

Modèles de calcul

Année 2009–2010

M1, Univ. Bordeaux

18 septembre 2009

Objectifs (fiche UE)






Définir, **indépendamment de la technologie** :

- ▶ ce qui est **calculable** et ce qui ne l'est pas (théorie de la calculabilité) ;
- ▶ ce qui est calculable **efficacement** et ce qui ne l'est pas (théorie de la complexité).




Modalités du cours

- ▶ 12 cours, 3 groupes de TD (débutent le 14/9/09).
- ▶ N. Saheb, G. Sénizergues, A. Zvonkine.
- ▶ Contrôle continu (CC) obligatoire sauf dispense.
- ▶ Note finale session 1 :
 $\frac{2}{3}$ Examen (3h) + $\frac{1}{3}$ CC.
- ▶ Note finale session 2 :
 $\max(\text{Examen}, \frac{2}{3} \text{ Examen (3h)} + \frac{1}{3} \text{ CC})$.

Bibliographie

-  [J.E. Hopcroft, R. Motwani, J. D. Ullman.](#)
Introduction to Automata Theory, Languages & Computation.
[Addison-Wesley, 2005.](#)
-  [M. Sipser.](#)
Introduction to the Theory of Computation.
[PWS publishing Company, 1997.](#)
-  [O. Carton.](#)
Langages formels, Calculabilité et Complexité.
[Vuibert, 2008.](#)
-  [J.M. Autebert.](#)
Calculabilité et Décidabilité.
[Masson, 1992.](#)
-  [P. Wolper.](#)
Introduction à la calculabilité.
[InterÉditions, 1991.](#)

Bibliographie complémentaire

-  Ch. Papadimitriou.
Computational complexity.
Addison-Wesley, 1995.
-  M. Garey, D. Johnson.
Computers and intractability.
W.H. Freeman & Co, 1979.
-  J. E. Savage.
Models of computation.
Addison-Wesley, 1998.

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP. NP-complétude

Fonction récursives

Plan du cours

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP. NP-complétude

Fonction récursives

Questions abordées dans ce cours

- ▶ Quels problèmes peut-on résoudre avec une machine (indépendamment de puissance de calcul des machines) ?
- ▶ Comment formaliser
 - ▶ ce qu'est un problème ?
 - ▶ ce qu'est une machine ?
- ▶ Quelles fonctions peut-on calculer avec une machine ?
- ▶ Comment comparer la complexité des problèmes ?
Y a-t-il des problèmes inhéremment difficiles ?

Bref historique

- 1900 Hilbert, 10^{ème} problème : peut-on décider, de façon mécanique, si une équation Diophantienne a une solution ?
- 1931 Gödel publie le théorème d'incomplétude.
- 1935 Turing formalise une définition de machine et de calcul.
- 1936 Church exhibe un problème non résoluble par machine.
- 1938 Kleene prouve l'équivalence entre machines de Turing, λ -calcul, fonctions récursives.
- 1947 Post & Markov prouvent qu'un problème posé par Thue en 1914 n'est pas résoluble mécaniquement.
- 1970 Matiyasevich répond négativement au 10^{ème} problème de Hilbert.
- 1971 Cook & Levin formalisent la notion de problème NP-complet.

Pourquoi ce cours (ou la théorie en général) ?

- pour comprendre les limites de la programmation, mais aussi...
- ... pour découvrir un côté esthétique du calcul - la théorie apporte le plus souvent un point de vue plus simple et plus élégant,
- ... pour chercher la pérennité : la technologie évolue rapidement, mais les concepts de base (théoriques) restent valides,
- ... pour entrainer l'abileté d'analyser et/ou décrire un problème avec clarté.

Bref historique (info fondamentale)

- 1936 la calculabilité émerge de la logique (Russel, Hilbert, Turing, Church sont logiciens)
- 1941 Zuse invente le premier ordinateur (Z3) ; suit ENIAC 1945 ; modèle de von Neumann
- 1950 premiers langages de programmation (Fortran, Cobol, Lisp)
- 1960 développement de la théorie des langages formels et des modèles de calcul (hiérarchie de Chomsky)
- 1970 développement de la théorie de la complexité, question P vs. NP.
- 70/80 algorithmique et structures de données
- 80/90 algorithmique parallèle, distribuée ; cryptographie

Exemples de problèmes de décision

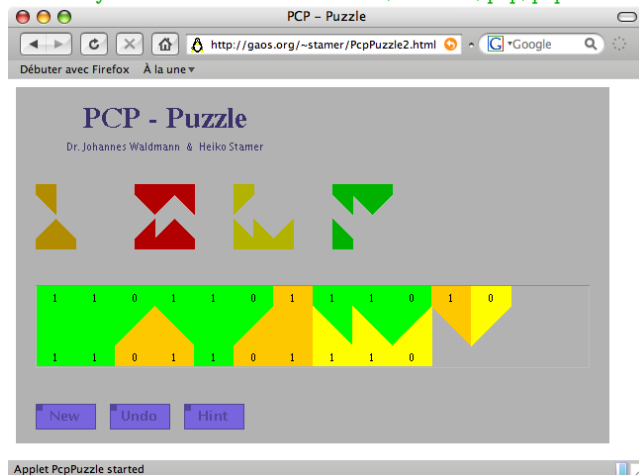
- | | |
|------------|--|
| Problème 1 | Donnée Un nombre entier positif n en base 2.
Question n est-il pair ? |
| Problème 2 | D. Un nombre entier positif n en base 10.
Q. n est-il premier ? |
| Problème 3 | D. Une séquence DNA s et un motif p .
Q. p apparait-il dans s ? |
| Problème 4 | D. Un programme C .
Q. Le programme est-il syntaxiquement correct ? |

Exemples de problèmes de décision (2)

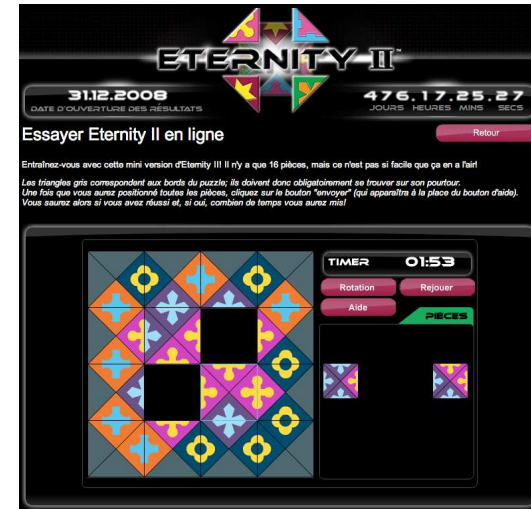
- Problème 5** **Donnée** Un graphe donné par une liste d'adjacence.
 Question Le graphe est-il 3-coloriable ?
- Problème 6** **D.** Un puzzle Eternity <http://fr.eternityii.com>.
 Q. Le puzzle a-t-il une solution ?
- Problème 7** **D.** Un programme C.
 Q. Le programme s'arrête-t-il sur au moins l'une de ses entrées ?
- Problème 8** **D.** Un programme C.
 Q. Le programme s'arrête-t-il sur toute entrée ?
- Problème 9** **D.** Des couples de mots $(u_1, v_1), \dots, (u_n, v_n)$.
 Q. Existe-t-il des entiers i_1, \dots, i_k tels que $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$?

Problème 9 : PCP, Problème de correspondance de Post

http://www.theory.informatik.uni-kassel.de/~stamer/pcp/pcpcontest_en.html



Problème 6 : Eternity II



Notion de problème de décision

Un problème de décision est

- ▶ une **question**...
- ▶ ...portant sur un **ensemble** de données...
- ▶ ...dont une **description** est fixée...
- ▶ ...et dont la réponse est **OUI** ou **NON**.

Ne pas confondre **problème** et **algorithme le résolvant**.

Une **instance** du problème est la question posée sur une donnée particulière.

- ▶ On s'intéressera aussi aux problèmes calculatoires, dont la réponse n'est pas nécessairement binaire (OUI/NON).
- ▶ Exemple : **calculer** un 3-coloriage.

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP. NP-complétude

Fonction récursives

Propriétés des ensembles dénombrables

Proposition

- 1a. Toute partie de \mathbb{N} est au plus dénombrable.
- 1b. Toute partie d'un ensemble dénombrable est au plus dénombrable.
2. Soit $f : A \rightarrow B$ une application.
 - 2.1 Si f est injective et B dénombrable, alors A est au plus dénombrable.
 - 2.2 Si f est surjective et A dénombrable, alors B est au plus dénombrable.

Ensembles dénombrables

- On note $\mathbb{N} = \{0, 1, 2, \dots\}$ l'ensemble des entiers naturels.
- Un ensemble E est dit **dénombrable** s'il est en bijection avec \mathbb{N} .
- Un ensemble fini ou dénombrable est dit **au plus dénombrable**.
- **Exemples.** Les ensembles suivants sont dénombrables :
 1. L'ensemble \mathbb{N}^* des entiers strictement positifs,
 2. Pour un entier $k > 0$ donné, l'ensemble des entiers divisibles par k ,
 3. L'ensemble des entiers qui sont des puissances de 2,
 4. L'ensemble des entiers naturels qui sont des cubes,
 5. L'ensemble des nombres premiers,
 6. L'ensemble \mathbb{Z} des entiers relatifs,
 7. L'ensemble $\mathbb{N} \times \mathbb{N}$,
 8. L'ensemble \mathbb{Q} des rationnels.

Propriétés des ensembles dénombrables

Proposition

- 1a. Si A et B sont dénombrables, alors $A \times B$ l'est aussi.
- 1b. Si A_1, \dots, A_n sont dénombrables, $\prod_{i=1}^n A_i$ l'est aussi (récurrence).
2. Si J est au plus dénombrable, et si A_i est au plus dénombrable, pour $i \in J$, alors $\bigcup_{i \in J} A_i$ est dénombrable.
3. Si $A \neq \emptyset$ est au plus dénombrable, alors A^* est dénombrable.
4. L'ensemble des automates finis sur alphabet A fini est dénombrable.
5. L'ensemble des programmes C est dénombrable.

Quelques ensembles non dénombrables

Proposition Les ensembles suivants ne sont pas dénombrables :

1. l'ensemble des nombres réels ;
2. l'ensemble des suites infinies d'entiers ;
3. l'ensemble des parties de \mathbb{N} ;
4. l'ensemble des suites infinies de 0 ou 1 ;
5. l'ensemble des applications de \mathbb{N} dans $\{0, 1\}$.

Note Les trois derniers ensembles sont en bijection.

À une partie X de \mathbb{N} , on associe la suite $x = (x_n)_{n \geq 0}$ définie par

$$x_n = \begin{cases} 1 & \text{si } n \in X \\ 0 & \text{si } n \notin X \end{cases}$$

De même, à toute suite $x = (x_n)_{n \geq 0}$ à valeurs dans $\{0, 1\}$, on associe bijectivement l'application $f_x : \mathbb{N} \rightarrow \{0, 1\}$ définie par $f_x(n) = x_n$.

