Calculabilité
Cours 1 : machines de Turing

Kévin Perrot – Aix Marseille Université – printemps 2019

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 compliquées : 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 calcul. 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`. 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**

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.

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 ?

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, mais plutôt si quelque chose est fondamentalement faisable ou non, fondamentalement vrai ou faux.

2 Machines de Turing

Pour montrer qu'une fonction est calculable ou qu'un langage est décidable (distinction discutée en 2.2) il faut donner un algorithme. Pour montrer qu'une fonction n'est pas calculable ou qu'un langage est indécibable, il faut d'abord définir l'ensemble des algorithmes (car cela définit l'ensemble de ce qui est calculable/décibable). L'intérêt des machines de Turing est qu'elles définissent les algorithmes de façon intuitive et simple ! Imaginez devoir définir mathématiquement votre langage de programmation préféré dans ses moindres détails...

Idée du calculateur humain devant sa feuille :

— feuilles découpées en cases : ruban ;

— crayon posé sur une case : tête de lecture/écriture, déplacement ;

— l'opérateur dispose d'une mémoire finie (son cerveau) : états.

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

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...

2.1 Définitions

Définition 1. *Une* **machine de Turing** *(***MT***) déterministe est un 7-uplet*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, q_F)$$

où

- Q *est un* **ensemble d'états** *fini ;*
- Σ *est l'***alphabet d'entrée** *;*
- Γ *est l'***alphabet de ruban**
  (il contient tous les symboles qui peuvent apparaître sur le ruban. En particulier $\Sigma \subset \Gamma$ *car l'entrée est initialement écrite sur le ruban. On supposera que* $Q \cap \Gamma = \emptyset$ *pour qu'il n'y ait pas de confusion entre états et symboles du ruban) ;*
- δ *est une* **fonction de transition** *décrite ci-après ;*
- $q_0 \in Q$ *est l'***état initial** *;*
- $B \in \Gamma \setminus \Sigma$ *est un* **symbole blanc** *spécial (ne fait pas partie de l'alphabet d'entrée) ;*
- $q_F \in Q$ *est l'état final.*

La fonction de transition δ est une application (partielle)

de l'ensemble $(Q \setminus \{q_F\}) \times \Gamma$ dans l'ensemble $Q \times \Gamma \times \{L, R\}$.

L'application est partielle : elle peut être indéfinie pour certains arguments, auquel cas la machine n'a pas de mouvement suivant et s'arrête. En particulier, il n'y a pas de transition depuis l'état final q_F. Une transition

$$\delta(q, a) = (p, b, L)$$

signifie que dans l'état q et en lisant le symbole de ruban a, la machine passe dans l'état p, remplace a par b sur le ruban, et déplace la tête de lecture/écriture d'une cellule sur la gauche (L pour *left* et R pour *right*).

Initialement, le mot d'entrée est écrit sur le ruban et toutes les autres cellules contiennent le symbole blanc B. La machine est dans l'état q_0, et la tête de lecture/écriture est positionnée sur la lettre la plus à gauche de l'entrée. Il y a trois possibilités :

- **acceptation** si au cours des transitions la machine entre dans l'état final q_F (et donc s'arrête),
- **rejet** si au cours des transitions la machine s'arrête dans un état non final (s'il n'y a pas de mouvement suivant à réaliser),
- **rejet** si la machine ne s'arrête jamais.

Définition 2. *Une* **description instantanée** *(***DI***) d'une MT décrit sa configuration courante. C'est un mot*

$$uqav \in (\{\epsilon\} \cup (\Gamma \setminus \{B\})\Gamma^*)Q\Gamma(\{\epsilon\} \cup \Gamma^*(\Gamma \setminus \{B\}))$$

avec $q \in Q$ *l'état courant,* $u, v \in \Gamma^*$ *le contenu du ruban à gauche et à droite de la tête, respectivement, jusqu'au dernier symbole non blanc, et* $a \in \Gamma$ *le symbole de ruban actuellement sous la tête.*

Définition 3. *Un* **mouvement**, *une* **transition**, *un* **déplacement** *de la MT à partir de la DI* $\alpha = uqav$ *vers la DI suivante* β *sera noté* $\alpha \vdash \beta$. *Plus précisément :*

1. *Si* $\delta(q, a) = (p, b, L)$,
   - *si* $u = \epsilon$ *alors* $\beta = pBbv$ *(potentiellement en supprimant les* B *à la fin de* bv*),*

— *si* $u = u'c$ *avec* $c \in \Gamma$ *alors* $\beta = u'pcbv$ *(potentiellement en supprimant les* B *à la fin de* bv*).*

2. *Si* $\delta(q, a) = (p, b, R)$,
   — *si* $v = \epsilon$ *alors* $\beta = ubpB$ *(potentiellement en supprimant les* B *au début de* ub*),*
   — *si* $v \neq \epsilon$ *alors* $\beta = ubpv$ *(potentiellement en supprimant les* B *au début de* ub*).*
3. *Si* $\delta(q, a)$ *est indéfini alors aucun mouvement n'est possible depuis* α*, et* α *est une* **DI d'arrêt**. *Si* $q = q_F$ *alors* α *est une* **DI acceptante**.

