UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

Année universitaire 2024-2025
Licence d'informatique
MAIN0602

Calculabilité, Décidabilité, Complexité

version du 30 janvier 2026

Pascal Mignot
Pascal.Mignot@univ-reims.fr
Département Mathématiques et Informatique
Bureau n°20

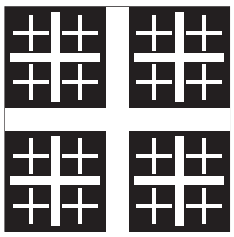


Table des matières

Chapitre I

Modèles abstraits de machines

Normalement, ce cours devrait commencer par l'étude du modèle des machines de Turing.

Nous allons repousser un peu l'étude de ce modèle formel afin de vous permettre d'en assimiler les prérequis dans le cours sur les langages.

Dans ce chapitre, nous allons donc expliquer ce qui doit être compris dans le cours sur les langages.

- tout objet informatique peut être codé comme un mot d'un langage formel,
- résoudre un problème informatique décisionnel est équivalent à reconnaître un langage.

1 Alphabets et mots

1.1 Prérequis

Définition I.1 (cardinal d'un ensemble)

Le cardinal d'un ensemble E est le nombre d'éléments qu'il contient.
On le note $\#E$.

Définition I.2 (ensemble fini)

Un ensemble E est fini s'il existe $k \in \mathbb{N}$ tel que $\#E = k$.

Autrement dit, un ensemble fini n'est pas infini.

Exemples 1:

- L'ensemble des chiffres en base 10 est fini.
- L'ensemble des nombres en base 10 est infini.

Définition I.3 (Ensemble des parties d'un ensemble)

On appelle l'ensemble des parties d'un ensemble E l'ensemble des ensembles qu'il est possible de construire à partir de cet ensemble.

Notation : on note $\mathcal{P}(E)$ l'ensemble des parties de l'ensemble E.

Exemple 2: Soit l'ensemble $E = \{a, b, c\}$.
Alors $\mathcal{P}(E) = \{\varepsilon, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$

Remarque 0: : $\#\mathcal{P}(E) = 2^{\#E}$
En effet, pour tout sous-ensemble de E, chaque élément a 2 choix : y appartenir ou pas. Comme il y a $\#E$ élément dans E, il est donc possible de construire $2^{\#E}$ ensembles différents.

1.2 Alphabet

Définition I.4 (alphabet)
Un **alphabet** Σ est un ensemble **fini** de symboles.

Exemples 3: d'alphabets

- $\Sigma = \{a, b, ..., z\}$: alphabet des lettres (de l'alphabet).
- $\Sigma = \{0, 1\}$: alphabet binaire.
- $\Sigma = \{0, 1, 2, ..., 9\}$: alphabet des chiffres de la base 10.
- ...

Remarques 1:

- Le symbole Σ dénotera toujours un alphabet.
- Si sa définition n'est pas précisée, on prendra $\Sigma = \{0, 1\}$.
- **(redite)** La taille d'un alphabet est toujours finie.

1.3 Mot

Définition I.5 (mot)
Un **mot** issu d'un alphabet Σ est une suite finie de symboles dont chacun appartient à Σ.

Remarque : Un mot peut être vide. On note ε le mot vide.

Exemples de mots :

- d est un mot issu de l'alphabet $\{a, b, ..., z\}$.
- 01101 est un mot issu de l'alphabet $\{0, 1\}$.
- ε est un mot issu de tout alphabet.

Définition I.6
La longueur d'un mot est le nombre total de symboles dont il est composé.

Notation : si m est un mot, on notera $|m|$ sa longueur.

Exemples :

- $|\varepsilon| = 0$.
- si $m = 01101$, alors $|m| = 5$.

On note m^k représente le mot m répété k fois.
exemple : si $m = 01$ alors $m^5 = 0101010101$

Définition I.7 (opérateurs réguliers sur des mots)

L'ensemble des opérations régulières est :
- l'**union** $\cup$: si m_1 et m_2 sont deux mots, alors $m_1 \cup m_2$ est l'ensemble $\{m_1, m_2\}$.
- la **concaténation** $\circ$: si m_1 et m_2 sont deux mots, alors $m_1 \circ m_2 = m_1 m_2$.
- l'**opérateur *** (étoile de Kleene) : si m est un mot, alors m^* est la répétition de m un nombre quelconque de fois, à savoir $m^* = \{\varepsilon\} \cup \bigcup_{k=1}^{\infty} \{m^k\}$.

Exemples 4: si m_1 = ab et m_2 = c

- $m_1 \cup m_2$ est l'ensemble de mots $\{ab, c\}$.
- $m_1 \circ m_2$ est le mot abc.
- ${m_1}^*$ est l'ensemble de mots $\{\varepsilon, ab, abab, ababab, abababab, \dots\}$.

Par extension, on notera :

- Σ^k l'ensemble tous les mots de longueur k issus de l'alphabet Σ.
  exemple : si $\Sigma = \{0, 1\}$, alors $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- Σ^* l'ensemble des mots de toute taille qu'il est possible de construire avec l'alphabet Σ, y compris la chaîne vide ε.
  $\Sigma^* = \{\varepsilon\} \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$
  exemple : si $\Sigma = \{0, 1\}$, alors Σ^* contient toutes les chaînes binaires possibles de longueur finie.
  Attention, Σ^* ne contient pas les chaînes binaires de longueur infinie.

2 Langages

2.1 Langage formel

Définition I.8 (langage formel)

Un **langage** L défini sur un alphabet Σ est un sous-ensemble de Σ^*.

Un langage L sur Σ est un ensemble de mots construit à partir de l'alphabet Σ.

Notes :

- un langage peut ne contenir aucun mot (i.e. $L = \emptyset$).
- le nombre de mots dans un langage n'est pas nécessairement fini.

Exemples de langages :

- pour $\Sigma = \{a, b, ..., z\}$, l'anglais est un langage.
- pour $\Sigma = \{0, 1, ..., 9\}$, les ensembles suivants sont des langages :
  - les nombres entre 1 et 100

  - les nombres pairs
  - les nombres premiers
- pour $\Sigma = \{0, 1\}$, les ensembles suivants sont des langages :
  - $\{w$ tels que w contient au moins 12 fois le symboles $0\}$
  - $\{0^n 1^n$ tel que $n \geq 0\}$
  - $\{w$ tels que w a le même nombre de symboles 0 et $1\}$

2.2 Opérateurs réguliers sur un langage

Définition I.9 (opérateurs réguliers sur des langages)

Soit L_1 et L_2 deux langages. L'ensemble des opérations régulières est :
- l'**union** $\cup$: $L_1 \cup L_2 = \{m \mid m \in L_1 \text{ ou } m \in L_2\}$.
- la **concaténation** $\circ$: $L_1 \circ L_2 = \{m_1 m_2 \mid m \in L_1 \text{ et } m \in L_2\}$.
- l'**opérateur *** : $L^* = \{\varepsilon\} \cup \bigcup_{k=1}^{\infty} \{m_1 m_2 \dots m_k \text{ où } \forall i, m_i \in L\}$.

Note : le résultat d'un opérateur régulier sur un ou plusieurs langages est un langage.

Exemple 5: Soit le langage $L_1 = \{0, 11\}$ construit sur l'alphabet $\Sigma_1 = \{0, 1\}$ et $L_2 = \{a, bb, ccc\}$ construit sur l'alphabet $\Sigma_2 = \{a, b, c\}$.

- $L_1 \cup L_2$ est le langage $\{0, 11, a, bb, ccc\}$.
- $L_1 \circ L_2$ est le langage $\{0a, 0bb, 0ccc, 11a, 11bb, 111ccc\}$.
- ${L_1}^*$ est le langage {ε, 0, 11, 00, 011, 110, 1111, 000, 0011, 0110, 01111, 1100, 11011, 11110, 111111, 0000, 00011, 00110, 001111, 01100, 011011, 011110, 0111111, 11000, 110011, 110110, 1101111, 111100, 1111011, 1111110, 11111111, ...}.
  Cet ensemble n'est pas de taille finie.

2.3 Langages réguliers

Définition I.10 (Langage régulier)

Un langage sur un alphabet Σ est régulier s'il peut être défini comme un ensemble fini de mots de Σ^* ou de toute composition finie d'opérateurs réguliers sur Σ.

Exemples 6: sur $\Sigma = \{0, 1\}$

- $\{010\} \cup \{010\}$ est un langage régulier.
- $\{01\}^*$ est un langage régulier.
- $(\{010\} \cup \{010\}) \circ \{01\}^*$ est un langage régulier.
- Σ^{10} est un langage régulier (l'ensemble des mots binaires de taille 10 peut être construit comme une union finie de mots de taille 10).
- $\{0^n 1^n \mid \forall n \leq 10\} = \varepsilon \cup \{01\} \cup \{0011\} \cup \dots \cup \left\{0^{10} 1^{10}\right\}$ est un langage régulier.
- $\{0^n 1^n \mid \forall n > 0\}$ n'est pas un langage régulier, car cet ensemble ne peut pas être écrit comme une composition finie d'opérateurs réguliers.

Remarques 2:

- On notera que tout langage fini est un langage régulier, car il peut être écrit comme l'union finie des mots qu'il contient.
- Le lemme de l'étoile permet de vérifier si un langage régulier (voir votre cours sur la théorie des langages).

2.4 Langages algébriques

Définition I.11 (Grammaire algébriques)

Une grammaire algébrique G est un ensemble de règles de construction de mots basés sur :

- un alphabet Σ.
- une variable de départ,
- des variables intermédiaires,
- des règles de transformations qui transforme une variable en toute concaténation de variables intermédiaires et de symboles de l'alphabet.

Exemple 7: Soit la grammaire algébrique G définie par l'alphabet $\Sigma = \{0, 1\}$, la variable de départ S, la variable intermédiaire U, et les règles de transformation $S \to 1U0$, $U \to 0U1$ et $U \to \varepsilon$.

Définition I.12 (Mot engendré par une grammaire algébrique)

Un mot w est engendré par une grammaire algébrique G :

- en partant de la variable de départ,
- en appliquant des règles de transformation jusqu'à ce que le mot obtenu ne contiennent plus que des symboles de l'alphabet.

Exemple 8: $S \overset{S\to 1U0}{\longrightarrow} 1U0 \overset{U\to 0U1}{\longrightarrow} 10U10 \overset{U\to 0U1}{\longrightarrow} 100U110 \overset{U\to \varepsilon}{\longrightarrow} 100110.$

Définition I.13 (Langages algébriques)

Un langage algébrique est un langage qui contient l'ensemble des mots engendrés par une grammaire algébrique.

Exemple 9: L'ensemble des mots engendrés par la grammaire précédente est $\left\{10, 1010, 10^2 1^2 0, 10^3 1^3 0, \ldots\right\}$.

3 Langages et modèles

Les langages réguliers et algébriques ont des applications directes en informatique, par exemple :

- avec les expressions régulières pour les langages réguliers,
- avec la syntaxe de tout objet informatique (langages programmations, représentation des données, …) pour les langages algébriques.

Afin de reconnaître les mots issus d'un langage L, des modèles de machines élémentaires sont utilisés. On utilise :

- un automate fini permet de reconnaître si une chaîne appartient à un langage régulier.
- un automate à pile permet de vérifier si un code n'a pas d'erreur de syntaxe ou si des données lues respectent bien une structure prédéterminée.

Si A est l'automate qui reconnaît le langage L, alors $A(w)$ accepte si $w \in L$, et rejette sinon.

Chaque automate est donc spécialisé pour un langage particulier.
Essentiellement, ces automates sont des machines à état auxquels on donne un mot en entrée :

- dont les symboles sont lus l'un après l'autre (dans l'ordre),
- chaque symbole lu provoque un changement d'état (= une transition),
- et qui n'est accepté que si le dernier symbole lu conduit à un état acceptant, et rejeté sinon.

De manière un peu plus formelle,

- si L est un langage régulier, alors il existe un automate fini A tel que, pour tout mot w de L, $A(w)$ accepte si $w \in L$, et rejette sinon.
- si L est un langage algébrique, alors il existe un automate à pile A tel que, pour tout mot w de L, $A(w)$ accepte si $w \in L$, et rejette sinon.

L'inverse est aussi vrai : si A est un automate fini (resp. à pile), alors le langage L reconnu par A est un langage régulier (resp. algébrique).

redite : un automate A ne reconnaît donc qu'un seul langage L. On note alors $L = \mathcal{L}(A)$ afin d'indiquer que le langage reconnu par A est L.

3.1 Non déterminisme

Un automate fini est par essence **déterministe**.

On entend pas là que, pour chaque état dans lequel l'automate se trouve, il existe n'existe qu'un seul choix possible pour traiter le symbole courant du mot en cours d'analyse, et passer à l'état suivant (= faire une transition).

Définition I.14 (Transition non déterministe)

Une transition est non déterministe s'il peut exister plusieurs choix possibles pour traiter le symbole courant, voir même aucun.

Dans ce cas,

- s'il existe plusieurs choix, la machine se duplique et poursuit toutes les exécutions possibles (= plusieurs branches d'exécution)
- s'il n'existe aucun choix, la branche d'exécution s'arrête.
- le mot est accepté si au moins une branche accepte.

Remarques 3:

- Un automate fini peut s'écrire sous forme déterministe ou non déterministe.
- Un automate à pile est, par définition, non déterministe.

Une exécution non déterministe a les caractéristiques suivantes :

- l'exécution d'une machine peut se représenter sous forme d'un arbre d'exécution, où chaque nœud correspond à un choix non déterministe (son arité est le nombre de choix).
- le temps d'exécution d'une machine non déterministe est celui de la branche acceptante de plus petite profondeur.
- le nombre de branches non déterministe est potentiellement exponentiel.

Le **non-déterminisme n'est pas du parallélisme** au sens où :

- lancer des threads supplémentaires à chaque choix non déterministe augmente la charge de la machine,
- le temps d'exécution dépendra nécessairement du nombre de processeurs nécessaires, ou du nombre de qubits sur un ordinateur quantique.

En conséquence, une machine non déterministe n'est pas physiquement réalisable.

3.2 Limitations

Mais ces modèles souffrent de beaucoup de limitation :

- tout ce qui n'est pas de l'ordre de l'expression régulière ou relever de la syntaxe ne peut pas être reconnu.
- un automate ne reconnaît qu'un seul langage (un automate n'est pas programmable). Ils sont donc l'équivalent d'un processeur spécialisé qui ne serait câblé pour ne réaliser qu'une seule tache.

Vous allez étudier essentiellement deux types de modèles en théorie des langages :

- les automates finis
- les automates à pile

Leurs capacités sont néanmoins bien trop limitées. Par exemple, on peut :

- savoir si un mot est un palindrome mais pas s'il est de la forme ww où w est un autre mot,
- savoir si un mot contient un entier en base 10, mais pas s'il est premier,
- savoir si un mot contient la description syntaxique d'un graphe, mais pas si le graphe est cohérent correctement,
- ...

3.3 Codage

Convenons tout d'abord qu'une chaîne de symboles aléatoires n'a pas de sens propre (tout ce à quoi on peut donner du sens dans une telle chaîne est le résultat d'un événement aléatoire).

Le codage d'une information avec les symboles d'un alphabet est obtenu :

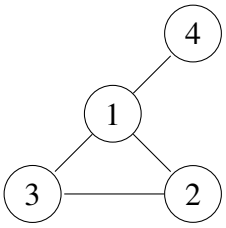
- en donnant un sens aux symboles,
- en organisant ceux-ci de manière à ce que leur interprétation ait un sens.

Autrement dit, un codage est la grammaire d'un langage.

Exemple 10: Le codage de l'entier 212 en binaire est la chaîne "11010100" ne prend son sens qu'en prenant chacun de ses symboles $s_7 \dots s_1 s_0$, et en construisant sa valeur $v = \sum_{i=0}^{7} s_i 2^i$.
Un autre codage binaire serait "AAZAZAZZ" avec le sens A=1 et Z=0.
Donc le codage d'un entier est déterminé par le choix de la base (ou toute autre méthode de codage) et par le choix de l'alphabet (sens des symboles).

On peut ainsi par exemple définir les grammaires suivantes :

- un **entier** avec comme une suite finie de chiffres par parmi $\{0, 1, 2, \ldots, 9\}$.
- une **liste d'entiers** $(12, 34, 56, 78)$ avec une "liste d'entiers séparés par des virgules et délimitée par des parenthèses."
- **un graphe** avec la "composition d'une liste d'entiers représentant les sommets, et d'une liste de couples d'entiers représentant les arêtes entre les sommets".



$(1; 2; 3; 4)((1; 2); (2; 3); (3; 1); (1; 4))$

- ...

Évidemment, différents types de codage auraient pu être utilisés pour ces mêmes objets. **Le codage utilisé n'a pas d'importance**, seul compte le fait que cela peut être codé.
Tout objet utilisable dans un traitement informatique peut ainsi être codé sous forme de grammaire.

Remarque 4:

On notera que tout mot accepté par la grammaire ci-dessus associée a un graphe ne représente pas nécessairement un graphe.

- la liste des arêtes ne doit utiliser des identifiants de sommet se trouvant dans le liste de sommets,
- chaque sommet dans la liste des sommets doit avoir un identifiant unique,
- ...

Il peut donc exister des mots **grammaticalement corrects** mais **sémantiquement faux**, et qui nécessitent des vérifications supplémentaires.

Remarque 5:

On utilisera les notations suivantes :

- si O est un objet, alors $\langle O \rangle$ représente l'encodage d'un objet,
- si $O_1, O_2, \ldots O_k$ sont des objets, alors $\langle O_1, O_2, \ldots O_k \rangle$ représente l'encodage de tous ces objets,

et ceci sans se soucier **ni de l'alphabet, ni du codage utilisé à cet effet**.

Exemples 11:

- `MATRIX` = $\{\langle M \rangle \mid M$ est une matrice$\}$
  `MATRIX` est le langage constitué de l'ensemble des mots qui représentent une matrice.
- `INT_LIST` = $\{\langle L \rangle \mid L$ est une liste d'entiers$\}$
  `INT_LIST` est le langage constitué de l'ensemble des mots qui représentent une liste d'entiers.
- `EXAMPLE` = $\{\langle M, L_1, L_2 \rangle \mid M \in$ `MATRIX` et $L_1, L_2 \in$ `INT_LIST`$\}$
  `EXAMPLE` est le langage constitué d'un triplet constitué d'une matrice et de deux listes d'entiers.

4 Introduction aux machines de Turing

4.1 Modèle théorique

Une machine de Turing M est essentiellement un automate déterministe fini avec une bande de mémoire dans laquelle il peut, lire, écrire et se déplacer. Le mot à analyser est placé au début la bande.

Les concepts d'automate fini étant abordé en cours de langage, nous repoussons l'étude du fonctionnement exact de la machine de Turing pour un peu plus tard.

Comme pour les automates précédents, une MT M qui reconnaît un langage L, et fonctionne de la même manière suivantes :

- si $w \in L$, alors $M(w)$ accepte,
- si $w \notin L$, alors $M(w)$ rejette ou boucle.

De la même façon qu'un ordinateur peut boucler, la possibilité qu'une MT le puisse également n'est donc pas étonnante.

Au final, une MT a les mêmes capacités qu'un ordinateur pour reconnaître un langage.
Avant de décrire le modèle théorique, nous considérerons qu'une machine de Turing comme un ordinateur classique qui :

- prend en paramètre une chaîne de symboles,
- ne sait répondre que deux choses : oui (accepter) ou non (rejet) suivant que le paramètre appartienne ou non au langage reconnu par la machine.

et pour laquelle la description de ce qu'elle exécute est décrit sous forme d'un pseudo-code **effectuant uniquement des opérations élémentaires**.

Exemple 12: de machine qui reconnaît le langage $L = \{xx \mid x \in \Sigma^*\}$ Ce langage contient tous les mots qui contient un mot qui se répète deux fois. Un code de MT M qui reconnaît L pourrait être :

$M(w) =$

```
n = |w|
SI n == 0 OU mod(n,2) == 1 ALORS REJETER
p = n/2
POUR i = 0 à p-1
    SI w[i] != w[p+i] ALORS REJETER
ACCEPTER
```

4.2 Machine de Turing universelle

Pour les MTs classiques, nous avions :

- $\mathcal{L}(M)$ est le langage reconnu par la MT M.
- M_L indique que la MT qui reconnaît le langage L.
  Évidemment, $\mathcal{L}(M_L) = L$.
- l'ensemble des mots reconnus par une MT M peut s'écrire :
  $\mathcal{L}(M) = \{w | M(w) \text{ accepte}\}$.

Une MT universelle U est une MT qui permet de simuler une autre MT M sur n'importe quelle entrée w, à savoir :

$$U(\langle M, w\rangle) = \begin{cases} \text{accepte} & \text{si } M(w) \text{ accepte} \\ \text{rejette} & \text{si } M(w) \text{ rejette ou boucle} \end{cases}$$

Le langage reconnu par U est :

$$\mathcal{L}(U) = \{\langle M, w\rangle \mid M(w) \text{ accepte}\}$$

Ce langage contient l'ensemble des mots couples de codes des machines M et d'entrées w tels que M accepte w.

Une MTU est donc un modèle de MT programmable.

L'existence d'une MTU implique la possibilité d'écrire son code équivalent :

$U(\langle M, w\rangle) =$
```
EXÉCUTER M(w)          // simulation de M sur l'entrée w
DÉCIDER comme M
```

On peut ainsi écrire des codes permettant de contrôler d'appeler d'autres machines ou de contrôler finement l'exécution :

1. **appel à plusieurs machines :**

   $D(\langle M_1, M_2, w\rangle) =$
   ```
   EXÉCUTER M1(w)          // simulation de M1 sur l'entrée w
   EXÉCUTER M2(w)          // simulation de M2 sur l'entrée w
   SI M1 ET M2 acceptent ALORS ACCEPTER SINON REJETER
   ```

2. **exécution pas à pas plusieurs machines en parallèle :**

   $D(\langle M_1, M_2, w\rangle) =$
   ```
   RÉPÉTER
       EXÉCUTER M1(w) sur une transition
       EXÉCUTER M2(w) sur une transition
       SI M1 ACCEPTE ALORS ACCEPTER
       SI M2 ACCEPTE ALORS ACCEPTER
   ```
   L'exécution sur une transition signifie que la machine s'exécute pas à pas.
   Ceci est utile lorsqu'on exécute des machines qui peuvent boucler.

3. **construction du code d'une machine à exécuter :**

   $D(\langle M, w\rangle) =$
   ```
   CONSTRUIRE le code:
       M1(x) = SI x ≠ w ALORS REJETER
               EXÉCUTER M(⟨w⟩)
               DÉCIDER M
   EXÉCUTER M1(w)
   DÉCIDER comme M1
   ```

Exercice 1 (Langage et machine de Turing)**.** *Donner les langages reconnus par chacune des machines de Turing précédentes.*
Ces langages L seront écrits sous la forme :

$$L = \{\langle w\rangle \mid \textit{ensemble des mots } w \textit{ tels que } \ldots\}$$

où la forme de $\langle w\rangle$ sera adaptée en fonction du langage.
Pour la dernière machine, on commencera par décrire le langage reconnu par la machine M_1.

4.3 Fonction calculable

Une fonction calculable est un type de MT qui retourne une valeur.

Définition I.15 (fonction (totalement) calculable)

Une fonction totalement calculable est une MT qui évalue une fonction $f : \Sigma^* \to \Sigma^*$. Elle commence avec l'entrée w sur sa bande, et s'arrête pour tout w avec la valeur $f(w)$ écrit sur sa bande.

Définition I.16 (fonction partiellement calculable)

Une fonction partiellement calculable est MT qui évalue une fonction f dont la valeur $f(w)$ ne peut pas être calculée pour certaines entrées w. Dans ce dernier cas, la MT retourne alors un caractère spécial $\perp$ ou boucle.

Remarques 6:

- marche aussi pour les fonctions avec plus d'une variable.
  Encoder les paramètres sur la chaîne d'entrée intercalée de séparateurs.
  Chaque paramètre peut représenter n'importe quel objet.
- même remarque pour la sortie.
- une bande particulière peut être utilisée pour la sortie.
- Toutes les fonctions arithmétiques classiques sur les entiers sont totalement calculables : addition, soustraction, multiplication, division (quotient, reste), ...
- Toutes les fonctions non-arithmétiques sont également totalement calculables : trigonométrique, logarithmique, ... en utilisant les séries de Taylor (ou d'autres techniques d'approximation) à une précision spécifiée.

  voir "Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables" d'Abramowitz et Stegun sur la façon pratique de réaliser ces approximations à une précision donnée.

5 Modèle de calcul

Un modèle de calcul peut s'interpréter comme une évaluation fonctionnelle de la forme $y = f(x)$ où f est la fonction qui résout le problème de trouver la sortie y à partir de l'entrée x.

L'adaptation d'un tel modèle pour correspondre à un calcul informatique suppose :

- le choix d'un alphabet Σ afin de coder des informations sous forme de chaînes de symboles,
- un codage de l'entrée x comme un mot de Σ^*,
- un modèle abstrait de machine capable de traiter des chaînes de symboles et de produire une sortie.

Exemple 13: Si on prend un ADF comme modèle abstrait de machine, alors il permettra de résoudre tous les problèmes devant déterminer si une chaîne de symboles appartient à un langage régulier particulier.

5.1 Problèmes

Nous allons en considérer deux types de problèmes résolubles par un traitement informatique :

- les problèmes de décision (ceux dont la réponse est oui ou non).
- les problèmes d'évaluation (ceux dont le but est de calculer une réponse).

5.1.1 Problèmes de décision

Un problème de décision est un problème dont la réponse ne peut être que oui ou non.

Exemples 14: de problèmes de décision

- Décider si une chaîne binaire contient un nombre pair de 0.
- Décider pour un couple (x, y) si x divise y.
- Décider si un nombre entier x est premier.
- Décider si une liste d'entiers est triée.
- Décider si une équation à coefficients entiers à une solution entière.
- ...

Soit un alphabet Σ quelconque.

Définition I.17 (Problème de décision)

Un problème de décision est une fonction $f : \Sigma^* \to \{0, 1\}$ telle que, pour toute entrée $w \in \Sigma^*$, $f(w) = 1$ si la décision est oui pour w et $f(w) = 0$ si la décision est non.

Remarque 7:

Un problème de décision permet de définir un langage.
Reprenons l'exemple de test de parité, avec $\Sigma = \{0, \ldots, 9\}$ et écrivons le problème de décision comme la fonction $f : \Sigma^* \to \{0, 1\}$ telle que $f(w) = 1$ si et seulement si w est un nombre premier.
Soit le langage PRIME = $\{w \in \Sigma^*$ tels que x est nombre premier$\}$.
Alors il est défini de manière équivalente comme :
PRIME = $\{w \in \Sigma^*$ tels que $f(w) = 1\}$
En conséquence, tout problème de décision définit un langage.

Remarque 8:

La définition n'exclue pas que la fonction f puisse boucler, donc puisse ne jamais prendre de décision.

5.1.2 Problèmes d'évaluation

Un problème d'évaluation est un problème pour lequel on cherche à évaluer au moins une solution.

Pour cela, on note :

- soit x un problème.
- soit $R(x)$ l'ensemble des solutions au problème x.
- soit R l'ensemble des couples tels que x est un problème et $y \in R(x)$.

Définition I.18 (Problème d'évaluation)

Soit un problème d'évaluation $R \subseteq \Sigma^* \times \Sigma^*$.
Une fonction $f : \Sigma^* \to \Sigma^*$ résout un problème d'évaluation R si :

1. si l'ensemble des solutions au problème x n'est pas vide, alors $f(x)$ retourne l'une des solutions (*i.e.* si $R(x) \neq \emptyset$, $f(x) \in R(x)$).
2. s'il n'y a pas de solution, alors $f(x) = \perp$ (sortie spécifique pour signaler le problème)

Exemples 15: de problème d'évaluation

- Évaluer le zéro d'une fonction.
  si $x =$ '$X \mapsto X^2 - 1$', alors $R(x) = \{-1, +1\}$ et $f(x) = 1$ ou $f(x) = -1$.

- Trouver la décomposition en facteur premier d'un entier.
  si $x = 18$, alors $R(x) = \left\{2 \cdot 3^2\right\}$ et $f(x) = 2 \cdot 3^2$
  si $x = 0$, alors $R(x) = \emptyset$ et $f(x) = \perp$
- Trouver un diviseur strict $d > 1$ d'un entier.
  si $x = 18$, alors $R(x) = \{2, 6, 9\}$ et $f(x) = 2$, 3 ou 9.
- Trouver un chemin entre deux sommets d'un graphe.
  si $x = \langle G, a, b\rangle$ alors $R(X) = \{\langle p_1\rangle, \langle p_2\rangle\}$ où on suppose que p_1 et p_2 sont deux listes de sommets décrivant un chemin dans G, commençant par a et terminant par b. Donc $f(x) = \langle p_1\rangle$ ou $\langle p_2\rangle$.
- Trouver le chemin le plus court entre deux sommets d'un graphe.
  si $x = \langle G, a, b\rangle$ alors $R(X) = \{\langle p\rangle\}$ où p est le chemin le plus court dans G parmi les deux précédents. D'où $f(x) = \langle p\rangle$.
- ...

Il y a une fonction f particulière par problème d'évaluation à résoudre. Noter que x et $f(x)$ sont toujours codés dans Σ^*.

5.1.3 Considérations

Les deux modèles abstraits principaux de calcul que nous allons étudier sont :

- **les machines de Turing**
  ce modèle est fait pour reconnaître les mots d'un langage.
  il est adapté aux problèmes de décision.
- **les fonctions calculables**
  ce modèle identiques aux machines de Turing mais capable de produire une sortie.
  il est adapté aux problèmes d'évaluation.

On se restreindra dans un premier temps à des problèmes décisionnels :

- plus simple dans un premier temps
- ne pose pas de problèmes fondamentaux.

Noter que pour les problèmes d'évaluation, dire si un problème a oui ou non une solution implique souvent de trouver cette solution (par exemple, tel entier peut-il être factorisé = trouver cette factorisation si elle existe, et décider selon).

6 Hypothèse de Church-Turing

6.1 Algorithmes

Dans ce contexte plus général, la fonctionnement d'une MT devient le suivant :

- soit P un objet mathématique (resp. une structure de données) sur lequel on se pose une question mathématique (resp. algorithmique) à laquelle il est possible de répondre par oui ou par non,
- soit M une MT qui prend en entrée $\langle P\rangle$

Alors, le code de la MT M se structure de la manière suivante :

- analyse de la syntaxe de $\langle P \rangle$,
- vérification que $\langle P \rangle$ est un codage cohérent de l'objet représenté,
- application sur P de l'algorithme permettant de résoudre cette question (s'il en existe un exécutable sur une MT)

Ce qui amène aux questions :

- qu'est-ce-qu'un algorithme ?
- est-ce-que tout algorithme peut-être exécuté par une MT ?
- est-ce-que tout problème algorithmique peut être résolu par un algorithme ?

Qu'est-ce qu'un algorithme ?

- ☐ une recette ?
- ☐ une procédure ?
- ☐ un programme sur un ordinateur ?
- ☐ quelle importance ? Je le sais quand j'en vois un !

Historiquement :

- la notion intéresse les mathématiciens depuis longtemps
  L'algorithme d'Euclide (300 av.J.-C.) pour le calcul du PGCD
- pas précisément définie avant le $20^{\text{ème}}$ siècle.

La notion était informelle, mais suffisante jusque là.

Avec l'étude mathématique des algorithmes, il devient maintenant nécessaire de définir rigoureusement la question.

6.2 Thèse de Church-Turing

D'où la **thèse de Church-Turing** :

> "La notion intuitive d'un modèle raisonnable de calcul informatique est équivalente à un algorithme s'exécutant sur une machine de Turing."

Donc,

- tout programme que s'exécute sur une machine de Turing est algorithme.
- tout algorithme peut être exécuté sur une machine de Turing.

D'après cette hypothèse, tout algorithme exécuté sur n'importe quelle machine de calcul physiquement réalisable peut aussi être exécuté sur une machine de Turing.
y compris pour un ordinateur quantique (et une machine de Turing quantique)

Attention, ceci n'est qu'une hypothèse.
mais elle n'a pas été démentie jusqu'à présent.

Cette thèse restera probablement valide jusqu'à ce que l'on conçoive une machine physique qui ne puisse pas être simulée sur une MT.
Il existe aussi une forme forte de l'hypothèse de Church-Turing :

> La notion intuitive d'un modèle raisonnable de calcul informatique est équivalente à un algorithme s'exécutant sur une machine de Turing avec une pénalité au plus d'ordre polynomial.

Il n'est pas encore vraiment clair que cette thèse ne soit pas vraie :

- les MTNDs n'ont actuellement pas d'implémentation physique,
- les ordinateurs quantiques sont de bons candidats pour invalider cette thèse,
  - — Un modèle de machine de Turing quantique (MTQ) a été créé afin d'étudier les propriétés des ordinateurs quantiques tels qu'on les conçoit.
    La thèse (faible) de Church-Turing reste vérifiée (= les MTQs ne reconnaissent pas d'autres langages que les MTs).
  - — Aucune méthode n'a été trouvée pour simuler efficacement une MTQ sur une MT (= pénalité exponentielle).
  - — Il n'est pas clair que les implémentations physiques des ordinateurs quantiques actuels utilisent bien les propriétés quantiques (la superposition d'état) qui rendent les ordinateurs quantiques intéressants.

7 Complétude de Turing

La MT est donc l'outil qui permet de simuler :

- tout modèle de machine physiquement réalisable,
- tout algorithme qu'il est possible d'écrire

En fait : Tous les modèles de langages de programmation "raisonnables" sont équivalents.
Java, C++, Lisp, Scheme, Prolog, Mathematica, Maple, Cobol, ...

Chacun de ces langages représente un formalisme en mesure de représenter plus facilement certains types d'algorithmes.

Définition I.19 (Complétude de Turing)

Le formalisme d'une machine est **Turing-complet** s'il permet de simuler une MT.

Donc, **Turing-complet** = peut exécuter tout algorithme.

8 Résumé

Nous avons vu dans ce cours que :

- une machine de Turing est modèle théorique qui permet de représenter l'exécution d'un algorithme sur un ordinateur,
- une fonction calculable est une machine de Turing qui s'arrête avec le résultat de la fonction écrit sur la bande.
- d'après la thèse de Church-Turing, un algorithme s'exécutant sur une machine de Turing est équivalent à un modèle raisonnable de calcul informatique,
- tout modèle raisonnable de calcul peut être simulé sur une machine de Turing,
- cette simulation peut être effectuée avec un facteur multiplicatif de pénalité au plus quadratique, exception faire des MTs quantiques.
- un modèle de machine est Turing-complet s'il permet de simuler tout algorithme (=simuler une machine de Turing),
- le modèle de la machine de Turing a servi de base pour concevoir le principe des ordinateurs modernes.

