Notes de cours d'Algorithmique : les barrières de l'informatique

Nantes, janvier 2020

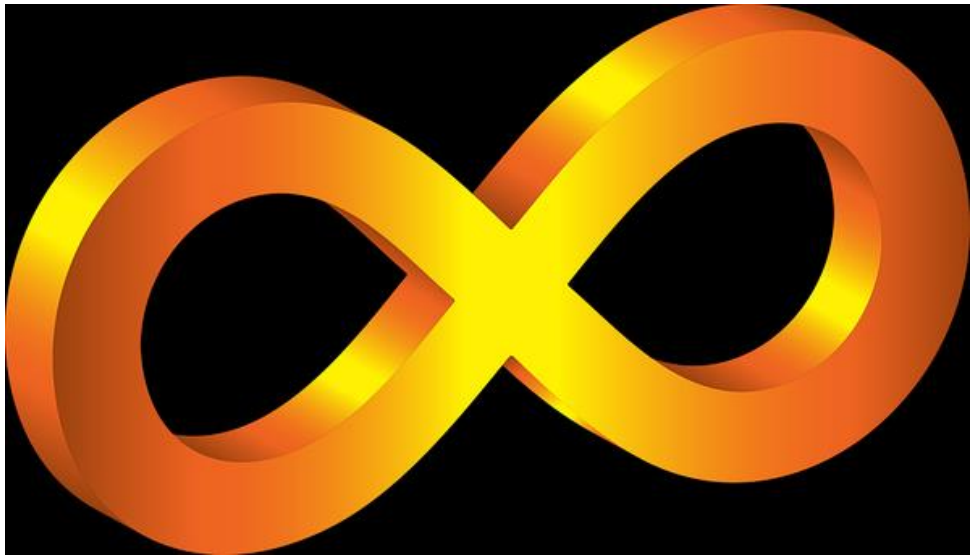
CC BY

Résumé

- L'informatique se base sur un modèle essentiel : la machine de Turing
- Ce modèle nous permet d'étudier différentes limites de l'informatique
- La représentation des données doit correspondre à un ensemble dénombrable (première limite)
- Le problème doit être décidable (deuxième limite)
- Le problème devrait pouvoir se résoudre en temps polynomial (troisième limite)

Avis pour les moins matheux

- Il faut prendre ce cours comme une introduction au sublime. La plupart des acteurs (mathématiciens/informaticiens) cités dans ce cours sont devenus fous. Pour avoir voulu comprendre l'infini…



Cantor et l'infini non dénombrable

Il était une fois les ensembles finis

- et certains ensembles finis ont la même **cardinalité** :
- {Alfred, Béa, Colin, Denis, Emilie, Fabienne, Guillaume} et {lundi,…,dimanche} ont la cardinalité 7
- L'ensemble vide a pour cardinalité 0

- Quelle est la cardinalité de $\mathbb{N}$?

- L'infini !
- Et $\mathbb{R}$?
- L'infini ?
- Cantor a démontré que c'était un peu plus compliqué que ça

Définitions

- Deux ensembles A et B sont **équipotents** s'ils ont la même **cardinalité**, c'est-à-dire s'il existe une bijection de A sur B
- Rappel : une bijection donne un appariement 1-1
- (c'est aussi une application injective et surjective)

- On peut décider de représenter chaque ensemble équipotent à {0,1,…*n*-1} par son cardinal *n*
- Notations usuelles : Card(E) ou |E|

Définitions

- Pour $\mathbb{N}$ inventons un **cardinal** pour exprimer sa cardinalité : $\aleph_0$ (aleph 0)
- Et on peut aussi dans ce cas parler d'équipotence
- On démontre (exercice) que les ensembles suivants sont équipotents à $\mathbb{N}$ (et ont donc cardinalité $\aleph_0$)
  - $\mathbb{N} \cup \{*\}$
  - $\mathbb{Z}$
  - $\mathbb{N}^2$
  - $\mathbb{Q}$
- Enfin, tous ces ensembles, ainsi que les ensembles finis sont dits **dénombrables** car on peut énumérer les éléments (le 12^e, le 477^e, le 9453382916^e)
- Remarque : si notre énumération a des trous ce n'est pas grave

Démonstration pour $\mathbb{Z}$

- Il suffit de pouvoir énumérer $\mathbb{Z}$
- On construit une fonction $\mathbb{Z} \mapsto \mathbb{N}$
- Si $x<0$ $f(x)=2|x|-1$
- Si $x \geq 0$ $f(x)=2x$

- Ainsi f(-5)=9, f(16)=32, f(0)=0
- On démontre (si, si…) que c'est une bijection
- On peut par exemple (exercice…) construire la fonction inverse

Paradoxe de l'hôtel infini

- Vous êtes le propriétaire d'un hôtel. Toutes vos chambres sont occupées. Un client arrive :
  - « bonjour, je voudrais une chambre »
  - « je suis désolé l'hôtel est complet »
- Votre hôtel est maintenant un hôtel infini. Vous avez $\aleph_0$ chambres qui sont donc numérotées 0, 1, 2,…
- Un client arrive :
  - « bonjour, je voudrais une chambre »
  - « je suis désolé l'hôtel est complet. Mais j'ai une solution »
- Voyez-vous quelle solution vous avez ?

La solution

- Par l'interphone vous demandez à chaque client de changer de chambre et d'aller dans celle de numéro supérieur
- La chambre #0 s'est ainsi libérée pour votre client
- Vous venez de construire une bijection de $\mathbb{N}$ sur $\mathbb{N} \cup \{*\}$

- Question : supposons que maintenant ce soit un bus infini de nouveaux clients qui arrive…

Passons à $2^{\mathbb{N}}$

- Considérons maintenant $2^{\mathbb{N}}$, l'ensemble des parties de $\mathbb{N}$
- A-t-il la même cardinalité que $\mathbb{N}$?
- Non.
- Preuve par l'absurde

Preuve par l'absurde (parenthèse pour les non mathématiciens)

- Une preuve par l'absurde se fait de la façon suivante

1. On émet une hypothèse (qu'on aimerait réfuter) H
2. On va suivre des implications logiques (le *modus ponens*) pour passer de proposition en proposition jusqu'à une conclusion.
3. $H \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \ldots \Rightarrow P_n \Rightarrow C$
4. Si la conclusion est fausse ou entraine une contradiction (absurde), il en résulte que P_n est fausse aussi. Et de proche en proche P_{n-1},…., P_1 et enfin H.