Un paradoxe ?

- ▶ On suppose qu'on dispose d'un ordinateur à mémoire infinie.
- ▶ On considère les programmes C qui écrivent une suite de 0 ou 1.
- ▶ On interprète une telle suite a_1, a_2, \dots comme le réel $0, a_1 a_2 \dots$.
- ▶ Un tel réel est dit **calculable**.
- ▶ L'ensemble de ces programmes est dénombrable : $\{P_1, P_2, \dots\}$.
- ▶ On considère le programme qui
 - ▶ lance P_1 jusqu'à ce que celui-ci s'apprête à écrire a_1 , et écrit à la place un entier différent de a_1 , puis,
 - ▶ lance P_2 jusqu'à ce que celui-ci s'apprête à écrire a_2 , et écrit à la place un entier différent de a_2 , etc.
- ▶ Notre programme écrit un réel calculable non calculable !
- ▶ Où est l'erreur ?

Quelques ensembles non dénombrables

L'argument ci-dessous, dû à Cantor, permet de montrer que l'ensemble $E = \{0, 1\}^{\mathbb{N}}$ des suites infinies de 0 ou 1 est non dénombrable.

- ▶ Soit $f : \mathbb{N} \rightarrow E$ une application.
- ▶ Soit $x = (x_n)_{n \geq 0}$ la suite infinie de 0 ou 1 définie par

$$x_n = 1 - f(n)_n$$

- ▶ Puisque $x_n \neq f(n)_n$, on a $x \neq f(n)$, et ce, pour tout $n \in \mathbb{N}$.
- ▶ Comme $x \in E$ n'est pas de la forme $f(n)$, f n'est pas surjective.
- ▶ En particulier, il n'y a aucune bijection de \mathbb{N} dans E .

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP, NP-complétude

Fonction récursives

Notion de problème : formalisation

- ▶ Un **problème de décision** est la donnée d'un ensemble (dénombrable) I d'**instances** et d'un sous-ensemble $P \subseteq I$ d'**instances positives**.
- ▶ **Exemples**
 - ▶ $I = \mathbb{N}$, $P = \{n \in \mathbb{N} \mid n \text{ est premier}\}$,
 - ▶ $I = \{G \mid G \text{ graphe fini}\}$, $P = \{G \in I \mid G \text{ est 3-coloriable}\}$,
 - ▶ $I = \{G \mid G \text{ grammaire hors contexte}\}$, $P = \{G \in I \mid G \text{ est ambiguë}\}$.
- ▶ Pour Σ fini, un **codage** des instances est une application **injective** $\langle \cdot \rangle : I \rightarrow \Sigma^*$ associant à chaque élément x de I un mot $\langle x \rangle \in \Sigma^*$.

Notion de problème

- ▶ Un **problème de décision** peut se voir comme une fonction

$$f_P : \Sigma^* \longrightarrow \{\text{OUI}, \text{NON}\}$$

calculant le langage L_P , au sens que $f_P(\langle x \rangle) = \text{OUI} \iff x \in P$.

- ▶ Plus généralement, on s'intéresse au calcul de **fonctions**

$$f : \Sigma_1^* \longrightarrow \Sigma_2^*$$

- ▶ **Exemples**
 - ▶ $\Sigma_1 = \Sigma_2 = \{0, 1\}$,
 - ▶ un élément de $\{0, 1\}^*$ représente un entier codé en base 2
 - ▶ f calcule la fonction $n \mapsto 3^n$.

Notion de problème : formalisation

▶ Exemples

- ▶ $I = \mathbb{N}$, $\Sigma = \{a\}$ et $\langle n \rangle = a^n$.
- ▶ $I = \mathbb{N}$, $\Sigma = \{0, 1\}$ et $\langle n \rangle = \text{codage en base 2 de } n$.
- ▶ $I = \text{graphes orientés } G = (V, E)$, $\Sigma = \{0, 1, \#, \$\}$.
 $V = \{1, 2, \dots, n\}$, $E = \{(i_1, j_1), \dots, (i_m, j_m)\}$:

$$\langle (V, E) \rangle = \underbrace{\$ \cdots \$}_n \langle i_1 \rangle \# \langle j_1 \rangle \$ \langle i_2 \rangle \# \langle j_2 \rangle \$ \cdots \$ \langle i_m \rangle \# \langle j_m \rangle$$

- ▶ Σ^* se partitionne en 3 ensembles :
 - ▶ Instances positives : $L_P = \{\langle x \rangle \mid x \in P\}$.
 - ▶ Instances négatives : $L_N = \{\langle x \rangle \mid x \in I \text{ et } x \notin P\}$.
 - ▶ Non instances : $\Sigma^* \setminus \{\langle x \rangle \mid x \in I\}$.

Qu'est-ce que c'est un ordinateur ?

- ▶ avant 1940 : circuits logiques (Boole, Shannon, etc)
- ▶ 1941, von Neumann : machines "modernes"
- ▶ 1950/60 : différencier la puissance des machines selon leur mémoire
Hiérarchie de Chomsky : automates finis, automates à pile, automates de mémoire linéaire, machines de Turing.
Chomsky introduit sa hiérarchie avec des grammaires (calcul = **réécriture**).

Notion de machine

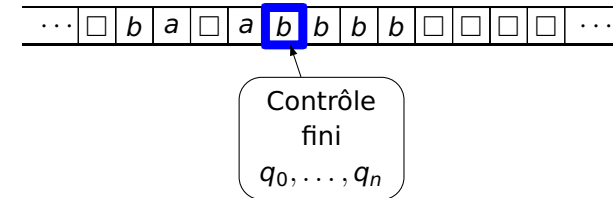
- ▶ Résoudre un problème consiste à déterminer pour $x \in I$, si $\langle x \rangle \in L_P$.
- ▶ On veut une machine « acceptant » les mots de L_P .
- ▶ **Automates** : mémoire bornée.
- ▶ **Automates à pile** : ne reconnaissent pas les langages
 - ▶ $\{a^n b^n c^n \mid n \geq 0\}$
 - ▶ $\{ww \mid w \in \{a, b\}^*\}$
- ▶ On peut reconnaître/accepter tous les langages $L \subseteq \Sigma^*$ avec un automate à nombre infini d'états.

Machines de Turing : intuition

- ▶ Le nombre d'états d'une machine de Turing est **fini** (par comparaison, un ordinateur a un nombre fini de registres et les programmes sont finis).
- ▶ La bande représente la mémoire de la machine. Elle est **infinie** : sur un ordinateur, on peut ajouter des périphériques mémoire (disques...) de façon quasi-illimitée.
- ▶ L'accès à la mémoire est **séquentiel** : la machine peut bouger sa tête à droite ou à gauche d'une case à chaque étape.

Machines de Turing

- ▶ Les machines de Turing sont des abstractions des ordinateurs.
- ▶ Une machine de Turing comporte :



- ▶ Une **bande infinie** à droite et à gauche faite de cases consécutives.
- ▶ Dans chaque case se trouve un symbole, éventuellement blanc \square .
- ▶ Une tête de lecture-écriture.
- ▶ Un contrôle à nombre **fini** d'états.

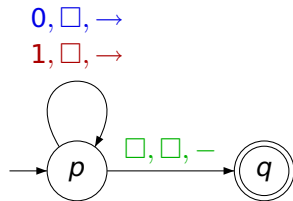
Machines de Turing : formalisation

Une **Machine de Turing** (MT) à une bande $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$ est donnée par

- ▶ Q : ensemble **fini** d'états.
- ▶ q_0 : état initial.
- ▶ $F \subseteq Q$: ensemble d'états finaux (ou acceptants).
- ▶ Γ : alphabet fini de la bande, avec $\square \in \Gamma$.
- ▶ Σ : alphabet d'entrée, avec $\Sigma \subseteq \Gamma \setminus \{\square\}$.
- ▶ δ : ensemble de transitions. Une transition est de la forme (p, a, q, b, d) , notée $p \xrightarrow{a, b, d} q$, avec
 - ▶ $p, q \in Q$,
 - ▶ $a, b \in \Gamma$,
 - ▶ $d \in \{\leftarrow, -, \rightarrow\}$.
- ▶ On supposera qu'aucune transition ne part d'un état de F .

Machines de Turing : représentation graphique

- ▶ On représente souvent une MT comme un automate.
- ▶ Seules changent les étiquettes des transitions.
- ▶ Exemple, avec $\Gamma = \{0, 1, \square\}$ et $\Sigma = \{0, 1\}$:



représente la machine avec $Q = \{p, q\}$, $q_0 = p$, $F = \{q\}$,
 $\delta = \{(p, 0, \square, \rightarrow, p), (p, 1, \square, \rightarrow, p), (p, \square, \square, -, q)\}$

Fonctionnement d'une MT

- ▶ Chaque pas consiste à appliquer une transition.
- ▶ Une transition de la forme $p \xrightarrow{a,b,d} q$ est possible seulement si
 1. la machine se trouve dans l'état p , et
 2. la lettre se trouvant sous la tête de lecture-écriture est a .
- ▶ Dans ce cas, l'application de la transition consiste à
 - ▶ changer l'état de contrôle qui devient q ,
 - ▶ remplacer le contenu de la case sous la tête de lecture-écriture par b ,
 - ▶ bouger la tête d'une case à gauche si $d = \leftarrow$, ou
 - ▶ bouger la tête d'une case à droite si $d = \rightarrow$, ou
 - ▶ ne pas bouger la tête si $d = -$.

Fonctionnement d'une MT

- ▶ Initialement, un mot w est écrit sur la bande entouré de \square .
- ▶ Un calcul d'une MT sur w est une suite de **pas de calcul**.
- ▶ Cette suite peut être **finie** ou **infinie**.
- ▶ Le calcul commence
 - ▶ avec la tête de lecture-écriture sur la première lettre de w ,
 - ▶ dans l'état q_0 .
- ▶ Chaque pas de calcul consiste à appliquer une transition, si possible (il peut y avoir des choix : **non-déterminisme**).
- ▶ Le calcul ne s'arrête que si aucune transition n'est applicable.

Configurations et calculs

- ▶ Une **configuration** représente un instantané du calcul.
- ▶ La configuration uqv signifie que
 - ▶ L'état de contrôle est q
 - ▶ Le mot écrit sur la bande est uv , entouré par des \square ,
 - ▶ La tête de lecture est sur la première lettre de v .
- ▶ La configuration initiale sur w est donc q_0w .
- ▶ Pour 2 configurations C, C' , on écrit $C \vdash C'$ lorsqu'on obtient C' par application d'une transition à partir de C .