Chapitre II

Décidabilité, calculabilité, réductibilité

Introduction

Ce chapitre se décompose de la manière suivante :

- **le problème de la décidabilité :**
  Les problèmes de décision peuvent-ils toujours être décidé ?
  Si non, lesquels peuvent l'être ?
- **le problème de la calculabilité :**
  Les problèmes d'évaluation peuvent-ils toujours être calculé ?
- **le principe de réductibilité :**
  Principe de démonstration permettant de déduire les propriétés d'un problème à partir des propriétés d'un autre problème (utilisé aussi lors des calculs de complexité).

Deux autres résultats importants seront également présentés :

- **le théorème de récursion :**
  Il affirme la possibilité pour une MT de se dupliquer, ce qui conduit à sa possibilité de disposer de son propre code.
- **le problème de la décidabilité des théories logiques**
  Est-il possible toujours possible de déterminer si un énoncé logique est vrai ou faux ?

1 Décidabilité

1.1 Classification

On voudrait maintenant classifier l'ensemble des programmes d'une machine de Turing qui peuvent servir à quelque chose.

On peut s'entendre sur les points suivants :

1) un programme qui ne s'arrête jamais quelque soit l'entrée ne sert à rien.
2) un programme qui s'arrête toujours quelque soit l'entrée fait quelque chose (peu importe quoi).

Si on veut considérer l'ensemble des programmes possibles sur une MT, il faut aussi considérer la catégorie suivante :

3) un programme qui s'arrête pour certaines entrées (celles pour lesquelles le programme a un sens ; trouve une solution par exemple), mais qui bouclent pour d'autres (n'a pas de sens ; ne trouve pas de solution).

Certes, si on ne connaît pas les entrées pour lesquelles le programme boucle, la machine ne s'arrête jamais dans ce cas.

Ces programmes sont, pour l'instant, même s'ils sont problématiques, impossible à ignorer car on ne peut pas savoir s'il est *a priori* possible de détecter qu'un programme boucle.
Renversons la perspective : regardons maintenant ce problème sous l'angle des langages.

Cet angle est plus intéressant car un même langage peut être reconnu par des programmes différents.

On veut connaître la puissance effective des MTs, en s'intéressant à ce qu'elles **savent faire** et non à la **façon de le faire**.

On définit un langage L acceptés par une machine de Turing M comme étant l'ensemble des mots $w \in \Sigma^*$ tels que $M(w)$ accepte. On note $L = \mathcal{L}(M)$.

Conséquence de la définition d'un langage : si une MT M boucle sur un mot, alors elle n'accepte pas ce mot. Boucler sur un mot est équivalent à le rejeter.

En ne présumant rien sur les capacités des MTs, on veut savoir si, pour n'importe quel langage L, on est dans l'un des cas suivants :

1. il existe une MT qui reconnaît L et qui ne boucle pas.
2. toute MT qui reconnaît L boucle nécessairement sur au moins un mot.
3. il n'existe aucune MT qui reconnaît L.

Formalisons maintenant ces notions.

1.1.1 Langage récursivement énumérable

Définition II.1 (langage récursivement énumérable)

Un langage L est dit (récursivement) énumérable s'il existe une machine de Turing qui l'accepte.

Notes : Initialement, la définition d'un langage énumérable était "qui peut être énuméré par un énumèrateur". Voir la définition d'un énumèrateur plus loin.

Définition II.2 (Classe des langages récursivement énumérables)

La classe des langages récursivement énumérables est constituée de l'ensemble des langages reconnus par une machine de Turing.
Notation : On note $\mathcal{RE}$ cette classe.

Pour un langage L récursivement énumérable, une MT M qui reconnaît L ne peut faire que trois choses sur un mot d'entrée w,

- l'accepter ($w \in L$),
- le rejeter ($w \notin L$),
- ne pas s'arréter (boucle infinie, $w \notin L$).

Si $w \notin L$, il est possible que M ne donne jamais de réponse.

C'est la définition la plus faible qui puisse être donnée au fait qu'un langage est reconnu par une MT.

1.1.2 Langage décidable

En même temps, ceci est problématique, car on s'attendrait à ce qu'une machine qui reconnaît un langage donne une réponse définitive pour toute entrée.

On introduit alors la notion de décidabilité :

Définition II.3 (langage décidable)

Un langage L est décidable s'il est récursivement énumérable (*i.e.* il existe une machine de Turing M qui accepte L) **et** que M s'arrête pour toutes les entrées.
On dit alors que la MT M décide L.

Autrement dit, si L est décidable, alors il existe une MT M telle que :

- si $w \in L$, alors M accepte w.
- si $w \notin L$ alors M rejette w.

i.e. M s'arrête dans tous les cas.

Définition II.4 (Classe des langages décidables)

La classe des langages décidables est constituée de l'ensemble des langages décidables.
Notation : On note $\mathcal{R}$ cette classe.

Cette notion est approfondie maintenant.

1.2 Lien entre énumérabilité et décidabilité

Donc : un langage L est :

- **énumérable** $\stackrel{\Delta}{=}$ s'il existe une MT M qui accepte tout $w \in L$.
  i.e. si $w \notin L$, alors M rejette ou boucle.
- **décidable** $\stackrel{\Delta}{=}$ s'il existe une MT M telle que M qui accepte tout $w \in L$ et s'arrête toujours ; *i.e.* si $w \notin L$, alors M rejette et ne boucle jamais.

La décidabilité est une notion plus forte que l'énumérabilité.

Proposition II.1

Si un langage L est décidable, alors il est énumérable.

Démonstration:

Par définition, une MT est décidable si elle est énumérable et ne boucle jamais. □

Proposition II.2

Soit un langage L et $\overline{L}$ son complémentaire.
$$(L \text{ est décidable}) \Leftrightarrow \left(\overline{L} \text{ est décidable}\right)$$

DÉMONSTRATION:

$\Rightarrow$ soit M une MT qui décide L.
Pour tout $w \in \Sigma^*$, M s'arrête toujours soit dans l'état acceptant q_a, soit dans l'état rejetant q_r.
Soit $\overline{M}$ la MT construite à partir de M en inversant q_a et q_r. Alors, trivialement, $\overline{M}$ s'arrête toujours et décide $\overline{L}$.

Même argument avec un code : soit M la MT qui décide L. Soit la machine $\overline{M}$ suivante construite à partir de M :

$\overline{M}(w) =$

```
EXÉCUTER M(w)              // s'arrête toujours car M décidable
DÉCIDER l'opposé de M      // = accepter si rejette , rejeter si accepte
```

Clairement, $\overline{M}$ décidable et décide $\overline{L}$
$\Leftarrow$ soit $\overline{M}$ une MT qui décide $\overline{L}$.
Même idée en utilisant $\overline{L}$ pour définir M. □

REMARQUES 9: Notations

- $\mathcal{RE} \stackrel{\Delta}{=}$ classe des langages énumérables.
- $co\mathcal{RE} \stackrel{\Delta}{=}$ classe des langages dont le complément est énumérable.
- $\mathcal{R} \stackrel{\Delta}{=}$ classe des langages décidables.

Attention : à bien comprendre la co-énumérabilité. La définition n'implique que les assertions suivantes :

- $(L \in co\mathcal{RE}) \Leftrightarrow (\bar{L} \in \mathcal{RE})$ = un langage est co-énumérable si est seulement si son complémentaire est énumérable.
- $(L \in co\mathcal{RE})$ n'implique pas $(L \in \mathcal{RE})$ = un langage co-énumérable n'est pas nécessairement énumérable.
- $(L \in \mathcal{RE})$ n'implique pas $(L \in co\mathcal{RE})$ = un langage énumérable n'est pas nécessairement co-énumérable.

Avec ces notations, les constatations deviennent :

- si un langage L est décidable, alors il est énumérable : $\mathcal{R} \subseteq \mathcal{RE}$.
- pour un langage L, si $\bar{L}$ est énumérable (*i.e.* $\bar{L} \in \mathcal{RE}$), alors $L \in co\mathcal{RE}$ (par définition).
- si L est décidable, alors $\bar{L}$ aussi ($\Rightarrow \bar{L} \in \mathcal{R}$). D'où $\mathcal{R} \subseteq co\mathcal{RE}$.
- donc $\mathcal{R} \subseteq \mathcal{RE} \cap co\mathcal{RE}$

Théorème II.1 (lien entre décidabilité et énumérabilité)

$\mathcal{R} = \mathcal{RE} \cap co\mathcal{RE}$

DÉMONSTRATION:

- $\mathcal{R} \subseteq \mathcal{RE} \cap co\mathcal{RE}$: déjà démontré.
- $\mathcal{R} \supseteq \mathcal{RE} \cap co\mathcal{RE}$

  $\Leftrightarrow$ si $L \in \mathcal{RE} \cap co\mathcal{RE}$, alors $L \in \mathcal{R}$
  $\Leftrightarrow$ si L et son complément sont énumérables, alors L est décidable.

  Soit M une MT qui accepte L.
  Soit $\overline{M}$ une MT qui accepte $\bar{L}$.
  Construisons une MT D qui exécute M et $\overline{M}$ en parallèle :

- si M accepte, alors M accepte.
- si $\overline{M}$ accepte, alors M rejette.

Comment exécuter M et $\overline{M}$ en parallèle : prendre 2 bandes (une pour chaque) et alterner entre les 2 machines (et éviter les boucles d'une des 2 machines).

Montrons que D décide L.
Toute chaîne w est soit dans L, soit dans $\bar{L} \Rightarrow$ soit M, soit $\overline{M}$ accepte w.
D s'arrête chaque fois que M ou $\overline{M}$ accepte $\Rightarrow D$ s'arrête pour tout w.
De plus par construction, D accepte les chaînes de L et rejette les chaînes de $\bar{L}$. Donc D décide L, et L est décidable. □

Démonstration: (idem avec un code)

Considérons la MT D suivante :

$D(w) =$

```
RÉPÉTER à l'infini
  EXÉCUTER M sur une transition
  SI M ACCEPTE ALORS ACCEPTER   // M accepte si w ∈ L
  EXÉCUTER M̄ sur une transition
  SI M̄ ACCEPTE ALORS REJETER    // M̄ accepte si w ∉ L
```

D s'arrête toujours car $w \in L \cup \overline{L} = \Sigma^*$, et n'accepte que si $w \in L$. Donc D décide L et L décidable. □

Théorème II.2 ((le même reformulé))

Un langage est décidable si et seulement si il est à la fois énumérable et co-énumérable.

Exercice 2 (Fermeture de $\mathcal{R}$)**.** *Montrer que $\mathcal{R}$ est :*

1. *fermé par complémentarité, à savoir $L \in \mathcal{R} \Leftrightarrow \overline{L} \in \mathcal{R}$*
2. *fermé par union, à savoir $(L_1, L_2) \in \mathcal{R}^2 \Leftrightarrow L_1 \cup L_2 \in \mathcal{R}$*
3. *fermé par intersection (modifier la démonstration précédente)*
4. *fermé par concaténation, $(L_1, L_2) \in \mathcal{R}^2 \Leftrightarrow L_1 \circ L_2 \in \mathcal{R}$,*
   où $\circ$ est l'opérateur de concaténation défini par $L_1 \circ L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ et } w_2 \in L_2\}$; à savoir l'ensemble des mots obtenus par concaténation d'un mot de L_1 et d'un mot de L_2.
5. *fermé par l'opérateur de Kleene $*$: $L \in \mathcal{R} \Leftrightarrow L^* \in \mathcal{R}^*$.*
   $w \in L^* \Leftrightarrow w = \varepsilon$ ou $\exists k \geq 1$ tel que $w = w_1 w_2 \dots w_k$ et $\forall k, w_k \in L$.

Exercice 3 (Fermeture de $\mathcal{RE}$)**.** *Montrer que $\mathcal{RE}$ est :*

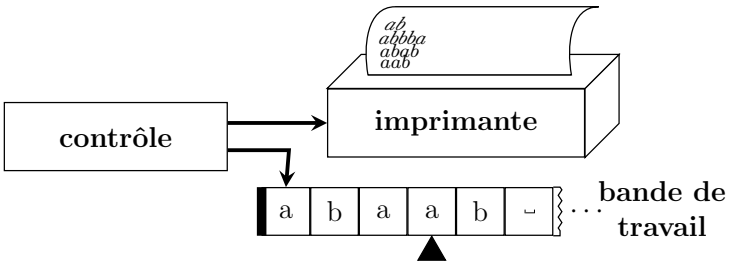
1. *fermé par union.*
2. *fermé par concaténation.*
3. *fermé par l'opérateur de Kleene.*

Attention : *si $L \in \mathcal{RE}$, alors toute machine de Turing M_L telle que $\mathcal{M} = L$ peut boucler sur n'importe quelle entrée.*

1.3 Énumèrateur

Énumèrateur : type particulier de MT avec une imprimante attachée servant à écrire l'ensemble des chaînes contenues dans le langage reconnu par cette MT.

- énumération dans n'importe quel ordre, éventuellement avec répétition.
- ne possède pas d'entrée (mais possède une bande pour travailler).



Le code d'un énumérateur E pour un langage L est schématiquement le suivant :

$E_L() =$
RÉPÉTER
  CALCULER le mot w suivant du langage L
  SI tous les mots ont déjà été tous calculés
  ALORS ARRÊTER
  SINON AFFICHER w

EXEMPLE 16: énumèrateur Le code de l'énumèrateur E_{MULTIPLE} qui écrit tout les couples de la forme (n, p) où $n \in \mathbb{N}$, $p \in \mathbb{N}$ et n est un multiple de p est le suivant :

$E_{\text{MULTIPLE}}() =$
POUR $n = 1, 2, \ldots$
  POUR $1 \leq p \leq n$
    SI $n\%p = 0$ **ALORS AFFICHER** (n, p)

REMARQUE 10:
Évidemment, si un énumèrateur E reconnaît un langage L, et que $\#L$ n'est pas fini, l'énumérateur ne s'arrête jamais.

Théorème II.3 (lien énumèrateur-MT)
Un langage est accepté par une MT si et seulement si il existe un énumèrateur qui l'énumère.

DÉMONSTRATION:
$\Leftarrow$: $\exists$ énumèrateur E qui énumère $L \Rightarrow \exists$ MT M qui reconnaît L

on construit la MT $M(w)$ comme : exécuter l'énumérateur pour obtenir l'ensemble des mots v, et accepter si $v = w$.

$M(w) =$
RÉPÉTER
  CALCULER le mot v suivant du langage L
  SI tous les mots ont été calculé
  ALORS REJETER
  SINON SI $w = v$ **ALORS ACCEPTER**

Clairement, M accepte les sorties de E, donc M reconnaît L.

$\Rightarrow$: $\exists$ MT M qui reconnaît $L \Rightarrow \exists$ énumèrateur E qui énumère L

Soit $s_1, s_2, s_3, \ldots$ l'ensemble des chaînes possibles de Σ^*.
On construit E de la manière suivante :

$E_L() =$
```
POUR i=1,2,3,...
  POUR s_j = s_1, s_2, ..., s_i
    EXÉCUTER M(s_j) sur i transitions
    SI M(s_j) ACCEPTE
    ALORS AFFICHER s_j
```

La limitation de l'exécution à i transitions permet d'éviter d'être bloqué au cas où un s_j ferait boucler M.

Inconvénient : la même sortie peut apparaître de très nombreuses fois.

□

1.4 Existence de langages non énumérables

Commençons par montrer l'existence de langages non énumérables, à savoir de langages reconnus par aucune machine de Turing.

Pour ce faire, nous avons besoin de quelques outils théoriques supplémentaires.

1.4.1 Compléments théorique

Comment comparer la taille de deux ensembles de tailles finies ?

Facile ! Il suffit de les compter.
Exemple : soit $n_A = \#A$ et $n_B = \#B$
si $n_A \neq n_B$, alors ils n'ont pas la même taille.

Définition II.5 (bijection (rappel))

une bijection de A dans B est une fonction f tq :

- si $(a_1, a_2) \in A^2$ et $a_1 \neq a_2$ alors $f(a_1) \neq f(a_2)$
- si $\forall b \in B$ alors $\exists! a \in A$ tel que $f(a) = b$

Autrement dit, tout élément de A est associé à un élément de B unique et vice-versa.

Autre méthode : s'il existe une bijection entre A et B
alors ils ont la même taille.

Comment comparer la taille de deux ensembles de tailles infinies ?

Il n'est plus possible de les compter.
Par contre, il est possible de définir une bijection entre 2 ensembles infinis.
On prend donc la **définition** suivante pour comparer la taille d'ensembles infinis.

Définition II.6 (Comparaison de la taille de deux ensembles)

Deux ensembles A et B ont la même taille s'il existe une bijection entre A et B.

Quelle conséquence pour des ensembles infinis ?

Première affirmation :
L'ensemble des entiers naturels $\mathbb{N}$ a la même taille que l'ensemble $\mathbb{E}$ des nombres pairs.
Démonstration : prendre la bijection $f(i) = 2i$.
$\mathbb{E}$ est un sous-ensemble de $\mathbb{N}$, mais ils ont la même taille !

Définition II.7 (Ensemble dénombrable)

Un ensemble A est dénombrable si :

- soit A est un ensemble fini.
- soit A a la même taille que $\mathbb{N}$ (ensemble des entiers naturels).

Notation : $\aleph_0 = \#\mathbb{N}$

Deuxième affirmation :

Proposition II.3

L'ensemble des entiers relatifs $\mathbb{Z}$ est dénombrable.

Démonstration:

prendre la bijection :

$$f(i) = \begin{cases} i/2 & \text{si } i \text{ est pair} \\ -(i+1)/2 & \text{si } i \text{ est impair} \end{cases}$$

□

Cette fois ci, $\mathbb{N}$ est un sous-ensemble de $\mathbb{Z}$.
Pourtant, ils ont encore la même taille.

Tous les ensembles sont-ils dénombrables ?

Ensemble des rationnels positifs $\mathbb{Q}^+ = \left\{ m/n \mid (m,n) \in \mathbb{N}^{*2} \right\}$:
Troisième affirmation :

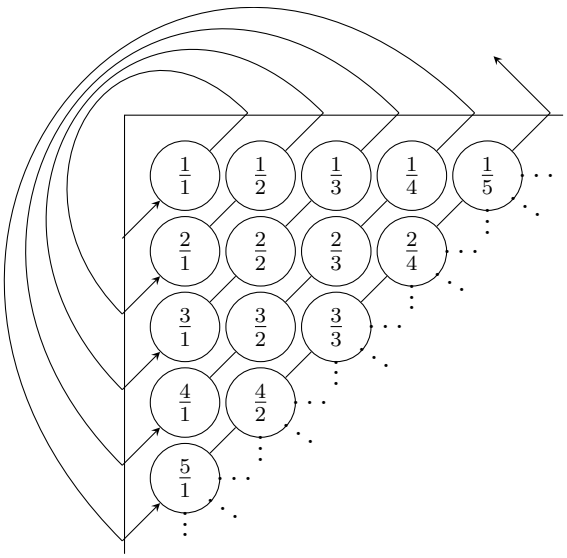
Proposition II.4

L'ensemble des entiers rationnels positifs $\mathbb{Q}^+$ est dénombrable.

Plus difficile à démontrer. **Démonstration:**

Première approche : On considère $\mathbb{Q}$ comme un tableau à 2 dimensions.
Commençons par compter la première ligne ...
Zut, elle est infinie !

Seconde approche : On compte le long des diagonales et en sautant les doubles comme ceci :



C'est une union dénombrable (= autant de diagonales que $\mathbb{N}$) d'ensembles (= éléments sur la diagonales) qui sont chacun fini (= la taille de chaque diagonale est finie).

Or, toute une union dénombrable d'ensembles finis est un ensemble dénombrable. Démontrons-le dans ce cas particulier.

La $n^{\text{ème}}$ diagonale contient n éléments. Par ailleurs, il y a $N_n = n(n-1)/2 = 1 + 2 + \cdots + n - 1$ éléments sur toutes les diagonales précédentes. En conséquence, on peut construire une bijection qui affecte les éléments $\{1, \ldots, n\}$ de la $n^{\text{ème}}$ diagonales aux élements $\{N_n + 1, \ldots N_n + n\}$ de $\mathbb{N}$.

Cette propriété est vraie dans le cas général.

Donc $\mathbb{Q}^+$ est dénombrable. □

Ensemble des nombres réels $\mathbb{R}$:

Théorème II.4

$\mathbb{R}$ n'est pas dénombrable.

Notation : $\aleph_1 = \#\mathbb{R}$

Remarque 11: Rappel

sur les nombres réels Tout nombre réel a une représentation décimale.
Exemple : $\pi = 3.1415926\ldots$, $\sqrt{2} = 1.4142136\ldots$
$0 = 0.\underline{0}$ où $\underline{0} = 0$ suivi d'une infinité de 0.

Équivalences : $1.6\underline{9} = 1.7\underline{0}$

Démonstration: par l'absurde

Hypothèse : supposons qu'il existe une bijection entre $\mathbb{N}$ et $\mathbb{R}$.
Il est donc possible de construire une table :

n	$f(n)$
1	$3.14159\ldots$
2	$55.55555\ldots$
3	$40.18642\ldots$
4	$15.20601\ldots$

Montrons que l'on peut trouver un réel x qui n'est pas dans cette liste.

Construisons un x, tel que $0 \leq x \leq 1$ qui soit différent de $f(n)$, $\forall n$.

La $n^{\text{ème}}$ décimale de x est construite à partir de la $n^{\text{ème}}$ décimale de $f(n)$.

n	$f(n)$
1	$3.\mathbf{1}4159\ldots$
2	$55.5\mathbf{5}555\ldots$
3	$40.18\mathbf{6}42\ldots$
4	$15.206\mathbf{0}1\ldots$

(méthode de diagonalisation)

La $n^{\text{ème}}$ décimale de x est choisie comme :

- différente de la $n^{\text{ème}}$ décimale de $f(n)$
- différente de 0 ou 9 (pour éviter les équivalences).

Ainsi construit, x a toujours au moins une décimale différente de $f(n)$ pour tout n.

Donc, il n'existe pas de bijection entre $\mathbb{N}$ et $\mathbb{R}$.
Donc, $\mathbb{R}$ n'est pas dénombrable. □

1.4.2 Preuve d'existence de langages non énumérables

Montrons maintenant qu'il existe des langages non énumérables.

On rappelle que si un langages n'est pas énumérable, alors il n'existe aucune machine de Turing qui accepte le langage associé à ce problème.

Lemme II.1
si Σ un ensemble fini, alors Σ^* est dénombrable.

Démonstration:
Notons Σ_n les chaînes de longueurs n de Σ^*. Il est clair que $\Sigma^* = \bigcup_{i\in\mathbb{N}} \Sigma_i$.
Or, tous ces ensembles sont finis : $\Sigma_n = (\#\Sigma)^n$.
Donc Σ^* est une union dénombrable d'ensembles finis.
Or, une union dénombrable d'ensembles finis est aussi dénombrable. □

Lemme II.2
L'ensemble des MTs est dénombrable.

Démonstration:
Chaque MT M est codée comme une chaîne $\langle M \rangle$.
Or, cette chaîne s'écrit à partir de symboles d'un alphabet Σ.
Donc, il existe une injection de l'ensemble des MTs dans Σ^* (i.e. toutes les MTs sont codées dans Σ^*, mais toutes les chaînes de Σ^* ne représentent pas des MTs).
Or, Σ^* est dénombrable. Donc l'ensemble des MTs aussi. □

Soit $\mathbb{B}$ l'ensemble des suites binaires de longueur infinie.

Lemme II.3
$\mathbb{B}$ n'est pas dénombrable.

Démonstration:
Réutiliser l'argument de diagonalisation pour construire une suite binaire de longueur infinie qui n'est dans aucun ensemble dénombrable d'éléments de $\mathbb{B}$. **Exercice :** faire la démonstration complète. □

Lemme II.4
L'ensemble des langages $\mathcal{L}$ sur un alphabet Σ fini n'est pas dénombrable.

Démonstration:
Soit $\Sigma^* = \{s_1, s_2, s_3, \dots\}$ l'ensemble (dénombrable) des mots reconnus par L. Pour $b \in \mathbb{B}$, on note b_i le $i^{\text{ème}}$ bit de b.
Soit $\chi : \mathcal{L} \to \mathbb{B}$ qui associe à tout langage $L \in \mathcal{L}$ une suite caractéristique unique $b \in \mathbb{B}$, définie par $b_i = 1$ si et seulement si $s_i \in L$ (et 0 sinon).
Or, l'application χ est une bijection (par construction).
Comme $\mathbb{B}$ n'est pas dénombrable, $\mathcal{L}$ n'est pas non plus dénombrable. □

Pour résumer :

- L'ensemble des MTs est dénombrable.
- l'ensemble $\mathcal{L}$ de tous les langages sur un alphabet fini Σ n'est pas dénombrable.

Corollaire II.1
Il existe des langages qui ne sont acceptés par aucune MT.

DÉMONSTRATION:

Une MT ne reconnaît qu'un seul langage.
Il y a infiniment plus de langages que de MT.
Donc, certains langages ne peuvent pas être reconnus par une MT. □

On a donc la preuve de l'existence de langages non énumérables (sans en exhiber un seul).

En terme informatique, cela signifie qu'**aucun programme** n'est capable de reconnaître de tels langages.

Pire : l'essentiel des langages ne sont pas énumérables (il y a infiniment plus de langages que de programmes qu'il est possible d'écrire).

Conséquence : le modèle de la machine de Turing est très limité sur le nombre de langages qu'il est capable de reconnaître.
Néanmoins, tout problème de décision ayant un sens pour nous se formule sous forme d'une chaîne de longueur finie avec un nombre fini de symboles (par exemple : langage constitué de l'ensemble des nombres premiers). Le nombre de ces problèmes est donc fini.

Est-il possible de formuler un problème de décision non énumérable ?

Nous n'avons pas de réponse à cette question pour le moment. Restreignons un peu nos ambitions, et regardons s'il est possible de trouver des problèmes énumérables mais pas décidables.

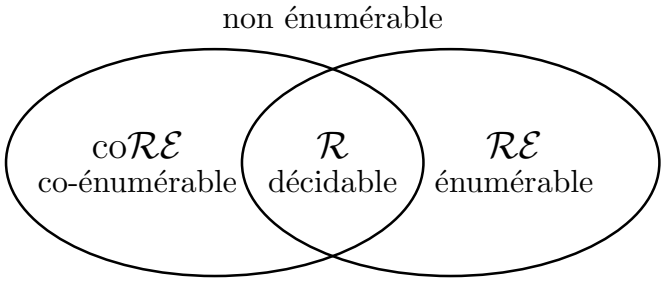
Exercice 4. *Prouver qu'il existe un sous-ensemble non récursivement énumérable de* $\{1\}^*$.

1.4.3 Première synthèse sur les classes de langages

Structuration des langages

- Tous les ensembles $\mathcal{RE}$, *co*$\mathcal{RE}$ et $\mathcal{R}$ sont dénombrables (car ils ne contiennent que des MTs).
- Il manque tous les langages non énumérables.

Les langages s'organisent donc comme suit :



1.5 Problèmes de décidabilité

On s'intéresse maintenant aux problèmes pour lesquels ils est possible d'écrire une machine de Turing (= au moins énumérables).

La décidabilité d'une problème (et donc de son algorithme) est une question fondamentale, en effet :

- si un problème est décidable, alors il existe **nécessairement** un programme qui décide le problème (= accepte ou rejette) et s'arrête pour n'importe quelle entrée.
  = pour tout mot d'entrée, si j'attends assez longtemps, je suis sûr que le programme me donnera la réponse.
- si le problème est indécidable, alors pour n'importe quel programme M qui accepte le langage L associé, il existe au moins un mot qui n'appartient pas au langage pour lequel M boucle.
  = pour tout mot d'entrée, si je n'ai pas de réponse, il n'y a aucun moyen de savoir si le programme est en train de calculer ou de boucler.

Clairement, on ne souhaite exécuter sur un ordinateur que des programmes correspondant à des problèmes décidables.

1.5.1 Premier exemple de problème indécidable

Au début du XX$^{\text{ème}}$ siècle, on pensait que tout problème mathématique correctement posé devait pouvoir être résolu, et que le seul obstacle à cette résolution était la complexité du problème.

Le 10$^{\text{ème}}$ problème de Hilbert :

En 1900, David Hilbert présente comme défi pour le 20$^{\text{ème}}$ siècle de résoudre 23 problèmes mathématiques centraux.

Le 10$^{\text{ème}}$ problème concerne directement l'algorithmique : "trouver un algorithme qui détermine si un polynôme multivarié à coefficients entiers possède une racine entière".

Clairement, la formulation de Hilbert induit que :

1. un tel algorithme existe.
2. il suffit de le trouver.

Exemples :

Soit $P_1(x, y, z) = 6x^3yzz^2 + 3xy^2 - x^3 - 10$
Existe-t-il $(x, y, z) \in \mathbb{Z}^3$ tel que $P_1(x, y, z) = 0$
Oui, prendre $(x, y, z) = (5, 3, 0)$.

Soit $P_2(x, y) = 3x^2 + 5y^2 + 7$.
Existe-t-il $(x, y) \in \mathbb{Z}^2$ tel que $P_2(x, y) = 0$
Non, toujours positif.

Soit $P_3(x, y) = x^2 - 991y^2 - 1$ (équation de Pell pour $d = 991$).
La plus petite solution entière est :

$$\begin{cases} x = 379516400906811930638014896080 \\ y = 12055735790331359447442538767 \end{cases}$$

Cas des polynômes à une variable (=univarié) :

Soit le langage :
$D = \{\langle p(x)\rangle \mid p(x)$ est un polynôme univarié avec une racine entière$\}$

On peut trouver une MT M qui reconnaît D :

$M(\langle p(x)\rangle) =$

```
POUR x = 0, 1, −1, 2, −2, ...
  ÉVALUER p(x)
  SI p(x) = 0 ALORS ACCEPTER
```

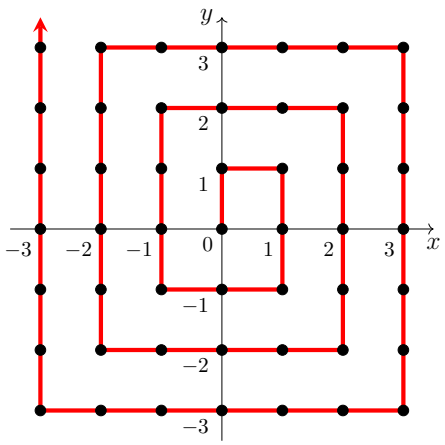
Si $p(x)$ a une racine entière, alors M finira par la trouver et l'accepter, sinon M ne s'arrêtera jamais.

Donc, ce langage est bien énumérable (= reconnu par une MT).

Cas des polynômes multivariés :

Une construction similaire est possible.

Par exemple en 2D pour les polynômes bivariés, on peut parcourir tous les couples entiers (x, y) ainsi :



Ceci est généralisable en dimension n.

Donc ce langage est aussi énumérable pour les polynômes multivariés.

Ces langages sont énumérables, mais sont-il décidables ?
Cas des polynômes à une variable

Théorème II.5 (Racines d'un polynôme univarié)

toutes les racines réelles de $p(x)$ vérifient $|x| < |kc_{\max}/c_1|$ où k = nombre de termes du polynôme, $c_{\max}$ = coefficient maximum et c_1 = coefficient d'ordre le plus élevé.

Modifions la MT M de la manière suivante :

$D(\langle p(x)\rangle) =$

$A = [-|kc_{\max}/c_1|, +|kc_{\max}/c_1|]$
POUR tous les x entiers dans A
  ÉVALUER $p(x)$
  SI $p(x) = 0$ **ALORS ACCEPTER**
REJETER

D s'arrête toujours en acceptant ou rejetant car A contient un nombre fini d'entiers. Donc, D **est décidable**.

Cas des polynômes multivariés :

C'est le $10^{\text{ème}}$ **problème de Hilbert**.

Ce problème n'est **pas décidable** (prouvé en 1970, par Yuri Matijasevic).

1.5.2 Problème de l'acceptation

Cas général

Le problème de l'acceptation consiste à construire un langage A constitué de couples $\langle M, w\rangle$ constitués :

- de l'ensemble des descriptions des machines M (qui reconnaisse chacune un certain langage),
- de l'ensemble des mots $w \in \Sigma^*$

tels que l'exécution de la machine M sur le mot w accepte.

A savoir :

$$A = \{\langle M, w\rangle |\ M \text{ est une machine qui accepte } w\}$$

Bien lire l'expression ci-dessus : A ne dépend pas d'un M ou un w particulier,

Le problème de l'acceptation est de savoir si A est décidable.

si A est décidable, cela signifie qu'il existe un programme D tel que, pour tout $\langle M, w\rangle \in A$, D peut décider si M accepte w.

Si A est décidable, le décideur D de A ne boucle jamais.

Proposition II.5

Si un décideur D de A existe, alors il existe un décideur pour toute machine T.

Démonstration:

Pour tout mot $w \in \Sigma^*$,

- si D accepte $\langle T, w\rangle$, alors $w \in \mathcal{L}(T)$.
- si D rejette $\langle T, w\rangle$, alors $w \notin \mathcal{L}(T)$.

Autrement dit, pour chaque machine T fixée, il est possible d'écrire le décideur D_T suivant :

$D_T(w) =$
construire $\langle T, w\rangle$ à partir de $\langle T\rangle$ **et** w
exécuter $D(\langle T, w\rangle)$
décider comme D // = accepter si accepte, rejeter si rejette

Chaque ligne du décideur D_T ne boucle jamais, et par conséquent, D_T ne boucle jamais. □

Cas des automates finis

Étudions maintenant le problème de l'acceptation dans le cas des automates finis.

Soient :

$$\begin{aligned} A_{\text{ADF}} &= \{< M, w > |\ M \text{ est un Automate Déterministe Fini qui accepte } w\} \\ A_{\text{ANF}} &= \{< M, w > |\ M \text{ est un Automate Non Déterministe Fini qui accepte } w\} \end{aligned}$$

Proposition II.6 (décidabilité des automates finis)

1. A_{ADF} est décidable.
2. A_{ANF} est décidable.

Intuitivement, cette proposition parait assez évidente.

La démonstration consiste à écrire une machine de Turing qui détermine ce que décide l'automate fini sur son entrée. Celle-ci doit donc simuler le fonctionnement de l'automate à partir de sa description $\langle M \rangle$ sur l'entrée w, et s'arrêter dans tous les cas.

Commençons partiellement la démonstration avec un exercice, que nous complétons plus tard.

Démonstration:

1. Le décideur doit d'abord vérifier que $\langle M \rangle$ est l'encodage d'un ADF, et que $w \in \Sigma^*$ où Σ est l'alphabet donné dans l'encodage de la description formelle de M.