- Note : c'est très compliqué de « comprendre » une preuve par l'absurde puisque justement on ne manipule que des arguments faux.

Passons à $2^{\mathbb{N}}$

- Considérons maintenant $2^{\mathbb{N}}$, l'ensemble des parties de $\mathbb{N}$
- A-t-il la même cardinalité que $\mathbb{N}$?
- Non.
- Preuve par l'absurde : supposons que $2^{\mathbb{N}}$ peut être mis en bijection avec $\mathbb{N}$. Alors on peut énumérer $2^{\mathbb{N}}$: P_0, P_1, P_2,…
- Considérons maintenant l'ensemble $D=\{i \in \mathbb{N} : i \notin P_i\}$
- Comme D est un sous ensemble de $\mathbb{N}$, D est un élément de l'énumération P_0, P_1, P_2,…
- Supposons (sans perte de généralité) que c'est P_k.
- Avons-nous $k \in P_k$?

Les deux cas

- Si $k \in P_k$ alors par définition de D, $k \notin D$, et comme $D=P_k$ il en résulte que $k \notin P_k$. C'est une contradiction.

- Si $k \notin P_k$ alors par définition de D, $k \in D$, et comme $D=P_k$ il en résulte que $k \in P_k$. C'est aussi une contradiction.

- Donc dans les deux cas possibles, on aboutit à une contradiction.
- Seule explication : l'hypothèse de départ « on peut énumérer $2^{\mathbb{N}}$ » était fausse.

La diagonalisation

	0	1	2	3	4	…		j	…						
P_0	0	0	0	1	0	…		1	…						
P_1	0	1	0	0	1	…		0	…						
P_2	1	1	1	0	0	…		1	…						
…	…														
P_i	0	0	0	0	1			1							
…			…												

Signifie que $j \in P_i$

La diagonalisation

	0	1	2	3	4	…		*i*	…						
P_0	0	0	0	1	0	…		1	…						
P_1	0	1	0	0	1	…		0	…						
P_2	1	1	1	0	0	…		1	…						
…	…														
P_i	0	0	0	0	1			1							
…			…												

Signifie que $i \in P_i$

La diagonalisation

	0	1	2	3	4	…		*i*	…						
P_0	0	0	0	1	0	…		1	…						
P_1	0	1	0	0	1	…		0	…						
P_2	1	1	1	0	0	…		1	…						
…	…														
P_i	0	0	0	0	1			1							
…			…												