Un **calcul** d'une machine de Turing est une suite de configurations.

$$C_0 \vdash C_1 \vdash C_2 \vdash \dots$$

Calculs acceptants

Un calcul d'une machine de Turing est une suite de configurations.

$$C_0 \vdash C_1 \vdash C_2 \vdash \dots$$

3 cas possibles

- ▶ Le calcul est infini,
- ▶ Le calcul s'arrête sur un état final (de F),
- ▶ Le calcul s'arrête sur un état non final (pas de F).

Exemples de machines de Turing

- ▶ Machine qui effectue `while(true)` ;
- ▶ Machine qui efface son entrée et s'arrête.
- ▶ Machine qui accepte 0^*1^* (**automate fini**).
- ▶ Machine qui accepte $\{a^n b^n \mid n \geq 0\}$ (**automate à pile**).
- ▶ Machine qui accepte $\{a^{2^n} \mid n \geq 0\}$ (**automate à mémoire linéaire**).
- ▶ Machine qui accepte $\{a^n b^n c^n \mid n \geq 0\}$ (**automate à mémoire linéaire**).
- ▶ Machine qui accepte $\{ww \mid w \in \{0, 1\}^*\}$ (**automate à mémoire linéaire**).

Langages acceptés

On peut utiliser une machine pour **accepter** des mots.

- ▶ Le langage $\mathcal{L}(M) \subseteq \Sigma^*$ des mots acceptés par une MT M est l'ensemble des mots w sur lesquels **il existe** un calcul **fini**

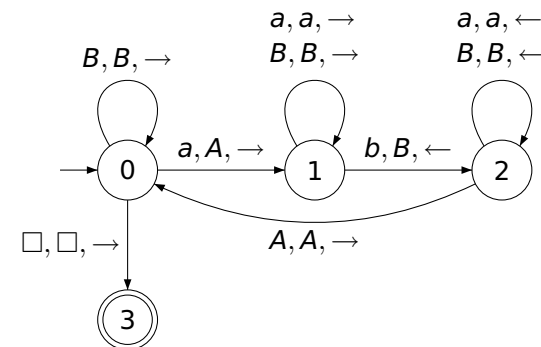
$$C_0 \vdash C_1 \vdash C_2 \vdash \dots C_n$$

avec $C_0 = q_0 w$ (w est le mot **d'entrée**) et $C_n \in \Gamma^* F \Gamma^*$.

- ▶ 3 cas exclusifs : un calcul peut
 - ▶ soit s'arrêter sur un état acceptant,
 - ▶ soit s'arrêter sur un état non acceptant,
 - ▶ soit ne pas s'arrêter.
- ▶ On dit qu'une machine est **déterministe** si, pour tout $(p, a) \in Q \times \Gamma$, il existe **au plus** une transition de la forme $p \xrightarrow{a, b, d} q$.
- ▶ Si M est déterministe, elle n'admet qu'un calcul par entrée.

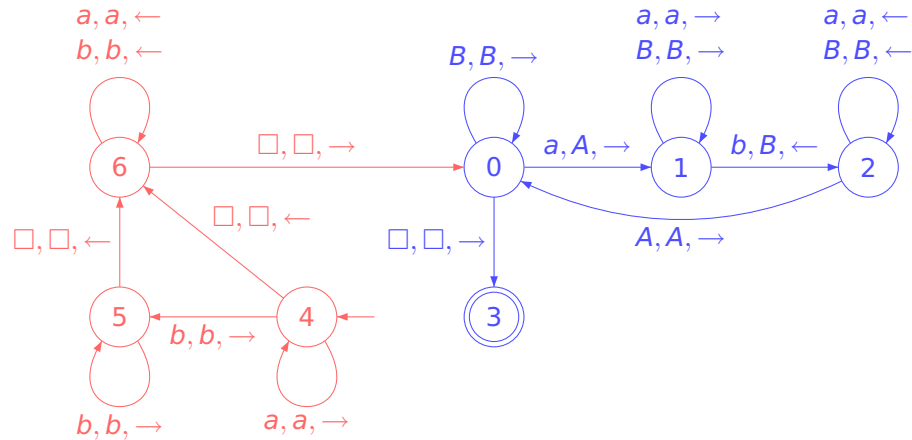
MT acceptant $(\{a^n b^n \mid n \geq 0\})^*$

Idée : marquer le 1^{er} a et le 1^{er} b, et recommencer.



MT acceptant $\{a^n b^n \mid n \geq 0\}$

Idée : idem en vérifiant qu'on est dans a^*b^* .



Les machines de Turing peuvent calculer

- On peut utiliser les MT pour **accepter** des langages ou **calculer**.
- Une MT déterministe acceptant un langage L calcule la fonction caractéristique de L définie par

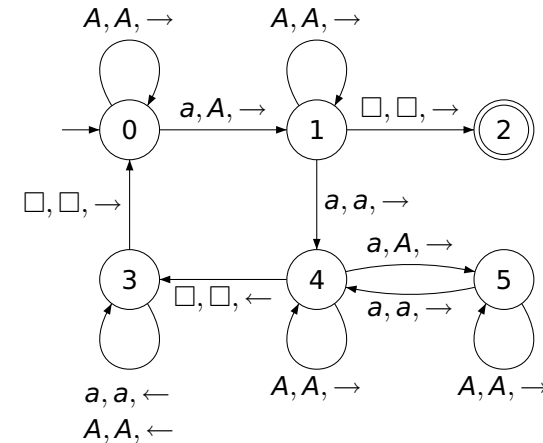
$$f : \Sigma^* \rightarrow \{0, 1\}$$

$$w \mapsto \begin{cases} 0 & \text{si } w \notin L, \\ 1 & \text{si } w \in L. \end{cases}$$

- Plus généralement, on peut associer à une MT déterministe M une fonction $f_M : \Sigma^* \rightarrow \Gamma^*$
 - On écrit la donnée $w \in \Sigma^*$ sur la bande,
 - Si la MT s'arrête avec sur la bande le mot $z \in \Gamma^*$, la fonction est définie par $f_M(w) = z$.

MT acceptant $\{a^{2^n} \mid n \geq 0\}$

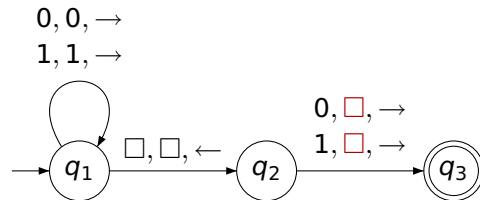
Idée : marquer un a sur 2.



Exemples de machines de Turing

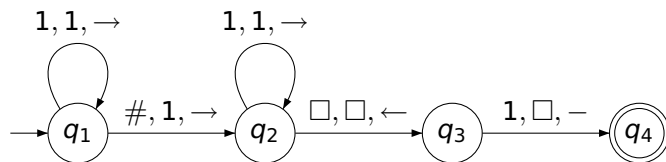
- Machine qui interprète son entrée comme un entier n , le remplace par $\lfloor n/2 \rfloor$ et s'arrête.
- Machine qui effectue l'incrément en binaire.
- Machine qui effectue l'addition de deux entiers en unaire.
- Machine qui effectue la multiplication de deux entiers en unaire.

Calcul de $\lfloor n/2 \rfloor$

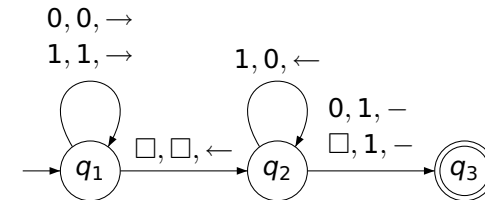


Addition en unaire

- Le mot d'entrée est de la forme $1^n \# 1^m$ interprété comme la donnée des entiers n et m .



Incrément en binaire



Il existe des langages non décidables

- Thèse de Church : les MT capturent tout ce qui est calculable.
- Un langage L est **décidable** (ou **récuratif**) s'il existe une MT qui
 - s'arrête sur F partant d'un mot de L ,
 - s'arrête sur $Q \setminus F$ partant d'un mot de $\Sigma^* \setminus L$.
- L'ensemble des langages sur $\{0, 1\}^*$ est non dénombrable.
- L'ensemble des machines de Turing est dénombrable.
- Il existe donc des langages **non décidables**.
- Peut-on décrire explicitement un tel langage ?

Variations des machines de Turing

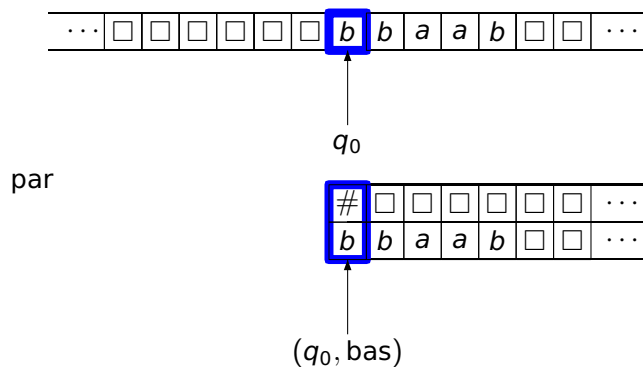
On peut changer la définition des MT de plusieurs façons :

- ▶ bande finie sur la gauche et infinie sur la droite,
- ▶ un unique état acceptant, un unique état rejetant,
- ▶ calculs remplaçant la tête en début de bande...
... et terminant avec le mot d'entrée écrit sur la bande,
- ▶ machines déterministes,
- ▶ petit alphabet de bande : $\Gamma = \Sigma \cup \{\square\}$,
- ▶ machine à plusieurs bandes et plusieurs têtes.

Ces variations n'affectent pas ce que l'on peut accepter ou calculer.

Bande finie à gauche

- ▶ On « replie » la bande.
- ▶ On représente la configuration initiale



Simulation : pour montrer l'équivalence des variantes

- ▶ Intuitivement, une machine M_2 simule une machine M_1 lorsque M_2 peut effectuer les mêmes calculs que M_1 .
- ▶ Un exemple (qui n'est pas général) où M_2 simule M_1 :

$$Q_1 \subseteq Q_2, \quad F_1 = F_2, \quad \Gamma_1 \subseteq \Gamma_2, \text{ et}$$

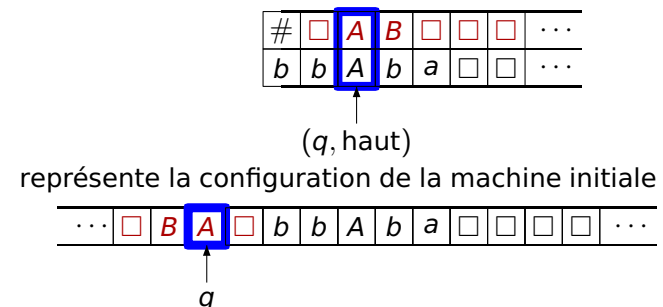
$$\forall C, C' \in \Gamma_1^* Q_1 \Gamma_1^*$$

$$C \vdash_1 C' \text{ dans } M_1 \Leftrightarrow C \vdash_2 C_1 \vdash_2 C_2 \vdash_2 \dots \vdash_2 C_k \vdash_2 C' \\ \text{dans } M_2 \text{ avec } C_i \notin \Gamma_2^* Q_1 \Gamma_2^*$$

- ▶ M_2 simule M_1 si elle peut effectuer les mêmes passages entre configurations (éventuellement par plusieurs transitions).

