 295 jours 9 heures 46 minutes et 40 secondes** [2]. Quoi qu'il en soit, l'important est qu'elle en soit capable, n'est-ce pas ?

1. Remarquons que l'existence de fonctions non calculables implique que pour d'autres problèmes (encore une fois il y en a énormément, infiniement plus que des calculales), même avec toute l'information qui définit la question, il n'existe pas de machine de Turing qui calcule le résultat.

2. Ce nombre est en réalité complètement faux, parce qu'une instruction de machine de Turing est différente d'une instruction d'un ordinateur moderne. Néanmoins, il est là pour souligner le fait que cette machine de Turing en Lego est très précisément équivalente à un ordinateur moderne.

C'est le coeur de la *calculabilité*. Les ordinateurs sont chaque année plus rapides, et leur vitesse continue d'augmenter. Cependant, ils restent restreints à l'ensemble des fonctions calculables, `C`. Ils peuvent calculer les fonctions de l'ensemble `C` toujours plus vite, mais ne peuvent pas s'échapper de `C` : leur *expressivité* reste la même. L'étude de cette expressivité, du sens de cette puissance de calcul, s'appelle la *théorie de la calculabilité*.

Vous pouvez parfois entendre que nous sommes aujourd'hui capables de calculer des choses qui étaient impossible à calculer les années passées, que les ordinateurs sont plus puissants aujourd'hui qu'hier. Ces phrases doivent être précisées. En réalité, ces calculs étaient simplement trop longs à réaliser les années passées (par exemple, il aurait fallut 100 ans si vous les aviez exécutés sur un ordinateur en l'an 2000), mais aujourd'hui vous pouvez les calculer en un temps raisonnable (par exemple 100 secondes) ce qui permet d'obtenir effectivement le résultat. Un exemple intéressant est le jeu des échecs : il est possible aujourd'hui, et il a toujours été possible, d'écrire un algorithme qui vous indique coup après coup la meilleure action possible, mais les ordinateurs actuels sont bien trop lents pour exécuter un tel algorithme jusqu'au bout... Néanmoins, un jour nous serons capables de réaliser ce calcul en un temps raisonnable, et ensuite jouer aux échecs avec un ordinateur deviendra définitivement ennuyant car nous serons sûrs et certains de perdre chaque partie [3]. Laissez moi répéter qu'un tel algorithme existe déjà et est facile à implémenter sur un ordinateur, les ordinateurs sont simplement trop lents pour réaliser les calculs en un temps raisonnable. La *théorie de la calculabilité* s'intéresse à des vérités mathématiques qui sont indépendantes du temps de calcul : la question n'est pas de savoir si quelque chose sera faisable dans 10 ou 20 ans ou même 1000 ans, mais plutôt si quelque chose est fondamentalement faisable ou non. De comprendre quels problèmes peuvent être résolus algorithmiquement, et quels problèmes ne le peuvent pas.

2 Limites des programmes : cardinalité

Nous proposons dans cette section un argument général énonçant l'existence de limites à nos capacités de calcul par ordinateur. Nous posons la question suivante :

quelles fonctions de $\mathbb{N}$ dans $\mathbb{N}$ peut-on programmer ?

Remarque 1. *La restriction aux fonctions de $\mathbb{N}$ dans $\mathbb{N}$ reste en fait un cas très général, si l'on pense que toute donnée stockée sur ordinateur est une suite de bits, que l'on peut voir comme un nombre. C'est une question d'encodage.*

Soit $\mathcal{P}$ votre langage de programmation favori (C, Python, Haskell, Java, *etc*). L'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ est de cardinalité infinie (elles sont données mathématiquement), et l'ensemble des programmes en $\mathcal{P}$ est également de cardinalité infinie (ils sont données par un code source). Cependant, on sait depuis les travaux du mathématicien Cantor en 1891, qu'il existe des ensembles infinis "plus grands" que d'autres, et l'on va voir maintenant qu'il est impossible d'avoir un programme pour chaque fonction.

Remarque 2. *Pour comparer les tailles d'ensembles infinis, la bonne notion est celle de l'existence d'une bijection. Si il existe une bijection entre deux ensembles, alors ils*

3. Je mens un peu. En réalité, puisque nous n'avons encore jamais calculé toutes les actions possibles aux échecs, nous ne savons pas si ce sont les blancs ou les noirs qui ont une stratégie gagnante, ou si nous arriverions à un match nul avec deux joueurs parfaits...