   Il s'agit d'une suite de vérifications syntaxiques simples sur des listes de symboles.

   Par exemple, avec $\langle M \rangle = \{\langle \Sigma \rangle, \langle Q \rangle, q_0, \langle F \rangle, \langle \delta \rangle\}$, on doit vérifier que : $\langle \Sigma \rangle$ est liste de symboles distincts, $\langle Q \rangle$ est liste d'états distincts, $q_0 \in \langle Q \rangle$, $\langle F \rangle$ est liste d'états q_i distincts tels que $\forall i, q_i \in Q$, $\langle \delta \rangle$ est une fonction de transition (à savoir une liste de triplet (u, s, v) tel qu'il existe un seul triplet pour l'ensemble $(u, s) \in Q \times \Sigma$, et que $v \in Q$.

   Puis, il faut simuler l'ADF (voir exercice) avec le décideur D de l'exercice pour obtenir la décision. D'où A_{ADF} est décidable.

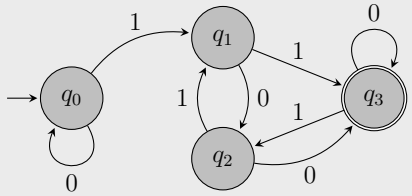
2. pour les ANFs, on peut démontrer (non fait) que tout ANF peut être converti en un ADF équivalent avec un algorithme T qui s'arrête toujours. Puis, on réutilise le décideur D de l'ADF.

   Le décideur D' d'un ANF M' est donc : $D'(\langle M', w \rangle) = D(\langle T(M'), w \rangle)$, et A_{ANF} est décidable.

□

Exercice 5 (Décidabilité des automates déterministes finis (ADF))**.**

1. *Pourquoi un ADF s'arrête-t-il toujours pour tout mot d'entrée w ?*
2. *Donner la définition formelle de l'ADF M suivant :*



3. *En déduire un codage $\langle M \rangle$ de M.*
4. *Avec des transformations simples, expliquer comment obtenir une machine de Turing qui reconnaît exactement le même langage que M.*
5. *Écrire une machine de Turing D qui prend $\langle M, w \rangle$ en entrée, et qui simule le fonctionnement de l'ADF M sur son entrée w. Indice : utiliser une machine multibande pour simplifier l'accès aux différentes caractéristiques de l'automate.*
6. *Montrer que D s'arrête toujours.*
7. *Peut-on déduire des questions précédentes que A_{ADF} est décidable ?*

Cas des grammaires algébriques

Passons maintenant aux grammaires algébriques :

Proposition II.7 (décidabilité des automates finis et des automates à pile)

1. $A_{\text{GA}} = \{< M, w > \mid M \text{ est une Grammaire Algébrique qui accepte } w\}$ est décidable.
2. $A_{\text{AP}} = \{< M, w > \mid M \text{ est une Automate à Pile qui accepte } w\}$ est décidable.

Démonstration: (grandes lignes)

1. savoir si un mot w est accepté par une GA peut être difficile en raison des règles du type $U \to \varepsilon|UU|0$ car il y a une infinité de façon d'obtenir une chaîne de longueur n.

   Mais, toute GA peut être mise sous forme normale de Chomsky = sous la forme $(S \to UU|a, U \to UV|a)$ où le symbole de départ S est le seul donnant ε, et toute variable est transformée en deux variables (hors S) ou en un terminal.

   En conséquence, pour savoir si un mot w de longueur n est généré par la GA, il suffit d'appliquer $n-1$ fois des règles de type $U \to UV$ et n fois des règles du type $U \to a$. Le nombre de combinaisons est alors fini (car le nombre de règles de chaque type est fini), et il suffit de les tester une à une.

   Si aucun mot généré ainsi n'est w, alors il ne peut pas être généré par la GA. Sinon, on accepte w. Cet algorithme s'arrête toujours, donc toute grammaire algébrique peut être décidée pour toute entrée. Donc A_{GA} est décidable.

2. tout AP peut être transformée en une GA équivalente avec un algorithme T qui s'arrête toujours.

□

Exercice 6 (Automate Linéaire Borné)**.** *Un automate linéaire borné (ABL) est une MT avec une bande de taille fixe (bornée à droite). Un déplacement à droite en fin de bande ne déplace pas le curseur de lecture.*

Soit M, un ABL à q états, g symboles de bande, et une bande de taille n.

1. *Montrer que le nombre de configurations différentes dans lequel un ABL peut se trouver est fini. On le calculera.*
2. *Soit $A_{ABL} = \{\langle M, w\rangle \mid M \text{ est un ABL qui accepte } w\}$. Montrer que A_{ABL} est récursivement énumérable.*
3. *Comment détecter si un ABL est en train de boucler ?*
4. *Montrer que le problème d'acceptation pour un ABL est décidable.*
5. *Est-il possible d'utiliser le résultat précédent pour détecter qu'un ordinateur limité (par exemple, avec 64 états, et 1Mo de mémoire) est en train de boucler ?*

Cas des machines de Turing

Et enfin, le cas de la machine de Turing :

$$A_{\text{MT}} = \{< M, w > \mid M \text{ est une Machine de Turing qui accepte } w\}$$

Théorème II.6 (décidabilité des machines de Turing)

1. A_{MT} est récursivement énumérable.
2. A_{MT} est indécidable.

Démonstration:

1. Une MT universelle U est une MT qui, $\forall M$, $\forall w$:
   - accepte $\langle M, w\rangle$ si M accepte w.
   - rejette ou boucle $\langle M, w\rangle$ si M rejette ou boucle sur w.

   Le langage reconnu par la MT universelle U est A_{MT}.

   A_{MT} est donc énumérable.

2. Montrons maintenant que A_{MT} n'est pas décidable.

Supposons qu'il existe un MT H qui décide A_{MT}.

Par définition, H doit donc avoir le comportement suivant pour toute machine M et chaîne w :

$$H(\langle M, w\rangle) = \begin{cases} \text{accepte} & \text{si } M \text{ accepte } w \\ \text{rejette} & \text{si } M \text{ n'accepte pas } w \end{cases}$$

Rappel sémantique : rejeter (= arrêt) ≠ ne pas accepter (= arrêt OU boucle).
Construisons maintenant une autre MT D qui utilise H de la façon suivante :

$D(\langle M\rangle) =$ **EXÉCUTER** $H(\langle M, \langle M\rangle\rangle)$
RETOURNER l'opposé de la décision de H.

Ainsi,

- D accepte $\langle M\rangle$ si H rejette $\langle M, \langle M\rangle\rangle$.
- D rejette $\langle M\rangle$ si H accepte $\langle M, \langle M\rangle\rangle$.

D s'arrête toujours (n'utilise que des opérations qui s'arrêtent toujours).

Note : exécuter une machine M sur sa propre description n'est pas aberrant. On peut écrire un compilateur C en C.

En reprenant la définition de H, D peut s'écrire :

$$D(\langle M\rangle) = \begin{cases} \text{accepte} & \text{si } M \text{ n'accepte pas } \langle M\rangle \\ \text{rejette} & \text{si } M \text{ accepte } \langle M\rangle \end{cases}$$

Que se passe-t-il maintenant si l'on utilise D sur lui-même ?

$$D(\langle D\rangle) = \begin{cases} \text{accepte} & \text{si } D \text{ n'accepte pas } \langle D\rangle \\ \text{rejette} & \text{si } D \text{ accepte } \langle D\rangle \end{cases}$$

Contradiction : D rejette $\langle D\rangle$ quand D accepte $\langle D\rangle$.

Donc, ni D, ni H n'existent.
Comme H n'existe pas, A_{MT} n'est pas décidable. □

On remarquera qu'il s'agit là d'une impossibilité logique, et non technique.
Une deuxième preuve de l'indécidabilité par la méthode dite de "diagonalisation".

DÉMONSTRATION:

(par diagonalisation)

Table $M(\langle M\rangle)$
si M_i accepte $\langle M_j\rangle$
alors (i, j) est "accept"

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	…
M_1	accept		accept		
M_2	accept	accept	accept	accept	
M_3					…
M_4	accept	accept			
⋮			⋮		

Table $H(\langle M, \langle M\rangle\rangle)$
si H accepte $\langle M_i, \langle M_j\rangle\rangle$
alors (i, j) est "accept"
sinon (i, j) est "reject"

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	…
M_1	accept	reject	accept	reject	
M_2	accept	accept	accept	accept	
M_3	reject	reject	reject	reject	…
M_4	accept	accept	reject	reject	
⋮			⋮		

Diagonalisation :
Ajout de $D(\langle M \rangle)$:

- opposé de la diagonale.
- problème pour $D(\langle D \rangle)$
- contradiction

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle D \rangle$	...
M_1	accept	reject	accept	reject		accept	
M_2	accept	accept	accept	accept		accept	
M_3	reject	reject	reject	reject	...	reject	...
M_4	accept	accept	reject	reject		accept	
$\vdots$			$\vdots$		$\ddots$		
D	reject	reject	accept	accept	...	???	
$\vdots$			$\vdots$				$\ddots$

□

Conséquences de l'indécidabilité du problème de l'acceptation

L'indécidabilité de A_{MT} a la conséquence suivante pour l'informatique :

> Il n'existe pas de MT (*i.e.* programme) permettant de décider si n'importe quelle MT (*i.e.* autre programme) accepte n'importe quelle entrée.

Autrement dit, je ne peux pas écrire de programme :

- auquel je passe un programme de décision M et une entrée w,
- capable de dire comment M décide w,
- et qui fonctionne dans tous les cas.

Ce programme contiendra nécessairement au moins une contradiction logique qui l'empêchera de fonctionner correctement.

Constat :

- Les notions intuitives sont suffisantes pour construire des algorithmes simples.
- des notions formelles sont nécessaires pour démontrer que certains problèmes ne sont pas solvables par un ordinateur.

Attention :

- il est évidemment possible d'écrire un algorithme indécidable pour un problème décidable (exemple : prendre un algorithme décidable et le faire boucler dans certains cas ou l'algorithme rejette).
- si un problème est indécidable, cela ne signifie pas qu'il ne sera pas décidable sur un sous-ensemble du problème (= il est peut-être décidable **dans certains cas**, à condition d'avoir un algorithme **décidable** permettant de déterminer si l'on est sur ce sous-ensemble).
- ne pas confondre indécidable avec non récursivement énumérable (= aucun algorithme n'existe, tout court, pour résoudre le problème).

Premier exemple de langage non-énumérable

Donnons maintenant un exemple de langage non-énumérable.

Rappels :

- un langage co-énumérable si son complément est énumérable.
- **théorème :** un langage est décidable si et seulement si il est énumérable et co-énumérable.

Corollaire II.2

si un langage n'est pas décidable, alors soit le langage lui-même soit son complément n'est pas énumérable.

Démonstration:
proposition logique inverse du théorème dans les rappels ci-dessus. □

Théorème II.7
$\overline{A_{\text{MT}}}$ n'est pas énumérable.

$\overline{A_{\text{MT}}}$ est l'ensemble des couples $\langle M, w\rangle$ tels que $M(w)$ rejette ou boucle (par défaut, une expression $\langle M, w\rangle$ mal formée rejette).

Démonstration:
On a vu que A_{MT} est énumérable, mais qu'il n'est pas décidable.
Donc, d'après le corollaire précédent $\overline{A_{\text{MT}}}$ n'est nécessairement pas énumérable. □

1.5.3 Autres problèmes de décidabilité

Nous allons maintenant étudier d'autres problèmes de décidabilité.

Est-il possible de construire un programme permettant de décider :

- **le problème de l'arrêt :** si un autre programme s'arrête sur une entrée ?

  $$H_{\text{MT}} = \{\langle M, w\rangle |\ M \text{ est une MT qui s'arrête sur } w\}$$

  contient toutes les paires (MT,mot) telle que la MT s'arrête sur le mot.
- **le problème du vide :** si un autre programme rejette toutes ses entrées ?

  $$E_{\text{MT}} = \{\langle M\rangle |\ M \text{ est une MT et } \mathcal{L}(M) = \emptyset\}$$

  contient toutes les MTs qui n'acceptent aucun mot.
- **le problème de l'égalité :** si deux autres programmes prennent les mêmes décisions ?

  $$EQ_{\text{MT}} = \{\langle M_1, M_2\rangle |\ M_1 \text{ et } M_2 \text{ sont des MTs telles que } \mathcal{L}(M_1) = \mathcal{L}(M_2)\}$$

  contient tous les couples de MTs qui acceptent le même langage.

Problème de l'arrêt

Théorème II.8
H_{MT} est récursivement énumérable.

Démonstration:
Reprendre la MT universelle U, et remplacer rejet par accepte, de façon à ce que U accepte tout le temps.
Ou de manière équivalente, en appelant M_H la MT qui énumère H_{MT} :

$M_H(\langle M, w\rangle) =$
```
SIMULER M(w)
// on arrive ici seulement si M ne boucle pas sur w
ACCEPTER
```

M_H accepte trivialement H_{MT}. □

Théorème II.9
H_{MT} est indécidable.

Démonstration:

Même type de preuve que pour A_{MT}.
On suppose qu'il existe une machine Halt($\langle M, w\rangle$) qui décide H_{MT}.
Considérons le programme D qui boucle si :

$D(\langle M\rangle) =$ **si** Halt($\langle M, M\rangle$) **alors boucler**() **sinon accepter**

$D(\langle M\rangle)$ = si Halt($\langle M, M\rangle$) alors boucler() sinon accepter

où *boucler*() est une MT qui boucle (par exemple, qui va à gauche puis à droite sur n'importe quel symbole, et recommence).
Alors, $D(\langle D\rangle)$ donne le résultat suivant :

- si Halt($\langle D, D\rangle$) = rejette, alors $D(\langle D\rangle)$ accepte
  $\Rightarrow$ Halt($\langle D, D\rangle$) devrait accepter.
- si Halt($\langle D, D\rangle$) = accepte, alors $D(\langle D\rangle)$ boucle
  $\Rightarrow$ Halt($\langle D, D\rangle$) devrait rejeter.
- si Halt($\langle D, D\rangle$) = boucle, alors il n'est pas décidable.

Dans les 3 cas, Halt($\langle M, M\rangle$) ne fonctionne pas, donc n'existe pas. □

Autre façon de faire cette démonstration :

Utilisons l'indécidabilité de A_{MT} pour prouver l'indécidabilité de H_{MT}.

Démonstration: ($2^{\text{ème}}$ version)

Supposons qu'il existe une MT R qui décide H_{MT}.
Utilisons R pour construire une MT S qui décide A_{MT} :

$S(\langle M, w\rangle) =$

```
// exécution du décideur de l'arrêt
EXÉCUTER R(⟨M, w⟩).
// R nous dit si M s'arrête sur w
// cas 1: R rejette = M(w) boucle.
SI R REJETTE ALORS REJETER
// cas 2: R accepte = M(w) ne boucle pas.
// on peut simuler M(w) sans crainte
SI R ACCEPTE ALORS SIMULER M(⟨w⟩).
DÉCIDER comme M
```

Clairement, si R décide H_{MT}, alors S décide A_{MT}.
Comme A_{MT} est indécidable, H_{MT} ne peut pas être décidable. □

Cette méthode est dite de **réduction** : consiste à réduire la résolution d'un problème inconnu à la résolution d'un problème déjà connu.

Problème du vide

On rappelle que : $E_{\text{MT}} = \{< M > \mid M \text{ est une MT et } \mathcal{L}(M) = \emptyset\}$

Théorème II.10

E_{MT} est indécidable.

Démonstration: (par réduction)

Idée de la démonstration : construire une MT à partir de $M(\langle w \rangle)$ qui réduit le problème A_{MT} au problème E_{MT}.

Soit la MT M_1 construite à partir d'une MT M et d'une entrée w comme suit :

$M_1(x) =$
```
SI x ≠ w
ALORS REJETER
SINON // exécuté seulement si x = w
  SIMULER M(⟨w⟩) // peut boucler
  DÉCIDER comme M // accepte ou rejette
```

Seul le mot w est accepté par M_1 si seulement si $w \in \mathcal{L}(M)$.
Donc, $\mathcal{L}(M_1) = \emptyset$ si $M(w)$ rejette ou boucle, et $\mathcal{L}(M_1) = \{w\}$ si $M(w)$ accepte.

Autrement dit : $\mathcal{L}(M_1) = \begin{cases} \emptyset & \text{si } w \notin \mathcal{L}(M) \\ \{w\} & \text{si } w \in \mathcal{L}(M) \end{cases}$

Supposons qu'il existe un décideur R de E_{MT}.
Construisons une MT S qui décide A_{MT} :

$S(\langle M, w \rangle) =$
```
CONSTRUIRE ⟨M1⟩ à partir de ⟨M⟩ ET w
EXÉCUTER R(⟨M1⟩)
SI R ACCEPTE
ALORS REJETER   // L(M1) = ∅ ⇒ M(w) rejette ou boucle
SINON ACCEPTER  // L(M1) = {w} ⇒ w ∈ L(M)
```

Le comportement de $R(\langle M_1 \rangle)$ est le suivant :

$$R(\langle M_1 \rangle) = \begin{cases} \text{accepte} & \text{si } \mathcal{L}(M_1) = \emptyset \Rightarrow w \notin \mathcal{L}(M) \\ \text{rejette} & \text{si } \mathcal{L}(M_1) = \{w\} \Rightarrow w \in \mathcal{L}(M) \end{cases}$$

Si R décide E_{MT}, alors S décide A_{MT}.
Comme A_{MT} est indécidable, E_{MT} n'est pas décidable. □

Remarque : ce problème n'est même pas énumérable (impossibilité d'assurer pour n'importe quel code M et par la seule analyse que l'état acceptant n'est jamais atteint). Évidemment, tester si aucun mot n'est accepté n'est pas envisageable non plus.

Problème de l'égalité

Soit $EQ_{\text{MT}} = \{< M_1, M_2 > \mid M_1 \text{ et } M_2 \text{ sont des MTs telles que } \mathcal{L}(M_1) = \mathcal{L}(M_2)\}$

Théorème II.11

EQ_{MT} est indécidable.

Démonstration: (par réduction)

Montrons que la décidabilité de EQ_{MT} se réduit à la décidabilité de E_{MT}.
Supposons qu'il existe une MT R qui décide EQ_{MT}.
Soit $M_\emptyset$ une MT qui rejette toutes les entrées ($\mathcal{L}(M_\emptyset) = \emptyset$) :

$$M_\emptyset(w) = \boxed{\textbf{REJETER}}$$

Construisons une MT S qui décide E_{MT} (problème du vide) :

$$S(\langle M \rangle) = \begin{array}{|l} \textbf{CONSTRUIRE } \langle M_\emptyset \rangle \text{ // } \mathcal{L}(M_\emptyset) = \emptyset \\ \textbf{EXÉCUTER } R(\langle M, M_\emptyset \rangle) \\ \text{// décide si } \mathcal{L}(M) = \mathcal{L}(M_\emptyset) = \emptyset \\ \textbf{DÉCIDER comme } R \end{array}$$

Si R décide EQ_{MT}, alors S décide E_{MT}.
Comme E_{MT} est indécidable, EQ_{MT} n'est pas décidable. □

Remarques 12:

- EQ_{MT} n'est pas récursivement énumérable non plus.
- Plus on résout de problèmes (par réduction), plus on a de choix de réduction.

Exercices

Exercice 7 (Indécidabilité du langage $USELESS_{\text{MT}}$)**.** *Soit :*

$$USELESS_{MT} = \{\langle M, q \rangle \mid q \text{ est un état jamais atteint lors de l'exécution de la MT } M \text{ sur n'importe laquelle de ses entrées}\}$$

Montrer que $USELESS_{MT}$ est indécidable en utilisant une réduction de $USELESS_{MT}$ à E_{MT}.

Exercice 8 (Indécidabilité du langage $A_{\varepsilon-\text{MT}}$)**.** *Soit :*

$$A_{\varepsilon-MT} = \{\langle M \rangle \mid M \text{ est une MT qui accepte } \varepsilon\}$$

Montrer que $A_{\varepsilon-MT}$ est indécidable en utilisant une réduction de $A_{\varepsilon-MT}$ à A_{MT}.

Exercice 9. *Soit $T = \{\langle M \rangle | M$ est une machine de Turing qui accepte w^R chaque fois qu'elle accepte $w\}$ où w^R est l'inverse de la chaîne w. Montrer par réduction que T est indécidable.*

1.5.4 Théorème de Rice

Les problèmes de décidabilité étudiés d'une propriété non triviale d'un code ont tous eu la même conclusion : le problème est indécidable.

Nous voulons savoir s'il existe un sous-ensemble de $\mathcal{RE}$ qui est décidable.

Voyons comment définir un sous-ensemble de langages à partir d'une propriété.

Définition II.8

- Une **propriété** P est un ensemble particulier de langages inclus dans $\mathcal{RE}$ dont le langage possède caractéristique particulière.
  Exemple : $P = \{\langle M \rangle \mid \mathcal{L}(M)$ est un langage décidable$\}$.
- Un langage L **possède la propriété** P ssi $L \in P$.
- Une propriété P est dite **triviale** ssi
  - $P = \emptyset$ (la propriété n'est vérifiée par aucun langage de $\mathcal{RE}$),
  - $P = \mathcal{RE}$ (la propriété n'est vérifiée par tous les langages de $\mathcal{RE}$).
- Une propriété P est **dite non-triviale** ssi il existe :
  - au moins une MT qui a la propriété P.
  - au moins une MT qui a la propriété $\overline{P}$.

Est-il possible de décider un ensemble de langages ayant une certaine propriété ? (n'importe laquelle)

Cas des propriétés triviales

Théorème II.12 (décidabilité des propriétés triviales)

1. La propriété $P_\emptyset = \emptyset$ qui n'est vérifiée par aucun langage est décidable.
2. La propriété $P_{\text{all}} = \mathcal{RE}$ qui est vérifiée par tous les langages est décidable.

Démonstration:

1. Construisons une MT $M_\emptyset$ qui décide $P_\emptyset$.
   Soit la MT $M_\emptyset$:
   $M_\emptyset(w) =$ | **REJETER**
   Il est clair que $M_\emptyset$ reconnaît $P_\emptyset$ (ou $\mathcal{L}(M_\emptyset) = P_\emptyset$, donc si $\mathcal{L}(M) = \emptyset$) et s'arrête toujours (car ne dépend pas de l'entrée).
   Donc, $M_\emptyset$ décide $P_\emptyset$ et $P_\emptyset$ est décidable.
2. Construisons une MT M_{all} qui décide P_{all}.
   Soit la MT M_{all} :
   $M_{\text{all}}(w) =$ | **ACCEPTER**
   Il est clair que M_{all} reconnaît P_{all} (ou $\mathcal{L}(M_{\text{all}}) = P_{\text{all}}$). et s'arrête toujours (car ne dépend pas de l'entrée).
   Donc, M_{all} décide P_{all} et P_{all} est décidable.

□

Théorème II.13 (Rice)

Soit P une propriété non-triviale des langages $\mathcal{RE}$. Alors P est indécidable.

Démonstration: (par réduction)

Sans perte de généralité, supposons que $\emptyset \notin P$

nécessité de la démonstration - sinon effectuer la démonstration avec $\bar{P}$, énumérable lui-aussi si P est décidable).

Comme P n'est pas vide (sinon la propriété n'existe pas), il existe un langage $L \in P$.

Supposons que P est décidable, notons D_P son décideur.

On veut construire une réduction de A_{MT} vers P, afin de prouver que si P est décidable, alors A_{MT} est lui-aussi décidable.

Pour cela, on veut construire une MT M_w à partir de :

- un couple $\langle M, w\rangle$ quelconque dont on veut vérifier s'il appartient à A_{MT}.
- la MT M_L accepte le langage L (ci-dessus) qui vérifie la propriété P.

telle que $(M_w \in P) \Leftrightarrow (\langle M, w\rangle \in A_{\text{MT}})$.

Soit la machine suivante :

$M_w(\langle x\rangle) =$

```
SIMULER M(w)
SI M ACCEPTE
ALORS EXÉCUTER M_L(⟨x⟩)
        SI M_L ACCEPTE
        ALORS ACCEPTER
        SINON REJETER
SINON REJETER
```

Examinons le comportement de M_w :
si $\langle M, w\rangle \in A_{\text{MT}}$
alors M_w exécute $M_L(\langle x\rangle)$ $\quad \mathcal{L}(M_w) = L$, par construction $L \in P$
sinon rejeter $\quad \mathcal{L}(M_w) = \emptyset$, par définition de P, $\emptyset \notin P$

On a donc bien une MT construite à partir de $\langle M, w\rangle$ et de L qui vérifie la propriété P si et seulement si M accepte w.

Construisons alors un décideur D de A_{MT} :

$D(\langle M, w\rangle) =$

```
CONSTRUIRE ⟨M_w⟩ à partir de ⟨M⟩, w ET ⟨M_L⟩
SIMULER D_L(⟨M_w⟩)
DÉCIDER comme D_L
```

Notons la machine M_w ne pose aucune difficulté à être construite. C'est la concaténation de :

- une MT universelle qui s'exécute sur $M(w)$ (M et w variables dans D)
- la MT M_L (toujours la même)

et qu'il s'agit bien d'une réduction de A_{MT} à P.

Donc D est un décideur de A_{MT}.

Or, A_{MT} est indécidable. Donc D n'existe pas.

Comme l'existence de D est basé sur celle de D_P, D_P n'existe pas non plus.

En conséquence, P est indécidable.

Comme la démonstration ci-dessus peut-être effectuée pour toute propriété P (ou $\overline{P}$), on en conclut que toute propriété non triviale est indécidable.

□

Nous verrons plus loin comment ce type de méthodes de réduction peut être formalisée.

Conséquences du théorème de Rice

Exemple : Est-ce qu'un programme trie un tableau d'entiers ?

Attention : il ne s'agit pas de trier un tableau d'entiers, mais de vérifier qu'un programme effectue véritablement cette tâche.

A savoir, **s'il a la propriété** de trier un tableau d'entiers.

Le problème est bien défini : les spécifications exprimées sous forme d'objets mathématiques précis.

Prouver qu'un programme respecte une spécification ne devrait pas être plus difficile que de savoir si un triangle ressemble à un autre.

Le théorème de Rice montre que ce n'est PAS DU TOUT le cas : ce problème est indécidable.

Interprétation en termes de programme :

Il n'existe pas de programme permettant de décider si une propriété non triviale **quelconque** d'un autre programme est vraie.
Ceci est vrai pour **TOUTE** propriété non triviale.

Par exemple :

- savoir si un programme fait ce qu'on lui demande.
- savoir si un programme s'arrête (=ne boucle pas).
- savoir si deux programmes font la même chose.
- ...

Ceci constitue une limite importante **à toute tentative de vérification sur programme**.

Par contre :

Cela ne veut pas dire que le problème ne peut pas être résolu dans certains cas particuliers.

Mais qu'il ne le sera **jamais** par une machine de Turing sur la classe complète des langages de $\mathcal{RE}$.

Attention : à bien comprendre le théorème de Rice

Il dit : pour une MT M, les questions suivantes sur $\mathcal{L}(M)$ sont non-décidables.

Est-ce que $\mathcal{L}(M)$:

- est vide ?
- est fini ?
- est infini ?
- est décidable ?
- est régulier ?
- $= \Sigma^*$?
- = {Hello, World} ?
- contient un palindrome ?
- contient une chaîne de longueur paire ?

La vérification de toute propriété sur le langage engendré par une machine de Turing est indécidable.
Attention : ne sont pas concernées par le théorème de Rice :

- Les propriétés concernant l'encodage d'une MT
  Exemple : "$\langle M \rangle$ a un nombre pair d'états ?" est décidable.
- Les propriétés concernant les étapes intermédiaires de l'exécution d'une MT :
  — si on passe par un état particulier.
  — si on passe par une configuration particulière (état de la bande).
  — le nombre de transitions ...

  Exemple : "M passe par q_6 pour l'entrée vide"
  Non décidable, mais théorème de Rice ne s'applique pas.

Exercice 10. *Utiliser le théorème de Rice pour prouver l'indécidabilité de* $\{\langle M \rangle | M$ *est une machine de Turing telle que* $1011 \in \mathcal{L}(M)\}$.

Exercice 11. *Utiliser le théorème de Rice pour prouver l'indécidabilité de* $INFINITE_{TM} = \{\langle M \rangle | M$ *is a TM and* $\mathcal{L}(M)$ *is an infinite language*$\}$.

Exercice 12. *Soit* $T = \{\langle M \rangle | M$ *est une machine de Turing qui accepte* w^R *chaque fois qu'elle accepte* $w\}$ *où* w^R *est l'inverse de la chaîne* w. *Montrer avec le théorème de Rice que* T *est indécidable.*

1.6 Conclusion

Nous avons vu qu'il existait plusieurs types de problèmes qui sont toujours indécidables :

- Les problèmes non-énumérables,
- Les problèmes devant décider des propriétés sur des programmes (th. de Rice)

Ce ne sont pas les seuls.

- Le $10^{\text{ème}}$ problème de Hilbert,
- La pavage de Wang (impossibilité de pavage automatique d'un plan),
- Le calcul de la complexité de Kolmogorov (description minimale d'un bloc de données),
- ...

Tant qu'il n'a pas été démontré qu'un problème est décidable, on ne doit pas supposer qu'il l'est.

Rappel : pour démontrer qu'un problème est décidable, il suffit de montrer qu'il **ne boucle jamais**, et qu'il **s'arrête toujours**.

2 Calculabilité

2.1 Lien entre calculabilité et décidabilité

Rappelons les définitions d'une fonction calculable et partiellement calculable.

Définition II.9 (fonction (totalement) calculable)

Une MT M calcule totalement une fonction $f : \Sigma^* \to \Sigma^*$ si M

- commence avec l'entrée w sur sa bande.
- s'arrête pour tout w et avec seulement $f(w)$ écrit sur sa bande.

Définition II.10 (fonction partiellement calculable)

Une MT M calcule partiellement une fonction $f : \Sigma \to \Sigma \cup \perp$ si M :

- commence avec l'entrée w sur sa bande.
- si $f(w)$ est défini, alors M s'arrête avec seulement $f(w)$ écrit sur sa bande.
- si $f(w)$ est indéfini, alors M ne s'arrête pas.

On peut effectuer un lien direct entre calculabilité et complexité.

Exemple 17:

- **Formulation du problème :** Soit trois entiers x, y, z. Est-ce que $z = x \times y$?
- **Problème de décision :**
  $L = \{\langle x, y, z\rangle \mid \ x, y, z$ sont des entiers et $z = x \times y\}$.
  Ce problème est **décidable**.
- **Problème d'évaluation :**
  $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ tel que $f(x, y) = x \times y$.
  Cette fonction est **calculable**.

Exemple 18:

- **Formulation du problème :** Soit une machine de Turing M. Est-ce que M a un nombre pair d'états ?
- **Problème de décision :**
  $L = \{\langle M\rangle \mid$ la MT M a un nombre pair d'états $\}$.
  Ce problème est **décidable**.
- **Problème d'évaluation :**
  $f : \{\langle M\rangle\} \to$ booléen tel que $f(\langle M\rangle) =$ est vrai si la MT $\langle M\rangle$ a un nombre d'états pairs et faux sinon.$\}$
  Cette fonction est **calculable**.

Exemple 19:

- **Formulation du problème :** Soit une machine de Turing M et une chaîne w. Est-ce que $M(w)$ s'arrête ?
- **Problème de décision :**
  $L = \{\langle M, w\rangle \mid$ la MT $M(w)$ s'arrête$\}$.
  Ce problème est **récursivement énumérable** (c'est le problème de l'arrêt).
- **Problème d'évaluation :**
  $f : \{\langle M, w\rangle\} \to \mathbb{N}$ tel que $f(\langle M, w\rangle) =$ le nombre de transitions au bout duquel M s'arrête sur w, sinon la réponse est indéfinie.
  Cette fonction est **partiellement calculable**.

EXEMPLE 20:

- **Formulation du problème :** Soit une machine de Turing M. Est-ce que M s'arrête sur toutes les entrées en moins de n transitions ?
- **Problème de décision :**
  $L = \{\langle M, n\rangle \mid$ la MT M s'arrête sur tous les mots de Σ^* en moins de n transitions $\}$.
  Ce problème est **indécidable**.
- **Problème d'évaluation :**
  $f : \{\langle M\rangle\} \rightarrow \mathbb{N}$ tel que $f(\langle M, w\rangle) =$ le nombre maximum de transitions au bout duquel M s'arrête sur n'importe quel mot de Σ^*, sinon la réponse est indéfinie.
  Cette fonction n'est pas **partiellement calculable** (on exclut les machines qui ne tiennent pas compte de la chaîne d'entrée).

2.2 Exemple de fonction non calculable

Donnons maintenant un exemple célèbre de fonction non calculable.
Le castor affairé est la fonction $\mathcal{S}(n)$ définie de la manière suivante.

- Soit l'ensemble de toutes les MTs à une bande avec $\Gamma = \{\sqcup, 1\}$, la bande infinie des deux côtées, initialisée avec $\sqcup$.
- Soit le sous-ensemble S_n des MTs M à n états tel que $M(\epsilon)$ s'arrête.

  L'ensemble S_n est fini (nombre fini d'états et de transitions possibles). Par définition, $M \in S_n$ s'exécute en un nombre fini de pas (il doit s'arrêter).
- Soit $\mathcal{S}(n)$ le nombre maximum de transitions exécutées par une machine de S_n sur l'entrée ϵ.

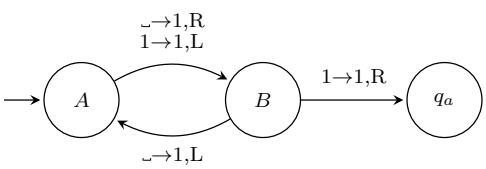
  De ce qui précède, $\mathcal{S}(n)$ est une fonction complètement définie.