Bande finie à gauche

- ▶ On a $\Gamma_2 = (\{\#\} \cup \Gamma_1) \times \Gamma_1$.
- ▶ $\#$ est un nouveau symbole servant à repérer le début de bande.
- ▶ On a $Q_2 = Q_1 \times \{\text{haut}, \text{bas}\}$.
- ▶ Une configuration de la nouvelle machine est du type



- ▶ Reste à écrire les transitions.

Bande finie vs. bi-infinie

- ▶ Inversement, toute machine travaillant sur une bande finie peut être simulée par une machine à bande bi-infinie.

Composition des MT

- ▶ En connectant l'état final (supposé unique) d'une machine M_1 à l'état initial d'une machine M_2 , on compose les fonctions calculées par les deux machines.
- ▶ Si M_1 termine en restaurant le mot d'entrée, on exécute la séquence $M_1; M_2$.
- ▶ **Exemple** : codage unaire \rightarrow binaire en utilisant l'incrément.

Un unique état acceptant, un unique état rejetant

A partir de $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$, on construit

$$M' = (Q', q'_0, F', \Sigma', \Gamma', \delta').$$

- ▶ On ajoute un état OK, seul état acceptant de la nouvelle machine M' ,
- ▶ ... et des transitions de F vers OK.
- ▶ $Q' = Q \uplus \{\text{OK}\}$, $q'_0 = q_0$, $F' = \{\text{OK}\}$, $\Sigma' = \Sigma$, $\Gamma' = \Gamma$,
 $\delta' = \delta \cup (F \times \Gamma \times \{\text{OK}\} \times \Gamma \times \{-\})$.

On peut de même transformer M pour que :

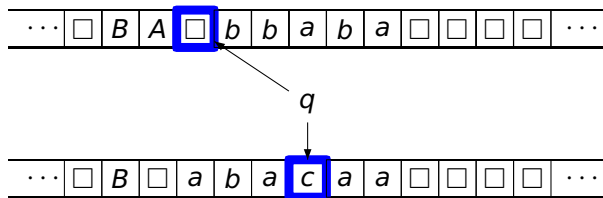
- ▶ il y ait un **unique état rejetant KO**.
- ▶ tout calcul de M qui s'arrête remplace la tête en **début** de mot écrit.
- ▶ M sauvegarde son mot d'entrée et le **restaure** sur la bande quand elle s'arrête.

Langage des machines de Turing

- ▶ Les simplifications précédentes permettent de composer les MT.
- ▶ Composition séquentielle : $M_1; M_2$.
- ▶ Test : **Si** M_1 **alors** M_2 **sinon** M_3 .
- ▶ Boucle : **Tant que** M_1 **faire** M_2 .
- ▶ Instructions de base
 - ▶ test de lecture $R == a$,
 - ▶ écriture de a : $W(a)$,
 - ▶ Déplacements : G ou D,
 - ▶ arrêt OK ou KO.

Plusieurs bandes

- ▶ On peut utiliser plusieurs bandes.
- ▶ Initialement, le mot d'entrée est écrit sur une bande,
- ▶ Chaque transition lit un symbole sur chaque bande, et en fonction des symboles lus et de l'état, la machine
 - ▶ change d'état,
 - ▶ écrit un nouveau symbole sous chacune des têtes.
 - ▶ déplace chaque tête indépendamment.

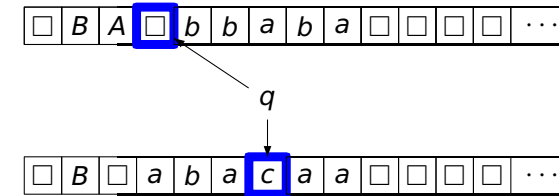


Plusieurs bandes

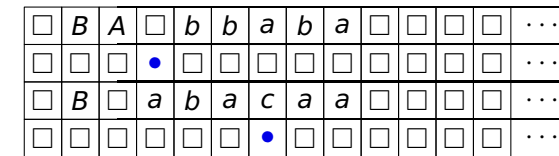
- ▶ **Idée** de la simulation : position des têtes stockée dans la bande.
- ▶ Chaque transition de la machine originale est simulée par
 - ▶ un aller sur la bande pour récupérer les symboles sous les têtes,
 - ▶ un retour pour simuler la transition.
- ▶ Contrairement à la simulation de bande infinie par bande finie, plusieurs mouvements sont nécessaires pour simuler un seul mouvement.
- ▶ La nouvelle machine est intuitivement « plus lente ».
- ▶ De combien ? [Retour aux classes P et NP](#)

Plusieurs bandes

- ▶ Simulation possible : regrouper les bandes.



est représentée par



Machines déterministes

- ▶ Comme pour les automates finis, pour toute machine M , on peut construire une machine déterministe qui simule M .
- ▶ Idée : on parcourt l'arbre des calculs en **largeur**.
- ▶ On note r le nombre maximal de choix dans M .
- ▶ On utilise 3 bandes :
 - ▶ bande 1 : sauvegarde la valeur du mot d'entrée w .
 - ▶ bande 2 : génère dans l'ordre hiérarchique les suites finies d'entiers sur $\{1, \dots, r\}$.
 - ▶ bande 3 : effectue le calcul de M sur w correspondant à la suite de choix écrite sur la bande 2.

$$\Gamma = \Sigma \cup \{\square\}$$

- ▶ On peut faire en sorte que Γ n'utilise pas de symboles supplémentaires, à part \square .
- ▶ Idée : coder chaque symbole de Γ sur $\Sigma \cup \{\square\}$.
- ▶ Représenter
 - ▶ 0 par $(0, \square, \dots, \square)$,
 - ▶ 1 par $(1, \square, \dots, \square)$,
 - ▶ \square par $(\square, \square, \dots, \square)$,

Machines RAM

Une machine RAM est une machine possédant

- ▶ Un programme, qui est une suite finie d'instructions I_0, I_1, I_2, \dots
- ▶ Une suite infinie de registres R_0, R_1, R_2, \dots
- ▶ Un registre spécial : compteur de programme **PC** (program counter).
- ▶ Une bande d'entrée sur laquelle la machine lit ses données.
- ▶ Une bande de sortie sur laquelle la machine écrit ses résultats.

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP. NP-complétude

Fonction récursives

Machines RAM : registres

- ▶ Chaque registre peut mémoriser n'importe quel entier positif ou nul.
- ▶ Initialement, tous les registres ont comme valeur 0.
- ▶ Le registre **PC** contient le numéro de la prochaine instruction à exécuter.
- ▶ Les autres registres contiennent des valeurs (entières positives) manipulées par la machine au cours du calcul.

Machines RAM : instructions

Les instructions sont de 4 types :

1. Manipulations de registres.
2. Opérations arithmétiques.
3. Sauts et arrêt.
4. Entrées/sorties.

Machines RAM : sauts et arrêt

- ▶ **jump** $\langle k \rangle$: saut inconditionnel à l'instruction I_k .
- ▶ **jz** $\langle k, n \rangle$: saut à l'instruction I_k si la valeur du registre R_n est 0.
- ▶ **stop** : fin du programme.

En pratique, on se permet de mettre des étiquettes au niveau d'instructions vers lesquelles on veut aller.

Machines RAM : Opérations arithmétiques

- ▶ **incr** (n) : incrémente de 1 la valeur contenue dans le registre R_n .
- ▶ **decr** (n) : décrémente de 1 la valeur contenue dans le registre R_n , si elle est strictement positive. Ne fait rien sinon.

Machines RAM : E/S

Sur la bande d'entrée se trouve une suite d'entiers (les entrées du programme).

- ▶ **read** (n) : lit la valeur courante de la bande d'entrée et la met dans le registre R_n . La prochaine valeur lue sera la suivante de la bande d'entrée.
- ▶ **write** (n) : écrit sur la bande de sortie la valeur contenue dans le registre R_n .

Exécution d'une machine RAM

- ▶ La machine exécute les instructions en commençant par I_0 .
- ▶ Une fois l'instruction I_k exécutée, c'est l'instruction I_{k+1} qui est exécutée, sauf après une instruction de saut ou **stop**.

Machines RAM : simulation par machines de Turing

Une MT peut simuler une RAM : tout programme RAM peut être réalisé par machine de Turing à plusieurs bandes :

- ▶ Une bande d'entrée sur laquelle on met les données séparées par \square .
- ▶ Une bande de sortie.
- ▶ Une bande pour l'accumulateur (en binaire par exemple).
- ▶ Une bande pour tous les autres registres, séparés par \square .
- ▶ Une bande de travail servant à retrouver le contenu d'un registre.

Jeux d'instruction des machines RAM vs. processeurs

Les machines RAM

- + peuvent manipuler des entiers arbitraires,
- + peuvent utiliser un nombre arbitraire de registres.
- ont un jeu d'instructions plus restreint que celui des processeurs habituels.

Mais on peut reprogrammer les instructions manquantes.

- ▶ $R_m := R_n, R_n := 0$.
- ▶ Opérations arithmétiques (addition, multiplication, soustraction « tronquée », ...).
- ▶ Appel de sous-programme : **call/return**
 \rightsquigarrow nécessite un décalage de registres.

Machines RAM : simulation par machines de Turing

- ▶ Chaque instruction est codée par une partie de la machine.
- ▶ Les instructions sont reliées entre elles grâce aux états.

Exemples.

- ▶ **incr/decr** : machines incrémentation/décrémentation déjà vues, travaillant sur la bande correspondant à l'accumulateur.
- ▶ **load <n>** : recherche du contenu de R_n et recopie dans **A**. Pour la recherche, on initialise la bande de travail à n et place la tête de la bande des registres derrière le n -ième \square .
- ▶ **jump q** : correspond aux transitions $p \xrightarrow{x/x/-} q$ pour $x \in \Gamma$.
- ▶ **jz q** : idem si l'accumulateur est nul. Sinon, aller en $p + 1$.
- ▶ **stop** : aller dans l'état **OK**.

À quoi sert cette simulation ?

- ▶ On peut compiler un programme C (sans entrée-sortie) vers un programme RAM.
- ▶ La simulation précédente montre qu'on peut compiler tout programme RAM en machine de Turing.
- ▶ Tout programme sans entrée-sortie dans n'importe quel langage de programmation actuel se compile en une machine de Turing.

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP. NP-complétude

Fonction récursives

Simulation inverse

Inversement, on peut simuler toute machine de Turing par une RAM. Idée de simulation, pour machine à bande infinie à droite :

- ▶ Mémoriser dans R_1 le contenu de la case 1, dans R_2 celui de la case 2, etc.
- ▶ Mémoriser dans R_0 la position de la tête de lecture.
- ▶ Initialiser la bande, et simuler chaque instruction par un bout de code.
- ▶ Suite en TD...