Notation 4. *Notre modèle de MT est* **déterministe***, ce qui signifie que pour tout* α *il y a au plus un* β *tel que* $\alpha \vdash \beta$*. Nous noterons*

$$\alpha \vdash^* \beta$$

si la MT change α *en* β *en n'importe quel nombre d'étapes (0 inclus, auquel cas* $\alpha = \beta$*),*
$\alpha \vdash^+ \beta$ *si la MT change* α *en* β *en au moins une étape, et*
$\alpha \vdash^i \beta$ *si la MT change* α *en* β *en exactement* i *étapes.*

Pour tout $w \in \Sigma^*$ nous pouvons définir la DI correspondante

$$\iota_w = \begin{cases} q_0 w, & \text{si } w \neq \epsilon \\ q_0 B & \text{si } w = \epsilon. \end{cases}$$

2.2 Décider et calculer

Définition 5. *Le langage* **reconnu** *(ou* **accepté***) par la MT* M *est*

$$L(M) = \{w \mid w \in \Sigma^* \text{ et } \iota_w \vdash^* uq_F v \text{ avec } u, v \in \Gamma^*\}$$

Définition 6. *Un langage est* **récursivement énumérable** *(***r.e.***) s'il est reconnu par une machine de Turing. Un langage est* **récursif***, ou* **décibable***, s'il est reconnu par une machine de Turing qui s'arrête sur toutes les entrées.*

Attention à la différence ! Bien entendu, tout langage récursif est également r.e.

Notation 7. *Le* **résultat** *du calcul de la MT* M *sur l'entrée* w *sera noté*

$$M(w) = \begin{cases} uv & \text{si } \iota_w \vdash^* uq_F v \text{ avec } u, v \in \Gamma^* \\ uav & \text{si } \iota_w \vdash^* uqav \text{ avec } u, v \in \Gamma^* \text{ et } \delta(q, a) \text{ non défini}\dagger \\ \uparrow & \text{si l'exécution ne termine pas.} \end{cases}$$

†potentiellement en supprimant les B à la fin de av.

Définition 8. *Une fonction* $f : \Sigma^* \to \Gamma^*$ *est* **calculable** *ou* **récursive** *si et seulement si il existe une MT* M *telle que pour tout* $w \in \Sigma^* : f(w) = M(w)$.

Remarque 9. Décider (un langage) est équivalent à calculer (une fonction).
⇐ *Décider un langage* L *revient à calculer sa* **fonction caractéristique**

$$f_L : \begin{array}{rcl} \Sigma^* & \to & \{0, 1\} \\ w & \mapsto & \begin{cases} 1 & \text{si } w \in L \\ 0 & \text{sinon.} \end{cases} \end{array}$$

$\Rightarrow$ *Calculer une fonction f revient à décider le langage*

$$\{(x, y) \mid y = f(x)\}.$$

Nous n'établirons pas de distinction très nette entre calculer et décider.
— Dans la vraie vie on dira plutôt calculer (plus parlant).
— Dans le monde des mathématiques on dira plutôt décider (plus minimaliste, et s'applique aux propriétés).
— Récursif est synonyme de calculable et décidable.

Remarque 10. *Le terme* **récursivement énumérable** *(définition 6) vient du fait qu'il est possible d'écrire (pour ces langages) une MT qui va, à partir d'une entrée vide, énumérer tous les mots du langage, un à un et sans en oublier aucun (dans n'importe quel ordre, possiblement en répétant plusieurs fois certains mots). C'est-à-dire que nous aurons un* état spécial d'énumération q_e *(quand on entre dans cet état c'est qu'on énumère le mot présent sur le ruban, par convention à la droite de la tête de lecture/écriture) tel que :*

$$\textit{pour tout mot } w \in L\textit{, il existe une étape } t \textit{ telle que } \iota_\epsilon \vdash^t w' q_e w.$$

Exemple 11. *Le langage suivant est récursif :*

$$\{w \in \{a, b\}^* \mid w \textit{ est un palindrome}\}$$

donc il existe une machine $M_{palindrome}$ qui le décide (répond oui/non sur toute entrée).

2.3 Propriétés de clôture

Théorème 12. *Les propriétés suivantes sont vraies :*

1. *la famille des langages récursifs est close par complémentation ;*
2. *les familles des langages récursifs et récursivement énumérables sont closes par union et intersection ;*
3. *Un langage $L \subseteq \Sigma^*$ est récursif si et seulement si L et $\Sigma^* \setminus L$ sont récursivement énumérables.*

Idées de démonstration.