La diagonale

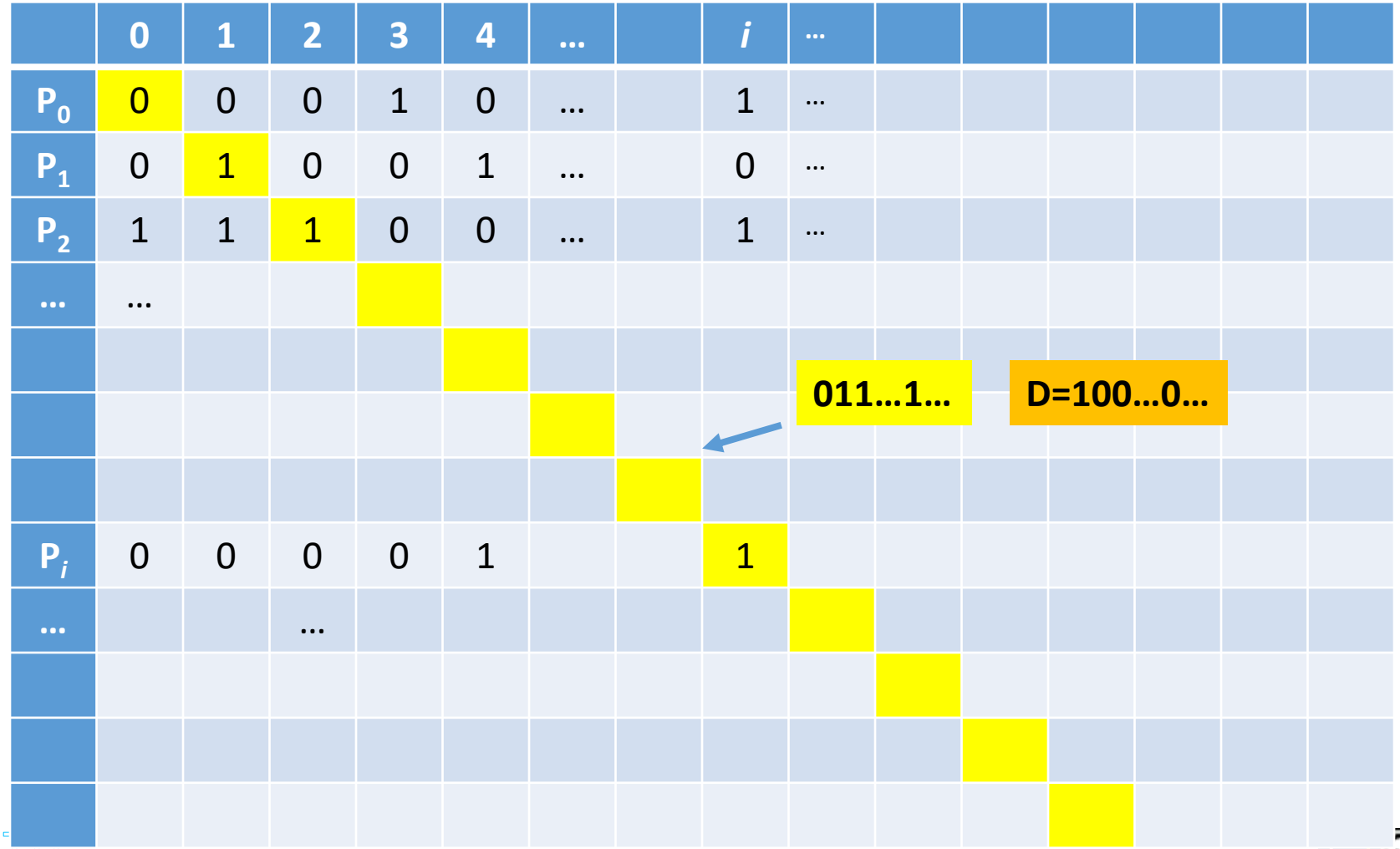
La diagonalisation

La diagonalisation

	0	1	2	3	4	…		*i*	…						
P_0	0	0	0	1	0	…		1	…						
P_1	0	1	0	0	1	…		0	…						
P_2	1	1	1	0	0	…		1	…						
…	…														
P_i	0	0	0	0	1			1							
…			…												

011…1… D=100…0…

La diagonalisation
D=100…0… ne peut correspondre à aucune ligne



Et...

- Ca nous oblige à inventer une nouvelle cardinalité pour $2^{\mathbb{N}}$.
- C'est $\aleph_1$.
- Et…
- On démontre que Card($\mathbb{R}$)= $\aleph_1$.

- Idée de la preuve : comme pour $2^{\mathbb{N}}$. Mais il faut

1. Se restreindre à [0,1]
2. Écrire les réels comme 0,….
3. Considérer le nouveau réel obtenu en modifiant la $i^{\text{ème}}$ décimale

Greg Cantor (1845-1918)

- Il est devenu fou… bien entendu

- Remarques. Parmi les problèmes qui fascinent les mathématiciens, l'hypothèse du continu s'énonce comme : y a-t-il quelque chose entre $\aleph_0$ et $\aleph_1$?

Conséquences pour l'informaticien(ne)

- Ce qui est compliqué pour $2^{\mathbb{N}}$ et $\mathbb{R}$ est qu'il existe des éléments qu'on n'a pas la capacité d'écrire de façon finie
  - Certains ensembles qui ne sont pas générés par un algorithme
  - Les nombres transcendants

- Il y a donc là une première limite de l'informatique : on ne peut travailler que sur les ensembles dénombrables (*)
- En particulier on ne peut pas travailler sur les réels (la solution est de les approximer –par exemple avec les nombres flottants)

(*) ensembles dénombrables ou ensembles finis ?

- Travailler sur les ensembles finis signifie arrêter la taille de l'ensemble. Ce qu'on ne veut pas faire.
- Si on écrit un algorithme pour déterminer si un nombre premier l'algorithme est écrit pour tout nombre, quelle que soit sa taille. Même si le nombre n'entre pas dans la mémoire d'un ordinateur !

Les machines de Turing, quelques questions générales

Pour en savoir plus

- L'article de Turing « Computing Machinery and Intelligence » se trouve facilement en ligne et se lit très agréablement. Peut-être la seule lecture obligatoire ! Il est existe une traduction française tout à fait bien écrite !
- On peut trouver des simulations de machines de Turing en ligne:

- https://videotheque.cnrs.fr/index.php?id_doc=3001&urlaction=doc
- https://interstices.info/comment-fonctionne-une-machine-de-turing/

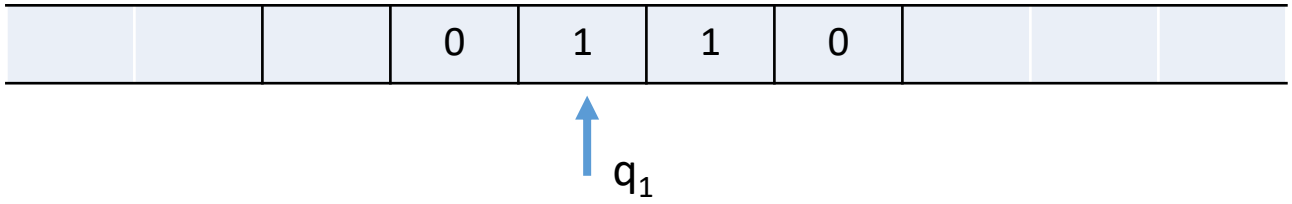
Les objets de la machine de Turing

- Un ensemble infini dénombrable d'**états** q_0, q_1, q_2,…
- Certains états sont dits **acceptants**
- Une **bande de lecture** écriture infinie composée d'une infinité dénombrable de cases (on imagine un ruban)
- Une **tête de lecture/écriture**
- Un **alphabet** : on peut se limiter à 2 valeurs 0/1 et au vide noté ⊡
- Un ensemble fini de règles

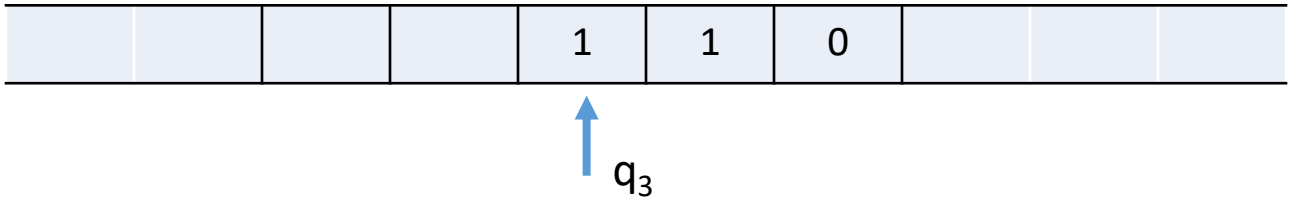
- Une règle (q, a, q', b, S)
  - q et q' sont des états
  - a et b sont des symboles (0, 1 ou ⊡)
  - S est le sens : **G**auche, **D**roite ou **I**mmobile