Définitions

- ▶ Un langage L est **récursivement énumérable** s'il est le langage accepté par une machine de Turing.
- ▶ **RE** = classe des langages récursivement énumérables.
- ▶ **Note** Sur un mot $\Sigma^* \setminus L$, la machine de Turing peut ne pas s'arrêter.
- ▶ Un langage L est **récursif** ou **décidable** s'il est le langage accepté par une machine de Turing **qui s'arrête sur toute entrée**.
- ▶ **R** = classe des langages récursifs.

Lagages RE et R

- ▶ On a clairement $R \subseteq RE$.
- ▶ La classe des langages RE est fermée par union.
- ▶ La classe des langages R est fermée par union et complément.
- ▶ Si L et $\Sigma^* \setminus L$ sont dans RE, alors ils sont dans R.
 - ▶ On construit une machine qui simule en parallèle les machines M et N acceptant L et $\Sigma^* \setminus L$ (elle alterne un pas de calcul de M , un pas de calcul de N).
 - ▶ Cette machine s'arrête si l'une des machines M et N s'arrête.
 - ▶ Un mot appartient soit à L , soit à $\Sigma^* \setminus L$, donc elle **s'arrête toujours**.

Le langage diagonal

- ▶ On numérote les mots de $\Sigma = \{0, 1\}^*$ dans l'ordre **hiérarchique**.
 - ▶ longueur d'abord,
 - ▶ ordre lexicographique pour une longueur donnée.

$$w_0 = \epsilon, w_1 = 0, w_2 = 1, w_3 = 00, w_4 = 01, w_5 = 10, w_6 = 11, \dots$$

- ▶ **Note** Une MT peut calculer le numéro d'un mot.
- ▶ On note M_i la machine telle que $\langle M_i \rangle = w_i$ (par convention $\mathcal{L}(M_i) = \emptyset$ si w_i n'est pas le code d'une machine de Turing).
- ▶ **Proposition** Le langage

$$L_d = \{w_i \mid w_i \notin \mathcal{L}(M_i)\}$$

n'est **pas** RE.

Codage des MT

- ▶ On travaille sur $\Sigma = \{0, 1\}$, et on peut supposer $\Gamma = \{0, 1, \square\}$.
- ▶ On ne considère que les MT à un unique état OK.
- ▶ Toute MT de ce type peut se coder sur l'alphabet $\{0, 1\}$.
- ▶ On code une transition $p_i \xrightarrow{a_j, a_k, d_\ell} q_m$ par $0^i 10^j 10^k 10^\ell 10^m$.
- ▶ On code la MT M par la suite de ces transitions, séparées entre elles par 11, avec deux blocs 111 en début et fin.
- ▶ On note ce code $\langle M \rangle$.
- ▶ Si w est un mot, on note $\langle M, w \rangle$ le code de M suivi par w .

Réductions

- ▶ Soient P_A et P_B deux problèmes.
- ▶ On note L_A l'ensemble des instances positives de P_A .
- ▶ On note L_B l'ensemble des instances positives de P_B .
- ▶ Une réduction de P_A vers P_B est une **fonction calculable par MT** $f : \Sigma^* \rightarrow \Sigma^*$ telle que

$$x \in L_A \iff f(x) \in L_B.$$

- ▶ On note $P_A \leq P_B$ (P_A **se réduit** à P_B)
- ▶ L'existence d'une réduction de P_A vers P_B assure que
 - ▶ si P_B est décidable, P_A l'est aussi,
 - ▶ si P_A est indécidable, P_B l'est aussi.

Langage et machine universelle

- Le langage

$$L_u = \{ \langle M, w \rangle \mid M \text{ accepte } w \}$$

est RE, mais pas R.

- S'il était dans R, alors $\Sigma^* \setminus L_d$ le serait aussi.
- À partir d'une machine de Turing hypothétique M qui s'arrête sur toute entrée et telle que $\mathcal{L}(M) = L_u$, on **construit** une machine de Turing M' qui s'arrête sur toute entrée et telle que $\mathcal{L}(M') = \Sigma^* \setminus L_d$.
- Or, une telle machine M' n'existe pas, car sinon, L_d serait dans R.
- On a construit une **réduction** de $\Sigma^* \setminus L_d$ à L_u : $\Sigma^* \setminus L_d \leq L_u$.

Théorème de Rice

- Soit \mathcal{E} l'ensemble des langages RE de $\{0, 1\}^*$.
- Une **propriété des langages RE** est un sous-ensemble \mathcal{P} de \mathcal{E} .
- Une propriété $\mathcal{P} \subseteq \mathcal{E}$ est **triviale** si $\mathcal{P} = \emptyset$ ou $\mathcal{P} = \mathcal{E}$.
- **Attention** Ne pas confondre $\mathcal{P} = \emptyset$ (\mathcal{P} ne contient aucun langage) et $\mathcal{P} = \{\emptyset\}$ (\mathcal{P} ne contient que le langage vide).

Un autre langage indécidable

- Le langage

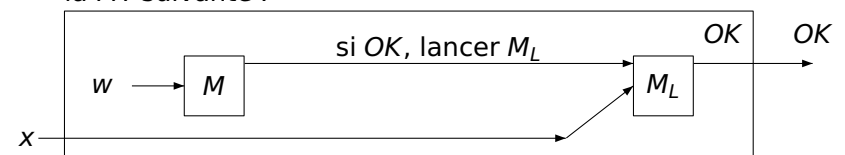
$$L_\emptyset = \{ \langle M \rangle \mid L(M) = \emptyset \}$$

est n'est pas RE.

- S'il était dans R, alors L_u le serait aussi.
- On montre que $\Sigma^* \setminus L_\emptyset$ est RE.
- On construit une réduction $L_u \leq \Sigma^* \setminus L_\emptyset$.
- À partir d'un algorithme pour **décider** L_\emptyset , on décrit un algorithme pour décider L_u .
- À partir de M et w , on construit M' qui efface son entrée et lance M sur w .
- M' accepte si et seulement si M s'arrête sur OK.

Théorème de Rice

- **Toute propriété non triviale \mathcal{P} des langages RE est non décidable.**
- **Attention** : il s'agit d'une propriété des **langages**, non des MT.
- **Preuve** A nouveau une réduction à partir de L_u .
- Quitte à changer \mathcal{P} et $\mathcal{E} \setminus \mathcal{P}$, on peut supposer $\emptyset \notin \mathcal{P}$.
- Comme $\mathcal{P} \neq \emptyset$, il existe $L \in \mathcal{P}$. Soit M_L telle que $\mathcal{L}(M_L) = L$.
- À partir de M, w donnés, on construit (par machine de Turing !) la MT suivante :



- Cette machine accepte $\emptyset \notin \mathcal{P}$ si $\langle M, w \rangle \notin L_u$, et $L \in \mathcal{P}$ sinon.
- Donc si \mathcal{P} était dans R, L_u le serait aussi.

Problème de l'arrêt

Les problèmes suivants sont **indécidables** :

- ▶ étant donnée une MT M et un mot w , M s'arrête-t-elle sur w ?
- ▶ étant donnée une MT M , s'arrête-t-elle sur ε ?
- ▶ étant donnée une MT M , s'arrête-t-elle sur au moins une entrée ?
- ▶ étant donnée une MT M , s'arrête-t-elle sur toute entrée ?
- ▶ étant donnée une MT M et un état q de M , M utilise-t-elle l'état q sur l'entrée ε ?

Le PCP modifié (PCPM)

- ▶ Problème de correspondance de Post (1946).
- ▶ **Donnée** : n paires de mots $(u_1, v_1), \dots, (u_n, v_n)$.
- ▶ **Question** : existe-il une suite finie i_1, \dots, i_k telle que $i_1 = 1$ et

$$u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$$

A noter : les suites d'indices sont les mêmes, ...
... et le premier indice est 1.

L'indécidabilité hors du monde des MT : le PCP

- ▶ Problème de correspondance de Post (1946).
- ▶ **Donnée** : n paires de mots $(u_1, v_1), \dots, (u_n, v_n)$.
- ▶ **Question** : existe-il une suite finie i_1, \dots, i_k telle que

$$u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$$

[A noter : les suites d'indices sont les mêmes.]

- ▶ \leadsto On peut voir les couples de mots comme des dominos.

a	aa	ba	bab
ab	a	aa	$abba$

- ▶ Une solution : $a.bab.ba.aa.aa = ab.abba.aa.a.a$
- ▶ Suite d'indices : 1, 4, 3, 2, 2.

Indécidabilité du PCP et PCPM

- ▶ On montre que

$$L_u \leq \text{PCPM} \leq \text{PCP}.$$

- ▶ Comme le langage universel L_u est non décidable, il en est de même de PCPM et de PCP.
- ▶ Accessoirement, on peut montrer que $\text{PCP} \leq \text{PCPM}$.

PCP \leq PCPM

- ▶ Si on a un algorithme pour résoudre PCPM, on a un algorithme pour résoudre PCP.
- ▶ Il suffit de résoudre n PCPM différents, selon le mot avec lequel on commence.

Indécidabilité du PCPM

- ▶ On rappelle que $L_u = \{\langle M, w \rangle \mid M \text{ accepte } w\}$ est indécidable.
- ▶ On montre $L_u \leq$ PCPM.
- ▶ Étant donné une MT M et un mot w , on construit une instance $((u_\ell, v_\ell))_{1 \leq \ell \leq n}$ de PCPM telle que

$$\langle M, w \rangle \in L_u \iff \text{PCP sur l'instance } ((u_\ell, v_\ell))_{1 \leq \ell \leq n} \text{ a une solution}.$$

- ▶ On peut supposer que
 - ▶ le seul état d'arrêt de M est q_{ok} ,
 - ▶ M déplace sa tête à chaque transition.

PCPM \leq PCP : plus difficile

- ▶ Supposons donné un algorithme pour résoudre le PCP.
- ▶ On introduit une nouvelle lettre \$, et pour $a_1 \cdots a_k \in A^+$, soient $p(a_1 \cdots a_k) = \$a_1 \cdots \a_k et $s(a_1 \cdots a_k) = a_1\$ \cdots a_k\$$.
- ▶ Soit $(u_1, v_1), \dots, (u_n, v_n)$ une instance de PCPM.
- ▶ Soient les $2n + 1$ mots suivants :

$$\begin{aligned} x_i &= p(u_i), & y_i &= s(v_i) \\ x_{n+i} &= p(u_i)\$, & y_{n+i} &= s(v_i) \\ x_{2n+1} &= p(u_1), & y_{2n+1} &= \$s(v_1). \end{aligned}$$

- ▶ Le PCPM sur l'instance $((u_\ell, v_\ell))_{1 \leq \ell \leq n}$ a une solution si et seulement si le PCP sur l'instance $((x_\ell, y_\ell))_{1 \leq \ell \leq 2n+1}$ en a une.