1. On veut prouver L récursif implique $\Sigma^* \setminus L$ récursif. Soit M la machine dont le langage est $L(M) = L$ et qui s'arrête toujours, nous allons construire une nouvelle machine M' dont le langage est $L(M') = \Sigma^* \setminus L$ et qui s'arrête toujours. Pour cela, on ajoute un puits global qui sera notre nouvel état final. Ainsi, la nouvelle machine s'arrête dans cet état final à chaque fois que M s'arrêtait sur un état non final ($w \notin L(M) \Rightarrow w \in L(M')$). De plus, chaque fois que M s'arrêtait dans l'état final, M' va dans un nouvel état non final à partir duquel aucune transition n'est possible ($w \in L(M) \Rightarrow w \notin L(M')$).
2. Intersection. Soient $L_1 = L(M_1)$ et $L_2 = L(M_2)$. On veut construire une machine M' dont le langage est $L(M') = L_1 \cap L_2$. Pour cela, sur une entrée w la machine M' simulera (pour cela il suffit de modifier l'état final des machines simulées) :
   — M_1 sur l'entrée w (on sait que le calcul termine),
   — puis M_2 sur l'entrée w (on sait que le calcul termine),

se souviendra du résultat de chaque simulation (arrêt dans l'état final ou non), et va dans l'état final si et seulement si M_1 et M_2 acceptent w (sinon M' va dans un nouvel état non final à partir duquel aucune transition n'est possible). Pour les langages r.e. on étudie en plus les cas où les machines M_1 et M_2 ne s'arrête pas.

3. Le sens $\Rightarrow$ est direct. Pour $\Leftarrow$, on simule en parallèle les deux machines qui reconnaissent L et $L \setminus \Sigma^*$ (mais ne s'arrête pas toujours). Puisque soit l'une soit l'autre accepte w, soit l'une soit l'autre entrera dans son état final, et nous pourrons alors :
   — entrer dans notre état final si c'est la machine qui reconnait L qui est entrée dans son état final,
   — entrer dans un nouvel état non final à partir duquel aucune transition n'est possible si c'est la machine qui reconnait $\Sigma^* \setminus L$ qui est entrée dans son état final.

□

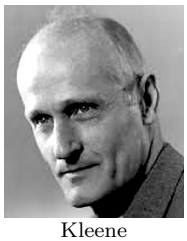
2.4 Un peu d'histoire


Cantor
(1845-1918)


Hilbert
(1862-1943)


Gödel
(1906-1978)


Church
(1903-1995)


Kleene
(1909-1994)


Turing
(1912-1954)


von Neumann
(1903-1957)

A la toute fin du XIX^{e} siècle, Georg Cantor définit les fondements de la théorie des ensembles, dont l'usage systématique (c'est-à-dire qui est utilisée dans tous les domaines) allait bouleverser les fondements de la logique mathématique. En 1900, pour fêter le passage au XX^{e} siècle, David Hilbert énonce 23 grands problèmes ouverts, dont le suivant : les propriétés qui s'expriment en langage mathématique sont-elles toutes décidables ? Si la réponse devait être affirmatives, les propriétés mathématiques valides seraient des théorèmes dérivables mécaniquement de quelques axiomes dans un système formel. Autrement dit : on pourrait remplacer les mathématicien·ne·s par des machines surpuissantes ! En 1931, Kurt Gödel met un terme à cette interrogation : il existe des propriétés mathématiques indécidables (dans tous les systèmes d'axiomes qui formalisent au moins l'arithmétique). Autrement dit : mathématicien·ne·s 1 - machines 0. Entre 1932 et 1936, Alsonso Church et Stephen Kleene proposent des modèles de calculs (le λ-calcul et les fonctions μ-récursives) qui semblent capturer la notion intuitive de fonctions calculables, mais il est un peu difficile de s'en convaincre... Notons tout de même que le λ-calcul est extrêmement minimaliste, ce qui rend sa compréhension mathématique fort intéressante : tout est capturé en quelques lignes de définition ! Indépendamment, en 1936, Alan Turing propose sa définition de machines. En 1937 il montre que la classe des fonctions λ-calculables est égale à la classe des fonctions programmables sur les machines de Turing. Les machines de Turing permettent de reformuler en termes intuitifs de calculs les résultats de Kurt Gödel (qui étaient exprimés en termes de démonstration). Avec l'aide de Von Neumann (et d'autres), les premiers ordinateurs programmables verront le jour quelques années plus tard !

La vie de Turing vaut le coup d'oeil (savez vous que le rôle de Turing durant la seconde guerre mondiale est resté secret d'Etat de nombreuses années ?).

e-penser (13') : `https://www.youtube.com/watch?v=7dpFeXV_hqs`

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.