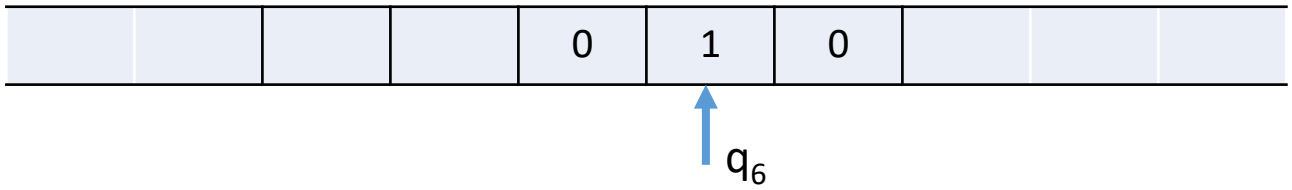
Configurations

- A tout moment la machine est dans une **configuration** : on est dans un état, la tête de lecture/écriture pointe sur une cellule



Règles





- La règle (q_3,1,q_6,0,D) s'interprète comme :

Si je suis dans l'état q_3 et que je lis un 1, je remplace le 1 par le 0 et je déplace vers la droite ma tête de lecture. Puis je passe dans l'état q_6

Calcul d'une machine de Turing

- La machine de Turing commence dans l'état q_0 avec un mot (fini) sur sa bande (sa donnée) et la tête de lecture/écriture sur le premier symbole différent de ⊡.
- Un **calcul** est une suite de changements de configurations valides
- Un calcul **s'arrête** si dans la configuration, aucune règle n'est applicable
- Un mot est **accepté** si un calcul s'arrête en partant de ce mot et l'état est acceptant

Fonctionnement de la machine de Turing

Soit la machine suivante.

- $(q_0,1,q_0,1,D)$
- $(q_0,0,q_0,0,D)$
- $(q_0, \boxdot,q_1, \boxdot,G)$
- $(q_1, 0,q_3, 1,I)$
- $(q_1, 1,q_2, 0,G)$
- $(q_2, 0,q_3, 1,I)$
- $(q_2, 1,q_2, 0,G)$
- $(q_2, \boxdot,q_3, 1,I)$

Que fait cette machine ?

Fonctionnement de la machine de Turing

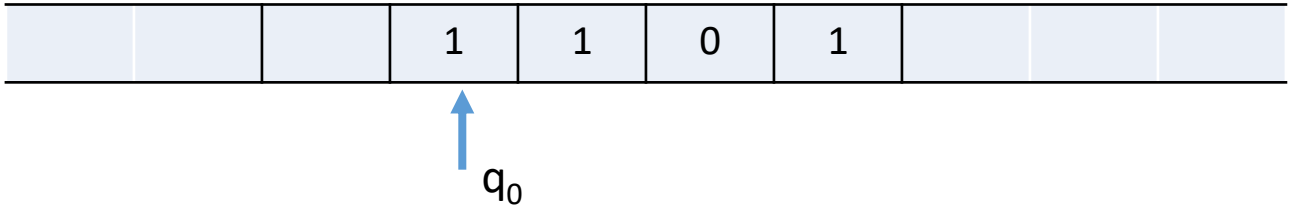
Soit la machine suivante.

- $(q_0,1,q_0,1,D)$ //on se déplace vers la fin du nombre
- $(q_0,0,q_0,0,D)$ //on se déplace vers la fin du nombre
- $(q_0, \boxdot,q_1, \boxdot,G)$ //on revient sur le dernier bit
- $(q_1, 0,q_3, 1,I)$ // si le dernier bit est un 0 on ajoute 1 et fin
- $(q_1, 1,q_2, 0,G)$ // si le dernier bit est un 1 on passe à 0 et on retient 1
- $(q_2, 0,q_3, 1,I)$ // si le bit est un 0 on passe à 1 et fin
- $(q_2, 1,q_2, 0,G)$ // si le bit est un 1 on passe à 0 et on retient 1
- $(q_2, \boxdot,q_3, 1,I)$ // si on est revenu au début c'est que le nombre était 1111 et il faut mettre un 1 devant.

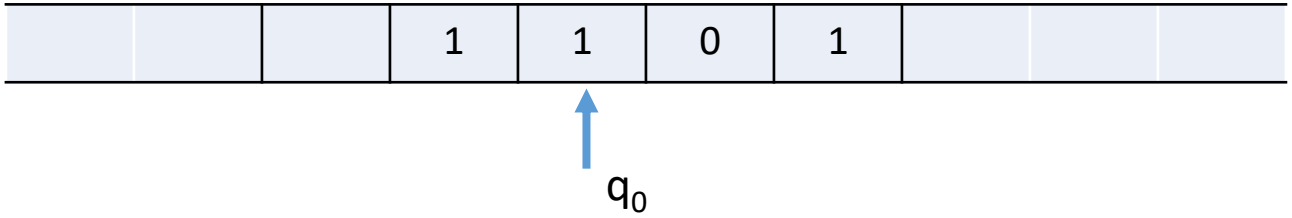
Elle ajoute 1

Vérifions (1)

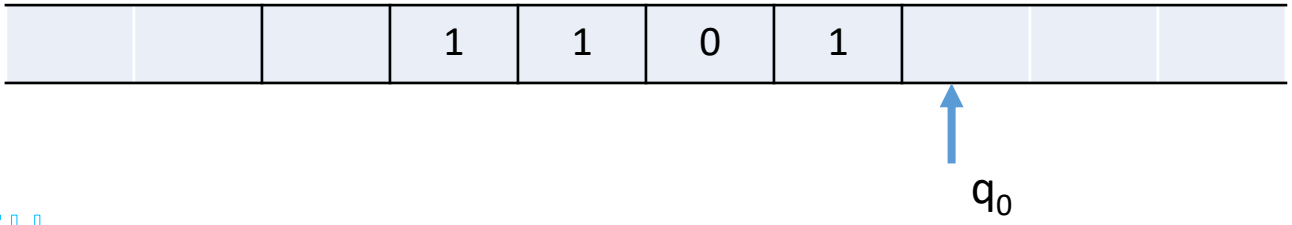
- On commence avec le codage binaire de 12