La réduction $L \leq$ PCPM

- ▶ **Idée** : la seule solution sera la suite des configurations sur w .
- ▶ La partie **bleue** est en retard d'une configuration.
- ▶ Domino 1 : $(\#, \#q_0w\#)$. Les autres dominos :
- ▶ Dominos de copie : (a, a) , $(\#, \#)$,
- ▶ Dominos de transitions :
- ▶ Si $p \xrightarrow{a,b,\rightarrow} q \in \delta$, on met un domino (pa, bq) .
- ▶ Si $p \xrightarrow{a,b,\leftarrow} q \in \delta$, on met un domino (xpa, qxb) pour tout $x \in \Gamma$.
- ▶ Si $p \xrightarrow{\square,b,\rightarrow} q \in \delta$, on met un domino $(p\#, bq\#)$.
- ▶ Si $p \xrightarrow{\square,b,\leftarrow} q \in \delta$, on met un domino $(xp\#, qxb\#)$ pour tout $x \in \Gamma$.
- ▶ Dominos de synchronisation en fin de calcul :
 - ▶ pour chaque $a, b \in \Gamma$: (aq_{ok}, q_{ok}) , $(q_{ok}b, q_{ok})$,
 - ▶ $(q_{ok}\#\#, \#)$.

Quelques autres problèmes indécidables

Les problèmes suivants sont indécidables :

- ▶ Étant donné un jeu fini de tuiles carrées, avec conditions de compatibilité entre côtés, déterminer si on peut paver le $1/4$ de plan.
- ▶ Étant donnée une grammaire, déterminer si elle est ambiguë.
- ▶ Étant donné un nombre fini de matrices 3×3 à coefficients entiers, déterminer si un produit permet d'annuler la composante $(3,2)$.
- ▶ Étant donnée une suite calculable d'entiers, déterminer si elle converge.

Révisions : problèmes indécidables

Les problèmes suivants sont indécidables :

1. P1 : Étant donnée une MT M , décider si elle accepte au moins 7 mots.
2. P2 : Étant donnée une MT M , décider si elle s'arrête toujours avec écrit sur sa bande 2008.
3. P3 : Étant donnée une MT M et un état p de M , décider si M utilise l'état p sur au moins une entrée.
4. P4 : Étant donnée une MT M et un état p de M , décider si M utilise l'état p sur toute une entrée.
5. P5 : Étant donnée une MT M , décider si elle accepte n'importe quel mot représentant (en binaire) un entier premier.

Révisions : vrai/faux

- (a) L'ensemble des graphes orientés finis est dénombrable.
- (b) Tout langage fini est décidable.
- (c) Pour tous $K, L \subseteq \Sigma^*$ avec $\Sigma = \{0, 1\}$, si K se réduit à L , alors $\Sigma^* \setminus K$ se réduit à $\Sigma^* \setminus L$.
- (d) Tout langage sur un alphabet à une lettre est décidable.
- (e) Étant donné une machine de Turing M , un mot w et un entier k , on peut décider si M accepte w en au plus k pas de calcul.
- (f) Étant données deux machines de Turing M_1 et M_2 , on peut décider si $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$.
- (g) Le langage des mots sur l'alphabet ASCII représentant un programme C syntaxiquement correct est décidable.
- (h) Le complément de tout ensemble récursivement énumérable est aussi récursivement énumérable.

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP. NP-complétude

Fonction récursives

Littéraux, clauses, et formules CNF

Étant données des variables x_1, x_2, \dots :

- ▶ un littéral est soit une variable x_i , soit la négation d'une variable $\neg x_i$.

- ▶ Une **clause** est une disjonction de littéraux.

Exemple : $x_1 \vee \neg x_2 \vee \neg x_4 \vee x_5$.

- ▶ Une **3-clause** est une clause avec 3 littéraux différents.

Exemple : $x_1 \vee \neg x_2 \vee \neg x_4$.

- ▶ Une formule CNF est une conjonction de clauses.

- ▶ Une formule 3-CNF est une conjonction de 3-clauses.

Exemple : $(x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_3)$.

3-SAT

Le problème **3-SAT** est le suivant :

- ▶ **Donnée** : une formule **3-CNF** sur des variables $\{x_1, x_2, \dots\}$.
- ▶ **Question** : existe-t-il une assignation de chaque variable x_i par **vrai** ou **faux** qui rend la formule vraie ?

Le problème **3-SAT** est donc moins général que **SAT**.

SAT

Le problème **SAT** est le suivant :

- ▶ **Donnée** : une formule **CNF** sur des variables $\{x_1, x_2, \dots\}$.
- ▶ **Question** : existe-t-il une assignation de chaque variable x_i par **vrai** ou **faux** qui rend la formule vraie ?

Réduction SAT vers 3-SAT

- ▶ À toute instance φ de SAT, on associe une instance $\tilde{\varphi}$ de 3-SAT tq.

φ est satisfaisable $\iff \tilde{\varphi}$ est satisfaisable.

Remarque. On va construire $\tilde{\varphi}$ en temps polynomial par rapport à $|\varphi|$.

Réduction SAT vers 3-SAT

Si $\varphi = c_1 \wedge \dots \wedge c_k$ où chaque c_i est une clause, on construit

$$\tilde{\varphi} = \varphi_1 \wedge \dots \wedge \varphi_k, \text{ où}$$

- ▶ Chaque φ_i est une conjonction de 3-clauses,
- ▶ φ_i utilise les variables de c_i , + éventuellement de nouvelles variables.
- ▶ Si une affectation des variables rend c_i vraie, on peut la compléter pour rendre φ_i vraie.
- ▶ Inversement, si une affectation des variables rend φ_i vraie, sa restriction aux variables de c_i rend c_i vraie.

Réduction SAT vers 3-SAT : exemple

▶ $\varphi = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4),$
alors

▶ La construction donne

$$\begin{aligned} \tilde{\varphi} = & (t_{1,1} \vee x_1 \vee \neg x_2) \wedge (\neg t_{1,1} \vee x_3 \vee t_{1,2}) \wedge (\neg t_{1,2} \vee \neg x_4 \vee x_5) \\ & \wedge (y_2 \vee x_1 \vee \neg x_2) \wedge (\neg y_2 \vee x_1 \vee \neg x_2) \\ & \wedge (\neg x_1 \vee x_2 \vee x_4) \end{aligned}$$

Réduction SAT vers 3-SAT : construction de φ_i

▶ Si $c_i = \ell_1$ (un littéral), on ajoute 2 variables y_i, z_i et

$$\varphi_i = (\ell_1 \vee y_i \vee z_i) \wedge (\ell_1 \vee \neg y_i \vee z_i) \wedge (\ell_1 \vee y_i \vee \neg z_i) \wedge (\ell_1 \vee \neg y_i \vee \neg z_i).$$

▶ Si $c_i = \ell_1 \vee \ell_2$ (2 littéraux), on ajoute 1 variable y_i et

$$\varphi_i = (y_i \vee \ell_1 \vee \ell_2) \wedge (\neg y_i \vee \ell_1 \vee \ell_2).$$

▶ Si c_i est une 3-clause : $\varphi_i = c_i$.

▶ Si $c_i = \ell_1 \vee \dots \vee \ell_k$ avec $k \geq 4$, on ajoute $k - 3$ variables

$$t_{i,1}, \dots, t_{i,k-3}$$

$$\varphi_i = (t_{i,1} \vee \ell_1 \vee \ell_2) \wedge (\neg t_{i,1} \vee \ell_3 \vee t_{i,2}) \wedge (\neg t_{i,2} \vee \ell_4 \vee t_{i,3}) \wedge \dots \wedge (\neg t_{i,k-3} \vee \ell_{k-1} \vee \ell_k)$$

Réduction SAT vers 3-SAT

On vérifie qu'avec la construction précédente :

- ▶ Si une affectation des variables rend chaque c_i vraie, on la complète facilement pour rendre chaque φ_i vraie.
- ▶ Inversement, si une affectation des variables rend chaque φ_i vraie, la restriction de cette affectation aux variables de c_i rend c_i vraie. Donc

c_i est satisfaisable

\iff

φ_i est satisfaisable avec les mêmes valeurs pour les variables de c_i .

- ▶ Comme les variables ajoutées dans φ_i n'apparaissent que dans φ_i :

$$\boxed{\varphi \text{ est satisfaisable} \iff \tilde{\varphi} \text{ est satisfaisable.}}$$

Réduction SAT vers 3-SAT

Récapitulatif. A partir de φ CNF, on a construit $\tilde{\varphi}$ 3-CNF telle que

$$\varphi \text{ est satisfaisable} \iff \tilde{\varphi} \text{ est satisfaisable.}$$

On a donc

$$\text{SAT} \leq \text{3-SAT}.$$

Inversement, comme 3-SAT est un problème moins général que SAT :

$$\text{3-SAT} \leq \text{SAT}.$$

Réduction 3-SAT vers 3-coloration

- ▶ À toute instance φ de 3-SAT, on associe une instance G_φ de 3-coloration tq.

$$\varphi \text{ est satisfaisable} \iff G_\varphi \text{ admet une 3-coloration.}$$

On utilise des sous-graphes (appelés *gadgets*) pour coder

- ▶ les littéraux vrais dans une évaluation qui satisfait φ ,
- ▶ les opérateurs logiques \wedge et \vee .

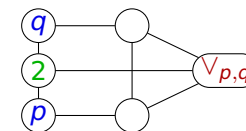
3-coloration

Le problème **3-coloration** est le suivant :

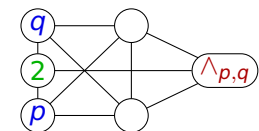
- ▶ **Donnée** : un graphe non orienté G .
- ▶ **Question** : existe-t-il une 3-coloration de G ?

3-SAT \leq 3-coloration

- ▶ On utilise 3 sommets particuliers 0, 1, 2 reliés entre eux, qu'on peut supposer, quitte à renommer les couleurs, coloriés par 0, 1, 2.
- ▶ Pour chaque variable x_i : 2 sommets x_i et $\neg x_i$ reliés entre eux et à 2.
- ▶ Opérateurs : OU $p \vee q$ codé par :



ET $p \wedge q$ codé par :



en reliant $\vee_{p,q}$ et $\wedge_{p,q}$ au sommet 2 (p et q le sont inductivement).

- ▶ $\vee_{p,q}$ coloriable par 1 si et seulement si p OU q sont coloriés 1.
- ▶ $\wedge_{p,q}$ coloriable par 1 si et seulement si p ET q sont coloriés 1.
- ▶ Sommet « résultat » relié à 0 (et 2 par la construction précédente).

Clique et ensemble indépendant

Dans un graphe G non orienté

- ▶ Une **clique** pour G est un ensemble de sommets tous reliés 2 à 2.