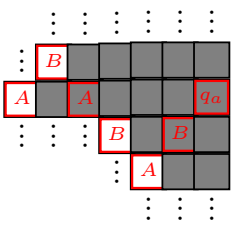
Cette fonction a les propriétés suivantes :

- Cette fonction est celle qui croît le plus rapidement

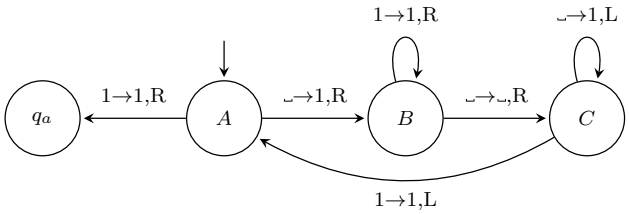
  $\mathcal{S}(1) = 1$, $\mathcal{S}(2) = 6$, $\mathcal{S}(3) = 14$, $\mathcal{S}(4) = 107$, $\mathcal{S}(5) = 47.176.870$, $\mathcal{S}(6) = 7.412.10^{36534}$, $\mathcal{S}(12) >$ 4096 mis 166 fois à la puissance 4096.
- les premières MTs associées à $\mathcal{S}(n)$ peuvent être trouvées :

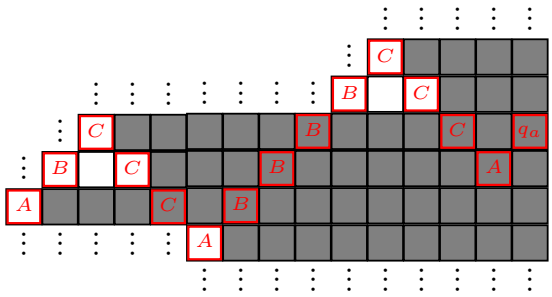
cas $n = 2$: $\mathcal{S}(2) = 6$



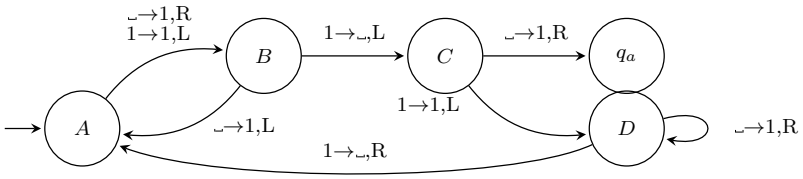


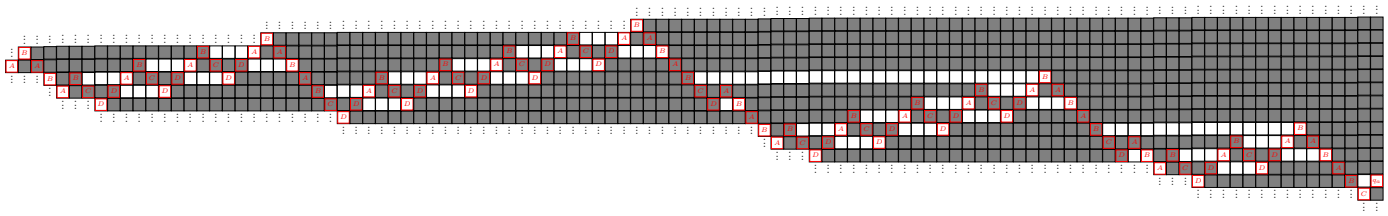
cas $n = 3$: $\mathcal{S}(3) = 14$





cas $n = 4$: $\mathcal{S}(4) = 107$





Montrons que cette fonction n'est pas calculable.

Théorème II.14

la fonction $\mathcal{S}(n)$ du castor affairé n'est pas calculable.

Démonstration:

Supposons que $\mathcal{S}(n)$ soit calculable.
Comme $\mathcal{S}(n)$ est le nombre maximum de transitions pour qu'une MT M à n états s'arrête, M ne s'arrêtera pas si elle dépasse ce nombre de transition.
On peut alors construire la MT S qui vérifie si un programme M s'arrête sur ε :

$S(\langle M \rangle)$ = calculer le nombre n d'états de M.
calculer $\mathcal{S}(n)$.
exécuter au plus $\mathcal{S}(n) + 1$ transitions de M.
si M s'est arrêté, alors accepter.
si M ne s'est pas arrêté alors rejeter.

Si un tel programme existait, H_{MT} serait décidable.
Comme H_{MT} n'est pas décidable, $\mathcal{S}(n)$ n'est pas calculable. □

Paradoxe :

On peut calculer $\mathcal{S}(n)$ pour certaines valeurs de n, mais $\mathcal{S}(n)$ n'est pas calculable ?

- Tout segment fini d'une suite de fonctions non calculables est calculable :
  pour tout n, il existe un algorithme qui calcule la suite $\mathcal{S}_n = \{\mathcal{S}(0), \mathcal{S}(1), \ldots, \mathcal{S}(n)\}$, ou pour une valeur de n particulière.
- mais il n'existe aucun algorithme qui calcule la suite entière de $\mathcal{S}(n)$ (*i.e.* $\mathcal{S}(n), \forall n$).

Toujours pas ? Cela tient à la définition d'un algorithme :

- Un algorithme est une méthode permettant de résoudre une **classe** de problème.
- $\mathcal{S}(n)$ ne se calcule que pour des valeurs de n déterminées.
- Comme il n'existe pas d'algorithme calculant $\mathcal{S}(n)$ pour tout n, cette fonction n'est donc pas calculable.

Cela revient à la différence entre être capable de résoudre un problème dans un cas particulier et dans le cas général.

3 Réductibilité

Lors de plusieurs démonstrations de décidabilité, nous avons utilisé le principe de réductibilité.

Ce modèle de démonstration est schématiquement le suivant :

- on connaît la propriété d'un certain langage A,
- on voudrait savoir si un autre langage B possède la même propriété.
- on construit une fonction dite "de réduction" capable de transformer tout mot de B en un mot de A.
- on en déduit que B a la même propriété que A.

Donnons-en maintenant une définition formelle.

3.1 Définition de la réductibilité

Définition II.11 (application réductive)

Soit A et B deux langages. Soit f une fonction **calculable**.

f est une application réductive (ou réduction) de A vers B s'il existe une fonction calculable $f : \Sigma^* \to \Sigma^*$ telle que pour tout w,

$$(w \in A) \Leftrightarrow (f(w) \in B)$$

Notation : $A \leq_m B$

Rappel : on a : $(u \Rightarrow v) \Leftrightarrow (\overline{v} \Rightarrow \overline{u})$. Donc :

$((w \in A) \Rightarrow (f(w) \in B)) \Rightarrow ((f(w) \notin B) \Rightarrow (w \notin A))$

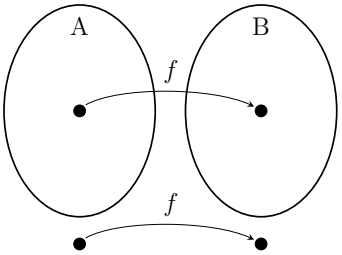
$((f(w) \in B) \Rightarrow (w \in A)) \Rightarrow ((w \notin A) \Rightarrow (f(w) \notin B))$

D'où on déduit $(w \notin A) \Leftrightarrow (f(w) \notin B)$

Remarque 13:

Une réduction de A vers B permet de convertir une question sur l'appartenance à A en une question sur l'appartenance à B.
A savoir :

- si $f(w) \in B$ alors $w \in A$.
- si $f(w) \notin B$ alors $w \notin A$.



Interprétation intuitive :

Comprendre $A \leq_m B$ comme :

- tout problème sur A peut être transformé en un problème sur B.

- tout problème de A n'est pas plus compliqué qu'un problème de B.
- tout problème de B est au moins aussi difficile qu'un problème de A

Donnons maintenant des exemples d'applications.

3.2 Lien avec la décidabilité

Théorème II.15 (décidabilité par réduction)

si $A \leq_m B$ et B est décidable alors A est décidable.

Démonstration:

Soit M_B une MT qui décide pour B, et f une réduction de A vers B.
Alors, la MT M_A suivante décide pour A :

$M_A(\langle w \rangle) =$

```
CALCULER ⟨v⟩ = f(⟨w⟩)      // ⇒ f(w) ∈ B
EXÉCUTER M_B(⟨v⟩)          // décide si f(w) ∈ B (ne boucle pas)
DÉCIDER comme M_B          // (f(w) ∈ B) ⇔ (w ∈ A)
```

f s'arrête toujours (fonction calculable), tout comme M_B (décideur de B). Donc M_A ne boucle jamais.
Ce code décide si $f(w) \in B$.
Or, $(f(w) \in B) \Leftrightarrow (w \in A)$.
Donc, si $f(w) \in B$ alors $w \in A$. Si $f(w) \notin B$, alors $w \notin A$.
Ainsi, M_A est un décideur de A. D'où A décidable. □

Corollaire II.3 (indécidabilité par réduction)

Si $A \leq_m B$ et A est indécidable alors B est indécidable.

Démonstration:

Supposons que B soit décidable. On se retrouve alors dans le cadre du théorème précédent : $A \leq_m B$ et B décidable implique que A est décidable. Or, A n'est pas décidable, donc B non plus. □

Lemme II.5

$A \leq_m B$ implique $\bar{A} \leq_m \bar{B}$

Démonstration:

Simple contraposée logique (rappel : $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$). Donc $(w \in A) \Leftrightarrow (f(w) \in B)$ est équivalent à $(w \notin A) \Leftrightarrow (f(w) \notin B)$ □

Remarque 14:

utile pour démontrer les propriétés sur les ensembles complémentaires.

3.3 Application à l'énumérabilité

Théorème II.16 (énumérabilité par réduction)

Si $A \leq_m B$ et B est récursivement énumérable alors A est récursivement énumérable.

Démonstration:

La démonstration est quasiment identique à la même proposition sur la décidabilité.
Soit M_B la MT qui énumère B.
Soit f la fonction calculable associée à la réduction de A à B.
Alors, la MT suivante énumère A :

$$M_A(\langle w\rangle) = \begin{array}{|ll|l} \textsc{calculer} & \langle v\rangle = f(\langle w\rangle) & // \Rightarrow f(w) \in B \\ \textsc{exécuter} & M_B(\langle v\rangle) & // \text{ accepte , rejette ou boucle} \\ \textsc{décider} & \text{comme } M_B & // (f(w) \in B) \Leftrightarrow (w \in A) \end{array}$$

$M_B(f(\langle w\rangle))$ peut faire 3 choses :

- **accepter :** comme $(f(w) \in B) \Leftrightarrow (w \in A)$. Donc M_A accepte quand $w \in A$.
- **rejeter :** comme $(f(w) \notin B) \Leftrightarrow (w \notin A)$. Donc M_A rejette quand $w \notin A$.
- **boucler :** dans ce cas $f(w) \notin B$, donc $(w \notin A)$. M_A boucle aussi, mais signifie que $w \notin A$, ce qui est bien le comportement attendu.

En conséquence, M_A énumère A, et A est énumérable. □

Corollaire II.4

si $A \leq_m B$ alors

- si A n'est pas $\mathcal{RE}$ alors B n'est pas $\mathcal{RE}$.
- si A n'est pas $co\mathcal{RE}$ alors B n'est pas $co\mathcal{RE}$.

Démonstration:

- Si B était $\mathcal{RE}$, le théorème ci-dessus impliquerait que A est $\mathcal{RE}$. Contradiction.
- Même démonstration avec les ensembles complémentaires, et grâce au lemme impliquant $\bar{A} \leq_m \bar{B}$.

□

3.4 Exercices

Exercice 13. *Montrer que A est récursivement énumérable si $A \leq_m A_{TM}$.*

Exercice 14. *Donnez un exemple de langage indécidable B tel que $B \leq_m \overline{B}$.*

Exercice 15 (Indécidabilité de $\text{USELESS}_{\text{MT}}$ par réduction formelle)**.** *Soit :*

$$USELESS_{MT} = \{\langle M, q\rangle \mid q \text{ est un état jamais atteint lors de l'exécution de la MT } M \text{ sur n'importe laquelle de ses entrées}\}$$

*Montrer que $USELESS_{MT}$ est indécidable en utilisant une **réduction formelle** de $USELESS_{MT}$ à E_{MT}.*

Exercice 16 (Indécidabilité de $A_{\varepsilon-\text{MT}}$ par réduction formelle)**.** *Soit :*

$$A_{\varepsilon-MT} = \{\langle M\rangle \mid M \text{ est une MT qui accepte } \varepsilon\}$$

*Montrer que $A_{\varepsilon-MT}$ est indécidable en utilisant une **réduction formelle** de $A_{\varepsilon-MT}$ à A_{MT}.*

Exercice 17. *Considérons le problème de déterminer si une machine de Turing M sur une entrée w tente jamais de déplacer sa tête vers la gauche lorsque sa tête est sur la première cellule de la bande. Formulez ce problème sous forme de langage et montrez par réduction qu'il est indécidable.*

Exercice 18. *Considérons le problème consistant à déterminer si une machine de Turing à une bande n'écrit à aucun moment un symbole vide sur un symbole non vide au cours de son exécution sur une chaîne d'entrée quelconque. Formulez ce problème sous la forme d'un langage et montrez par réduction qu'il est indécidable.*

3.5 Fonction à exécution contrôlée

Une classe intéressante de fonctions :
fonction qui modifie la machine qu'elle exécute.

Exemple 21: fonction à exécution contrôlée La machine F suivante termine l'exécution de la machine M si M essaie d'aller à gauche alors que le pointeur de lecture est déjà au début de la bande.

$F(\langle M, w\rangle) =$

```
exécution pas à pas de M sur w
  trouver la transition suivante ( état ,symbole, direction )
  SI on est en début de bande
  ALORS
    SI direction = L
    ALORS ÉCRIRE ϵ
      ARRÊTER
  appliquer la transition
  SI état = q_a ALORS ARRÊTER
```

3.6 Exécution contrôlée

Réduction par exécution contrôlée :

Technique de réduction qui consiste à construire la MT pas à pas.

Prenons par exemple, $L_\infty = \{\langle M\rangle \mid \mathcal{L}(M) \text{ est infini}\}$.
Par le théorème de Rice, on sait que $L_\infty \notin \mathcal{R}$.
Montrons aussi que $L_\infty \notin \mathcal{RE}$ en utilisant une réduction de $\overline{H_{\text{MT}}}$.
Rappel : $\overline{H_{\text{MT}}}$ = complémentaire de H_{MT} = ensemble des MTs qui ne s'arrêtent pas.

Lemme II.6

$\overline{H_{\text{MT}}} \notin \mathcal{RE}$

Démonstration:

On sait que $H_{\text{MT}} \notin \mathcal{R}$, mais $H_{\text{MT}} \in \mathcal{RE}$. Donc, nécessairement $H_{\text{MT}} \notin co\mathcal{RE}$, d'où $\overline{H_{\text{MT}}} \notin \mathcal{RE}$. □

Théorème II.17

$L_\infty \notin \mathcal{RE}$

DÉMONSTRATION:

La ligne de preuve est la suivante :

- on sait que $\overline{H_{\text{MT}}}$ n'est pas récursivement énumérable.
- si on trouve une réduction telle que : $\overline{H_{\text{MT}}} \leq_m L_\infty$
- on en déduit que L_∞ n'est pas récursivement énumérable.

Cherchons une réduction f : $\overline{H_{\text{MT}}} \to L_\infty$ telle que : $\langle M, w\rangle \mapsto \langle M_0\rangle$

- si M s'arrête sur w alors $\mathcal{L}(M_0)$ est fini.
- si M ne s'arrête pas sur w alors $\mathcal{L}(M_0)$ est infini.

On rappelle que $|w|$ retourne la longueur du mot w.

Soit la MT M_0 suivante construite à partir de $\langle M, w\rangle$ et une MT universelle U.
Si $M(w)$ ne boucle pas, notons $k_{M,w}$ le nombre de transitions nécessaire à $M(w)$ pour s'arrêter.

$M_0(\langle v\rangle) =$

```
EXÉCUTER |v| transition de U(⟨M, w⟩)
SI U s'est arrêtée
ALORS REJETER    // M(w) s'arrête, rejette si |v| > k_{M,w}
SINON ACCEPTER   // M(w) ne s'est pas arrêtée ou boucle, accepte tout
```

Examinons le comportement de $\mathcal{L}(M_0)$:

- si M **boucle** sur w, alors M_0 accepte v en entier, donc $\mathcal{L}(M_0) = \Sigma^*$, et $\langle M_0\rangle \in L_\infty$.
- si M **s'arrête** sur w après k transitions, alors M_0 accepte tous les mots $v \in \Sigma^*$ de longueur inférieure ou égale à k donc $\mathcal{L}(M_0)$ est fini, et $\langle M_0\rangle \notin L_\infty$.

Pour résumer, le comportement de $L(M_0)$:

- $\langle M, w\rangle \in \overline{H_{\text{MT}}} \Rightarrow \langle M_0\rangle \in L_\infty$.
- $\langle M, w\rangle \notin \overline{H_{\text{MT}}} \Rightarrow \langle M_0\rangle \notin L_\infty$.

D'où $(\langle M, w\rangle \in \overline{H_{\text{MT}}}) \Leftrightarrow (\langle M_0\rangle \in L_\infty)$
Soit la fonction de réduction f définie par :

$f(\langle M, w\rangle) =$

```
CONSTRUIRE ⟨M_0⟩ à partir de ⟨M, w⟩ ET U
RETOURNER ⟨M_0⟩
```

f est bien une fonction calculable (MTU exécutant partiellement $M(w)$).
Donc, f est une réduction de $\overline{H_{\text{MT}}}$ dans L_∞ et $\overline{H_{\text{MT}}} \leq_m L_\infty$. Or $\overline{H_{\text{MT}}} \notin \mathcal{RE}$, ce qui implique $L_\infty \notin \mathcal{RE}$ (cf corollaire réductibilité).

□

REMARQUES 15: technique utilisée pour des preuves comme

- tester l'existence de certains objets.
- est-ce qu'une MT bornée linéairement accepte le langage vide ?
- est-ce qu'une GLC génère Σ^* ?

4 Théorème de récursion

4.1 Principe

Nous abordons dans cette partie la possibilité de créer des machines capables de construire des répliques d'elles-mêmes.

Une première approche naïve serait de dire :

- si une machine A construit une machine B,
  alors A doit être plus complexe que B.
- Mais une machine ne peut être plus complexe qu'elle-même.
- Donc, aucune machine ne peut se construire elle-même, et donc l'auto-reproduction est impossible.

Les assertions ci-dessus sont fausses : créer une machine qui se reproduit elle-même est possible.

Ce fait est affirmé par le théorème de récursion, qui nécessite le lemme suivant.

Lemme II.7

Il est existe une fonction calculable $q : \Sigma^* \to \Sigma^*$ telle que pour toute chaine $w \in \Sigma^*$, $q(w)$ est la description de la machine de Turing P_w qui écrit w sur la bande et s'arrête.

DÉMONSTRATION:

La construction de P_w et l'écriture de $\langle P_w \rangle$ peuvent être trivialement effectuée.

Soit la fonction suivante qui remplace le contenu de la bande par w est clairement calculable :

$$P_w(\langle x \rangle) = \begin{array}{|l} \textsc{effacer } \langle x \rangle \text{ de la bande} \\ \textsc{écrire } \langle w \rangle \end{array}$$

On considère la fonction calculable Q suivante qui calcule $q(w)$:

$$Q(\langle w \rangle) = \begin{array}{|l} \textsc{construire } \langle P_w \rangle \\ \textsc{écrire } \langle P_w \rangle \end{array}$$

La MT Q s'arrête donc bien dans tous les cas avec $\langle P_w \rangle$ écrit sur sa bande.
Donc, q existe et est bien une fonction calculable.

□

On cherche maintenant à écrire une MT `SELF` capable de se répliquer (= de s'exécuter et de produire en fin d'exécution son propre code sur la bande).

Pour ce faire, `SELF` doit nécessairement être constituée d'au moins deux parties A et B (à savoir $\langle \texttt{SELF} \rangle = \langle AB \rangle$) telles que :

- la partie A qui écrit une description de B :
  on utilise la machine $P_{\langle B \rangle}$ (= remplace le contenu de la bande par $\langle B \rangle$).
  A nécessite donc d'avoir la description de B : on ne peut donc pas décrire A avant d'avoir construit B.
- la partie B qui écrit une description de A :
  On ne peut pas utiliser $q(\langle A \rangle)$ car A est lui-même défini en fonction de B (= définition circulaire = transgression logique).
  Autre stratégie : B calcule A à partir de la sortie que A devrait produire.

Si B obtient $\langle B\rangle$ (= sa propre description), il peut calculer $\langle A\rangle = q(\langle B\rangle)$.
mais comment obtenir $\langle B\rangle$?
Puisque A (après son exécution) écrit $\langle B\rangle$, il suffit donc pour B de lire la bande.
Donc, B lit $\langle B\rangle$ sur la bande, calcule $q(\langle B\rangle) = A$, combine A et B en une seule machine, et écrit la description $\langle AB\rangle$ sur la bande.

En résumé :

A = $Q(\langle B\rangle)$ = écrit $\langle B\rangle$ sur la bande
$B(\langle M\rangle)$ =

```
                                          // avec M = B
CALCULER Q(⟨M⟩)                           // = Q(⟨B⟩)
concaténer le résultat avec ⟨M⟩ // = Q(⟨B⟩)⟨B⟩ = ⟨A⟩⟨B⟩
ÉCRIRE Q(⟨M⟩)⟨M⟩
```

SELF = AB

Le code de SELF est $\langle \texttt{SELF}\rangle = \langle A\rangle\langle B\rangle = Q(\langle B\rangle)\langle B\rangle$.

L'exécution de SELF donne donc :

1. l'exécution de A ecrit $< B >$ sur la bande.
2. l'exécution de B lit $< B >$ sur la bande, calcule $q(\langle B\rangle)$, puis y réécrit la concaténation $Q(\langle B\rangle)\langle B\rangle$.

A la fin de l'exécution, le contenu de la bande est $Q(\langle B\rangle)\langle B\rangle$,
soit le code de SELF.

Exercice 19. *Donnez un exemple d'un programme dans un langage de programmation réel (ou une approximation raisonnable de celui-ci) qui s'affiche lui-même.*

Le théorème de récursion fournit la possibilité d'implémenter l'auto-référence *this* à toute MT (= à tout langage de programmation).

Grâce à ce théorème, tout programme peut faire référence à sa propre description.

Théorème II.18 (de récursion)

Soit la MT T qui calcule la fonction $t : \Sigma^* \times \Sigma^* \to \Sigma^*$.
$\exists$ MT R qui calcule la fonction $r : \Sigma^* \to \Sigma^*$ telle que $\forall w \in \Sigma^*$, $r(w) = t(\langle R\rangle, w)$.

Autrement dit, on voudrait construire une MT R qui utilise sa propre description lors de son exécution $R(w)$ sur une entrée w.

Pour cela, on peut construire une MT T qui effectuerai le même travail mais en recevant la description de R en paramètre (*i.e.* avec $T(\langle R\rangle, w)$).

Dire qu'il existe R telle que $r(w) = t(\langle R\rangle, w)$ signifie qu'il est possible :

- de créer une machine qui fait référence à sa description,
- d'intégrer sa propre description au code d'une MT.

Voyons maintenant comment.
On reprend l'idée de SELF, afin qu'un programme puisse obtenir sa propre description et l'utiliser pour des traitements.

Démonstration:

On construit une MT R en trois partie A, B et T (*i.e.* $\langle R\rangle = \langle ABT\rangle$).

$A = P_{\langle BT\rangle}$ est décrite par $q(\langle BT\rangle)$.
Mais, pour conserver l'entrée w, on modifie q telle que $P_{\langle BT\rangle}$ écrive sa sortie après l'entrée w déjà présente sur la bande (*i.e.* après l'exécution de A, la bande contient $w\langle BT\rangle$).

Ensuite, B lit la bande et applique q à son contenu donnant A.
Puis, B combine A, B et T en une MT R dont la description est $\langle ABT\rangle$.
Cette description est de nouveau combinée avec w, afin d'écrire $\langle R, w\rangle$ sur la bande.

Enfin, le contrôle est passé à T, ce qui permet bien d'obtenir la fonction recherchée. □

Pour résumer, à la fin de l'exécution T a bien pour entrée $\langle R, w\rangle$:

exécution		A	B	T
contenu de la bande	w	$w\langle BT\rangle$	$\langle\langle ABT\rangle, w\rangle = \langle R, w\rangle$	

Quel est l'intérêt du théorème de récursion ?

- permet d'inclure au code d'une MT M l'instruction «obtenir la propre description de M» (avec le théorème de récursion).

  `SELF` = | obtenir la propre description de `SELF`
  écrire ⟨`SELF`⟩
- une fois cette description obtenue, M peut calculer son nombre d'états, simuler $\langle M\rangle$, ...
- certains virus informatique utilisent ce principe pour se dupliquer.
- certaine démonstration d'indécidabilité ou de non récursive-énumérabilité peuvent aussi être effectués avec le théorème de récursion.

4.2 Problème de l'arrêt

Le théorème de récursion permet de fournir une nouvelle preuve du problème de l'acceptation.

Théorème II.19 (par le théorème de récursion)

$A_{\mathtt{MT}}$ est indécidable.

Démonstration:

Par l'absurde : supposons qu'il existe une MT T qui décide $A_{\mathtt{MT}}$ (*i.e.* $T(\langle M, w\rangle)$ accepte si $M(w)$ accepte et rejette sinon).

Soit la MT M suivante :

$M(\langle w\rangle) =$
```
OBTENIR ⟨M⟩                // th. de récursion
EXÉCUTER T(⟨M, w⟩)
DÉCIDER l'opposé de T
```

Comme M décide l'inverse de ce que T indique décider, T ne peut pas exister et $A_{\mathtt{MT}}$ est indécidable. □

4.3 Problème de la machine de Turing minimale

Définition II.12 (longueur de la description d'une MT)

La longueur de la description d'une MT M est le nombre de symbole de la chaîne de caractère décrivant $\langle M\rangle$.

Définition II.13 (MT minimale)

Une MT M est dite minimale s'il n'existe aucune autre MT équivalente (*i.e.* reconnaissant le même langage) et dont la longueur de la description est plus petite.

On définit le langage MIN_{MT} le langage constitué de l'ensemble des MTs minimales.

$$\textit{i.e. } \text{MIN}_{\text{MT}} = \{\langle M\rangle \mid M \text{ est une MT minimale}\}$$

Théorème II.20

MIN_{MT} n'est pas récursivement énumérable.

Démonstration:

Supposons qu'il existe un énumérateur E qui énumère MIN_{MT}.

Soit la MT C suivante :

$C(\langle w\rangle) =$
```
OBTENIR ⟨C⟩        // th. de récursion
EXÉCUTER l'enumérateur E jusqu'à trouver < D >
            telle que |⟨D⟩| > |⟨C⟩|
SIMULER D(w)
DÉCIDER comme D
```

Comme MIN_{MT} est infini, il existe nécessairement une machine D avec une description plus longue que C. Donc, l'énumération s'arrête toujours.

Or C simule D (et produit donc le résultat de D). Donc C et D sont équivalents. Mais C a une description plus petite que D (D choisi avec une description plus grande), donc D n'est pas minimale.

Or, D fait partie des machines énumérées par E. Contradiction : l'énumérateur E n'existe pas, et MIN_{MT} n'est pas récursivement énumérable. □

Exercice 20. *Montrer que tout sous-ensemble infini de MIN_{TM} n'est pas récursivement énumérable.*

5 Décidabilité des théories logiques

5.1 Idée

On aimerait pouvoir écrire un programme qui nous permettrait de tester des expressions mathématiques telles que :

1. $\forall q, \exists p, \forall x, y\,[p > q \wedge (x, y > 1 \Rightarrow xy \neq p)]$
   existence d'un nombre infini de nombres premiers (preuve Euclide, -300 ans)
2. $\forall a, b, c, n\,[(a, b, c > 0 \wedge n > 2) \Rightarrow a^n + b^n \neq c^n]$
   dernier théorème de Fermat (conjecture Pierre de Fermat 1937, preuve Andrew Wiles, 1993)
3. $\forall q \exists p \forall x, y\,[p > q \wedge (x, y > 1 \Rightarrow (xy \neq p \wedge xy \neq p + 2))]$
   conjecture des nombres premiers jumeaux (non prouvé)

En terme d'informatique théorique, cela revient à :

- définir un alphabet contenant les symboles utilisés pour définir ces expressions,
- définir un langage pour lequel l'ensemble des propositions vraies font parties du langage,
- déterminer si ce langage est décidable.

5.2 Enoncé

On définit l'alphabet du langage $\Sigma = \{\wedge, \vee, \neg, (,), \forall, \exists, x, R_1, \ldots, R_k\}$.
composé des opérateurs booleens ($\wedge, \vee, \neg$), des parenthèses, des quantificateurs, des variables ($x_1 = x$, $x_2 = xx$, $x_3 = xxx$, ...), des relations ($R_1, \ldots, R_k$, par exemple $R_1 \equiv <$).
Définitions :

- Une formule atomique est une chaine de la forme : $R_i(x_1, \ldots, x_j)$.
- Une chaine ϕ est une formule si ϕ est :
  - — une formule atomique,
  - — de la forme $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\neg\phi_1$ où ϕ_1 et ϕ_2 sont des formules,
  - — de la forme $\exists x_i\,[\phi_1]$ ou $\forall x_i\,[\phi_1]$ où ϕ_1 est une formule.
- Une variable libre est une variable sans quantificateur.
- Une formule sans variable libre est un énoncé (ou formule close).

Exemple 22: Enoncé $\forall x_1, \exists x_2, \exists x_3\,[R_1(x_1) \wedge R_2(x_1, x_2, x_3)]$ est un énoncé.

Ce type d'énoncé logique est du premier ordre (= quantification des variables), par opposition aux logiques d'ordre supérieur où les relations peuvent être quantifiées.

5.3 Univers et modèle

Pour définir complètement un langage, il est encore nécessaire d'ajouter :

- un univers sur lequel les variables peuvent prendre leurs valeurs (exemple : $\mathbb{N}$),
- un modèle $\mathcal{M}$ est un $k+1$-uple $(U, P_1, \ldots, P_k)$ où U est l'univers, et P_i l'assignation de R_i.

Exemple 23: Soit l'énoncé $\phi = \forall x_1 \forall x_2\,[R_1(x_1, x_2) \vee R_1(x_2, x_1)]$.
Soit le modèle $\mathcal{M}_1 = (\mathbb{N}, \leq)$ (*i.e.* $x_i \in \mathbb{N}$ et $R_1 = \leq$). ϕ est vrai dans $\mathcal{M}_1$.
Par contre, ϕ est faux dans $(\mathbb{N}, <)$ dans le cas où $x_1 = x_2$.

Exemple 24: On définit la relation $\texttt{PLUS}(a, b, c) = \texttt{vrai}$ si $a + b = c$.
Soit $\psi = \forall x_1, \exists x_2\,[R_1(x_1, x_1, x_2)] = \forall x_1, \exists x_2\,[x_1 + x_1 = x_2]$.
Soit le modèle $\mathcal{M}_2 = (\mathbb{R}, \texttt{PLUS})$.
ψ est vrai (signifie que x peut être divisé par 2) dans $\mathcal{M}_2$.
ψ est faux dans $(\mathbb{N}, \texttt{PLUS})$.

5.4 Théorie

Si $\mathcal{M}$ est un modèle, on appelle la théorie de $\mathcal{M}$ (notée $\mathrm{Th}(\mathcal{M})$) l'ensemble des énoncés vrais dans le langage du modèle.

En 1928, Hilbert posa le «Entscheidungsproblem», à savoir s'il existait un algorithme capable de déterminer si tout énoncé issu d'une logique du premier ordre était décidable.

Les résultats suivants ont été obtenus (non démontrés ici, voir Sipser) :

Théorème II.21
Th($\mathbb{N}, +$) est décidable.

Théorème II.22
Th($\mathbb{N}, +, \times$) est récursivement énumérable.

Théorème II.23 (Church, Turing, 1936)
Th($\mathbb{N}, +, \times$) est indécidable.

Autrement dit, il est impossible de démontrer si certains énoncés de Th($\mathbb{N}, +, \times$) sont vrais ou faux.

5.5 Théorème de Gödel

Une preuve formelle π d'un énoncé ϕ est une suite d'énoncé $S_1, S_2, \ldots, S_n$ où $S_n = \phi$, et où chaque S_i est obtenu à partir de S_{i-1}, d'axiomes et de règles d'implication.

On attend d'une preuve les deux propriétés suivantes :

- la validité de la preuve d'un énoncé peut être vérifiée par une machine (*i.e.* $\{\langle \phi, \pi \rangle \mid \pi$ est une preuve de $\phi\}$ est décidable),
- le système de preuve est consistant : si un énoncé est démontrable (*i.e.* a une preuve formelle), alors il est vrai.

Pour un système de preuve ayant ces deux propriétés, on a le théorème suivant :

Théorème II.24 (Gödel)
Certains énoncés vrais de Th($\mathbb{N}, +, \times$) ne sont pas démontrables.

On dit qu'une théorie est complète si l'ensemble des énoncés vrais sont démontrables.

Théorème II.25 (d'incomplétude, Gödel, 1931)
Pour tout système raisonnable formalisant la notion de preuve en théorie des nombres, certains énoncés vrais ne sont pas démontrables.

Le théorème de Gödel affirme donc qu'aucune théorie peut être à la fois consistante et complète.

Ce théorème a mis fin à 50 ans d'effort et de tentatives pour trouver un ensemble d'axiomes permettant de construire un système logique complet utilisable de manière générale par les mathématiques.

Néanmoins, certaines logiques restreintes du premier ordre sont décidables.

Résumé

- un langage est décidable si il existe une MT qui le reconnaisse et s'arrête toujours.
- si un langage n'est pas décidable, alors lui ou son complémentaire n'est pas énumérable.
- Le problème d'acceptation, le problème de l'arrêt est indécidable, le problème de déterminer si un langage est vide … sont indécidables.
- (Rice) toute propriété non triviale sur un langage est indécidable.

- une exécution contrôlée consiste à simuler une MT sur une MT universelle afin de modifier la simulation si certaines conditions sont remplies.
- une fonction calculable est une MT qui s'arrête avec le résultat de la fonction écrit sur la bande.
- une réduction consiste à transformer un problème sur un langage A en un problème sur un langage B (plus complexe et connu),
- le théorème de récursion rend possible à une MT de faire référence à son propre code lors de son exécution.
- Les théories logiques sont en général récursivement énumérables mais pas décidables.
- Pour tout système raisonnable formalisant la notion de preuve en théorie des nombres, certains énoncés vrais ne sont pas démontrables (Gödel).

Chapitre *III*

Complexité des données

1 Ordre asymptotique

Rappelons maintenant la notion de borne asymptotique supérieure.

Soit f et g des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$.

Définition III.1 (grand O)

définition : $f(n) = O(g(n))$ si $\exists c > 0, n_0 \in \mathbb{N}^* \mid \forall n \geq n_0, f(n) \leq c.g(n)$
se lit : g est une borne asymptotique supérieure pour f.
se comprend : f ne grandit pas plus vite que g.

Cette définition s'interprète de la manière suivante :

- On borne supérieurement une fonction $f(n)$ par une fonction $c.g(n)$.
- La constante c ne dépend pas de n. Il n'y a aucune contrainte sur son ordre de grandeur (pourrait être 10^9).
- Le n_0 est le n à partir duquel la relation $f(n) \leq c.g(n)$ est vérifiée.
  On tient compte ainsi de la tendance générale quand n devient assez grand.