- On applique la règle $(q_0,1,q_0,1,D)$

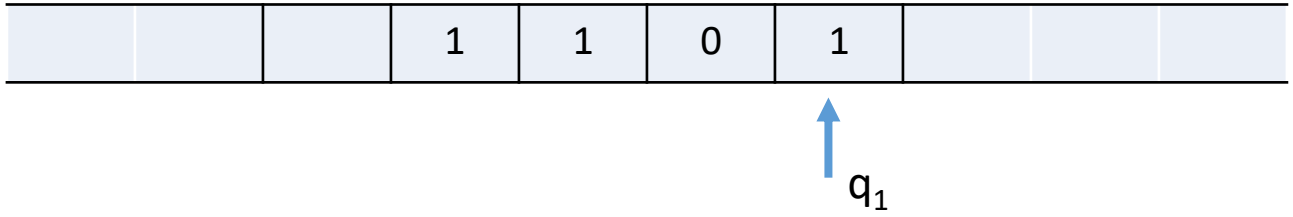


- Puis les règles $(q_0,1,q_0,1,D)$ $(q_0,0,q_0,0,D)$ $(q_0,1,q_0,1,D)$

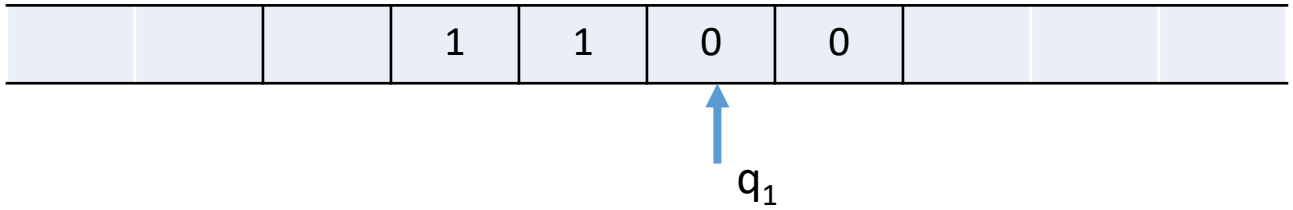


Vérifions (2)

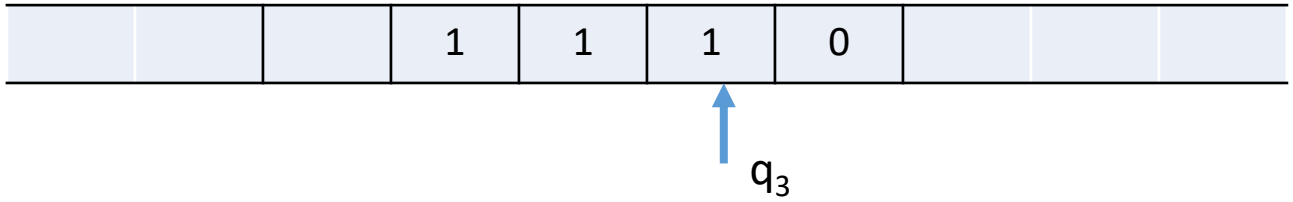
- On est allé « trop loin ». On revient en changeant d'état avec la règle (q_0, ⊡,q_1, ⊡,G)



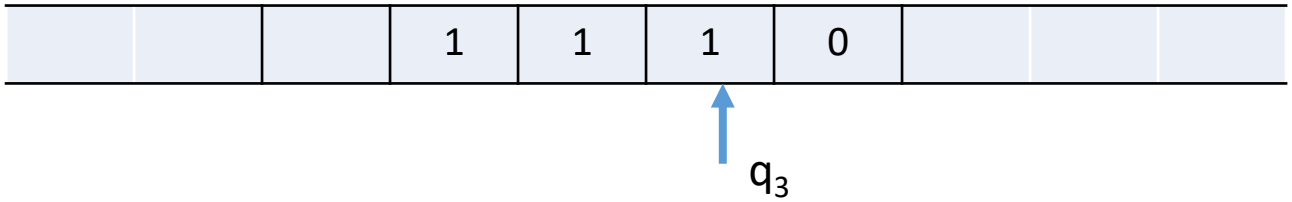
- La règle qui s'applique maintenant est (q_1,1,q_1,0,G)



- Puis la règle (q_1,0,q_3,1,I)

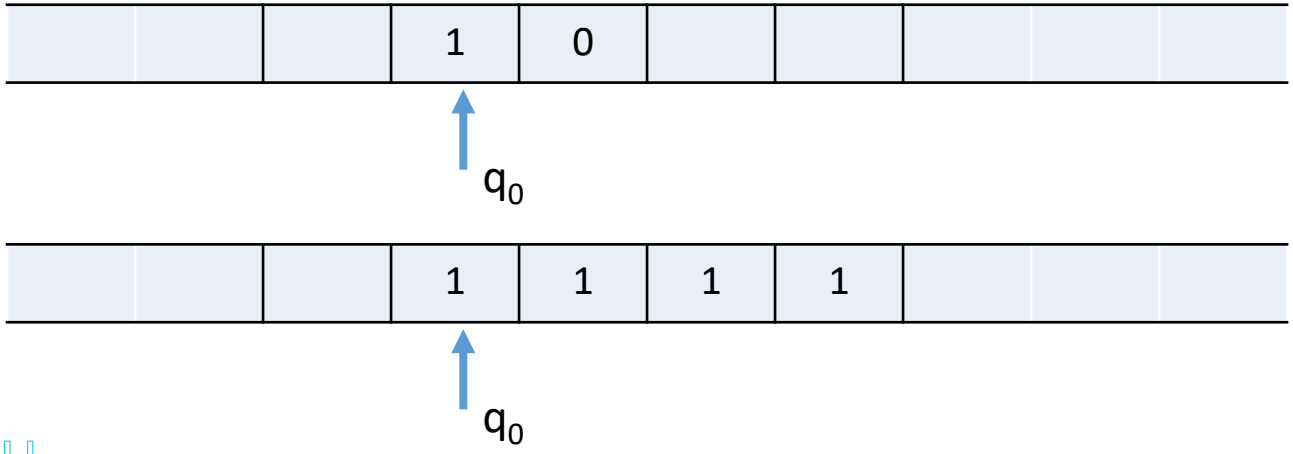


Vérifions (3)