Le problème **Clique** est le suivant :

- ▶ **Donnée** : un graphe G non orienté et un entier $K > 0$.
- ▶ **Question** : existe-t-il une clique de G de taille K ?

Réduction 3-SAT vers clique

- ▶ Soit $\varphi = (\ell_0 \vee \ell_1 \vee \ell_2) \wedge \dots \wedge (\ell_{3k-3} \wedge \ell_{3k-2} \wedge \ell_{3k-1})$.
- ▶ Le graphe G_φ a $3k$ sommets ℓ_1, \dots, ℓ_{3k} .
- ▶ Deux sommets ℓ_i, ℓ_k sont reliés si
 - ▶ ils ne proviennent pas de la même clause ($i/3 \neq k/3$), et si
 - ▶ ils ne sont pas de la forme $\ell, \neg \ell$.
- ▶ On choisit l'entier K_φ égal à k .
- ▶ On vérifie que G_φ a une clique de taille K_φ ssi φ est satisfaisable.

Réduction 3-SAT vers clique

- ▶ À toute instance φ de 3-SAT, on associe une instance G_φ, K_φ de **Clique** tq.

φ est satisfaisable $\iff G_\varphi$ a une clique de taille K_φ .

et tq. on peut construire G_φ, K_φ en temps polynomial par rapport à $|\varphi|$.

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP, NP-complétude

Fonction récursives

Complexité en temps

- ▶ Soit M une machine de Turing (s'arrêtant sur toute entrée).
- ▶ Soit $\text{step}_M(w)$ le **nombre maximal** d'instructions exécutées sur l'entrée w jusqu'à l'arrêt de M .
- ▶ Soit $f_M(n) = \max \{ \text{step}_M(w) \mid |w| = n \}$.
- ▶ $f_M(n)$ mesure le temps passé par M sur une entrée de taille n , **dans le pire des cas**.

Classe NP. Remarques et exemples

- ▶ A priori, ce n'est pas parce qu'un problème est dans NP que son complémentaire l'est aussi.
- ▶ On vérifie que les problèmes
SAT, 3-SAT, 3-COLORATION et CLIQUE
sont dans la classe NP.

Les classes P et NP

$$\text{NP} = \{ \mathcal{L}(M) \mid \exists p \text{ polynôme tel que } f_M(n) \leq p(n) \}.$$

$$\text{P} = \{ \mathcal{L}(M) \mid \exists p \text{ polynôme tel que } f_M(n) \leq p(n) \text{ et } M \text{ **déterministe**} \}.$$

- ▶ On a bien sûr $\text{P} \subseteq \text{NP}$.
- ▶ Déterminer si cette inclusion est stricte est un problème ouvert.
- ▶ Pour montrer qu'un problème est NP, on utilise souvent
 - ▶ une phase non-déterministe, où la machine **devine** le résultat.
 - ▶ une phase déterministe, où elle **vérifie** si ce résultat est **correct** en temps polynomial.
- ▶ Exemple : machine qui teste si un nombre **n'est pas premier**.

Les classes P et NP

- ▶ L'appartenance à P ne dépend pas du nombre de bandes utilisées.
- ▶ **Théorème** Si L est décidé par une MT à k bandes déterministe en temps $f(n)$, alors L est décidé par une MT usuelle en $O(f^2(n))$.
Preuve cf. construction.
- ▶ On peut passer d'une MT non déterministe à une MT déterministe au prix d'une exponentielle en complexité en temps.
- ▶ **Théorème** Si L est décidé par une MT non-déterministe en temps $f(n)$, alors L est décidé par une MT usuelle en temps $2^{O(f(n))}$.
Preuve cf. construction.

Réductions polynomiales

- ▶ Soient L_A et L_B deux langages de Σ^* .
- ▶ Une **réduction polynomiale** de L_A vers L_B est une fonction $f : \Sigma^* \rightarrow \Sigma^*$ telle que
 - ▶ f est **calculable par une MT déterministe en temps polynomial**.
 - ▶ On a l'équivalence suivante :

$$x \in L_A \iff f(x) \in L_B.$$

- ▶ On note $L_A \leq_P L_B$ (L_A **se réduit** à L_B de façon polynomiale).
- ▶ L'existence d'une réduction de L_A vers L_B assure que
 - ▶ si $L_B \in P$, alors $L_A \in P$.
 - ▶ si $L_B \in NP$, alors $L_A \in NP$.

Complexité en temps : résumé

- ▶ On s'intéresse à des machines de Turing s'arrêtant sur toute entrée.
- ▶ Les machines non-déterministes ont plusieurs calculs sur un mot.
- ▶ fonction de complexité f_M d'une machine M associe à tout entier n le temps du plus long calcul de M sur une entrée de taille n .
- ▶ **P** = langages décidés en temps polynomial par une MT **déterministe**.
- ▶ **NP** = langages décidés en temps polynomial par une MT.
- ▶ On ne sait pas si $P=NP$, ni si NP est close par complément.
- ▶ Réduction polynomiale : réduction qui se calcule en temps polynomial.
Exemple : de **SAT** vers **3-COLORATION**.
- ▶ Un problème A est **NP-complet** si tout problème NP B se réduit à A en temps polynomial.

Réductions polynomiales

Plusieurs réductions vues précédemment sont bien polynomiales :

$$\text{SAT} \leq 3\text{-SAT} \leq 3\text{-COLORATION} \\ 3\text{-SAT} \leq \text{CLIQUE}$$

Problèmes NP complets

- ▶ Un langage L (ou problème) est **NP-complet** si
 1. $L \in NP$, et
 2. pour tout langage $K \in NP$, on a $K \leq_P L$.
- ▶ On va montrer qu'il existe des problèmes NP-complets.



Ne pas confondre les termes **NP** et **NP-complet**.

Théorème de Cook-Levin

- ▶ **Théorème (Cook, Levin) SAT** est NP-complet.
- ▶ **Conséquence.** D'après les réductions polynomiales vues précédemment, les problèmes 3-SAT, 3-COLORATION, et CLIQUE sont NP-complets.

Théorème de Cook-Levin — idée de preuve

- ▶ Tout calcul de M sur w prend $p(n)$ étapes, où p est un polynôme et $n = |w|$.
- ▶ Donc tout calcul de M sur w utilise au plus $p(n) + 1$ cases.
- ▶ On introduit des variables, avec leur signification intuitive
 - ▶ $Q(i, k)$: **vrai** ssi l'état à l'instant i est q_k .
 - ▶ $P(i, j)$: **vrai** ssi la position à l'instant i est j .
 - ▶ $L(i, j, a)$: **vrai** ssi la lettre à l'instant i dans la case j est a .
- ▶ **Note.** Seulement un nombre **polynomial** de variables !

Théorème de Cook-Levin — principe de la preuve

- ▶ On part d'un langage quelconque de NP, décidé par une MT M .
- ▶ On construit à partir de M et d'un mot d'entrée w une formule $\varphi_{M,w}$ de taille polynomiale, et telle que

M accepte $w \iff \varphi_{M,w}$ est satisfaisable.

Théorème de Cook-Levin — idée de preuve

- ▶ Grâce aux variables Q, P, L , on encode le calcul : À tout instant
 - ▶ on se trouve dans un et un seul état,
 - ▶ une et une seule case est lue,
 - ▶ il y a une et une seule lettre dans chaque case.
- ▶ On code également que
 - ▶ au temps 0, la configuration est $q_0 w$.
 - ▶ au temps $p(n)$, la machine est dans l'état q_{OK} .
 - ▶ entre le temps i et le temps $i + 1$, on applique une transition.

Un autre problème NP-complet : Somme d'entiers

Le problème **Somme d'entiers** est le suivant :

- **Donnée** : des entiers $x_1, \dots, x_k > 0$ et un entier s .
- **Question** : existe-t-il $1 \leq i_1 < i_2 < \dots < i_p \leq k$ tels que

$$x_{i_1} + \dots + x_{i_p} = s.$$

C'est clairement dans NP : on devine i_1, \dots, i_p et on teste.

On montre que c'est NP-complet par une réduction 3-SAT \leq Somme d'entiers.

Réduction Somme d'entiers vers Partition

Soit x_1, \dots, x_k, s une instance de **Somme d'entiers**. Soit $x = \sum x_i$.
On construit (en temps polynomial) l'instance $x_1, \dots, x_k, x - 2s$ de **Partition**.

- Si **Somme d'entiers** a une solution sur x_1, \dots, x_k, s ,
Partition a une solution sur $x_1, \dots, x_k, x - 2s$.
- Inversement, si **Partition** a une solution sur $x_1, \dots, x_k, x - 2s$,
Somme d'entiers a une solution sur x_1, \dots, x_k, s .

Partition

Le problème **Partition** est le suivant :

- **Donnée** : des entiers $x_1, \dots, x_k > 0$.
- **Question** : existe-t-il $X \subseteq \{1, \dots, k\}$ tel que

$$\sum_{i \in X} x_i = \sum_{i \notin X} x_i.$$

C'est clairement dans NP : on devine X et on teste.

Plan

Présentation, bref historique

Ensembles dénombrables. Un paradoxe

Machines de Turing

Machines RAM

Problèmes indécidables

Réductions : logique, graphes, et problèmes sur entiers

Classes de complexité : P, NP, NP-complétude

Fonction récursives

Fonctions calculables

- ▶ Une MT peut être utilisée pour **calculer**.
- ▶ Exemple : pour calculer une fonction de $f : \mathbb{N}^k \rightarrow \mathbb{N}$, on peut utiliser $k + 2$ bandes
 - ▶ k bandes d'entrée pour les arguments,
 - ▶ une bande de calcul,
 - ▶ une bande de sortie, sur laquelle le résultat sera écrit.
- ▶ Lorsque la machine est lancée avec l'argument w sur sa bande d'entrée, elle doit s'arrêter avec $f(w)$ sur sa bande de sortie.
- ▶ On s'intéresse aux fonctions qu'il est possible de calculer par MT.

Exemples de fonctions primitives récursives

- ▶ **Remarque** Les fonctions primitives récursives se calculent en C **sans boucle while()**.
- ▶ Les fonctions suivantes sont primitives récursives.
 - ▶ La fonction prédécesseur : $f(0) = 0$, $f(x) = x - 1$ si $x > 0$.
 - ▶ L'addition, la multiplication de deux entiers.
 - ▶ La différence tronquée : maximum de 0 et de la différence de deux entiers.
$$x \dot{-} y \equiv \max\{0, x - y\}$$
- ▶ Le minimum de deux entiers.
Attention La récursion ne doit porter que sur **un** paramètre.