*contiennent "autant" d'éléments l'un que l'autre, ils ont la même cardinalité. Il y a une excellente analogie pour se rendre compte de cela, c'est l'*hotel de Hilbert *!*

Lemme 3. *Il existe une bijection entre l'ensemble des programmes en $\mathcal{P}$ et $\mathbb{N}$.*

Démonstration. Bien que l'on ne connaisse pas exactement le langage $\mathcal{P}$ dans tous ses détails [4], on sait qu'un programme consiste en un fichier de texte, dont les caractères sont choisis parmi un alphabet fini : en général ASCII, 127 caractères. On peut alors mettre en correspondance les entiers naturels et les textes, en regardant chaque texte comme un nombre écrit en base 128 (on utilise pas le zéro pour bien différencier 0001 de 1 par exemple). On a ainsi un ordre pour énumérer tous les textes en partant du texte correspondant à 1, et en comptant en base 128. Ensuite à chaque fois qu'un texte correspond à un programme qui respecte la grammaire du langage $\mathcal{P}$, on lui attribue un entier naturel, en partant de 0 pour le premier programme rencontré. Chaque programme en $\mathcal{P}$ sera associé à un entier naturel unique, et puisqu'il existe des programmes toujours plus longs les uns que les autres (on peut simplement imaginer rajouter des caractères espace) alors chaque entier naturel se verra au bout d'un moment associer un programme en $\mathcal{P}$. Cette correspondance est donc bien une bijection (injective et surjective). □

Remarque 4.
Enumérer *un ensemble S* $=$ *donner (au moins) un numéro à chaque élément*
$=$ *donner une fonction totale surjective de $\mathbb{N}$ dans S.*
Dans ce cas S ne peut pas être plus grand que $\mathbb{N}$.
Une énumération de S sans répétition (= injective) est une bijection de $\mathbb{N}$ dans S.

Rappel : $[0, 1]$ est l'ensemble des nombres réels entre 0 et 1 (inclus), et les réels peuvent avoir une infinité de décimales (comme par exemple π).

Lemme 5. *Il existe une bijection entre l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, et $[0, 1]$.*

Démonstration. Soit F l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Nous allons donner une fonction injective de F dans $[0, 1]$ (pour montrer $|F| \leq |[0, 1]|$), et une fonction injective de $[0, 1]$ dans F (pour montrer $|[0, 1]| \leq |F|$). Nous pourrons alors en déduire [5] l'énoncé du lemme (c'est-à-dire $|[0, 1]| = |F|$).

Commencons par construire une fonction injective de $[0, 1]$ dans F. Soit [6] $x = 0.x_0x_1x_2x_3...$ un réel entre 0 et 1 avec $x_i \in \{0, ..., 9\}$ pour tout $i \in \mathbb{N}$, nous pouvons lui faire correspondre la fonction $f : \mathbb{N} \to \mathbb{N}$ définie par $f(i) = x_i$.

Construisons maintenant une fonction injective de F dans $[0, 1]$. Une fonction f de $\mathbb{N}$ dans $\mathbb{N}$ est une suite infinie d'entiers naturels : $f(0)$, $f(1)$, $f(2)$, etc (attention : chaque $f(i)$ est un nombre fini car $+\infty \notin \mathbb{N}$). Nous pouvons donc faire correspondre à chaque fonction un nombre réel $0.f(0)f(1)f(2)\ldots$. Cette fonction n'est cependant pas injective. Pour la rendre injective nous pouvons utiliser le codage suivant :

— les $f(i)$ sont codés en binaire dédoublé (chaque bit est écrit deux fois, par exemple 9 en décimal devient 11000011),

4. Ce sera un très gros avantage des machines de Turing : leur définition complète tient en quelques lignes. La définition complète d'un langage de programmation est implicitement donnée par le code source d'un compilateur...

5. Cette déduction est donnée par l'application d'un résultat classique de théorie des ensemble, appelé théorème de Cantor-Schröder-Bernstein.

6. Remarquons entre parenthèse qu'un nombre réel peut avoir plusieurs représentations, comme par exemple 1 qui peut s'écrire 1.00000... ou 0.99999..., mais cela n'a pas d'influence sur notre argumentation.