- Dans cette configuration aucune règle ne s'applique, la machine de Turing s'arête avec 1110 (donc 14)

- On pourra tester avec les configurations de départ suivantes :

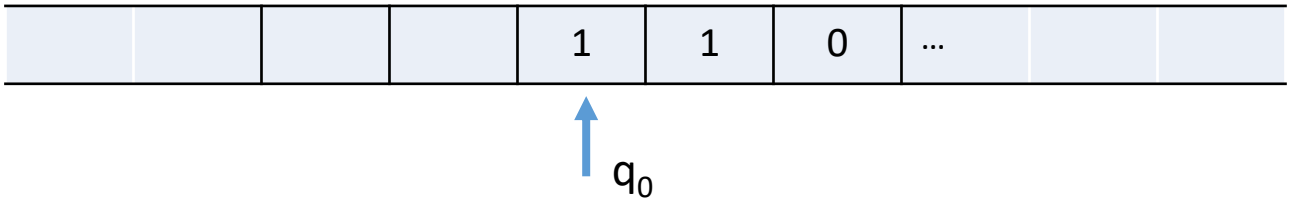


Attention au non déterminisme

- Il peut y avoir plusieurs calculs possibles
- C'est le non déterminisme

- Une Machine de Turing est déterministe si, quelle que soit la configuration, au plus une règle s'applique

- Donc une MT commence dans la configuration



- Et se met à calculer

Langage accepté par une MT

- Un mot est accepté s'il existe un calcul valide qui termine dans un état fini.

- Le **langage accepté** est celui de tous les mots acceptés.

- Cas non déterministe :
- Si un des calculs permet d'arriver à un état final et s'arrêter, le mot est accepté.

Les langages

- Supposons que l'**alphabet** est fixé.
- Il est composé de **symboles**.
- Un **langage** est un ensemble de mots
- Un **mot** est une séquence de symboles

- Ainsi, en fixant l'alphabet à {0,1} on peut parler du langage composé des nombres pairs L_{pair} = {0, 10, 100, 110,1000…}

Un langage récursif

- est un langage pour lequel il existe une MT qui termine sur toutes ses entrées et le fait dans un état acceptant si m $\in$ L
- En termes informatiques : un langage est récursif s'il existe un programme qui s'arrête toujours et résout la question m $\in$ L.
- Exemple : L_{pair} est récursif
- Question : comment sait-on que le programme s'arrête toujours ?
- Parce qu'il est écrit sans récursivité ni boucle `tant que`

Un langage récursivement énumérable

- On a une MT qui a un calcul acceptant chaque fois que $m \in L$.
- Si $m \notin L$, on ne sait pas. La MT peut s'arrêter (mais pas dans un acceptant) ou ne pas s'arrêter

L_{pair} est un langage récursivement énumérable

L_{pair} peut être reconnu par la MT suivante (avec l'état acceptant q_1)

- $(q_0,1,q_0,1,D)$
- $(q_0,0,q_0,0,D)$
- $(q_0, \boxdot,q_1, \boxdot,G)$
- $(q_1, 1,q_1, 1,I)$

- Que fait cette machine ?
- Comme dans l'exemple précédent, on va jusqu'à la fin, puis on revient en arrière et on regarde le dernier bit :
  - Si c'est un 0 aucune règle ne s'applique donc la MT s'arrête dans l'état acceptant q_1
  - Si c'est un 1 on ne change ni l'état ni le bit et donc la règle s'appliquera indéfiniment

L_{pair} est un langage récursif

L_{pair} peut être reconnu par la MT suivante (avec l'état acceptant q_1)

- $(q_0,1,q_0,1,D)$
- $(q_0,0,q_0,0,D)$
- $(q_0, \boxdot,q_1, \boxdot,G)$
- $(q_1, 1,q_2, 1,I)$

- Que fait cette machine ?
- Comme dans l'exemple précédent, on va jusqu'à la fin, puis on revient en arrière et on regarde le dernier bit :
  - Si c'est un 0 aucune règle ne s'applique donc la MT s'arrête dans l'état acceptant q_1
  - Si c'est un 1 on change d'état aucune règle ne s'applique donc la MT s'arrête mais n'est pas acceptant donc le mot est rejeté (et n'est pas dans L_{pair})

Les bons problèmes : les problèmes de décision

- Un problème de décision est défini par un ensemble d'instances, et une question. Sur chaque instance la réponse doit être **oui** ou **non**
- Par exemple : l'ensemble des entiers positifs. Le problème sur un entier *n* est « *n* est-il premier ? »
- Par contre le problème « quel est le double de *n* ?» n'est pas un problème de décision.
- Mais il peut être résolu en résolvant plusieurs problèmes de décision du genre « est ce que le $i^{\text{ème}}$ digit de 2*n* est un 1 ? »

Lien entre langages et problèmes

- Soit un problème de décision. On peut parler du langage de toutes les instances positives

- Inversement, soit un langage L. On lui associe le problème de décision « $m \in L$? »

- Et on dira que le problème est décidable si le langage associé est récursif
- Le problème est semi-décidable si le langage associé est récursivement énumérable