La classe des fonctions primitives récursives

Plus petite classe de fonctions $\mathbb{N}^k \rightarrow \mathbb{N}$

- ▶ contenant
 - ▶ La fonction nulle de $\mathbb{N}^k \rightarrow \mathbb{N}$ pour tout $k \geq 0$,
 - ▶ La fonction Succ : $\mathbb{N} \rightarrow \mathbb{N}$ telle que $\text{Succ}(x) = x + 1$.
 - ▶ La projection $\pi_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$, telle que $\pi_i^k(x_1, \dots, x_k) = x_i$.
- ▶ fermée par
 - ▶ **composition** : si $f_1, \dots, f_k : \mathbb{N}^n \rightarrow \mathbb{N}$ et $g : \mathbb{N}^k \rightarrow \mathbb{N}$ sont primitives récursives, alors $h : \mathbb{N}^n \rightarrow \mathbb{N}$ l'est aussi, où h est définie par

$$h(x_1, \dots, x_n) = g(f_1(x_1, \dots, x_n), \dots, f_k(x_1, \dots, x_n))$$

- ▶ **récursion** : Si $f : \mathbb{N}^n \rightarrow \mathbb{N}$ et $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ sont primitives récursives, alors $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ l'est aussi, où h est définie par

$$h(0, x_1, \dots, x_n) = f(x_1, \dots, x_n)$$
$$h(x + 1, x_1, \dots, x_n) = g(x, h(x, x_1, \dots, x_n), x_1, \dots, x_n)$$

Propriétés des fonctions primitives récursives

- ▶ La somme, le produit, la différence tronquée de fonctions primitives récursives est primitive récursive.
- ▶ Si $f : \mathbb{N}^{p+1} \rightarrow \mathbb{N}$ est primitive récursive, alors

$$g(n, x_1, \dots, x_p) = \sum_{i=0}^n f(i, x_1, \dots, x_p)$$

et

$$h(n, x_1, \dots, x_p) = \prod_{i=0}^n f(i, x_1, \dots, x_p)$$

sont primitives récursives.

Fonctions définies par cas

- ▶ Une fonction dont les valeurs sont 0 ou 1 est appelée un **prédicat**.
- ▶ On interprète 0 comme **faux** et 1 comme **vrai**.
- ▶ Les opérateurs Booléens classiques, appliqués à des prédicats primitifs récursifs fournissent des prédicats primitifs récursifs.
- ▶ Si $P(x, x_1, \dots, x_p)$ est un prédicat primitif récursif, il en est de même de
 - ▶ $A_P(n, x_1, \dots, x_p) \equiv \forall x \leq n P(x, x_1, \dots, x_p)$
 - ▶ $E_P(n, x_1, \dots, x_p) \equiv \exists x \leq n P(x, x_1, \dots, x_p)$
- ▶ ... car $A_P(n, x_1, \dots, x_p) = \prod_{i=0}^n P(i, x_1, \dots, x_p)$, et $E_P = 1 \dot{-} A_{1 \dot{-} P}$.

Minimisation bornée

- ▶ Si $P(x, x_1, \dots, x_p)$ est un prédicat primitif récursif, la fonction

$$\mu_B P(n, x_1, \dots, x_p) \equiv \begin{cases} \min \{x \leq n \mid P(x, x_1, \dots, x_p)\} & \text{si un tel } x \text{ existe} \\ n + 1 & \text{sinon} \end{cases}$$

est primitive récursive.

- ▶ On a $\mu_B P(n, x_1, \dots, x_p) = \sum_{k=0}^n \prod_{i=0}^k (1 \dot{-} P(i, x_1, \dots, x_p))$.
- ▶ On ne peut pas se passer de **borner** cette minimisation (sinon, la fonction n'est plus nécessairement primitive récursive).

Fonctions définies par cas

- ▶ Si P_1, \dots, P_k sont des prédicats primitifs récursifs, et f_1, \dots, f_{k+1} des fonctions primitives récursives, il en est de même de

$$g(x_1, \dots, x_n) = \begin{cases} f_1(x_1, \dots, x_n) & \text{si } P_1(x_1, \dots, x_n), \\ f_2(x_1, \dots, x_n) & \text{sinon, et si } P_2(x_1, \dots, x_n), \\ \dots & \dots \\ f_k(x_1, \dots, x_n) & \text{sinon, et si } P_k(x_1, \dots, x_n), \\ f_{k+1}(x_1, \dots, x_n) & \text{sinon.} \end{cases}$$

Fonctions calculables, non primitives récursives

- ▶ Ackermann et Sudan ont trouvé en 1927-28 des fonctions
 - ▶ non primitives récursives,
 - ▶ calculables mécaniquement.

$$A(m, x) = \begin{cases} x + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0, x = 0 \\ A(m - 1, A(m, x - 1)) & \text{si } m, x > 0. \end{cases}$$

- ▶ Une variante [D. Kozen] : $B(m, x) = B_m(x)$, où

$$\begin{aligned} B_0(x) &= x + 1 \\ B_{m+1}(x) &= \underbrace{B_m \circ \dots \circ B_m}_{x \text{ fois}}(x) \end{aligned}$$

Fonctions calculables, non primitives récursives

- ▶ On vérifie que les récurrences **terminent**.
- ▶ B **croît trop vite** pour être primitive récursive (idem pour A).

$$B_0(x) = x + 1, B_1(x) = 2x, B_2(x) \geq 2^x$$

$$B_3(x) \geq \underbrace{2^{2^{\dots^2}}}_x = 2 \uparrow x$$

$$B_4(x) \geq 2 \uparrow \uparrow x,$$

$$B_4(2) \geq 2^{2048} \dots$$

où $2 \uparrow (x+1) = 2^{2^x}$, et $2 \uparrow \uparrow (x+1) = 2 \uparrow (2 \uparrow \uparrow x)$, et plus généralement $B_{m+2}(x) \geq 2 \uparrow \dots \uparrow_m \text{ fois}(x)$, où

$$2 \uparrow \dots \uparrow_m (x+1) = 2 \uparrow \dots \uparrow_{m-1} \text{ fois} (2 \uparrow \dots \uparrow_m \text{ fois}(x))$$

On retrouve la définition d'Ackermann.

Bijection $\mathbb{N}^2 \rightarrow \mathbb{N}$ primitive récursive

- ▶ La bijection $c : \mathbb{N}^2 \rightarrow \mathbb{N}$ définie par

$$c(i, j) = \frac{(i+j)(i+j+1)}{2} + i$$

est primitive récursive
(montrer que $n \mapsto n/2$ est primitive récursive).

- ▶ Les fonctions $d_1, d_2 : \mathbb{N} \rightarrow \mathbb{N}$ telles que $c^{-1} = (d_1, d_2)$ sont aussi primitives récursives. Par exemple, pour d_1 , utiliser

$$d_1(x+1) = \begin{cases} 0 & \text{si } \exists y \leq x, \quad x = \frac{y(y+1)}{2}, \\ d_1(x) + 1 & \text{sinon.} \end{cases}$$

La fonction d'Ackermann n'est pas PR

- ▶ On démontre que si $f : \mathbb{N}^p \rightarrow \mathbb{N}$ est primitive récursive, alors il existe k tel que

$$f(x_1, x_2, \dots, x_n) < B_k(\max(x_1, \dots, x_n)). \quad (1)$$

- ▶ Or, la fonction B ne vérifie pas (1).
- ▶ La fonction B n'est donc pas primitive récursive.
- ▶ Idem pour A .
- ▶ (En revanche, chaque B_k est primitive récursive).

Récurrence simultanée

- ▶ On note $\vec{x} = x_1, \dots, x_p$.
- ▶ **Propriété** Si $f_1, \dots, f_k : \mathbb{N}^p \rightarrow \mathbb{N}$ et $g_1, \dots, g_k : \mathbb{N}^{k+p+1} \rightarrow \mathbb{N}$ sont des fonctions primitives récursives, alors les fonctions $h_1, \dots, h_k : \mathbb{N}^{p+1} \rightarrow \mathbb{N}$ définies comme suit le sont aussi.

$$h_i(0, \vec{x}) = f_i(\vec{x})$$

$$h_i(n+1, \vec{x}) = g_i(n, h_1(n, \vec{x}), \dots, h_k(n, \vec{x}), \vec{x})$$

- ▶ **Preuve.** Pour $k = 2$, on utilise le codage $c : \mathbb{N}^2 \rightarrow \mathbb{N}$ et ses décodages d_1 et d_2 pour exprimer $h = c(h_1, h_2)$ à l'aide du schéma de récurrence.

Fonctions récursives

- ▶ Soit $P(z, \vec{x}) : \mathbb{N}^{p+1} \rightarrow \{0, 1\}$ un prédicat **non nécessairement total**, (c'est-à-dire, non nécessairement défini sur tout \mathbb{N}^{p+1}).
- ▶ On écrit $P(z, \vec{x}) = a$ si P est **défini** sur (z, \vec{x}) et vaut a . Soit

$$\mathcal{E}_P(\vec{x}) = \left\{ y \in \mathbb{N} \mid P(y, \vec{x}) = 1 \wedge \bigwedge_{z < y} P(z, \vec{x}) = 0 \right\}.$$

- ▶ $\mathcal{E}_P(\vec{x})$ est soit un singleton, soit l'ensemble vide. Soit $\mu P : \mathbb{N}^p \rightarrow \mathbb{N}$

$$\mu P(\vec{x}) = \begin{cases} \min \mathcal{E}_P(\vec{x}) & \text{si } \mathcal{E}_P(\vec{x}) \neq \emptyset, \\ \text{indéfini} & \text{sinon.} \end{cases}$$

- ▶ La fonction μP **peut ne pas être totale**, même si P l'était.



Attention : malgré les définitions similaires, la minimisation (générale) permet de construire des fonctions non primitives récursives, contrairement à la minimisation bornée.

La classe des fonctions récursives

Plus petite classe de fonctions $\mathbb{N}^k \rightarrow \mathbb{N}$

- ▶ contenant la **fonction nulle**, la fonction **successeur**, les **projections**.
- ▶ fermée par
 - ▶ **composition**,
 - ▶ **réursion**,
 - ▶ **minimisation** : si P est un prédicat récursif, alors μP l'est aussi.

Calcul des fonctions récursives

- ▶ Les fonctions **primitives** récursives sont calculables par programme.
- ▶ Si $P(y, x)$ est un prédicat **récursif**, $\mu P(x)$ se « calcule » par

```
int muP(int x)
{
    y = 0;
    while (P(y, x) == 0)
        y++;
    return y;
}
```

- ▶ Le calcul peut ne pas terminer
 - ▶ soit parce que l'un des appels $P(y, x)$ ne termine pas,
 - ▶ soit parce que $P(y, x)$ vaut toujours 0.

Fonctions récursives et fonctions MT-calculables

Théorèmes

- ▶ Les fonctions récursives sont exactement les fonctions de \mathbb{N}^k dans \mathbb{N} , pour $k \geq 0$, qui sont calculables par machine de Turing.
- ▶ **Conséquence** Il existe une fonction récursive, définie sur \mathbb{N} tout entier, qui n'est pas primitive récursive.
- ▶ Il existe des fonctions non récursives.