Remarques 16:

- attention à l'ordre d'écriture.
- $f(n) = O(g(n)) = O(h(n))$ signifie $f(n) = O(g(n))$ et $g(n) = O(h(n))$
- sert à quantifier la vitesse de croissance d'une fonction le plus souvent croissante.

Exemples 25:

- si $f_1(n) = 3n$ alors $f_1(n) = O(n)$ et $100f_1(n) = O(n)$.
- $f_1(n) = 3n$ et $f_2(n) = 5n$. Et $f_1(n) = O(n)$, $f_2(n) = O(n)$ et $f_1(n) + f_2(n) = O(n)$.
- si $f(n) = 3n^2 + 4n + 5$, on a $3n^2 = O(n^2)$, $4n = O(n)$ et $5 = O(1)$. Alors le terme de plus haut degré domine tous les autres : $O(f(n)) = O(n^2)$.
- $O(3n + 2\log(n)) = O(3n) + O(2\log(n)) = O(n)$: les termes en $\log(n)$ sont dominés par les termes en n.
- $O(4\log(n) + 5) = O(4\log(n)) + O(5) = O(\log(n))$: tout terme constant est dominé par les termes dépendant de n.
- $O(100) = O(1)$

On obtient toujours l'ordre asymptotique à une constante multiplicative près en mettant en facteur le terme dominant et en faisant tendre n vers l'infini. Si on reprend l'exemple ci-dessus : $f(n) = n^2\left(3 + \frac{4}{n} + \frac{5}{n^2}\right)$. Alors $\lim_{n \to \infty} f(n) = 3n^2 = O(n^2)$.

2 Premières approches

2.1 Définition de l'information

Si on considère les chaînes de 24 bits suivantes :

1. 010101010101010101010101
2. 110100110010110100101100
3. 100111011101011100100110

Chacune pourrait représenter 24 tirages d'un pile (0) ou face (1).

Que peut-on dire de ces chaînes ?

- pour la première, c'est 12 répétitions de 01.
- pour la seconde, le $i^{\text{ème}}$ est à 1 si le nombre de 1 dans l'écriture en binaire de i est impair.

i	1	2	3	4	5	6	7	8	9	…
binaire	1	10	11	100	101	110	111	1000	1001	…
résultat	1	1	0	1	0	0	1	1	0	…

- la dernière chaîne n'expose pas de structure visible, et semble aléatoire.

Pourquoi s'intéresser à savoir si une chaîne est aléatoire ou pas ?

- en compression,
  - — une chaîne aléatoire semble ne pas pouvoir être compressée, (= impossibilité de prévoir le symbole suivant ou de trouver une structure).
  - — une chaîne particulière a généralement une représentation plus courte.
- en cryptographie, une chaîne qui contient une information redondante peut être exploitée pour casser le code (cryptographie).
- …

Mais comment savoir si une chaîne est aléatoire ?

2.2 Théorie des probabilités

Si on prend un jeu de pile ou face, la probabilité p de chaque face pour une pièce non truquée est de $\frac{1}{2}$.

Si on considère que les tirages de chaque pile-ou-face sont indépendants, alors toute suite de n tirages est aussi probable que n'importe quelle autre.

Par exemple, la suite de 16 tirages $s_1 = 0000000000000000$ est aussi probable que la suite $s_2 = 1000110111010110$. En effet, dans les deux cas, $\Pr[0] = \Pr[1] = \frac{1}{2}$.

Comme chaque tirage est indépendant, chacune de ces deux chaînes a le même probabilité d'occurrence :

$\Pr[s_1] = \Pr[s_2] = \frac{1}{2^{16}}$.

Avec la théorie classique des probabilités :

- il n'y a **aucun** moyen d'exprimer l'aléa d'une suite individuelle,
- on ne peut exprimer des propriétés que sous la forme de leurs espérances[1] dans le cadre de processus aléatoire.
  Par exemple, on pourrait exprimer les probabilités conditionnelles du bit b_i suivant connaissant le bit b_{i-1} précédent, et vérifier l'espérance de ces probabilités sur la chaîne.
  Plus la chaîne est longue, plus l'erreur sur l'estimation est faible, permettant ainsi de vérifier qu'elle respecte bien les propriétés attendues.

2.3 Approche de la théorie de l'information

En théorie de l'information, si un variable aléatoire X avec n réalisations possibles $\{s_1, \ldots, s_n\}$, chaque symbole s_i avec une probabilité p_i, alors l'entropie de X est :

$$E[X] = -\sum_{i=1}^{n} p_i \log_2 p_i$$

dont l'unité est en bits.

Alors, la théorie de l'information nous dit que toute chaîne produite par une réalisation de X de taille n peut être codée avec $n\,E[X]$ bits.

Cette limite peut effectivement être atteinte en utilisant des techniques de compression pour une chaîne assez longue.

Par exemple, pour $p_0 = \frac{1}{4}$ et $p_1 = 1 - p_0 = \frac{3}{4}$, alors son entropie est :

$$E[X] = -\left(\tfrac{1}{4}\log_2\tfrac{1}{4} + \tfrac{3}{4}\log_2\tfrac{3}{4}\right) \simeq 0.81 \text{ bits}$$

Donc pour $n = 1.000.000$ symboles, toute chaîne pourra être codé avec $n\,E[X] = 811.278$ bits.

Supposons maintenant que la chaîne ne soit constituée que de 1. Sa représentation peut donc être constitué du caractère répété, suivi du nombre de répétitions (codage de type runlength). Le codage de n utilise 20 bits + 1 bit pour le caractère répété, donc 21 bits.

En conséquence, si une chaîne possède une certaine forme de régularité, alors sa représentation peut être bien plus courte que celle obtenue avec la théorie de l'information.

L'entropie représente donc seulement le nombre de bits nécessaires au codage d'une chaîne quelconque dont les bits suivent une loi particulière. Elle n'est donc pas nom plus adaptée aux suites individuelles.

2.4 Caractérisation d'une suite aléatoire infinie

Une première caractérisation (par von Mises, 1919) d'une suite **aléatoire** binaire infinie $b_1 b_2 \ldots$ est la suivante :

1. soit f_n le nombre de 1 parmi les n premiers bits de la suite. Alors $\lim_{n \to \infty} \frac{f_n}{n} = p$.
2. Une règle de sélection de place est une fonction partielle ϕ qui sélectionne un sous-ensemble de bits de la suite. Le bit b_n est sélectionné si et seulement si $\phi(b_1 \ldots b_n) = 1$. La fonction retourne $\perp$ sinon.
   Notons n_i l'indice du $i^{\text{ème}}$ bit sélectionné de cette manière.
3. Alors la suite est aléatoire si pour toute règle de sélection de place, la fonction f_n tends aussi vers p pour toutes les sous-suites $b_{n_1} b_{n_2} \ldots$ obtenues par sélection.

On lui donne le nom de **collectif**.

1. **Exemple :** espérance de la valeur d'une variable aléatoire = sa moyenne.

Exemple 26: Soit la chaîne 10101010101010101 . . .
On a $f_n = 0.5$. Considérons les deux règles de sélection :

1. $\phi_1(b_1 \dots b_n) = 1$ si $n\%2 = 0$ et $\perp$ sinon.
   La sous-suite obtenue est 0000000 . . . et $f_n = 0$.
2. $\phi_2(b_1 \dots b_n) = 1$ si $n\%2 = 1$ et $\perp$ sinon.
   La sous-suite obtenue est 1111111 . . . et $f_n = 1$.

En conséquence, cette chaîne n'est pas aléatoire.

Autre façon de voir : si la suite est le résultat d'un jeu de pile-ou-face, alors un joueur qui miserait toujours sur pile aurait un gain nul, même en choisissant n'importe quelle sous-suite.

Il s'avère que cette définition n'est pas assez rigoureusement définie, et qu'elle induit qu'aucun collectif n'existe (*i.e.* aucune chaîne n'est aléatoire).

Exemples 27:

- Si on définit $\phi_1(b_1 \dots b_n) = 1$ si $b_n = 1$ et $\perp$ sinon, alors $f_n = 1$.
- Si on définit $\phi_1(b_1 \dots b_n) = 1$ si $b_n = 0$ et $\perp$ sinon, alors $f_n = 0$.

Wald (1930) démontre qu'en restreignant les ϕ admissibles à un ensemble dénombrable de fonctions, alors des collectifs existent. Mais, il ne précise pas l'ensemble de fonctions à utiliser.

Church (1940) propose de choisir l'ensemble des fonctions calculables. La définition devient alors rigoureuse. Il montre alors que, avec sa définition, non seulement les suites aléatoires existent, mais qu'elles sont abondantes.

Pour $p = \frac{1}{2}$, elles forment un ensemble de mesure 1. Autrement dit, les chaînes non aléatoires sont extrêmement rares (aussi rare que les nombres entiers parmi les nombres réels).

Remarque 17: nombres normaux

> Toutes les suites de von Mises-Wald-Church sont des nombres normaux. À savoir le nombre de 0 et de 1 sont asymptotiquement égaux (= pour toute suite assez longue, ce qui est induit par $f_n \to \frac{1}{2}$) sur tout bloc de toute longueur (induit par la règle de sélection).
>
> Mais le critère de normalité des suites est insuffisant. Le nombre de Champernowne construit de la manière suivante :
> 12345678910111213141516171819020212223242526 . . .
> est un nombre normal si on considère les répartitions de $\{0, \dots, 9\}$, alors que le $i^{\text{ème}}$ chiffre de cette suite peut facilement être calculé.

Malheureusement, cette définition ne suffit pas. Ville (1940) montre qu'il existe des suites aléatoire au sens de von Mises-Wald-Church, avec $p = \frac{1}{2}$, mais telles que $\frac{f_n}{n} \leq \frac{1}{2}$ pour tout n.

Autrement dit, si ces suites particulières sont le résultat d'un pile-ou-face, alors un joueur misant toujours sur pile accumulerait des gains. Elles ne sont donc pas aléatoires.

Le problème de fond est que des tests effectifs (règles de sélection + f_n) échouent à détecter **toutes les formes de régularité**.

Cela ne signifie pas que la régularité de certaines suites peut être prouvée, mais qu'il est impossible

de calculer effectivement le critère de von Mises-Wald-Church.

La solution est apportée un peu plus tard par Martin-Löf (1966).

Définition III.2 (test de Martin-Löf)

Un test de Martin-Löf est une suite calculable d'ensembles $\{U_n\}_{n\in\mathbb{N}}$ telle que :

- chaque U_n est un ensemble de chaînes finies.
- il existe un algorithme qui liste tous les éléments de U_n (=un énumérateur).
  x représente **toutes les chaînes qui commencent** par x.
- les ensembles U_n sont imbriqués ($U_{n+1} \subseteq U_n$).
- la mesure de U_n est bornée par 2^{-n} (à savoir $\mu(U_n) = \sum_{x\in U_n} 2^{-|x|} \leq 2^{-n}$).

La dernière condition assure que les ensembles U_n deviennent de plus en plus petit. Par exemple, si U_n contient toutes les chaînes de longueur $n+1$ commençant par 0 (donc $U_n = \{0\Sigma^n\}$), alors sa mesure est $\mu(U_n) = 2^n \cdot 2^{-(n+1)} = 2^{-1} = \frac{1}{2}$ car il y a 2^n chaînes de ce type.

Définition III.3 (passage du test de Martin-Löf)

- une suite x échoue au test de Martin-Löf si x est dans l'intersection de tous les U_n (*i.e.* $x \in \bigcap_n U_n$).
- une suite x réussi le test de Martin-Löf si x n'échoue à aucun test.

Si x est dans l'intersection de tous les U_n,

- chaque ensemble U_n contient une version partielle de x.
- tous les bits de x sont calculables.

Autrement dit, **il n'existe aucun programme capable de générer une suite aléatoire infinie**.

En revanche, pour toute suite aléatoire x de taille finie, il existe trivialement un programme qui génère x.

Exemple 28: Soit la chaîne x tirée aléatoirement 100111011101011100100110. Alors le fonction calculable qui génère x est :

```
x='100111011101011100100110'
print(x)
```

2.5 Complexité linéaire d'une suite binaire finie

Signalons pour finir l'algorithme de Massey-Berkelamp.

Les registres à décalage à rétroaction linéaire (Linear Feedback Shift Register = LFSR) constituent un modèle permettant d'estimer la complexité linéaire d'une suite binaire. Un LFSR de longueur N est une suite $\{s_i\}$ définie par :

- ses coefficients $c = (c_0, c_1, \ldots, c_{N-1})$ où $c_i \in \mathbb{B}$,
- son état initial $s = (s_0, s_1, \ldots, s_{N-1})$ où $s_i \in \mathbb{B}$
- les termes suivants $i > N$ sont calculés par la fonction récursive : $s_{i+1} = (c_0.s_i + c_1.s_{i-1} + \ldots + c_{N-1}.s_{i-(N-1)}) \mod 2$

On définit la complexité linéaire d'une suite binaire $(x_0, \ldots, x_{n-1})$ de longueur n comme étant la longueur N du plus petit LFSR telle qu'il existe des coefficients et un état initial permettant de générer la totalité de la suite.

L'algorithme de Massey-Berkelamp est un algorithme simple permettant de déterminer en temps linéaire les coefficients du plus petit LFSR générant une suite binaire finie.

Un LFSR de longueur N peut produire des suites de longueur $2^N - 1$ avant de se répéter.

3 Introduction à la complexité de Kolmogorov

3.1 Définition

Kolmogorov (1960)[2] définit une nouvelle notion de la complexité d'une chaîne.

Définition III.4 (Complexité de Kolmogorov)

La complexité de Kolmogorov d'un objet x (notée $K(x)$) est la longueur du plus petit programme qui produit la sortie x.
Autrement dit :

$$K(x) = \min_{\langle P\rangle \in \Pi} \{|\langle P\rangle| \text{ tel que } U(\langle P\rangle, \varepsilon) = x\}$$

où Π est l'ensemble des programmes P possibles, U est la machine de Turing universelle optimale, et ϵ est la chaîne vide.

On entend par $U(\langle P\rangle, \varepsilon) = x$ que la simulation de l'exécution de P sur la machine de Turing universelle avec pour entrée la chaîne vide s'arrête avec x sur sa bande.

3.2 Invariance

Mais pourquoi donner une définition avec une machine de Turing et non pas mon langage de programmation préféré ?

On rappelle que :

- Tout algorithme peut être exécuté sur une machine de Turing,
- Un langage de programmation est Turing-complet s'il peut simuler une machine de Turing. Notons $U_L(\langle M\rangle)$ le simulateur d'une machine de Turing Universelle avec le langage L.

Mais remarquons que l'inverse est également vrai : une machine de Turing peut simuler tout langage de programmation Turing-complet.

Notons $T_L(\langle P_L\rangle)$ la fonction qui transforme l'algorithme du programme P_L écrit avec langage L en son équivalent sur une machine de Turing.

En conséquence, pour tout couple de langage de programmation (L_1, L_2) Turing-complet, on peut simuler :

- avec le langage L_1 tout programme P_{L_2} avec $U_{L_1}(T_{L_2}(P_{L_2}))$,
- avec le langage L_2 tout programme P_{L_1} avec $U_{L_2}(T_{L_1}(P_{L_1}))$,

2. voir aussi Solomonoff (1960), Chaitin (1969)

Par la suite, on appelle $U_{L_j}(T_{L_i}(P))$ un interpréteur du langage L_i en langage L_j, et on le note $I_{i \to j}$.
La complexité de Kolmogorov dépend du langage utilisé, mais d'une manière bien particulière.
On a le résultat suivant ;

Théorème III.1 (invariance)

Pour tout couple de langages L_1 et L_2, il existe une constante c telle que :
$$\forall x, \left|K_{L_1}(x) - K_{L_2}(x)\right| \leq c$$

A savoir, la constante c ne dépend pas de x.

Démonstration:

Notons $I_{i \to j}$ l'interpréteur qui permet de simuler le langage L_i avec un langage L_j.
Mais, soit $P_{\min}(L_i, x)$ le plus petit programme qui produit x avec le langage L_i.
Avec ces notations, on a : $K_{L_i}(x) = |P_{\min}(L_i, x)|$.
Quelle est la différence de taille entre $K_{L_1}(x)$ et $I_{1 \to 2}(P_{\min}(L_1, x))$?

Dans le langage L_2, $I_{1 \to 2}(P_{\min}(L_1, x))$ est constitué de la taille de l'interpréteur et de la taille de $P_{\min}(L_1, x)$, à savoir $K_{L_1}(x)$.
Donc :
$|I_{1 \to 2}(P_{\min}(L_1, x))| = |I_{1 \to 2}| + K_{L_1}(x)$.
Remarquons que $|I_{1 \to 2}|$ ne dépend pas de x.
Nécessairement, comme $K_{L_i}(x)$ est la taille minimale pour le code qui produit x dans le langage L_i, il borne inférieurement la version interprétée du code minimal pour les autres langages.

- $K_{L_2}(x) \leq |I_{1 \to 2}| + K_{L_1}(x)$,
- $K_{L_1}(x) \leq |I_{2 \to 1}| + K_{L_2}(x)$

D'où, en choisissant $c = \max(|I_{1 \to 2}|, |I_{2 \to 1}|)$,
$$\forall x, \left|K_{L_1}(x) - K_{L_2}(x)\right| \leq c.$$

□

En conséquence, la complexité de Kolmogorov exprimée dans deux langages différents est la même à une constante fixe près.

3.3 Premiers exemples

Le but de cette section est de majorer la complexité de Kolmogorov à partir des codes qu'il est possible d'écrire afin d'obtenir le résultat souhaité, et non de produire le code minimal associé.

Les exemple seront dans un premier temps donné en Python. On affichera la valeur de x plutôt que de la retourner.

Exemple 29: $K(x)$
Soit une chaîne x quelconque (par exemple $x =' 10100111011100110101110 10'$).
Peut-on donner une borne supérieure de son code minimal ?
Considérons le code Python suivant :

```
x='10100111011100110101110 10'
print(x)
```

Il existe peut-être une façon plus courte de construire x, mais en l'absence d'information, on est sûr que pour n'importe quel x :

$\forall x, K(x) \leq |x| + c$
La pire représentation (au sens du code minimal) de x est x lui-même.
La constante c représente la longueur de tous les autres caractères du programme qui ne contiennent pas la représentation de x.

Exemple 30: $K(xx)$ Sachant que $\forall x, K(x) \leq |x| + c$, peut-on obtenir une borne supérieure de $K(xx)$?

```
x='10100111011100110101110 10'
x.append(x)
print(x)
```

On a donc $\forall x, K(xx) \leq |x| + c'$.
La constante c' est différente car le code n'est pas le même, mais la complexité de Kolmogorov de xx est bornée par celle de x.
Autrement dit, xx n'est pas plus complexe que x.

Exemple 31: $K(x^n)$ Et si on concatène n fois x ?

```
x='10100111011100110101110 10'
n=4392
out=''
for i in range(n): out.append(x)
print(out)
```

La partie variable du code est :

- x lui-même,
- n un nombre entier, dont la longueur en binaire est $\log_2(n)$.

Tout le reste est du code constant (on note c sa longueur).
En conséquence,
$K(x^n) \leq |x| + O(\log_2(n)) + c$.
Mais peut-on obtenir une borne encore plus fine ?

Oui ! Parce que la valeur de n peut éventuellement être elle-aussi construite et nécessiter moins de bits que sa propre description.

Exemple 32: $K(x^n)$ **et** $n = 2^p$
On peut obtenir x en doublant de manière itérative p fois sa longueur afin d'obtenir une chaîne de longueur 2^p.

```
# ici le code qui construit x dans a
out=a
p=43
for i in range(p): out.append(out)
print(out)
```

Le code minimal peut maintenant ne stocker que p à la place de 2^p.
D'où : $K(x^n) \leq K(x) + O(\log_2(p))$. Le coût de stockage de la répétition passe donc de $\log_2 2^p = p$ et $\log_2 \log_2 2^p = \log_2 p$.

On peut généraliser le résultat précédent en incluant aussi le coût de construction de p, à savoir : $K(x^n) \leq K(x) + K(p) + c$

3.4 Machine de Turing universelle optimale

Malheureusement, le code Python ne permet pas d'analyser finement ce qu'il se passe en terme de taille du code dès que les exemples deviennent un tout petit peu moins évidents. Par exemple, il est impossible d'évaluer le coût sur la taille du code d'un appel fonctionnel.

Il faut alors s'intéresser à machine de Turing universelle optimale utilisée pour évaluer la taille des codes minimaux.

Mais qu'entend-on par machine de Turing universelle optimale dans le cadre de la complexité de Kolmogorov ?

C'est une MTU U, avec :

- **une bande d'entrée :** bande en lecture seule contenant l'entrée de la MTU.
- **une bande de travail :** bande en lecture/écriture utilisable lors de l'exécution du programme.
- **une bande de sortie :** bande en écriture seule qui contient, lorsque la MTU s'arrête, la valeur retournée.

Ce type de machine est aussi utilisé dans le chapitre 5 (voir la notion de transducteur).

On adopte alors la stratégie suivante pour la MTU optimale U :

- cas d'un seul bloc :
  - un programme M est codé $10\langle M\rangle$: U exécute $\langle M\rangle(\varepsilon)$.
  - une chaîne x est codée $01\langle x\rangle$: U recopie x sur la bande de sortie.
- cas de plusieurs blocs :
  - un bloc suivi d'un autre bloc est codé $\ell(|\langle B_i\rangle|)\langle B_i\rangle$ où $\ell(|\langle B_i\rangle|)$ est le codage de la longueur de B_i (permet de lire les bits du bloc B_i, et de savoir où commence le bloc suivant).
  - le tout dernier bloc est codé sans sa longueur ($\langle B_k\rangle$) car on sait où il commence (fin du bloc précédent) et la fin du bloc est marquée par un blanc.

L'encodage de la longueur utilise le code auto-délimitant suivant : chaque bit est doublé, et la fin est marquée par 01 pour un paramètre, et de 10 pour un programme. Par exemple, pour coder un paramètre de longueur 5, on a : $\ell(5) = \ell(101_2) = 11\,00\,11\,01$. La longueur du codage d'un nombre binaire b est donc $|\ell(b)| = 2|n| + 2$.

Ainsi, si l'encodage du programme P commence par 10, c'est un programme sans paramètre. S'il commence par 01, c'est directement la chaîne de sortie. S'il commence par 00 ou 11, c'est le début du codage de la longueur du bloc qui suit. Ce type de codage est dit préfixe.

Exemples 33: de codage d'entrée pour la MTU optimale U

- $01\langle x\rangle$ = la chaîne x est directement recopiée en sortie.
- $10\langle M\rangle$ = le programme à exécuter est $M(\varepsilon)$
- $\ell\,(|M|)\,\langle M\rangle p$ = le programme à exécuter est $M(p)$
- $\ell\,(|M|)\,\langle M\rangle\langle F\rangle$ = le programme à exécuter est $M(\varepsilon)$ où M fait appel à la fonction F dans son code.

- $\ell(|M|)\langle M\rangle\ell(|F|)\langle F\rangle p$ = le programme à exécuter est $M(p)$ où M fait appel à la fonction F dans son code.
- ...

Un appel de fonction pour une MTU se déroule ainsi :

- On place le curseur en fin de bande de travail avec un marqueur de début de bande.
- On exécute la fonction qui poursuit l'écriture en fin de bande de sortie.
- A la fin de l'exécution, la portion de bande de travail utilisée par la fonction est effacée.

Regardons le codage de l'entrée d'une MTU universelle dans les cas déjà étudiés.

Exercice 21. *Donner le code minimal à placer en entrée de la MTU optimale pour les cas suivants. Puis donner une borne supérieure de $K(x)$.*

1. *$K(x)$ où x est une chaîne quelconque.*
2. *$K(x)$ où on connaît une fonction f_x qui construit x et telle que $|f_x| \ll |x|$.*
3. *$K(xx)$ où x est une chaîne quelconque.*
4. *$K(xx)$ où on connaît une fonction f_x qui construit x et telle que $|f_x| \ll |x|$.*
5. *En déduire une majoration de $K(xx)$ si on connaît le code minimal pour f_x.*
6. *$K(x^n)$ où x est une chaîne quelconque.*
7. *$K(x^n)$ où on connaît une fonction f_x qui construit x (avec $|f_x| \ll |x|$).*
8. *$K(x^n)$ où on connaît une fonction f_x qui construit x (avec $|f_x| \ll |x|$), et une fonction f_n qui construit n (avec $|f_n| \ll |n|$)*
9. *En déduire une majoration de $K(x^n)$ si on connaît le code minimal pour f_x et f_n.*
10. *La formule de Bellard permet de calculer $i^{ème}$ décimale de π pour tout i. Évaluer le complexité de Kolmogorov des n premières décimales de π.*

Exemple 34: $K(xy)$ Il faut écrire un code qui produit la concaténation de x et y. Ce code contient un code f_x qui génère x et le recopie sur la bande de sortie, et f_y qui procède de manière identique pour y.
Un code pour xy pour la MTU optimale est :
$\langle P\rangle = \ell(|\langle M\rangle|)\langle M\rangle\ell(|\langle f_x\rangle|)\langle f_x\rangle\langle f_y\rangle$
où M est le programme (de taille constante) ci-dessous :
$M(\varepsilon) = \begin{array}{|l} f_x(\varepsilon) \\ f_y(\varepsilon) \end{array}$
La taille de M est une constante c (indépendante de $|f_x|$ et de $|f_x|$).
Ainsi, $K(xy) \leq |f_x| + |f_y| + O(\log_2 |\langle f_x\rangle|)$.
Notons que la constant c disparaît dans $O(\log_2 |\langle f_x\rangle|)$
Mais, remarquons que la place de f_x et de f_y peut être intervertie dans le codage. Une écriture alternative de l'encodage de P est :
$\langle P'\rangle = \ell(|\langle M\rangle|)\langle M\rangle\ell\left(\left|\langle f_y\rangle\right|\right)\langle f_y\rangle\langle f_x\rangle$
De façon similaire $K(xy) \leq |f_x| + |f_y| + O(\log_2 |\langle f_y\rangle|)$.

Comme on veut une borne supérieure du code minimal, le meilleur encodage de P est celui tel que f_x et f_y sont les codes minimaux, en codant en premier le code le plus court.
$K(xy) \leq K(x) + K(y) + O(\log_2 \min(K(x), K(y)))$.

On notera que la finesse de la borne précédente est impossible à obtenir avec un code Python, car celui-ci ne permet pas de prendre en compte la longueur effective du codage.

3.5 Information

Cette nouvelle notion de complexité lui permet alors de donner la définition suivante :

Définition III.5 (information)
La quantité d'information dans une chaîne x est $K(x)$.

La théorie de l'information qui donnait une borne minimale pour toutes les chaînes, y compris celles aléatoires.

Nous avons ainsi un moyen de définir la quantité d'information dans une chaîne individuelle.

4 Compression

4.1 Lien avec Kolmogorov

La définition de la complexité de Kolmogorov donne l'impression qu'elle pourrait y avoir un lien avec la compression.

```
y=# une version compressée de x
# code de décompression de y dans a
print(a)
```

Si `y` est la compression maximale de `x`, est-ce que ce code ne serait pas en mesure de produire le code de longueur minimale qui construit `x` ?

Clairement, on a : $K(x) \leq |y| + c$ où c est le code de décompression de y.

Pour répondre à cette question, il est nécessaire de faire une brève introduction à la compression. Les parallèles avec la complexité de Kolmogorov apparaîtront au fur et à mesure.

Définissons tout d'abord les termes du problème.

4.2 Qu'est ce que la compression ?

Qu'est ce que la compression ?
C'est la science de la représentation de l'information sous une forme compacte.

Définition III.6 (compression sans perte)
Soit C une fonction de compression et C^{-1} la fonction de décompression. Alors C est une fonction de compression sans perte si $\forall x \in \mathbb{B}^*$, $C^{-1}(C(x)) = x$.

Dans le cadre de ce cours, on considérera que la compression sans perte.

Qu'est-ce-qu'une forme compacte ?
C'est une forme dans laquelle on a tenté de réduire au maximum la redondance, en la codant d'une manière particulière.

Qu'est ce que l'information ?
Vaste question, mais nous donnons une première réponse qui peut sembler circulaire : c'est ce qui ne peut pas être compressé.
Principe de la compression : Il s'agit pour une suite de bits de départ :

1. de trouver une suite alternative de bits avec un algorithme de compression,
2. tel que cette suite alternative occupe une place de stockage moins importante que la suite initiale,
3. et qu'il soit possible de reconstituer la suite de bits initiale à partir de cette suite alternative avec l'algorithme de décompression associé.

Intuitivement, un algorithme de compression dépend des données à compresser (la redondance dans un texte n'est pas de même nature que celle issue d'une suite de mesures sur un capteur).

4.3 Digression philosophique

En terme métaphysique,

- le langage permet d'exprimer une représentation compressée de la réalité ou de la pensée,
- les mathématiques sont une représentation symbolique compressée d'une pensée logique,
- la physique consiste à "compresser" la réalité en un ensemble d'équation permettant de la décrire,
- ...

Sur le fond, l'intelligence humaine fonctionne :

- en structurant le réel ou les idées sous forme de concept (symboles),
- en compressant le réel ou les idées sous forme d'une description de celui-ci à partir des concepts (suite de symboles),

Autrement dit, nous compressons l'information dans notre cerveau en :

- faisant sens de ce que nous voulons compresser,
- ne retenant et n'organisant que les informations qui nous semblent significatives.

4.4 Définition d'un compresseur

Mais avant de répondre à cette question, nous devons tout d'abord définir ce qu'est un compresseur.

On rappelle les notations utilisées :

$\mathbb{B}^*$ l'ensemble des mots binaires.
$\mathbb{B}^n$ l'ensemble des mots binaires de longueur n exactement.
$|x|$ la longueur du mot x.

Définition III.7 (compresseur)

Un compresseur est une fonction C injective de $\mathbb{B}^* \to \mathbb{B}^*$ qui possède au moins un mot u tel que $|C(u)| \leq |u|$.

Si $|C(u)| < |u|$, la compression est dite stricte.

Remarques 18:

- Un injection est une application telle que $C(u) = C(v) \Rightarrow u = v$.
  Si deux mots différents sont compressés, alors leurs compressions est différentes.
- Autre conséquence, si un mot u peut être compressé par F, alors $C(u)$ est inversible, et $C^{-1}(C(u)) = u$.
  Attention, cela ne signifie pas que pour tout v, $C^{-1}(v)$ existe (*i.e.* toute chaîne aléatoire de bits ne représente pas nécessairement le résultat d'une compression par C).

Cette définition est vraiment peu contraignante : elle demande juste à ce que le compresseur n'augmente pas la longueur d'au moins un mot. On aimerait évidemment qu'un compresseur soit capable d'effectuer une compression stricte sur un sous-ensemble non négligeable de $\mathbb{B}^*$.

4.5 Premières propriétés

Proposition III.1 (nombre de mots inférieur à n bits)

Il y a moins de mots de taille strictement inférieure à n que de mots de taille n.

Démonstration:

L'ensemble $\mathbb{B}^{<n} = \cup_{i=1}^{n-1} \mathbb{B}^i$ des mots de taille strictement inférieure à n a pour cardinal $\#\mathbb{B}^{<n} = \sum_{i=1}^{n-1} \#\mathbb{B}^i$ (car les $\mathbb{B}^i$ sont disjoints).
D'où $\#\mathbb{B}^{<n} = 2 + 2^2 + \cdots + 2^{n-1} = \frac{2^n-1}{2-1} - 1 = 2^n - 2$.
On a donc bien $\#\mathbb{B}^n > \#\mathbb{B}^{<n}$ car $2^n > 2^n - 2$. □

Regardons comment est structuré l'ensemble des mots compressibles et compressés :

- si on note $E = \mathbb{B}^*$ et l'image $F = \mathbb{B}^*$. Le compresseur est une fonction de E dans F.
- comme $E = F$, chacun de ces ensembles a 2 éléments de 1 bit, 4 éléments de 2 bit, 8 éléments de 3 bits, …, 2^n éléments de n bits, …

Supposons que le compresseur C ne compresse qu'un seul élément $u \in E$, alors il y a nécessairement un élément qui est dilaté.

En effet, si un u est compressé ($|C(u)| < |u|$),

- comme il n'y a que $2^{|C(u)|}$ éléments de taille $|C(u)|$, au moins un élément v de $\mathbb{B}^{|C(u)|}$ a été soit compressé, soit dilaté.
- s'il est dilaté, alors on a montré qu'au moins un élément a été dilaté.
- s'il est compressé, alors on reprend récursivement l'argument avec v.

Tout se passe comme un jeu de chaises musicales : le nombre de chaises auxquelles correspondent des mots binaires de n bits ou moins est très limité. Donc, tout élément u telle que $|u| > n$ et $|C(u)| < n$, oblige au moins un élément quelconque de ne plus avoir de place.

Que se passe-t-il si on applique itérative un compresseur sur un mot déjà compressé ?

Proposition III.2

Pour n'importe quel mot de départ sur lequel on applique de manière répétée un compresseur, on se trouve nécessairement dans l'un des deux cas suivants :

1. soit on est dans un cycle (= suite finie de mots qui se répète indéfiniment),
2. soit les mots successifs obtenus deviennent arbitrairement grands.

Démonstration:

Soit C la méthode de compression. Notons $C^n(u)$ l'application de n compressions successives C sur le mot u et $\mathbb{C}(u) = \cup_{k>0} C^k(u)$ l'ensemble contenant toutes ces compressions successives de u.
Deux cas sont alors possibles :

- soit $\mathbb{C}(u)$ est fini, et dans ce cas $C^n(u)$ cycle nécessairement entre les valeurs de $\mathbb{C}(u)$ (puisqu'il y a une infinité dénombrable de $C^n(u)$).
  On notera que si $\exists p$ tel que $C^p(u) = u$, alors on entre dans un cycle de longueur p et $\#\mathbb{C}(u) = p$.
- soit $\mathbb{C}(u)$ n'est pas fini, mais comme tout $\mathbb{B}_{<p}$ est fini, si on compresse plus de $\#\mathbb{B}_{<p}$ fois, alors on trouvera nécessairement un $C^n(u)$ tel que $|C^n(u)| > p$, et ceci pour toute valeur de p.

□

Proposition III.3 (principe des tiroirs de Dirichlet)

Soit E et F deux ensembles **finis** tels que $\#E > \#F$. Alors, il n'existe pas d'application injective de E dans F.

Démonstration:

Si E est un ensemble de n paires de chaussettes, F un ensemble de p tiroirs, avec $n > p$, alors il est impossible de ranger toutes les paires de chaussettes dans les tiroirs si on ne met qu'une seule paires par tiroir.
La démonstration mathématique est équivalente. □

Proposition III.4 (rareté des mots compressibles)

La proportion de mots de taille n qui sont compressibles d'au moins p bits par une méthode de compression C quelconque est strictement inférieur à 1 pour 2^{p-1}.