Thèse de Church

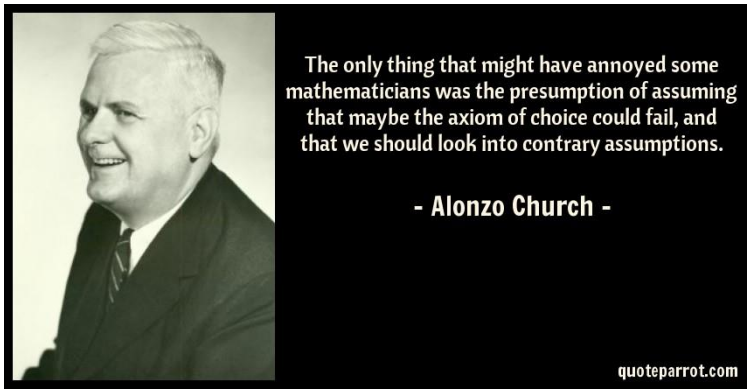
- Tout problème qui peut être résolu peut l'être par une machine de Turing
- Tout traitement systématique réalisable par un processus physique ou mécanique, peut être exprimé par une Machine de Turing

- Autrement dit : tout ce que le plus puissant des ordinateurs peut faire, une (simple) machine de Turing peut le faire aussi.

Pourquoi la thèse de Church serait elle vraie ?

- On ne peut pas espérer la prouver
- Par contre on pourrait la réfuter
- Elle résiste depuis 80 ans
- Des modèles très différents se sont avérés tous équivalents

- Alonzo Church (1903-1995)



Une machine qui s'emballe…

- C'est quoi une MT qui ne s'arrête pas ?

- **Cas 1 : elle boucle**

i ← 1

Tant que i<10 faire:

    Écrire(i)

- **Cas 2 : elle diverge**

i ← 1

Tant que i<10 faire:

    i ← i-1

Récapitulons

- Un langage L est **récursif** si le problème $m \in L$ est **décidable** si la fonction f(m)=1 si $m \in$ L, f(m)=0 sinon est **calculable**.
- Un langage L est **récursivement énumérable** si le problème $m \in L$ est **semi-décidable**.

- Et la question se pose :
  - Y a-t-il des langages non récursifs ? Non récursivement énumérables ?
  - Y a-t-il des fonctions non calculables ?
  - Y a-t-il des problèmes indécidables ?

Le problème de l’arrêt

Pour parler de la machine de Turing #i

- Comment faire pour énumérer les machines de Turing ?
- Il faut coder une machine de Turing en machine
- $(q_i, a) \rightarrow (q_j, b, s)$ est codé par 1 .. 1 0 1 .. 1 0 1 .. 1 0 1 .. 1 0 1 .. 1

1=1 1=1 G=1

i +1 0=11 j +1 0=11 D=11

⊡=111 ⊡=111 I=111

- Par exemple la règle $(q_3, 1) \rightarrow (q_0, \boxdot, D)$ est codée par 11110101011101 1
- Mais certaines suites de 0 et de 1 ne correspondent pas à des MTs !

Pour parler de la machine de Turing #i

- On classe ensuite les règles par ordre lexicographique (celui du dictionnaire)
  - Soient les règles r_1, r_2,…,r_k
  - On code la MT comme $r_1 00 r_2 00 \ldots 00 r_k$

- Remarque : codage unique. Les séquences de 00 permettent de séparer les règles
- Mais certaines suites de 0 et de 1 ne correspondent pas à des MTs !
- Ce code peut aussi être interprété comme un entier et on peut maintenant parler de la MT #323
- On peut écrire (en python !) un décodeur Décode(i)

Résumé

1. On peut attribuer un numéro à chaque MT
2. On peut le faire effectivement (avec un algorithme)
3. On peut donc énumérer les MTs

Pour énumérer toutes les MTs

i ←0

j←0

boucler (on s'arrête quand on veut):

    si i est le code d'une MT:

        Ecrire(« la MT »,j, « est », décoder(i))

        j ← j+1

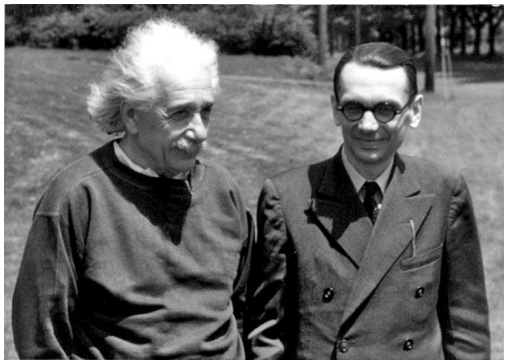
    i ← i+1

Problème de l’arrêt

- La MT #i boucle-t-elle sur la donnée i ?
- Il fallait y penser à ce problème…

Kurt Gödel (1906-1978)

- En 1931 il signe l'arrêt de mort de l'ambitieux programme des mathématiciens de l'époque en démontrant qu'on ne peut pas tout prouver. Il existe des théorèmes (donc des propositions vraies) qui ne peuvent pas être prouvées (*). C'est le théorème d'incomplétude.
- Notons la similarité avec le problème qui nous intéresse : on ne peut pas tout calculer.
- Il est devenu fou.



- (*) *c'est bien entendu un peu plus compliqué que ça mais on peut trouver de nombreux livres et articles qui nous expliquent ça.*

La MT #*i* s'arrête-t-elle sur la donnée *i* ?

- Notons $M_i(j)\circlearrowleft$ la proposition « la machine de Turing M_i ne s'arrête pas quand on lui donne la donnée *j* »
- Inversement $M_i(j)\downarrow$ est la proposition « la machine de Turing M_i s'arrête quand on lui donne la donnée *j* »
- Le problème qui nous intéresse est de savoir si une MT **ne** s'arrête **pas** sur son propre code.
- Supposons (l'hypothèse à réfuter) que ce problème soit décidable. Il existe donc une MT T qui prend en entrée un *i* et répond non quand $M_i(i)\downarrow$ et oui quand $M_i(i)\circlearrowleft$. Maintenant, transformons T en T' de façon à la faire boucler quand elle découvre que $M_i(i)\downarrow$.
- Ainsi $T'(i)\downarrow$ quand $M_i(i)\circlearrowleft$ et $T'(i)\circlearrowleft$ quand $M_i(i)\downarrow$
- Comme T' est une Machine de Turing, elle a un numéro. Soit *k* ce numéro.
- Que se passe-t-il si on donne *k* comme donnée à T' ?