— les $f(i)$ et $f(i+1)$ sont séparés par la séquence 01. □

Théorème 6. $\aleph_0 < 2^{\aleph_0}$, *avec* $\aleph_0 = |\mathbb{N}|$ *et* $2^{\aleph_0} = |[0,1]|$.

Démonstration (Cantor, 1891). Nous allons démontrer ce résultat par l'absurde. Supposons qu'il existe une bijection entre $\mathbb{N}$ et $[0,1]$ (auquel cas $\aleph_0 = 2^{\aleph_0}$), alors il est possible d'énumérer tous les nombres réels entre 0 et 1 sans en oublier aucun :

$$\begin{aligned} r_1 &= 0.\boxed{1}234567890\ldots \\ r_2 &= 0.5\boxed{3}49236423\ldots \\ r_3 &= 0.72\boxed{9}1655000\ldots \\ r_4 &= 0.239\boxed{3}218693\ldots \\ &\ldots \end{aligned}$$

Nous allons montrer qu'il est impossible d'avoir énuméré tous les éléments de $[0,1]$, ce qui est une conradiction. En effet, nous avons forcément oublié le nombre suivant :

$$r_+ = 0.2404\ldots$$

construit en prenant pour première décimale la première décimale de r_1 plus 1 modulo 10, pour seconde décimale la seconde décimale de r_2 plus 1 modulo 10, pour troisième décimale la troisième décimale de r_3 plus 1 modulo 10, etc, à l'infini (les nombres réels peuvent avoir une infinité de décimales). On a bien $r_+ \in [0,1]$, et pour tout $i \in \mathbb{N}$: $r_i \neq r_+$ car ils diffèrent sur leur $i^{\text{ème}}$ décimale. □

Corollaire 7.
Dans tout langage de programmation il existe des fonctions non calculables.

Démonstration. Application du théorème 6 d'après les lemmes 3 et 5. □

Remarque 8. *Il existe donc des infinis de tailles différentes. On dira qu'un ensemble est*
— **fini** *s'il contient un nombre fini d'éléments (sa taille est un entier naturel),*
— **dénombrable** *s'il est en bijection avec* $\mathbb{N}$ *(de taille* $\aleph_0$*),*
— **indénombrable** *sinon.*

Le corollaire 7 ne nous donne pas d'exemple de fonction non calculable, mais il nous dit qu'elles sont très nombreuses (infiniment plus que les fonctions calculables) !

Digression sur l'hypothèse du continu

Le théorème 6 amène une question naturelle : existe-t-il des infinis strictement plus grands que $\aleph_0$, mais strictement plus petits que $2^{\aleph_0}$? En d'autres termes, si l'on dénote $\aleph_1$ le second plus petit infini après $\aleph_0$, est-ce que

$$\aleph_1 = 2^{\aleph_0} ?$$

Cette question fameuse est appelée **hypothèse du continu** (HC), et fut posée par Cantor. Il fallut attendre l'axiomatisation de la théorie des ensembles [7] par Zermelo et Fraenkel, notée ZFC, au début du XX^{e} siècle, une preuve par Gödel en 1938 que HC ne peut pas être réfutée dans ZFC, et une preuve par Cohen en 1963 que HC ne peut pas être prouvée dans ZFC, pour arriver à la conclusion suivante : HC est indépendante de ZFC. Pour reformuler, la théorie des ensembles qui fait consensus comme capturant "l'ensemble des mathématiques", ne permet pas de dire si l'hypothèse du continu est vraie ou fausse, les deux éventualités sont consistantes (n'amènent pas de contradiction).

Références

[1] J. Kari. *Automata and formal languages*. University of Turku, 2013. Course notes available at `http://users.utu.fi/jkari/`.

[2] M. Sipser. *Introduction to the theory of computation*. Course Technology, 2006.

7. C'est-à-dire la définition précise d'axiomes à partir desquels on dérive des théorèmes (vérités mathématiques). Avant cela (et pour Cantor notamment), on utilisait une définition intuitive (≪ naïve ≫) des ensembles. Par exemple, rien n'interdisait de considérer l'ensemble de tous les ensembles, $\mathcal{S}$. Russell souleva a ce propos un paradoxe divertissant : soit $X = \{A \in \mathcal{S} \mid A \notin A\}$, est-ce que $X \in X$? C'est à ce moment que je vous conseille la lecture de la bande dessinée *Logicomix* par Doxiadis et Papadimitriou.