Démonstration:

Au mieux, C est une injection de $\mathbb{B}^n$ dans $\mathbb{B}^{\leq n-p}$ (compression de mots de taille n vers n'importe quel mot de taille $\leq n-p$).
Par la proposition *III*.1, on sait que $\#\mathbb{B}^{<n-p+1} < 2^{n-p+1}$.
Donc, par le principe des tiroirs de Dirichlet, seuls 2^{n-p+1} peuvent au mieux être compressé parmi les $\#\mathbb{B}^n = 2^n$; soit une fraction de 1 pour $2^n/2^{n-p+1} = 2^{p-1}$. □

Exemple 35: Soit les chaînes de longueur $n = 1024$ bits. La proportion de chaînes de longueur n compressible de 10% est donc de 1 pour $2^{102-1} \simeq 2,5 \times 10^{30}$.

4.6 Conséquences

Les propriétés démontrées précédemment ont les conséquences suivantes :

- l'application itéré d'un compresseur sur un mot d'entrée
  - soit a une borne minimale (cyclage : c'est le plus petit mot du cycle),
  - soit dilate le mot indéfiniment.
- un compresseur universel (à savoir, un compresseur qui compresserait tout mot) n'existe pas. Il y a moins de mots de taille inférieure à n que de mots de taille n.

- le nombre de mots compressibles est très faible.
  La fraction du nombre de mots compressibles de p bits diminue **au moins** exponentiellement en 2^{p-1}.
  En réalité, c'est encore beaucoup moins (la démonstration ne compressait que les mots de n bits).

Comme le nombre de mots compressible de manière appréciable est très faible, est-il intéressant de compresser ?

Exercice 22. *Soit C un algorithme de compression sans perte. Notons $C(x)$ comme le résultat de l'exécution de C sur x et $C^n(x) = \underbrace{C(C(C(\dots C}_{n\,fois}(x))))$ la compression itérée n fois. Par ailleurs, si $C(x) > |x|$, alors C ne compresse pas.*

1. *Qui signifie que C est un algorithme de compression sans perte et interpréter $C(x) > |x|$? On se souviendra de la proposition III.1.*
2. *La compression itérée a-t-elle un intérêt ?*
3. *Donnez une borne inférieure sur la longueur de $C^n(x)$ en fonction de n, x et d'une constante indépendante des deux.*

Évidemment, oui car :

- l'humain produit naturellement de l'information redondante (langue orale, langue écrite, ...)
- le stockage d'information dans la mémoire d'un ordinateur est :
  — guidé avant tout par la rapidité d'accès à l'information (alignement).
  — exceptionnellement peu efficace (exemple : **ü** apparaît dans 4 mots du dictionnaire sur 100000 mots, alors qu'il est codé sur le même nombre de bits que le **e** qui a une fréquence d'apparition d'environ 15%).
  — il existe des ensembles de méthodes de compression différentes adaptées aux types de données que l'on souhaite compresser.
- évidemment, lorsqu'un compresseur C est appliqué sur une chaîne u et que $|C(u)| > |u|$, alors u n'est pas compressé (sortie $0u$ si u n'est pas compressé, et $1u$ si u ne l'est pas. Coût=1 bit).

4.7 Pour aller plus loin

Pour compléter et donner une idée sur la façon dont les principales méthodes de compression fonctionnent :

- le codage entropique qui utilise la fréquence individuelle des symboles (codage de Huffman, codage Arithmétique, ...),
- le codage par dictionnaire qui construit un dictionnaire de motifs (LZ77, LZ78, LZW, ...),
- le codage prédictif qui utilise les probabilités conditionnelles pour prédire les symboles suivants à partir du contexte (Partial Prédiction Matching, Dynamic Markov Coding, ...)

Un logiciel de compression utilise un mélange de ces techniques compresser des données. La méthode utilisée dépend également du type des données.

Deux références pour aller plus loin :

- **Data compression : the complete reference** de David Salomon,
- **Théorie des codes : compression, cryptage, correction** de Jean-Guillaume Dumas.

5 Propriété de la complexité de Kolmogorov

5.1 Calculabilité

Théorème III.2 (calculabilité de $K(x)$)

$K(x)$ n'est pas calculable.

Démonstration: par l'absurde

Supposons que $K(x)$ soit calculable.
Considérons la machine de Turing suivante :

$M(\varepsilon) =$
```
POUR tout x ∈ Σ*
    SI K(x) > |⟨M⟩|
    ALORS RETOURNER x
```

Il existe des programmes minimaux de toute taille (voir le lemme III.2 plus loin).
Donc, on finit toujours par trouver un x vérifiant $K(x) > |\langle M \rangle|$.
Mais M produit la sortie x, et sa taille est inférieure à $K(x)$.
Donc $K(x)$ ne produit pas un résultat correct, et elle n'est pas calculable. □

Une preuve similaire a été effectuée pour le problème des MTs minimales en utilisant le théorème de récursion.

5.2 Incompressibilité

Proposition III.5

Considérons l'ensemble des chaînes binaires de longueur n.
La fraction des chaînes x de longueur n telles que $K(x) < n - k$ est inférieure à 2^k.

Démonstration:

On sait que $K(x) \leq |x| + c$ où $c \ll |x|$. Autrement dit, le code minimal pour produire x n'est pas plus grand que la longueur de x, à une constante c près dépendante du langage et négligeable devant x quand celui-ci est grand.
Dans cette démonstration, $\mathbb{B}^n$ est l'ensemble des codes minimaux de longueur n. Comme déjà démontré dans la partie compression (voir la proposition III.1), $\#\mathbb{B}^{<n-k} < 2^{n-k}$.
Cela signifie qu'il y a moins de 2^{n-k} codes de longueur $n - k$.
Or, il y a 2^n chaînes de longueur n.
En conséquence, parmi ces chaînes, seules 2^{n-k} sont susceptibles d'avoir un code minimal de longueur inférieure à $n - k$ (principes des tiroir de Dirichlet, proposition III.3).
Donc, la fraction des chaînes de longueur n telles que $K(x) < n - k$ est de $2^n/2^{n-k} = 2^k$. □

Lemme III.1

Pour tout $n > 1$, il existe une chaîne incompressible de longueur n.

Démonstration:

On a vu que $\mathbb{B}^{<n} < \mathbb{B}^n$ (il existe moins de mots de longueurs strictement inférieures à n que de mots de longueurs n).
En conséquence, il existe moins de codes minimaux de longueurs strictement inférieure à n qui de code chaîne de longueur n

Par le principes des tiroirs de Dirichlet (proposition III.3), il existe des chaînes dont le code minimal est de longueur n.
Ceci est vrai pour tout $n > 1$. □

Il existe donc des chaînes incompressibles de toutes tailles.
Puis le lemme utilisé dans la démonstration que la complexité de Kolmogorov n'est pas calculable.

Lemme III.2
Il existe une constante c, telle que $\forall n \geq c$, il existe un code minimal de longueur n.

Démonstration:
On a vu dans l'introduction sur la complexité de Kolmogorov qui la machine de Turing universelle optimale U associée était en mesure de produire x avec l'appel $U(00x)$.
En conséquence, si x est une chaîne non compressible de longueur $K(x) = |x| + 2$.
En prenant $c = 2$, n'importe quelle chaîne incompressible x de longueur $n - 2$ a bien un code minimal de longueur n. □

Ce lemme démontre donc qu'il existe donc des codes minimaux de toutes tailles supérieure à 2.

Exercice 23.

1. *Soit x et y deux chaînes incompressibles. Montrer que* $K(x) + K(y) > K(xy) + O(1)$.
2. *Soit les chaînes de la forme* $s = 0^n y$ *construite avec n 0 consécutifs suivis d'une chaîne y. Montrer que l'on peut choisir un* $c = O(1)$ *tel que qu'il existe des chaînes de la forme s qui sont incompressible.*
3. *En déduire que pour tout c, il existe des chaînes x et y pour lesquelles* $K(xy) > K(x) + K(y) + c$.

Exercice 24. *Montrer que l'ensemble des chaînes incompressibles ne contient aucun sous-ensemble infini récursivement énumérable.*
Ligne de preuve : *supposer qu'un tel ensemble A existe, donc il existe une MT M qui reconnaît A (attention M peut boucler). Construire alors un énumérateur* $N(k)$ *qui retourne la première chaîne de A de longueur au moins k (attention, il doit être décidable). Calculer la* $K(N(k))$ *et conclure.*

Théorème III.3 (indécidabilité de la compressibilité)
Soit l'ensemble des chaînes incompressibles :
$$L = \{x \in \Sigma^* \mid K(x) \geq |x|\}$$
L est indécidable.

Démonstration: par l'absurde
Supposons que L soit décidable. Alors il existe un décideur R qui décide L (*i.e.* $R(x)$ accepte si $x \in L$ et rejette si $x \notin L$).
Mais si un tel décideur existe, alors $K(x)$ devient calculable, à une constante près, pour tous les x tels que $R(x)$ accepte.
Notons qu'on peut avoir une idée précise de cette constante, puisque le code minimal associé à une chaîne incompressible ne fait que la retourner.
Or $K(x)$ n'est pas calculable (voir théorème III.2). Donc, R n'existe pas, et L n'est pas décidable. □

Comme l'ensemble des chaînes compressible est le complémentaire de L, il n'est pas décidable non plus.

5.3 Lien avec la notion d'aléatoire

Définition III.8 (Aléa de Kolmogorov)

Les chaînes x telles que $K(x) \geq |x|$ sont aléatoires au sens de Kolmogorov.

Pour résumer,

- chaîne incompressible = chaîne aléatoire.
- chaîne compressible = chaîne non aléatoire.

Remarque 19:

Pour tout x, le code minimal est incompressible, sinon il serait possible de trouver un code plus court en le compressant.

Comme l'information que nous générons est naturellement redondante, une chaîne incompressible est donc, soit aléatoire, soit de l'information compressée.

Remarque 20:

Soit une suite infinie $s = s_1 s_2 \ldots$. On note $s^n = s_1 \ldots s_n$ les n premiers symboles de cette suite.

Les deux notions suivantes sont **équivalentes** :

- une chaîne aléatoire au sens de Martin-Löf : à savoir une suite infinie s qui passe tous les tests de Martin-Löf,
- une chaîne aléatoire qui a la complexité de Kolmogorov [3] maximale pour tous les segments initiaux finis ($\forall n, \exists c, K(s^n) \geq n - c$).

5.4 Comportement de la complexité de Kolmogorov

Quel est le comportement de $K(x)$?
Prenons $x \in \mathbb{N}$ (ce qui est équivalent à considérer son écriture en binaire).

- $K(x)$ est bornée inférieurement par une fonction croissante
  exemple : $K(1^n) \geq \log_2(n)$
- $K(x) \simeq \log_2(x)$ presque partout
  car la plupart des chaînes sont incompressibles
- $K(x)$ décroche infiniment souvent
  si x est compressible, alors $K(x) \simeq K(xx) \simeq K(xxx) \simeq \ldots$, $K(x) \simeq K(x^R)$, ... à savoir pour toutes transformations de x par un code de taille fixe.
- $K(x)$ a une certaine forme de continuité.
  $\forall x, |K(x) - K(x \pm 1)| < c$ (une fois x obtenu, il est facile de lui ajouter une constante avec un code de taille fixe).

5.5 Autre applications

La complexité de Kolmogorov permet dans certains cas des démonstrations surprenantes de résultats bien connus. Notamment,

- la non régularité de $0^n 1^n$ (normalement avec lemme de l'étoile),
- le théorème d'Euclide (nombre infini de nombres premiers),
- ...

Proposition III.6

$L = \{0^n 1^n \mid n \in \mathbb{N}\}$ n'est pas régulier.

Démonstration:

Supposons que L soit régulier. Alors il existe un ADF A qui le décide.
En conséquence, pour toute chaîne $x \in L$, $K(x) \leq |\langle A \rangle| + c$.
À savoir, toute expression régulière peut être décidée avec un code de taille finie, et sa complexité de Kolmogorov peut être bornée par la longueur description de A et du code qui simule A. La somme de ces deux longueur **est une constante**.
Supposons maintenant que $x = 0^n 1^n$. Alors nécessairement, la valeur de n doit être calculée par le code minimal (comptage du nombre de 0, et vérification du nombre de 1).
Or, le stockage de n dans ce code minimal occupe $\log_2 n$ bits, ce qui implique qu'il vérifie : $K(0^n 1^n) \geq \log_2 n$.
Donc, $\log_2 n \leq K(x) \leq |\langle A \rangle| + c$
Mais on peut faire en sorte que $\log_2 n > |\langle A \rangle| + c$, ce qui contredit l'hypothèse que L est régulier. □

Théorème III.4 (Euclide)

Il existe une infinité de nombres premiers.

Démonstration: (par l'absurde)

Supposons qu'il y ait un nombre k fini de nombres premiers.
Soit x une chaîne incompressible, et n la représentation entière de x.
Alors il existe $\{e_1, \ldots, e_k\}$ tels que $n = p_1^{e_1} \ldots p_k^{e_k}$ (sa décomposition en facteurs premiers).
Nécessairement $e_i \leq \log_2 n$. Donc $|e_i| \leq \log_2 \log_2 n$ (par exemple, si $n = 2^k$ alors $p_1 = 2$, $e_1 = \log_2 n = k$, et $|e_1| = \log_2 \log 2^k = \log k$). L'inégalité est stricte pour $p_i > 2$.
Son code minimal contient les k nombres premiers (constants), et la représentation de tout n a besoin au plus de k entiers (les e_k), chacun étant inférieur à $\log_2 \log_2 n$.
Donc, $K(x) < k \log_2 \log_2 n + c$.
Ceci contredit le fait que x soit incompressible. Donc, l'hypothèse est fausse et le nombre de nombres premiers n'est pas fini. □

Chapitre IV

Retour sur les machines de Turing

Dans ce chapitre, nous revenons sur le modèle de la machine de Turing. Il nécessite la compréhension :

- des automates finis, déterministe et non déterministe,
- des automates à pile

pour lesquels nous ferons quelques rappels.

Ces concepts vu en théorie du langage étant maintenant considérés comme acquis, nous allons donc étudier :

- le modèle abstrait de la machine de Turing,
- différentes variations,
- la simulation d'un ordinateur minimaliste sur une machine de Turing.

A la fin de ce chapitre, nous disposerons donc du détail du fonctionnement de la machine de Turing, ce qui nous permettra d'étudier avec la complexité avec la finesse nécessaire.

1 Modèles de machine étudiés en théorie des langages

Nous allons maintenant considérer des premiers modèles théoriques de machines :

- les automates déterministes
- les automates non déterministes finis
- les automates à pile
- les grammaires algébriques

qui sont des modèles élémentaires de machines utilisé en théorie des langages, et en informatiques pour de nombreuses tâches.

Nous verrons pourquoi ces langages ont des capacités bien trop limitées d'où la nécessité de disposer d'un modèle plus évolué.

Il convient néanmoins de ne pas négliger ces premiers modèles formels car leurs principes de bases sont réutilisés dans les machines de Turing.

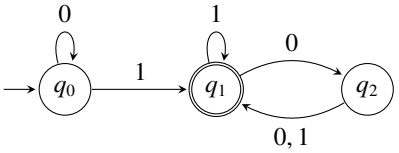
REMARQUES 21:

- Ces modèles de machine ne sont pas des machines programmables.
- Chaque machine n'est définie pour ne reconnaître qu'un seul langage.
- Ils sont donc l'équivalent d'un processeur spécialisé qui ne serait câblé que pour réaliser une seule tache.

1.1 Automate fini

Un automate déterministe fini (ADF) est une machine à état qui permet de résoudre un problème de décision pour un langage L.

Exemple 36:



Soit un alphabet $\Sigma = \{0, 1\}$. Pour décider un mot $w = s_1 \dots s_n \in \Sigma^*$ (où $n = |w|$ et $\forall i, s_i \in \Sigma$) avec cet ADF :

1. se placer sur le premier symbole s_1 de w ($i = 0$), et partir de l'état de départ q_1 (pointé par une flèche).
2. prendre la transition associée au symbole courant s_i et aller dans l'état correspond, puis passer au symbole suivant ($i = i + 1$).
3. recommencer (2) jusqu'à lire la totalité des symboles.
4. si l'état courant est un état acceptant (cercle double) alors accepter le mot, sinon le rejeter.

Remarque 22:

cet automate s'arrête toujours (longueur du mot d'entrée = nombre de transitions).

Il est déterministe :

- un symbole = une seule transition possible.
  pour chaque état, il y a une seule transition pour chaque symbole de l'alphabet.
- un mot = un seul chemin dans le graphe.

Définition IV.1 (formelle d'un automate déterministe fini)

Un automate déterministe fini est un 5-uple $(Q, \Sigma, \delta, q_0, F)$ où :

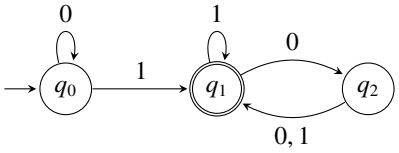
- Q est un ensemble fini d'états (ensemble des états).
- Σ est un ensemble fini de symboles (alphabet).
- $\delta : Q \times \Sigma \to Q$ est la fonction de transition.
- $q_0 \in Q$ est l'état de départ.
- $F \subseteq Q$ est l'ensemble des états acceptants.

Ce 5-uple est une description complète de l'ADF.

Remarques 23:

- La fonction de transition : $q' = \delta(q, s)$ est la fonction qui, lorsqu'on est dans l'état q et sur le symbole s, indique dans quel état q' on se retrouve.
- Si $F = \emptyset$ alors l'ADF n'accepte aucun mot ($L = \emptyset$). Si $F = Q$ alors l'ADF accepte tous les mots ($L = \Sigma^*$).

Exemple 37: de description d'un ADF



Cet ADF M a pour caractéristique :

- l'ensemble des états est $Q = \{q_1, q_2, q_3\}$
- l'alphabet est $\Sigma = \{0, 1\}$
- la fonction de transition $\delta : Q \times \Sigma \to Q$ (ou table de transition) est :

Q \ Σ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

δ est une fonction de transition déterministe : pour chaque état $q \in Q$, il n'y a qu'une seule transition pour chaque symbole $s \in \Sigma$.

- q_1 est l'état de départ,
- l'ensemble des états acceptants est $F = \{q_2\}$.

Sa description formelle est $(Q, \Sigma, \delta, q_1, F)$.

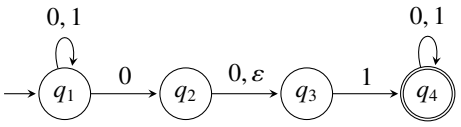
1.2 Automates non déterministes

Un automate non-déterministe fini (ANF) est un ADF modifié pour lequel, depuis un état, on modifie les transitions possibles de la manière suivante :

- un même symbole peut être associé à plusieurs transitions (voir aucun).
- un symbole supplémentaire (noté ϵ) permet une transition sans lecture de symbole sur la chaîne d'entrée.

L'automate n'est alors plus déterministe car plusieurs choix peuvent être possibles lors d'une transition, ou aucun !

Exemple 38:



- transitions supplémentaires : q_1 a deux transitions associées à 1.
- transition manquante : q_2 n'a pas de transition pour 1 (aussi q_3 pour 0).
- transition ϵ : la transition entre q_2 et q_3 peut se faire sans lecture de symbole.

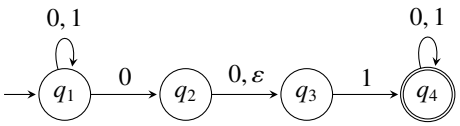
Un ANF s'exécute de la façon suivante :

- si plusieurs transitions sont possibles :
  - — la machine se découple en des copies multiples.
  - — chaque copie (= une branche d'exécution) suit une possibilité.
  - — l'ensemble des branches continuent et suivent toutes les possibilités.
- une transition ϵ correspond à une transition sans lecture de symbole.
- si aucune transition n'est possible
  la branche d'exécution ne peut pas se poursuivre : elle s'arrête (=rejet)

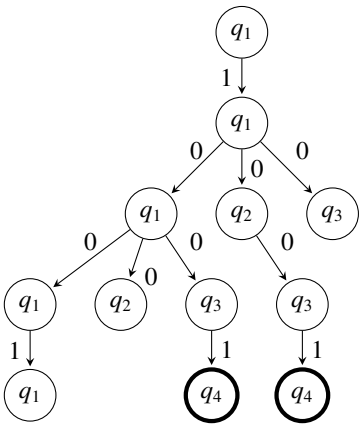
La chaîne d'entrée est :

- **acceptée** si au moins l'une de ses branches d'exécution accepte (= est dans un état acceptant à la fin de la lecture de la chaîne).
- **rejetée** si toutes les branches d'exécution ont rejeté ou se sont arrêtées avant la fin de la chaîne.

Exemple 39: ANF :



Exécution sur le mot 1001 **:**



Note : pour une transition ϵ, $q_1 \xrightarrow{0} q_2 \xrightarrow{\epsilon} q_3$ devient $q_1 \xrightarrow{0} q_3$.

Définition IV.2 (Fonction de transition)

La fonction de transition δ est définie de $Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$
où Q est l'ensemble des états, Σ l'alphabet du langage et $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

Exemple 40: On définit l'ensemble des états $Q = \{q_1, q_2, q_3\}$ et l'alphabet $\Sigma = \{0, 1\}$.

- $\mathcal{P}(Q)$ contient bien l'ensemble de toutes les transitions possibles entre un état et tous les autres (voir exemple ci-avant).
- Σ_ϵ est l'ensemble des symboles provoquant une transition (= symboles de l'alphabet + transition ϵ).
- $\delta(q, s)$ est définit pour tout $q \in \mathcal{P}(Q)$ et $s \in \Sigma_\epsilon$ (i.e. sur le produit cartésien $\mathcal{P}(Q) \times \Sigma_\epsilon$).

Exemples de transitions :

- $\delta(q_1, \epsilon) = \{q_2\}$: transition simple $q_1 \xrightarrow{\epsilon} q_2$
- $\delta(q_2, 0) = \{q_1, q_3\}$: transitions multiples $q_3 \xleftarrow{0} q_2 \xrightarrow{0} q_1$
- $\delta(q_1, 1) = \{\}$: pas de transition pour le symbole 1 depuis q_1

Définition IV.3 (description formelle d'un ANF)

Un automate non-déterministe fini est un 5-uple $(Q, \Sigma, \delta, q_0, F)$ où :

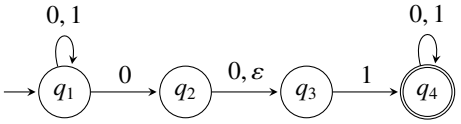
- Q est un ensemble fini d'états(ensemble des états).
- Σ est un ensemble fini de symboles (alphabet).
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ est la fonction de transition.
- $q_0 \in Q$ est l'état de départ.
- $F \subseteq Q$ est l'ensemble des états acceptants.

Le 5-uple $(Q, \Sigma, \delta, q_0, F)$ définit complètement l'automate non-déterministe fini.

Remarque : par rapport à la définition formelle d'un ADF seule la définition de la fonction de transition indique que l'on a affaire à un ANF.

Exemple 41: description formelle d'un ANF

Soit l'ANF suivant :



$N = (Q, \Sigma, \delta, q_1, F)$ où :

- états : $Q = \{q_1, q_2, q_3, q_4\}$
- langage : $\Sigma = \{0, 1\}$
- fonction de transition : $\delta(q, s)$

Q \ Σ	0	1	ϵ
q_1	$\{q_1, q_2\}$	$\{q_1\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

  δ est une fonction de transition non-déterministe : pour chaque état $q \in Q$, il n'y a qu'une ou plusieurs transitions pour chaque symbole $s \in \Sigma$, voire **aucune** ($\emptyset$ dans la table ci-contre).
- état de départ : q_1
- états acceptants : $F = \{q_4\}$

1.3 Automates à pile

Un Automate à Pile (AP) est un Automate Non-déterministe Fini (ANF) auquel a été ajouté pile LIFO [1] (=FILO) de taille infinie.

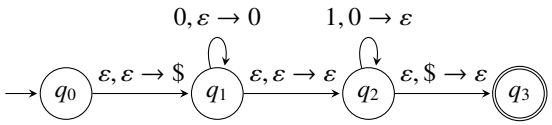
A chaque transition, un automate à pile :

- lit un caractère de la chaîne d'entrée (sauf s'il s'agit d'une transition ϵ)
- dépile (éventuellement) le symbole au sommet de la pile.
- empile (éventuellement) un symbole sur la pile.

La notation $a, b \to c$ signifie que la machine lit a depuis l'entrée, dépile b et pousse c sur la pile. ε signifie une absence d'opération.

1. Last In Firt Out.

Exemple 42:



Fonctionnement :

tant que l'on lit un 0 : le pousser sur la pile.

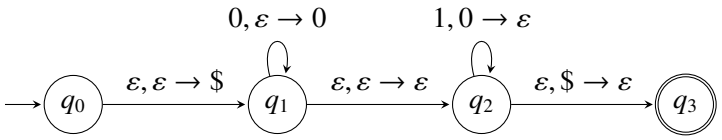
tant que l'on lit un 1 : dépiler un 0 de la pile, et s'il n'y en a pas, rejeter.

si la pile est vide, alors accepter, sinon rejeter.

Le langage reconnu est $L = \{0^n 1^n \mid n \geq 0\}$.

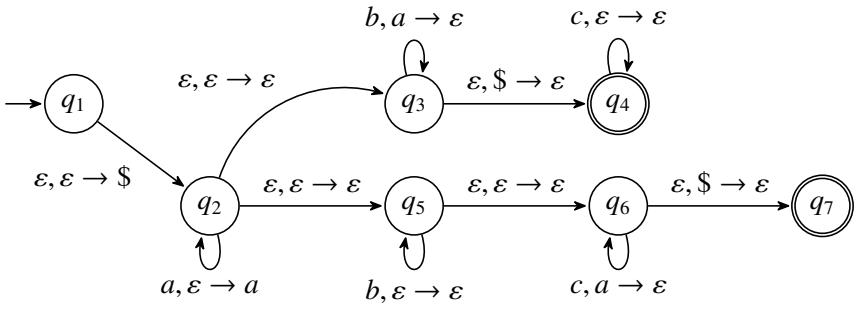
Remarques 24:

- on ne peut pas tester si la pile est vide.
  Une solution consiste à utiliser un symbole particulier (par exemple \$).
  On commence avant le début de la lecture par placer un \$ au sommet de la pile (transition de la forme $\epsilon, \epsilon \to \$$).
  Si on lit le symbole \$ sur le sommet de la pile, alors la pile est vide.
- on ne peut pas tester si la chaîne d'entrée a été traitée.
  Comme pour un ADF, la chaîne d'entrée est acceptée si après le dernier symbole lu, on se trouve dans un état acceptant.
- Dans l'exemple précédent (reproduit ci-dessous), le non-déterminisme est utilisé :
  — pour empiler autant de 0 à la lecture des 0s qu'il sera nécessaire à d'en dépiler à la lecture des 1,
  — pour s'assurer que la pile est vide lorsque l'on a fini de lire tous les symboles

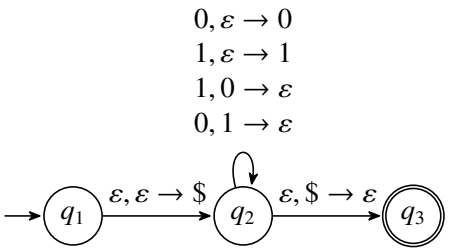


Exemple 43: $L = \left\{a^i b^j c^k \mid i, j, k \geq 0 \text{ et } i = j \text{ ou } i = k\right\}$
Comment construire l'automate à pile qui reconnaît ce langage ?

- pousser les a lus sur la pile.
- deviner si le nombre de a doit être comparé avec le nombre de b ou de c (fait par une transition non déterministe).



Exemple 44: $L = \{w \mid \#_0 w = \#_1 w\}$ ou $\#_i w$ = nombre de i dans la chaîne w.
Comment construire l'automate à pile qui reconnaît ce langage ?

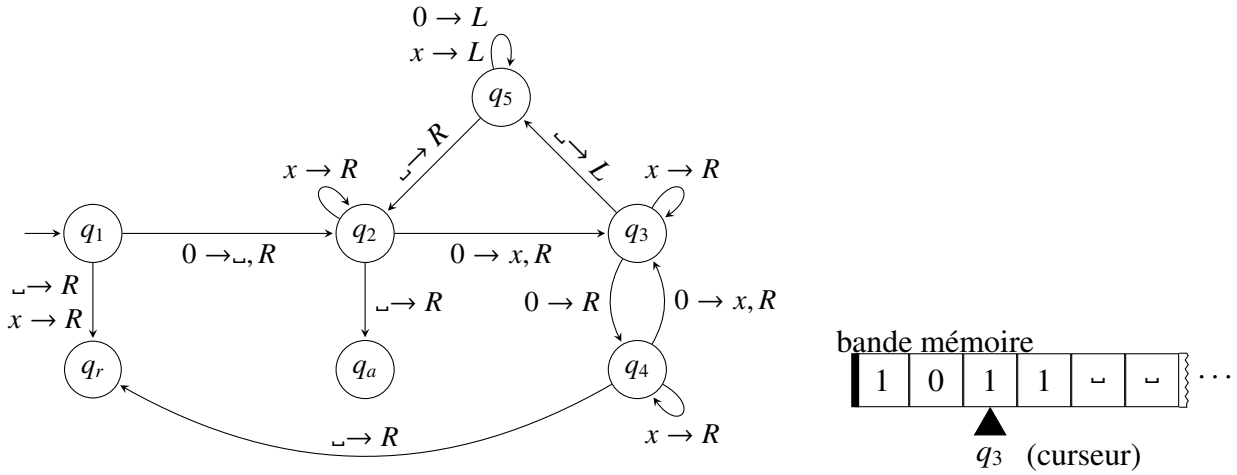


- si le sommet de la pile est 0 (= il n'y a que des 0 dans la pile)
  - si je lis un 1, alors je dépile le 0.
  - si je lis un 0, alors j'empile le 0.
- si le sommet de la pile est 1 (= il n'y a que des 1 dans la pile)
  - si je lis un 0, alors je dépile le 1.
  - si je lis un 1, alors j'empile le 1.
- à savoir, on dépile toujours autant de 0 (resp. 1) qu'on lit de 1 (resp. 0).
- si la pile est vide, le symbole suivant détermine celui qui sera empilé.
- le non déterminisme permet de trouver le bon chemin.

2 Machine de Turing

Une machine de Turing (MT) est un modèle abstrait de machine :

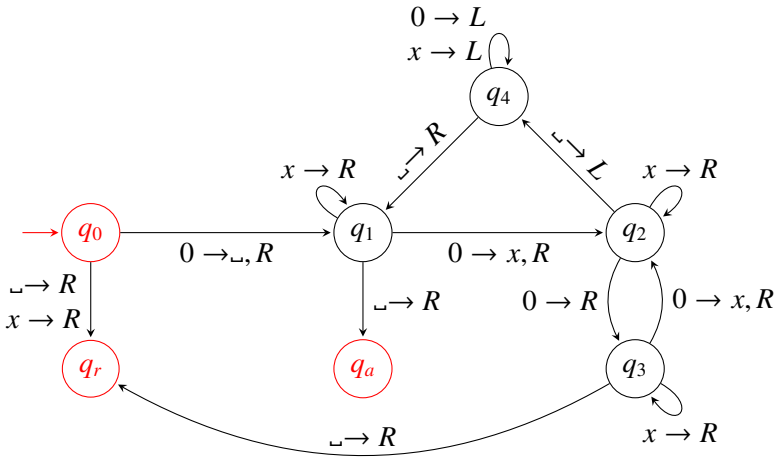
- qui prend en paramètre un mot d'entrée w,
- avec une bande mémoire (finie à gauche) permettant d'y stocker des symboles, et initialisée avec le mot d'entré,
- un curseur permet de se déplacer sur la bande (dans les deux sens), afin de lire ou écrire le symbole sous le curseur.
- le curseur est contrôlé par une machine à état déterministe,
- les transitions s'exécutent jusqu'à accepter ou à rejeter.



Une MT a donc pour but de résoudre des **problèmes de décision** (*i.e.* de décider des langages).

Une MT dispose de **trois états particuliers** :

- un **état de départ** (ci-dessous q_0) qui est le point de départ de l'exécution du graphe d'état, marquée par une petite flèche entrante.
- un **état acceptant** q_A : lorsqu'il est atteint, la machine s'arrête immédiatement en acceptant.
- un **état rejetant** q_R : lorsqu'il est atteint, la machine s'arrête immédiatement en rejetant.



Remarques 25:

- Les états q_A et q_R n'ont que des connexions entrantes (puisqu'elles provoquent l'arrêt de la machine),
- Si la machine n'entre jamais dans l'un des états q_A ou q_R, **elle boucle**.

Une MT dispose de **deux alphabets différents** :

- l'alphabet du langage Σ auquel appartient le mot d'entrée w (*i.e.* $w \in \Sigma^*$).
- l'alphabet de bande Γ contient l'ensemble des symboles qui peuvent être trouvé sur la bande.

Γ contient évidemment Σ mais également :

- le caractère blanc $\sqcup$ qui est le caractère spécifique avec lequel l'ensemble des cellules de la bande est initialisé.

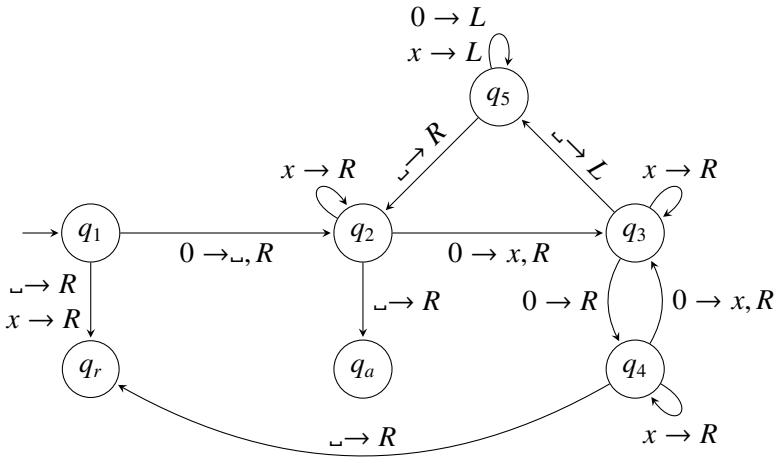
- tout autre caractères spécifiques auquel on affecte un rôle ou un sens particulier lors de l'exécution de la machine, par exemple :
  - — des caractères spéciaux :
    # pour indiquer le début de la bande,
    `x` pour barrer un symbole déjà traité,
  - — des marqueurs : pour un symbole 0, on peut le barrer Ø, le marquer de différente manière $\bar{0}$, $\dot{0}$, $\tilde{0}$, $\hat{0}$ afin d'indiquer un sens particulier tout en conservant le symbole, et éventuellement rétablir l'original.