Les deux cas

- Si T'(k)↓. Dans ce cas, comme T'= M_k on a M_k (k)↓. Mais on a aussi T'(k) ↺ par définition de T'. Contradiction !
- T'(k) ↺. Dans ce cas, comme T'= M_k on a M_k (k) ↺. Mais dans ce cas T'(k)↓. Contradiction !

- Aucun des deux cas ne tient. C'est donc l'hypothèse qui est fausse : la machine de Turing n'existe pas, et notre problème n'est pas décidable.

Ce que nous venons de démontrer

- Intuitivement, nous avons démontré qu'il est impossible de tester si une machine boucle sans le tester, mais comme cela prend un temps infini, on ne peut pas.

- Plus informatiquement nous espérions trouver un programme qui permettrait de savoir si un programme va planter sans le faire planter.

Le théorème de Rice

Peut-être le plus fascinant des théorèmes informatiques…

- Un problème de décision est trivial si la réponse est Vrai sur toutes les instances ou Faux sur toutes les instances.
- Un problème sur une MT est sur le fond (et non sur la forme) si ce problème concerne le résultat de la MT (*).
  - « La MT *i* contient-elle 3 instructions » est un problème de forme
  - « La MT *i* retourne-telle 0 sur l'input 327 » est un problème de fond

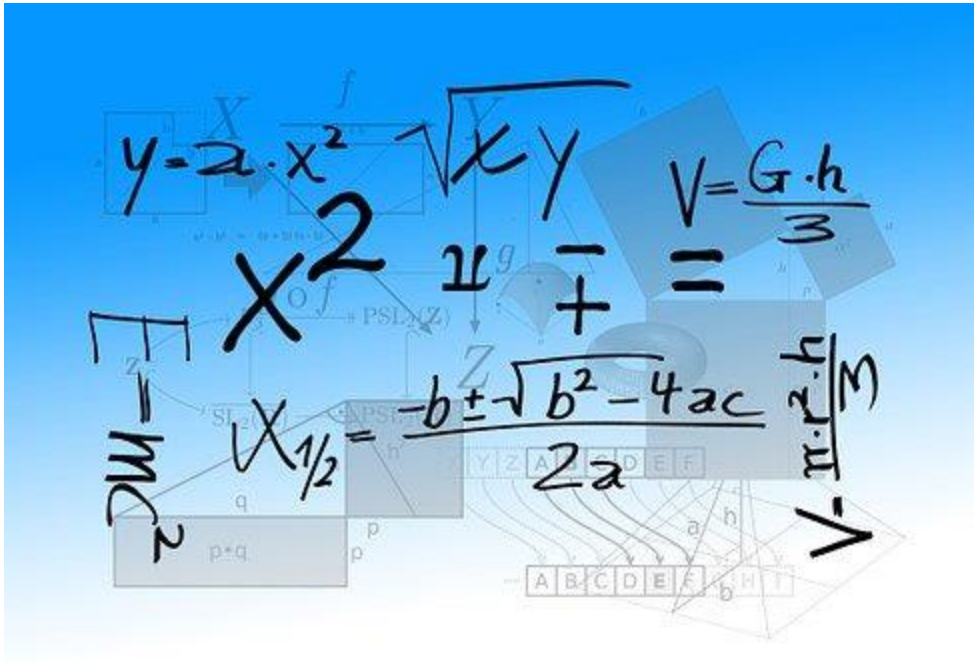
Théorème de Rice (1953)

- Si un problème de fond n'est pas trivial il est indécidable.

- Preuve : celle de l'article de Wikipedia se suit assez bien
- (*) On parle de *sémantique*

Les conséquences pour les informaticien(ne)s

- L'analyse statique des programmes a des limites : on ne peut pas savoir ce que fait un programme sans le faire tourner
- La preuve automatique de programmes est difficile
- Il n'existe pas d'anti-virus totalement efficace : un anti-virus qui regarde si une signature est présente se contente de regarder la forme (et donc le théorème de Rice n'est pas concerné). S'il regarde le fond (« ce que fait ce code est-il dangereux ? ») il suffit d'appliquer le théorème de Rice pour voir que c'est impossible.



Les modèles de calcul

Un modèle de calcul…

- … est un ensemble de règles permettant de simuler les calculs.
- Le modèle de calcul est **Turing complet** s'il est aussi puissant que les Machines de Turing

- Remarque : si on construisait un modèle de calcul raisonnable qui serait plus puissant que les MTs, cela signifierait que la thèse de Church est fausse

Et les machines quantiques ?

- Les machines quantiques ne remettent pas en cause la thèse de Church
- Des obstacles physiques…
- Pire, on peut penser que les ordinateurs quantiques sont moins puissants que les machines de Turing

Et les machines analogiques

- Non plus. L'argument essentiel est qu'on ne peut pas gérer le bruit
- https://fr.wikipedia.org/wiki/Calculateur_analogique

Conclusion

A retenir

- Avant de résoudre un problème par algorithme il faut pouvoir coder les données. Ce qui n'est possible que si on travaille sur des ensembles dénombrables.
- Avec la Machine de Turing on peut tout faire
- Il existe des problèmes décidables (la MT s'arrête toujours) et des problèmes indécidables
- Le « 1er » problème indécidable est le problème de l'arrêt
- Mais on peut en construire de nombreux autres en appliquant le théorème de Rice.
- Et même si le problème est décidable on peut peut-être pas le résoudre en temps polynomial…. A suivre !!!!