Au minimum, on a $\Gamma = \Sigma \cup \{\sqcup\}$.
A une étape de l'exécution, l'état d'une MT est caractérisé par :

- l'état courant q_i dans lequel on se trouve sur le graphe d'état,
- l'endroit de la bande où est placé le curseur de bande,
- le symbole contenu dans la cellule pointée par le curseur de bande,

La transition $a \to b, L$ signifie : si la cellule pointée par le curseur contient le symbole a, alors écrire un b dans la cellule, et déplacer le curseur dans la direction L (gauche).

Une machine de Turing est **déterministe**, à savoir que **pour chaque état de la machine**, **pour chaque symbole de bande**, il n'y a qu'une seule transition possible depuis cet état.

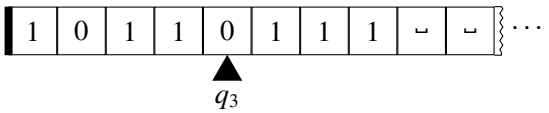


Dans l'exemple précédent, les symboles de l'alphabet sont $\Sigma = \{0\}$, les symboles de bande sont $\Gamma = \{0, x, \}$. Depuis chaque état, il y a une seule transition pour chaque état de Γ.

Définition IV.4 (Configuration d'une machine de Turing)

Notation qui contient l'**état**, la **position du pointeur**, le **contenu de la bande** d'une MT à un instant donné.

Exemple 45: Une configuration $1011q_30111$ signifie que la MT est dans l'état suivant :



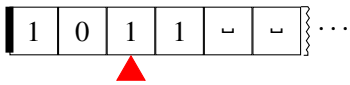
à savoir, la machine est dans l'état q_3, la bande contient 10110111, et le pointeur se trouve sur le cinquième caractère de la bande (= sur le symbole qui suit l'état dans la configuration).

Comme une machine de Turing est déterministe, une configuration décrit totalement l'état d'une MT ainsi que son évolution future.

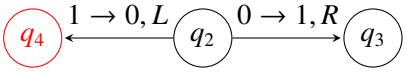
Exemple 46: On est dans l'état q_2 et le symbole dans la cellule sous le curseur de lecture est 1.

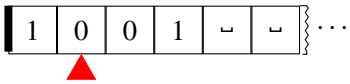
extrait du graphe de la machine de Turing

bande mémoire



On choisit donc la transition $1 \to 0, L$: on écrit 0, on déplace le curseur à gauche. et on passe dans l'état q_4.



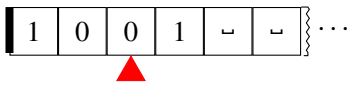


La machine passe donc de la configuration $10q_211$ à la configuration $1q_4001$.

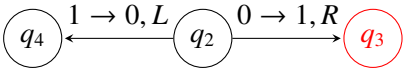
Exemple 47: On est dans l'état q_2 et le symbole dans la cellule sous le curseur de lecture est 0.

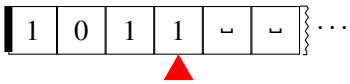
extrait du graphe de la machine de Turing

bande mémoire



On choisit donc la transition $0 \to 1, R$: on écrit 1, on déplace le curseur à droite. et on passe dans l'état q_3.





La machine passe donc de la configuration $10q_201$ à la configuration $101q_31$.

Plusieurs types de transitions sont possibles :

- Les transitions en lecture :
  $0 \to L$ = si le symbole courant est un 0, déplacer le curseur à gauche.
  $1 \to R$ = si le symbole courant est un 1, déplacer le curseur à droite.
- Les transitions en écriture :
  $0 \to 1, L$ = si le symbole courant est un 0, écrire un 1 à la place sur la bande, et déplacer le curseur à gauche.
  $1 \to 1, R$ = si le symbole courant est un 1, écrire un 1 à la place sur la bande (donc ne change rien), et déplacer le curseur à droite.

On utilise aussi les notations suivantes :

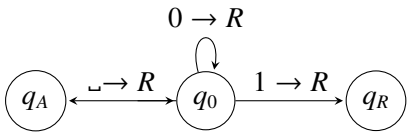
- $0, 1 \to L$ = si le symbole courant est 0 ou 1, déplacer le curseur à gauche.
- $\Sigma \to R$ = pour tout symbole de l'alphabet Σ, déplacer le curseur à droite.

Si on se déplace à gauche alors que le curseur est déjà au début de la bande, alors le curseur ne bouge pas.

Soit M une machine de Turing, et L le langage accepté par la machine. Le fonctionnement de M sur un mot d'entrée w est le suivant :

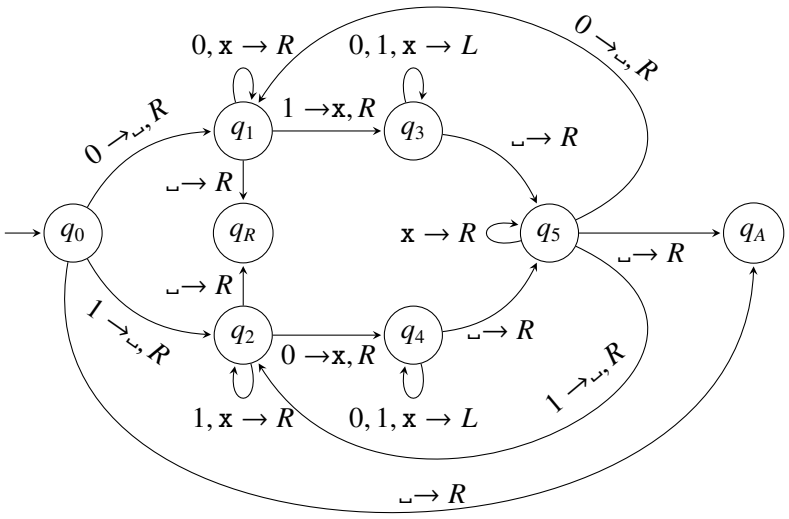
- **initialisation :**
  - placer le mot w au début de la bande mémoire,
  - placer le curseur sur la première cellule de la bande,
  - initialiser l'état courant à l'état de départ q_0 du graphe d'état.
- **exécution :**
  - en fonction de l'état courant et du symbole sous le curseur, effectuer la transition associée (écrire le nouveau symbole dans la cellule, déplacer le curseur, et passer à l'état suivant).
  - si le nouvel état est q_A, alors M s'arrête en acceptant ($w \in L$).
    si le nouvel état est q_R, alors M s'arrête en rejetant ($w \notin L$).
  - pour tous les autres états, recommencer avec ce nouvel état.

Exemple 48: langage qui ne contient que des 0



Remarquez que la machine s'arrête dès que le premier 1 est rencontré, sans lire le reste de la chaîne.

Exemple 49: langage qui contient autant de 0 que de 1

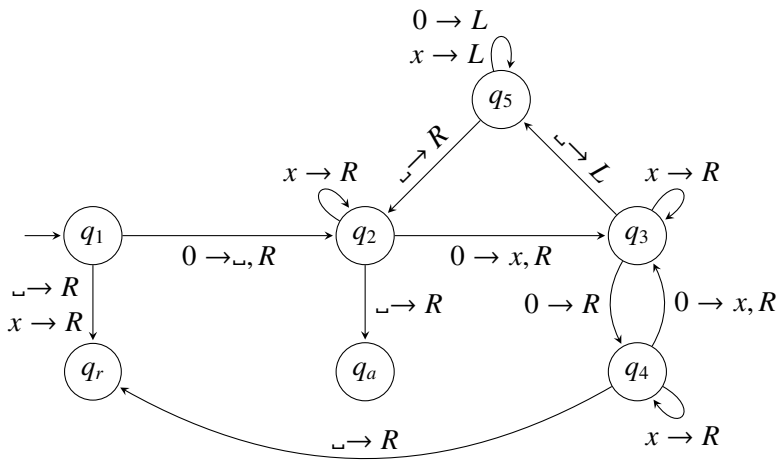


Exemple 50: Soit la machine de Turing M qui décide $L = \{0^{2^n} \mid n \geq 0\}$.

Idée de fonctionnement d'une MT qui accepte le mot w que s'il appartient à L :

1. parcourir la bande de gauche à droite, en barrant un 0 sur deux.
2. si à l'étape 1, la bande contient un seul 0, accepter.
3. si à l'étape 1, la bande contient plus d'un 0 et que le nombre de 0 est impair, rejeter.
4. replacer le pointeur au début de la bande.
5. recommencer à l'étape 1.

Variation sur la méthode indiquée : on barre le premier 0 (= le dernier 0 qui reste), et on le remplace par un blanc afin de marquer le début de la bande.



$q_1 0\,0\;\;0\,0 \sqcup$
$\sqcup q_2 0\;\;0\,0 \sqcup$
$\sqcup\, x q_3\, 0\,0 \sqcup$
$\sqcup\, x\, 0\, q_4 0 \sqcup$
$\sqcup\, x\, 0\;\; x q_3 \sqcup$
$\sqcup\, x\, 0\, q_5 x \sqcup$
$\sqcup\, x q_5\; 0\; x \sqcup$
$\sqcup q_5 x\;\; 0\; x \sqcup$
$q_5 \sqcup x\;\; 0\; x \sqcup$
$\sqcup q_2 x\;\; 0\; x \sqcup$
$\sqcup\, x q_2\; 0\; x \sqcup$
$\sqcup\, x\, x\, q_3 x \sqcup$
$\sqcup\, x\, x\;\; x q_3 \sqcup$
$\sqcup\, x\, x\, q_5 x \sqcup$
$\sqcup\, x q_5\; 0\; x \sqcup$
$\sqcup q_5 x\;\; 0\; x \sqcup$
$q_5 \sqcup x\;\; 0\; x \sqcup$
$\sqcup q_2 x\;\; 0\; x \sqcup$
$\sqcup\, x q_2\; 0\; x \sqcup$
$\sqcup\, x\, x\, q_3 x \sqcup$
$\sqcup\, x\, x\;\; x q_2 \sqcup$
$\sqcup\, x\, x\;\; x \sqcup q_a$

Trouver la fin de la chaîne est facile. C'est le premier caractère blanc rencontré lorsqu'on déplace le curseur de lecture à droite.
Il est plus difficile de trouver le début de la chaîne :

- soit il faut y placer un caractère spécial (par exemple # ou ␣)
  dans ce cas, il faut remplacer le premier caractère. On a donc deux possibilités :

  — si on n'a plus besoin du premier caractère car on n'a déjà pris en compte sa valeur, on peut le remplacer directement,
  — sinon, on insère le caractère au début de la bande et on décale la totalité de la chaîne d'entrée d'un caractère vers la droite.

  Trouver le début de la chaîne est alors facile : se déplacer à gauche jusqu'à trouver le caractère de début de chaîne.

- soit on utilise le fait que le curseur ne bouge pas lorsqu'on est en début de bande de la manière suivante :

  — on retient le caractère courant, on le remplace par #, et on va à droite.
  — si on relit le même symbole #, alors on est au début de la bande.
  — on rétablit le caractère courant et on poursuit l'exécution en fonction du cas dans lequel on se trouve.

  Le procédé est à recommencer à chaque fois qu'on veut tester si on est en début de chaîne.

Exercice 25 (détection du début de la chaîne)**.** *1. écrire la machine de Turing qui insère un caractère spécial dans la première cellule et décale les caractères de la chaîne d'entrée d'un caractère à droite.*

2. *écrire alors la machine de Turing qui permet de retourner en début de chaîne indépendamment de l'endroit où se trouve le curseur de bande.*
3. *écrire la machine de Turing qui teste si le curseur est en début de bande en utilisant le fait que le curseur de lecture ne bouge pas lorsqu'on va à gauche en début de bande.*

Exercice 26 (premiers langages)**.** *Écrire la machine de Turing qui reconnaît les mots :*

1. *de la forme* 0^n1^n *avec* $n \geq 0$.
2. *de la forme* $0^n1^n2^n$ *avec* $n \geq 0$.
3. *qui commencent et se terminent par les deux mêmes caractères pour* $\Sigma = \{0, 1\}$.
4. *de la forme* $\#w\#w$ *avec* $w = \{0, 1\}^*$.
5. *de la forme* $\#w\#w^R$ *avec* $w = \{0, 1\}^*$ *où* w^R *est le mot écrit à l'envers.*
6. *de la forme* ww^R *avec* $w = \{0, 1\}^*$.
7. w *tels que* $w = w^R$ *avec* $w = \{0, 1\}$.

Exercice 27 (Automate déterministe fini)**.** *Décrire une méthode permettant de transformer tout automate déterministe fini en sa machine de Turing équivalente (i.e. qui accepte le même langage).*

Exercice 28. *Montrer qu'une machine de Turing à bande unique qui ne peut pas écrire sur la partie de la bande contenant la chaîne d'entrée ne reconnaît que les langages réguliers.*

Autre façon de décrire une machine de Turing : décrire le "programme" que la machine exécute en décrivant les mouvements du curseur et les actions sur la bande.

Exemple 51: Construction de la MT qui décide $L = \{a^ib^jc^k \mid ij = k \text{ où } i, j, k \geq 1\}$

1. lire la chaîne de gauche à droite et vérifier qu'elle est bien de la forme $a^*b^*c^*$.
2. retourner au début de la bande.
3. rayer un a, et scanner à droite jusqu'à ce l'on rencontre un b.
4. faire la navette entre les b et les c, en en rayant un de chaque jusqu'à ce qu'il n'y ait plus de b (rejeter si je ne peux pas rayer de c).
5. rétablir tous les b barrés.
6. s'il existe encore des a non barrés, alors retourner à l'étape 3.
7. sinon si tous les c sont barrés, alors accepter la chaîne.
8. sinon refuser la chaîne.

Exercice 29 (premiers programmes)**.** *Écrire le code de la machine de Turing qui reconnaît les langages suivants :*

1. $L_1 = \{0^n1^n2^n \text{ où } n \geq 0\}$
2. $L_2 = \{ww \mid w \in \{0, 1\}^*\}$
3. $L_3 = \left\{0^i1^j2^k \mid i < j < k\right\}$
4. $L_4 = \left\{\#x_1\#x_2\#\ldots\#x_k \mid x_i \in \Sigma^* \text{ et } \forall i \neq j, x_i \neq x_j\right\}$ *avec* $\Sigma = \{0, 1\}$.

2.1 Définition

Notations :

- $Q = \{q_0, q_1, \ldots\}$ ensemble des états ; $\{q_A, q_R\}$ exclus.
- $Q' = Q \cup \{q_A, q_R\}$
- Γ alphabet de la bande = alphabet des symboles du langage + symboles de bande supplémentaires.
- $\{L, R\}$ mouvement du pointeur sur la bande (L = gauche, R = droite).

Fonction de transition $\delta : Q \times \Gamma \to Q' \times \Gamma \times \{L, R\}$
$\delta(q, a) = (r, b, L)$ signifie :

- **état courant** :
  sur le graphe : l'état est q.
  sur la bande : la tête est sur le symbole a.
- **transition** :
  sur le graphe : l'état devient r.
  sur la bande : la machine écrit b (remplace le a) et la tête va à gauche (L).

δ est une fonction **déterministe** : à savoir pour tout état, pour tout symbole sous le curseur, il existe une seule transition possible qui écrit sur la bande une valeur unique et se déplace dans la direction déterminée associée.

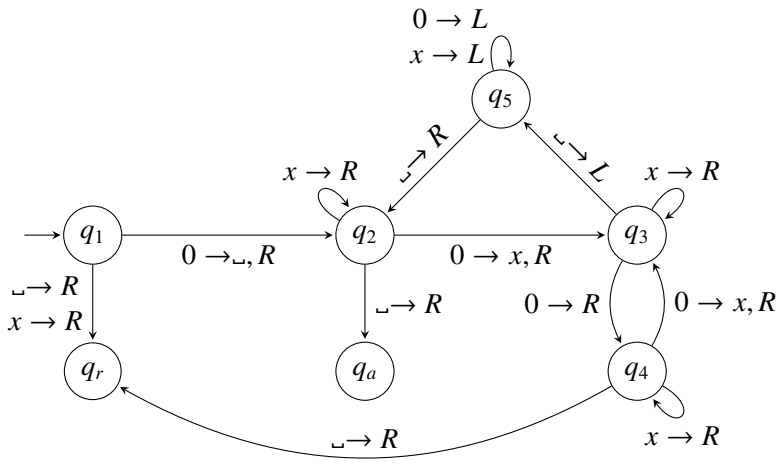
D'où la définition formelle d'une machine de Turing :

Définition IV.5

Une machine de Turing déterministe est un 7-uple $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ où :

- q_a est l'état acceptant, et q_r est l'état rejetant.
- Q est l'ensemble fini des états (q_a et q_r exclus). On ote $Q' = Q \cup \{q_A, q_R\}$.
- $q_0 \in Q$ est l'état de départ.
- Σ est l'alphabet du langage
- Γ est l'alphabet de la bande
- $\delta : Q \times \Gamma \to Q' \times \Gamma \times \{L, R\}$ est la fonction de transition.

Exemple 52: de description formelle



Description formelle de la MT $M = (Q, \Sigma, \Gamma, \delta, q_1, q_a, q_r)$ associée avec $Q = \{q_1, q_2, q_3, q_4, q_5\}$, $\Sigma = \{0\}$, $\Gamma = \Sigma \cup \{\mathtt{x}, \sqcup\}$

$\delta(q, r)$	0	x	$\sqcup$
q_1	$(q_2, \sqcup, R)$	$(q_r, \mathtt{x}, R)$	(q_r, R)
q_2	$(q_3, \mathtt{x}, R)$	$(q_2, \mathtt{x}, R)$	$(q_a, \sqcup, R)$
q_3	$(q_4, 0, R)$	$(q_3, \mathtt{x}, R)$	$(q_5, \sqcup, L)$
q_4	$(q_3, \mathtt{x}, R)$	$(q_4, \mathtt{x}, R)$	$(q_r, \sqcup, R)$
q_5	$(q_5, 0, L)$	$(q_5, \mathtt{x}, L)$	$(q_2, \sqcup, R)$

Exercice 30. *Donner la description formelle de la machine de Turing qui reconnaît le langage contenant autant de 0 que de 1 (voir le graphe donné dans l'exemple 49)*

2.2 Modèle de calcul d'une machine de Turing

On rappelle que, pour une machine de Turing déterministe, une configuration définit complètement l'état de la machine.

Le déterminisme fait aussi qu'une configuration donnée ne peut conduire qu'à une seule autre configuration.

Il existe certaines configurations particulières :

- si w est le mot mis en entrée de la machine, et q_0 l'état de départ, alors $q_0 w$ est la configuration de départ.
- si C est une configuration qui contient l'état q_a, alors c'est une configuration acceptante,
- si C est une configuration qui contient l'état q_r, alors c'est une configuration rejetante.

Ainsi, toute configuration de départ ne produit qu'une seule exécution possible, et donc, qu'une seule suite de configurations.

Définition IV.6 (chaîne acceptée par une MT M)

M accepte un chaîne w en entrée s'il existe une suite de configurations $C_1, C_2, \cdots C_k$ telles que :

- C_1 est la configuration de départ de M sur w.
- pour tout i, la configuration C_i conduit à la configuration C_{i+1}.
- C_k est une configuration acceptante.

Définition IV.7 (langage accepté par une machine de Turing)

Un langage L reconnu par une machine de Turing M est constitué de l'ensemble des mots $w \in \Sigma^*$ tels que $M(w)$ accepte.

Notation : $L = \mathcal{L}(M)$

3 Autres modèles de machine de Turing

Le modèle de la machine de Turing est simple, mais l'ensemble des langages reconnus les MTs change-t-il si on utilise les améliorations suivantes :

- une commande S (=stay) qui permet d'effectuer une transition sans bouger le pointeur sur la bande,
- un alphabet avec plus de symboles (#Σ dans sa définition formelle) ?
- une bande de travail infinie des deux cotés ?
- plusieurs bandes de travail ?
- en utilisant des transitions non déterministes ?

Afin de démontrer ces points, la ligne de démonstration sera toujours la même :

Est-il possible de simuler sur le modèle de MT le plus simple (avec un alphabet binaire) la MT améliorée ?

Par simuler, on entend : prendre les mêmes décisions (accepter, rejeter ou boucler) que la machine que l'on simule **en utilisant une machine de Turing classique**.

Si la réponse est oui, alors les deux modèles reconnaissent exactement le même ensemble de langages. On se propose de déterminer l'impact d'une modification du modèle de la machine de Turing.

Modifications étudiées :

- Ajout d'une commande S (= stay) permettant à la MT de lire/écrire sans se déplacer,
- Restriction de l'alphabet Σ à $\{0, 1\}$ (= codage d'un alphabet fini en binaire),
- Utilisation d'une bande de travail infinie des deux cotés,
- Utilisation de plusieurs bandes de travail au lieu d'une seule,
- Utilisation du non-déterminisme

Cet impact peut être de plusieurs ordres :

- sur la performance,
- sur l'ensemble des langages reconnus par rapport à une MT classique.

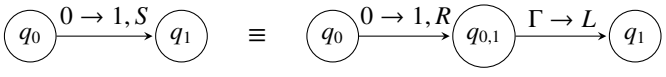
3.1 Ajout de commande

On veut ajouter une commande S permettant à la machine de Turing de lire/écrire sans se déplacer.

La fonction de transition est alors définie dans :

$$\delta : Q \times \Gamma \to Q' \times \Gamma \times \{L, R, S\}$$

Analyse : il est possible de simuler une MT LRS sur une MT LR en remplaçant chaque transition S par deux transitions (R puis L)



note : pas L puis R (problème au début de la bande).
Conséquences :

- une MT LRS modifiée n'est pas plus puissante puisqu'elle peut être simulée sur une MT LR.
  ⇒ L'ensemble des langages reconnus par les MT LRS et les MT LR est le même.
- la simulation ralentit la MT LR au plus d'un facteur 2 par rapport à la MT LRS.

On montre ainsi l'équivalence entre les MT classiques et les MT LRS.

3.2 Restriction d'alphabet

La taille de l'alphabet a-t-il une influence sur l'ensemble des langages qu'il est possible de reconnaître ?

Considérons une MT M_Σ utilisant un alphabet Σ étendu. Est-il possible de simuler M_Σ sur une MT $M_\mathbb{B}$ en utilisant un alphabet binaire $\Sigma_\mathbb{B} = \{0, 1\}$?

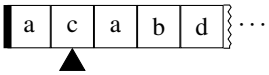
Évidemment, cette simulation ne peut s'effectuer qu'à travers l'encodage des symboles de Σ dans $\mathbb{B}$.

Avec un codage à taille fixe, il faut $p = \lceil \log_2 \#\Sigma \rceil$ bits pour coder l'ensemble des symboles de Σ dans $\mathbb{B}^p$.

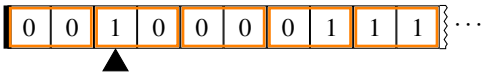
La bande est découpée en blocs de p bits dont chacun représente un symbole de Σ.

Exemple 53: Codage avec l'alphabet binaire $\Sigma_\mathbb{B}$ de l'alphabet $\Sigma = \{a, b, c, d\}$: $a = 00$, $b = 01$, $c = 10$, $d = 11$.

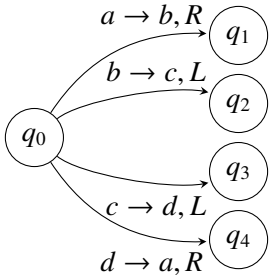
bande sur M_Σ



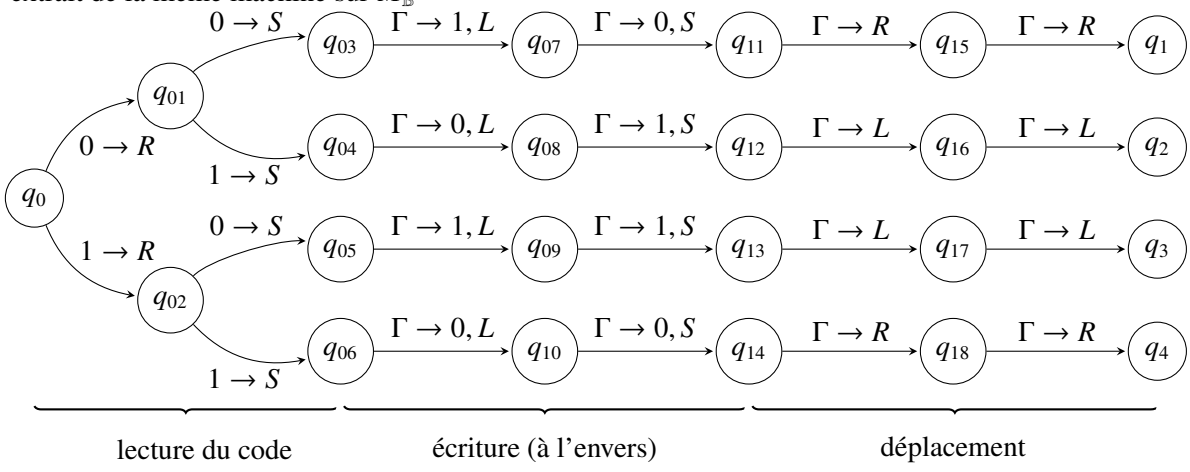
bande sur $M_\mathbb{B}$



extrait de machine sur M_Σ



extrait de la même machine sur $M_\mathbb{B}$



Montrons que le comportement de M_Σ sur tout mot d'entré peut être simulé sur une machine $M_\mathbb{B}$ sur ce même mot d'entré codé en binaire.

Sur $M_\mathbb{B}$, la simulation d'une transition $a \to b, L$ (resp. $a \to b, R$) de M_Σ consiste en :

- lecture à droite du bloc de p bits contenant le code de $a = p$ transitions.
- retour d'un caractère à gauche (= retour sur le dernier caractère du bloc),
- écriture inversée à gauche du bloc de p bits contenant le code de $b = p$ transitions.

- retour d'un caractère à droite (= retour sur le premier caractère du bloc),
- déplacement du pointeur de lecteur à gauche (resp. à droite) sur le bloc précédent = p transitions.

Soit un facteur multiplicatif de l'ordre de $3p$ transitions.

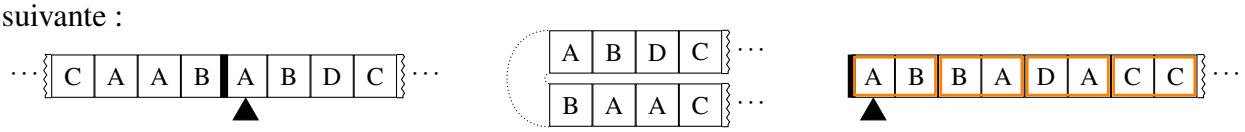
Conséquences : une MT M_Σ peut être simulée sur une MT $M_{\mathbb{B}}$.

- Elle n'est pas plus puissante qu'une $M_{\mathbb{B}}$,
- La simulation $M_{\mathbb{B}}$ augmente d'un facteur $3.\lceil \log_2 \#\Sigma \rceil$ le nombre de transitions nécessaires sur une entrée par rapport à M_Σ.

3.3 Bande doublement infinie

Considérons maintenant le cas où la bande est infinie des deux côtés.

Cette MT M_b peut être transformée en une MT M classique en transformant la bande de la manière suivante :

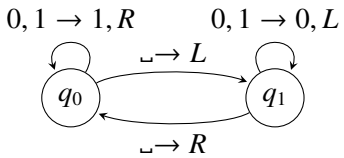


- "plier" la bande en deux à partir de son milieu (= position à laquelle le pointeur de lecteur est initialisé).
- chaque symbole de la bande est codé par un couple de symboles.
- pour un déplacement sur la portion droite (resp. gauche), considérer le symbole de droite (gauche) (= une version de M_b gauche et une droite).
- pour savoir quand passer de l'un à l'autre, placer un marqueur # au centre de la bande. Sa détection s'effectue en 2 transitions :
  - si le symbole courant $\neq$ # alors on continue sur le M_b courant.
    sinon passer sur l'autre M_b (avec inversion déplacement).

Conséquences : une MT M_b peut être simulée sur une MT M.

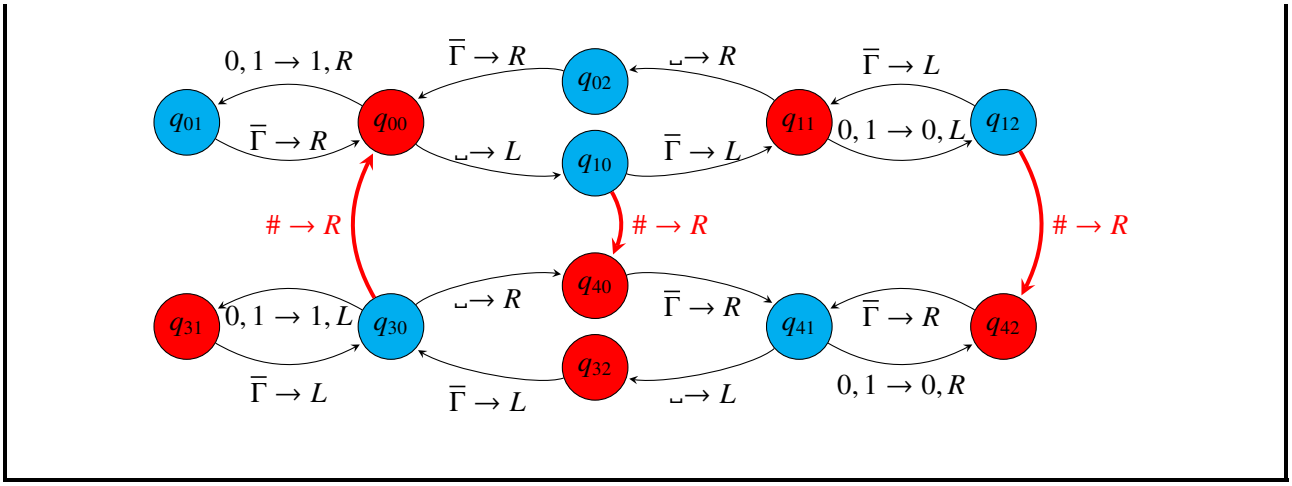
- Elle n'est pas plus puissante qu'une M,
- La simulation M_b augmente d'un facteur multiplicatif 2 le nombre de transitions nécessaires sur une entrée par rapport à M.

Exemple 54: Machine originale : va à droite en écrivant des 1, puis va à gauche en écrivant des 0, et recommence entre le début et la fin de la bande.



Machine avec bande doublement infinie :

- Ajouter des états intermédiaires pour passer un symbole sur deux.
- Dupliquer la machine en échangeant les sens d'exécutions
- Relier les deux machines lorsque le caractère de pliage de bande est rencontré en allant à gauche.



3.4 Machine de Turing multibandes

Une MT est modifiée en une MT_k à k bandes de la manière suivante :

- la mémoire est constituée de k bandes (infinies à droite),
- il y a k pointeurs de bande indépendants (un par bande),
- à l'initialisation, la chaîne d'entrée placée sur la première bande, les autres bandes sont blanches.
- la fonction de transition est définie par $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$
  $\delta(q_i, a_1, \cdots, a_k) = (q_j, b_1, \cdots, b_k, L, \cdots, R)$ signifie :

  si la machine est dans l'état q_i, et que les pointeurs $\{1, \cdots, k\}$ lisent $\{a_1, \cdots, a_k\}$ sur les k bandes,

  alors la machine passe à l'état q_j, les pointeurs $\{1, \cdots, k\}$ écrivent $\{b_1, \cdots, b_k\}$ sur les k bandes puis bougent respectivement dans les directions $\{L, \cdots, R\}$.

Typiquement, pour ce type de modèle de MT, un rôle particulier est affecté à chaque bande (exemple pour un transducteur : bande 1 = bande d'entrée en lecture seule, bande 2 = bande de travail, bande 3 = bande de sortie en écriture seule).

Exercice 31. *Écrire une MT multibande (avec au moins deux bandes) qui reconnaît :*

1. *le langage* $0^n 1^n$.
2. *l'ensemble palindromes.*
3. *l'ensemble des mots ayant le même nombre de 0 et de 1.*
4. *le langage* $\left\{\#x_1\#x_2\#\ldots\#x_k \mid x_i \in \Sigma^* \text{ et } \forall i \neq j, x_i \neq x_j\right\}$ *avec* $\Sigma = \{0, 1\}$.

Exercice 32. *Expliquer comment simuler un automate à pile avec une machine de Turing multibande, avec l'une des bandes qui joue exactement le rôle de la pile.*

Théorème IV.1 (équivalence MT simple bande - MT multi-bande)

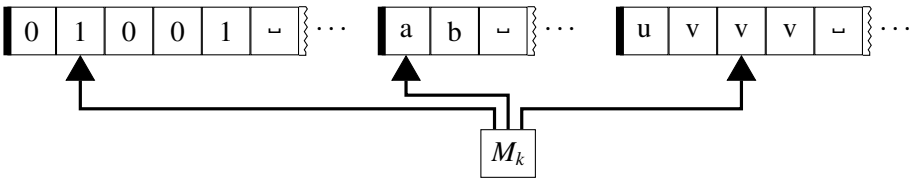
Toute MT M_k a k bandes a une MT M simple-bande équivalente.

DÉMONSTRATION: (constructive)

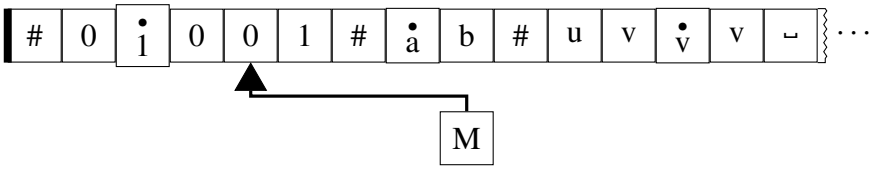
On stocke les k bandes de M_k sur la bande de M avec :

- un marqueur séparateur de bande : symbole #.
- un marqueur de position de pointeur : pour tout symbole $x \in \Gamma_k$, on ajoute une version pointée $\dot{x}$ à Γ signifiant que le pointeur est placé sur ce symbole.

Exemple de MT multibande :



Equivalent simple bande :



Fonctionnement de M : avec l'entrée $w = w_1 \cdots w_n$

1. **initialisation de la bande** : $\#\dot{w}_1 w_2 \cdots w_n \# \dot{\sqcup} \# \dot{\sqcup} \cdots \# \dot{\sqcup}$
2. **simulation d'une transition** :
   - **détermination de l'état** : scanner la bande du premier # au $(k+1)^{\text{ème}}$ qui marque la fin de la dernière bande pour déterminer les symboles pour chaque pointeur de chaque bande.
   - **transition** : faire une seconde passe pour effectuer la transition associé à chaque symbole pointé, en accord avec la fonction de transition de M_k.
3. **ajustement de la taille d'une bande** :
   si l'un des pointeurs virtuels se déplace sur un # (arrivée en bout de bande virtuelle), alors M écrit un symbole ␣, et décale tous les symboles à droite d'une case vers la droite.

M reproduit ainsi très exactement le comportement des k bandes de M_k. □

On montrera que la complexité supplémentaire engendrée par la MT à une bande est un facteur multiplicatif $5.k.T(n)^2$ où $T(n)$ est le nombre de transitions de la MT à k bandes pour une entrée de taille n.

3.5 Machine de Turing non déterministe

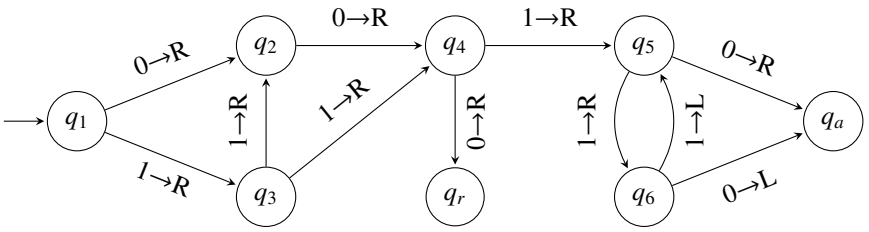
Une machine de Turing non déterministe (MTND) est une généralisation similaire à celle des ADFs en ANFs :

- pour chaque état, il existe une ou plusieurs transitions possibles associées au symbole sous le pointeur, voir aucune.
- pour chaque choix non déterministe lors de l'exécution, la machine se découple en autant de copies que nécessaires, chaque copie s'occupant d'une branche d'exécution.

Comportement possible d'une MTND :

- si une branche accepte, alors la MTND s'arrête en acceptant.
  peu importe le comportement des autres branches (rejet ou boucle).
- si toutes les branches rejettent, alors la MTND s'arrête en rejetant.
  = aucune branche de calcul n'a amené à accepter la chaîne d'entrée.
- si aucun branche n'accepte, et qu'**au moins une branche boucle**, alors la MTND boucle.
  branche qui boucle = calcul sur la branche non terminé ⇒ pas de décision.

Exemple : avec $\Sigma = \{0, 1\}$



- q_1,q_4, q_6 : états avec transition déterministe (une transition par symbole de Σ).
- q_3, q_5 : états avec une transition non déterministe.
- q_2 : n'a pas de transition pour 1 (=arrêt de la branche si cela est le cas).
  exemple : les mots commençant 01 ou 10 sont rejetés sans que la machine n'atteigne l'état q_r.
- la transition 1 pour q_5 et q_6 peut produire une boucle.
  exemple : pour le mot 001110, la machine boucle.

Pour une entrée w, une MTND :

- accepte w s'il existe (au moins) une branche qui l'accepte.
- rejette w si toutes les branches la rejettent ou bouclent à l'infini.

On considère donc qu'un mot qui fait boucler une machine n'appartient pas au langage que cette dernière reconnaît.

En terme de configuration,

- une exécution d'une MT déterministe est un chemin dans l'espace des configurations de la MT.
- une exécution d'une MTND est un arbre dans l'espace des configurations de la MTND.

Exercice 33. *Toutes les MTNDs proposées devront utiliser activement le non-déterminisme.*

1. *écrire une MTND qui reconnaît les mots qui commencent et se terminent par les deux mêmes caractères. On prendra* $\Sigma = \{0, 1\}$.
2. *écrire une MTND qui reconnaît les mots contenant la chaîne* 011.
3. *écrire une MTND qui reconnaît les mots de la forme ww.*
4. *écrire une MTND qui reconnaît le langage* $0^i1^j2^k$.
5. *écrire le code d'une MTND qui trouve deux nombres binaires* w_1 *et* w_2 *de la même longueur que le mot d'entrée tels que* $w_1 \oplus w_2 = 0$ *(où* $\oplus$ *est l'opérateur* `XOR`*).*

La fonction de transition est de la forme :

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Une MTND semble visiblement plus puissante qu'une MT.
une MT = une branche d'exécution d'une MTND.

Donc, tout langage récursivement énumérable peut être aussi reconnu par une MTND.

Théorème IV.2 (équivalence MTND-MT)

Toute MTND M' a une MT M équivalente.

DÉMONSTRATION:

idée : simuler une MTND M' avec une MT M = explorer toutes les branches de la MTND M'.

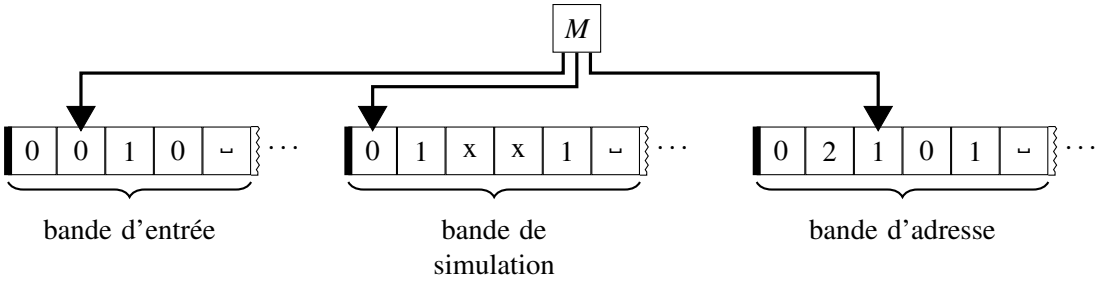
- M essaie toutes les branches possibles.
- si M accepte l'une de ces branches, alors M accepte.
- si toutes les branches sont rejetées, alors M rejette.
- si toutes les branches sont rejetées ou bouclent, alors M boucle.

L'exécution de M est un arbre :

- chaque branche de l'arbre est une branche de l'exécution non déterministe de M'.
- le nœud de l'arbre est la configuration initiale.
- chaque nœud de l'arbre est une configuration de M.
- le nombre d'enfants d'un nœud est au plus $n = \#Q \times \#\Gamma \times 2$
  note : $2 = \#\{L, R\}$.
- parcours de l'arbre :
  - — la recherche en profondeur d'abord ne fonctionne pas.
    la branche courante peut entrer dans une boucle infinie.
  - — la recherche en largeur d'abord est la solution.
    on trouve ainsi la plus petite configuration qui accepte la chaîne.

On simule la MTND avec une MT à 3 bandes avec :

- **une bande d'entrée** : contient la chaîne d'entrée, et n'est jamais modifiée,
- **une bande de simulation** : contient la même chose que la bande de M',
- **une bande d'adresses :** contient de la position de M dans l'arbre d'exécution de M'.



Utilisation de la bande d'adresses :

- chaque nœud a au plus n enfants.
- à chaque nœud, on peut affecter une adresse qui est une chaîne dans l'alphabet $\Delta = \{1, \cdots, n\}$ chaque symbole correspondant à une configuration.
- **utilisation d'une adresse** :
  pour se rendre au nœud d'adresse 231
  - — choisir le second fils de la racine.

— puis choisir son troisième fils de ce second fils.
— puis choisir son premier fils de ce troisième fils.

- ignorer les adresses qui n'ont pas de sens
  et qui correspondent à des configurations impossibles

Génération de tous les chemins possibles dans l'arbre :

exemple : pour 2 enfants par nœud ($\Delta = \{1, 2\}$), avec une recherche en largeur d'abord, revient à construire toutes les chaînes possibles dans l'ordre :

$1, 2, 11, 12, 21, 22, 111, 112, 121, 122, 211, 212, 221, 222, ...$

Tous les chemins partant de la racine sont ainsi simulés.

Simulation :

1. **Initialisation** : la bande d'entrée contient w, les deux autres sont vides.
2. copier la bande d'entrée sur la bande de simulation.
3. utiliser la bande de simulation pour simuler M' sur l'entrée w sur une portion limitée d'une des branches déterministes (l'entrée est simulée depuis le début).
   A chaque choix,
   - consulter le symbole suivant sur la bande d'adresse.
   - si un état acceptant est atteint, alors accepter w.
   - passer à l'étape suivante si :
     — les symboles sur la bande d'adresse sont épuisés.
     — un choix non déterministe est invalide.
     — un état rejetant est atteint.
4. remplacer la chaîne sur la bande d'adresse par l'entrée suivante pour chaque choix possible.
5. sauter à l'étape 2 et simuler cette branche de l'exécution de M'.

On simule ainsi toutes les branches de la MTND. □

Théorème IV.3 (simulation d'une MTND sur une MT (non démontré))

Une MTND N en temps $t(n)$ peut être simulée sur une MT déterministe M en temps proportionnel à $c(N)^{t(n)}$ où $c(N)$ est une constante dépendante de N.

La constante $c(N)$ dépend du nombre d'états, du nombre de bandes et de la taille de l'alphabet de N.

Conséquences : une MTND N peut être simulée sur une MT M,

- N n'est pas plus puissante qu'une M,
  Les MTNDs reconnaissent (exactement) les langages récursivement énumérables.
- en revanche, N peut reconnaître le langage potentiellement exponentiellement plus vite que M.
  Ce point sera étudié dans le chapitre sur la complexité temporelle.

3.6 Conséquence

Le modèle de la MT est extrêmement stable : les variations sur ce modèle sont incapables de reconnaître autre chose que les langages récursivement énumérables.

En terme de performance,

- les variations déterministes peut être simulées sur une MT classique et n'entraine qu'au plus un gain quadratique (proportionnel à $k.t^2$ pour une MT à k bandes),
- les variations non déterministes peuvent également être simulées mais le gain dans ce cas peut être beaucoup plus conséquent (exponentiel) dans certains cas particuliers que nous repréciserons.

4 Fonction calculable

Nous avons vu que les machines de Turing étaient une classe de machine permettant de résoudre (au moins) certains problèmes de décision.

Intéressons-nous à une modification d'une machine de Turing afin de résoudre (au moins) certains problèmes d'évaluations.

On parle alors de fonctions (partiellement) calculables.

Pour ce faire, nous allons aussi utiliser des machines de Turing dont la seule caractéristique supplémentaire sera de s'arrêter avec l'évaluation attendu sur sa bande.

4.1 Définitions

Définition IV.8 (fonction (totalement) calculable)

Une MT M calcule totalement une fonction $f : \Sigma^* \to \Sigma^*$ si M

- commence avec l'entrée w sur sa bande.
- s'arrête pour tout w et avec seulement $f(w)$ écrit sur sa bande.

Remarques 26:

- marche aussi pour les fonctions avec plus d'une variable.
  Encoder les paramètres sur la chaîne d'entrée intercalée de séparateurs.
  Chaque paramètre peut représenter n'importe quel objet.
- même remarque pour la sortie.
- une bande particulière peut être utilisée pour la sortie.

Notation : on note $\perp$ la sortie d'une fonction calculable si l'entrée n'est pas son ensemble de définition.

Définition IV.9 (fonction partiellement calculable)

Une MT M calcule partiellement une fonction $f : \Sigma \to \Sigma \cup \perp$ si M :

- commence avec l'entrée w sur sa bande.
- si $f(w)$ est défini, alors M s'arrête avec seulement $f(w)$ écrit sur sa bande.
- si $f(w)$ est indéfini, alors M retourne $\perp$ ou ne s'arrête pas.

Remarque 27:

Les fonctions calculables sont également appelées fonctions (totalement ou partiellement) récursives.

4.2 Exemples de fonctions calculables

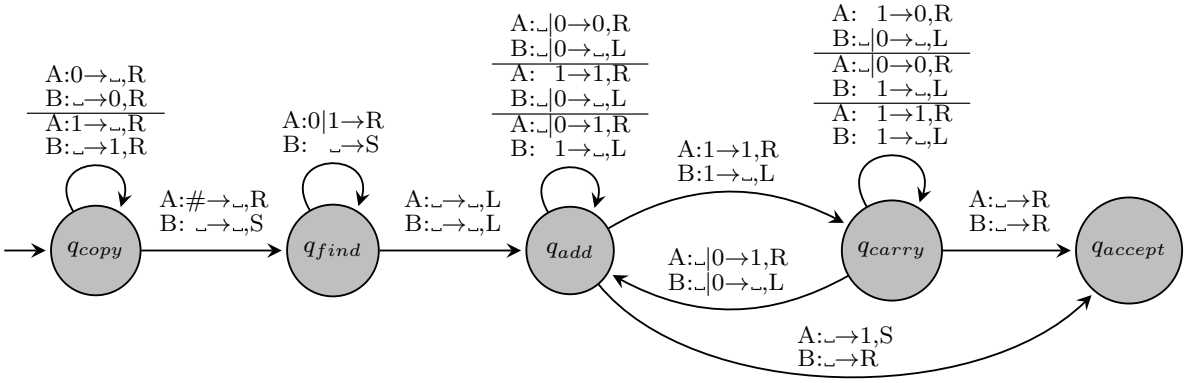
- Toutes les fonctions arithmétiques classiques sur les entiers sont totalement calculables : addition, soustraction, multiplication, division (quotient, reste), ...
- Toutes les fonctions non-arithmétiques sont également totalement calculables : trigonométrique, logarithmique, ... en utilisant les séries de Taylor (ou d'autres techniques d'approximation) à une précision spécifiée.

  voir "Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables" d'Abramowitz et Stegun sur la façon pratique de réaliser ces approximations à une précision donnée.

Nous donnons ici l'implémentation de l'addition binaire sur une MT à 2 bandes (notée A et B), infinie des deux côtés, initialisées avec des blancs, et avec un pointeur LSR (S =stay (ne pas bouger)).

L'entrée de la MT est la forme 1011#110 où # est le séparateur entre les deux mots binaires que l'on souhaite additionner.

- q_{copy} : copie le 1$^{\text{er}}$ mot binaire sur la deuxième bande (état de départ).
- q_{find} : place le pointeur sur le LSB du 2$^{\text{éme}}$ mot binaire de la 1$^{\text{ère}}$ bande.
- q_{add} : somme de 2 chiffres binaires sans retenue pour 0+0, 0+1, 1+0, avec retenue pour 1+1.
- q_{carry} : somme de 2 chiffres binaires avec retenue r=1 (1+0+r, 0+1+r, 1+1+r), sans retenue pour 0+0+r.
- q_{end} : fin du calcul.



Exercice 34. *Écrire la fonction calculable qui :*

1. *calcule un xor entre deux mots binaires. Supposera que l'entrée de la fonction est la suivante $f(\langle w_1\#w_2\rangle)$ où w_1 et w_2 sont deux mots binaires. La fonction pourra être multibande.*
2. *incrémente de 1 le nombre binaire en entrée.*
3. *calcule, en binaire, le nombre de* 1 *présent dans une chaîne binaire.*
4. *effectue la multiplication de deux nombres unaires. Par exemple $f(\langle x, y\rangle) = f(11\#111) =$ 111111.*
5. *effectue $f(\langle n\rangle) = \underbrace{1\ldots1}_{n\ fois}$.*
6. *triplique la chaine d'entrée (i.e. si la chaine d'entrée est $w =$ 11, la fonction retourne $www =$ 111111). On prendra l'alphabet $\Sigma = \{1\}$*
7. *retourne le caractère au centre d'une chaîne de caractères si sa longueur est impaire, et une erreur sinon.*

5 Machines Universelles

De ce qui précède, nous avons utilisés les machines de Turing sous forme de «circuit spécialisé».

i.e. si nous avions construit une implémentation physique d'une MT M qui reconnaît un langage particulier. nous aurions été obligé de construire une autre machine différente pour reconnaître un autre langage.

On se demande alors s'il ne serait pas possible de construire une modèle de machine qui prenne en paramètre un programme et serait ainsi capable de simuler l'algorithme décrit dans ce programme ?

5.1 Machine de Turing Universelle

Exercice 35. *Soit la description formelle d'ADF $A = \langle(Q, \Sigma, \delta, q_0, F)\rangle$.*

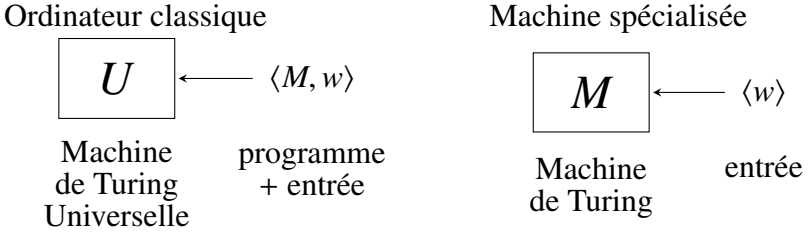
1. *Rappeler le sens de chacun des éléments de la description formelle d'un ADF.*
2. *Donner un codage de A sous forme d'une chaîne de symboles.*
3. *Écrire le code de la MT qui vérifie si une chaine w est bien le codage d'un ADF, à savoir d'une MT M telle que $M(\langle A\rangle)$ accepte si A représente la description formelle d'un ADF, et rejette sinon.*
4. *Écrire le code de la MT qui vérifie si un mot w est accepté par l'ADF A, à savoir écrire une MT M tel que $M(\langle A, w\rangle)$ accepte si A accepte w et rejette sinon.*

Il est évident que la description d'une MT M peut elle aussi être encodée (*i.e.* $\langle M\rangle = \langle(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)\rangle$).

Exercice 36. *Donner un codage de la description formelle donnée à l'exercice 30.*

On définit alors :
MT universelle $\stackrel{\Delta}{=}$ MT qui peut simuler une MT M arbitraire sur une entrée arbitraire w.

Sur une MT universelle, le programme a exécuter est donné sous forme de la description d'une MT (= langage de programmation).



Historique : C'est la MT universelle qui a inspiré les premiers ordinateurs programmables des années 40 et 50.

Comment réaliser cela ?

La MT universelle U fonctionne de la façon suivante :

- Mettre $\langle M, w\rangle$ en entrée de la MT.
- Vérifier que $\langle M, w\rangle$ est un encodage correct d'une MT, suivi par une chaîne de Σ^*.
- Simuler M sur w.
  voir ci-après le détail.
- A l'entrée w,
  - si M entre dans un état acceptant, alors U accepte.

— si M entre dans un état rejetant, alors U rejette.
— si M boucle, alors U boucle aussi.

Comment simuler M sur U ?

- **Organisation :**
  Pour une MT M à une bande et alphabet de travail Γ,
  la MT universelle U a :
  — 5 bandes :
    1. **bande d'entrée** : contient $\langle M, w\rangle$
    2. **bande du programme** : contient $\langle M\rangle$
    3. **bande de simulation** : contient w
    4. **bande d'état** : contient l'état de la MT M
    5. **bande de travail** : stockage intermédiaire.
  — un alphabet de travail Γ' étendu
  marquage de symboles : pointé (position de lecture), barré, ...
- **Initialisations**
  — copies : $\langle M\rangle$ sur la bande de programme, w sur la bande de simulation.
  — placer la tête de la bande de simulation au début de w (pointer son premier symbole).
  — placer l'état de départ sur la bande d'état.
- **Exécution de U**
  1. placer une copie de l'état q_i (depuis la bande d'état) et une copie du caractère courant a (depuis la bande de simulation) sur la bande de travail.
  2. comparer (q_i, a) aux entrées de table de transition de la bande du programme.
  3. lorsque la correspondance est trouvée (ex : $\delta(q_i, a) = (q_j, b, L)$)
     — mettre à jour la bande d'état avec q_j.
     — écrire la lettre b et déplacer le pointeur sur le caractère dans la direction L sur la bande de simulation.
  4. test de l'état q écrit sur la bande d'état.
     — si $q = q_{\text{accept}}$ alors U accepte $\langle M, w\rangle$
     — si $q = q_{\text{reject}}$ alors U rejette $\langle M, w\rangle$
     — sinon on recommence au 1.

Remarque : la MT universelle U

- a un alphabet de travail beaucoup plus grand
  pour faciliter les comparaisons, les copies, l'effaçage
- a un nombre d'états **fini** (environ une centaine)
  — le nombre d'états est indépendant de la machine à simuler.
  — U peut donc simuler une *MT* M avec beaucoup plus d'états.

Théorème IV.4 (MT universelle efficace)

Soit une MT $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ où $T(w)$ est le nombre de transitions avant son arrêt sur l'entrée w.

Il existe une MT universelle capable de simuler une machine M en $C.T \log T$ transitions où C dépend de $|w|$, $\#\Sigma$, $\#Q$ et du nombre de bandes.

Démonstration :

voir "Two-tape simulation of multitape Turing machines", Hennie et Stearns, 1966.

La méthode présentée ci-avant est en T^2. □

Intéressons-nous aux descriptions de MT possibles :

- il est clair que l'ensemble des descriptions MTs possibles est Σ^*.
- on peut faire en sorte que la MT universelle rejette toutes les entrées si la description de la machine $\langle M \rangle$ n'est pas valide (de façon à ce que toute chaîne de Σ^* soit associée à une MT).
- on peut faire en sorte que la MT universelle ignore tous les symboles qui suivent une description correcte d'une MT correcte (*i.e.* $\langle M \rangle \equiv \langle M \rangle \Sigma^*$).
  de la même façon que l'ajout de commentaires à un programme ne change pas son sens.

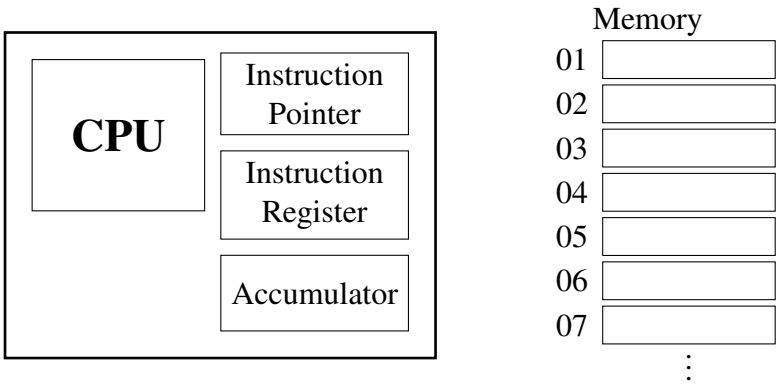
En conséquence,

- il y a un nombre d'algorithmes infini (autant que $\#\Sigma^*$),
- il y a un nombre infini d'algorithmes qui acceptent le même langage.

Ces notions d'infini seront précisés plus tard.

5.2 Random Access Machine

Composants d'une machine à accès aléatoire (ou RAM) :



Notons qu'une RAM est une machine programmable.

Comment définir une machine de Turing permettant de simuler une RAM ?
Une RAM est définie à partir :

- d'une mémoire (`MEM`).
  `MEM[i]` représentant le contenu de la mémoire à l'adresse i.
- de registres :
  - `IP` : instruction pointer (ligne du code en cours d'exécution)
    connu aussi sous le nom de Program Counter
  - `IR` : instruction register (code de l'instruction à exécuter)
  - `A` : accumulateur (mémoire locale)
- d'un CPU qui fonctionne comme suit :
  1. `IR` ← `MEM[IP]` (= charger l'instruction `IR`)
  2. incrémenter `IP`
  3. exécuter l'instruction dans `IR`

  voir ci-après pour le type d'instructions exécutable

Jeux d'instructions du CPU :

Code	Instruction	Signification
000	HALT	Arrêt
1xx	LOAD xx	A←MEM[xx]
2xx	LOADI xx	A←xx
3xx	STORE xx	MEM[xx]←A
4xx	ADD xx	A←A+MEM[xx]
5xx	ADDI xx	A←A+xx
6xx	SUB xx	A←A-MEM[xx]
7xx	SUBI xx	A←A-xx
8xx	JUMP xx	IP←xx
9xx	JZERO xx	IP←xx if A = 0

Exemple de programme : multiplication $n_r = n_1 \times n_2$

organisation mémoire :

adresses	50	51	52	53
contenu	n_1	n_2	n_r	compteur

Memory	Code	Assembleur
01	150	LOAD 50
02	353	STORE 53
03	200	LOADI 0
04	352	STORE 52
05	253	LOAD 53
06	912	JZERO 12
07	701	SUBI 1
08	353	STORE 53
09	152	LOAD 52
10	451	ADD 51
11	804	JUMP 04
12	000	HALT

Algorithme :

```
START  A←MEM[50]
       MEM[53]←A
       A←0
LOOP   MEM[52]←A
       A←MEM[53]
       IF A==0 GOTO END
       A←A-1
       MEM[53]←A
       A←MEM[52]
       A←A+MEM[51]
       JUMP LOOP
END    HALT
```

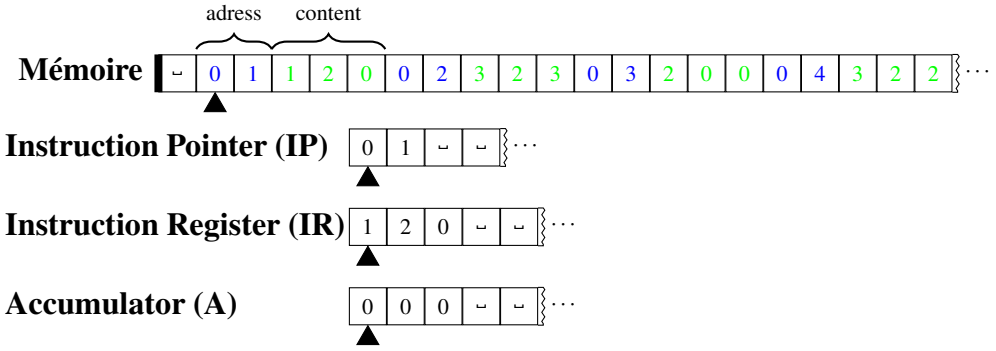
Théorème IV.5 (simulation d'une RAM)

Toute RAM peut être simulée par une MT multi-bande.

DÉMONSTRATION: Très long ! Juste une idée de la preuve.

Organisation des bandes :

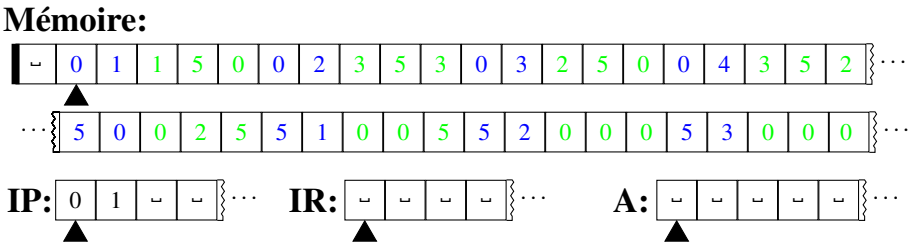
- utiliser une bande par registre (IR, IP et A).
- utiliser une bande pour la mémoire
- organisation de la mémoire :
  avec des entrées : <adresse> <contenu>
  utilisée pour stocker **le code** et **les données**



Initialisation : (de la MT)

Placer les données et le code dans la mémoire.

Initialiser le contenu de l'IP à `0001`



Exécution : (de la MT comme RAM)

1. parcourir la mémoire à la recherche de l'adresse correspondant à l'IP.
2. copier le contenu de cette adresse dans l'IR.
3. incrémenter l'IP.
4. en fonction de la valeur de l'IR
   - copier une valeur dans l'IP (jump)
   - copier une valeur / l'accumulateur dans la mémoire (store)
   - copier une valeur / le contenu mémoire dans l'accumulateur (load)
   - faire une addition/soustraction
5. recommencer au 1.

Note : montrer aussi que chaque instruction peut se simuler sur la MT. □

Des graphes simplifiés de la machine de Turing simulant une RAM seront donnés au cours de la simulation avec les notations suivantes :

- $* \to R$ signifie que la transition a lieu pour tout symbole sous le curseur.
- $B : 0 \to R$ signifie que la transition concerne la bande B.
- normalement, pour une machine multibande, toute transition entre deux états doit être spécifiée pour toutes les bandes.
- toute bande B non spécifiée dans une transition ne sera pas modifiée par cette dernière. Donc équivalent à $B : * \to S$ = le curseur ne bouge pas pour tout symbole sous le curseur pour cette bande).
- si une même variable x (i.e. x est un symbole quelconque) est utilisé sur plusieurs bandes, alors ce symbole doit être identique pour les deux bandes.

M:x→R IP:x→R	transition à droite sur les bandes M et IP si elles sont sur le même symbole x
M:x→ R IP:*→x,R	transition à droite sur les bandes M et IP, le symbole sur la bande M est écrit à la place du symbole courant sur la bande IP.

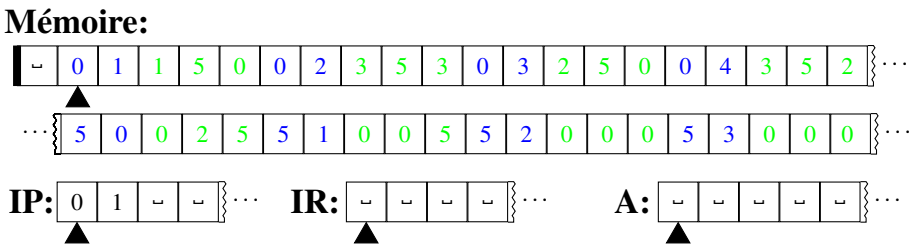
- si deux variable x et y sont utilisés sur plusieurs bandes, alors elles représentent des symboles différents.

M:x→R IP:y→R	transition à droite sur les bandes M et IP si elles sont sur des symboles différents

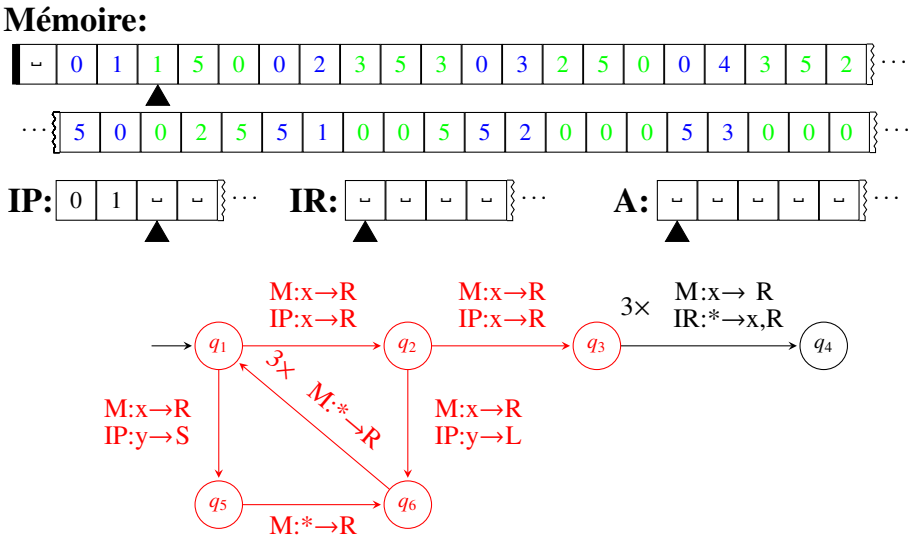
- l'écriture 2 × M:x→R signifie que la même transition est effectuée deux fois.

Exécution ! (multiplication)

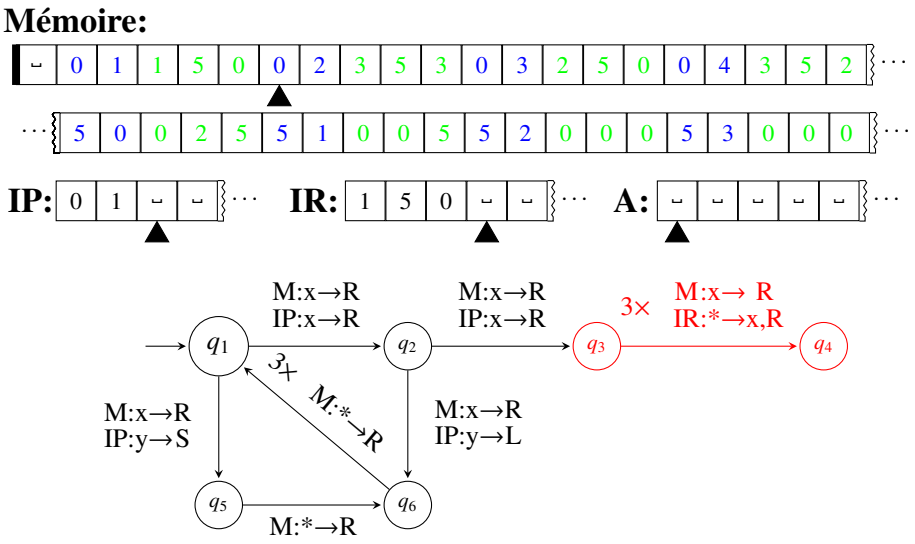
• Initialisation



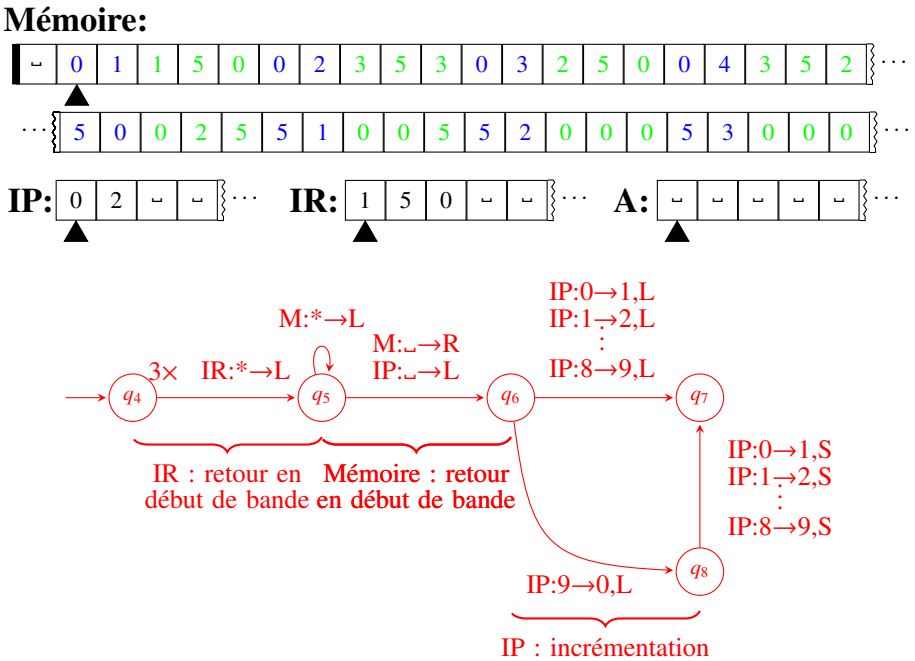
• Recherche de l'IP dans la mémoire (1ère instruction)



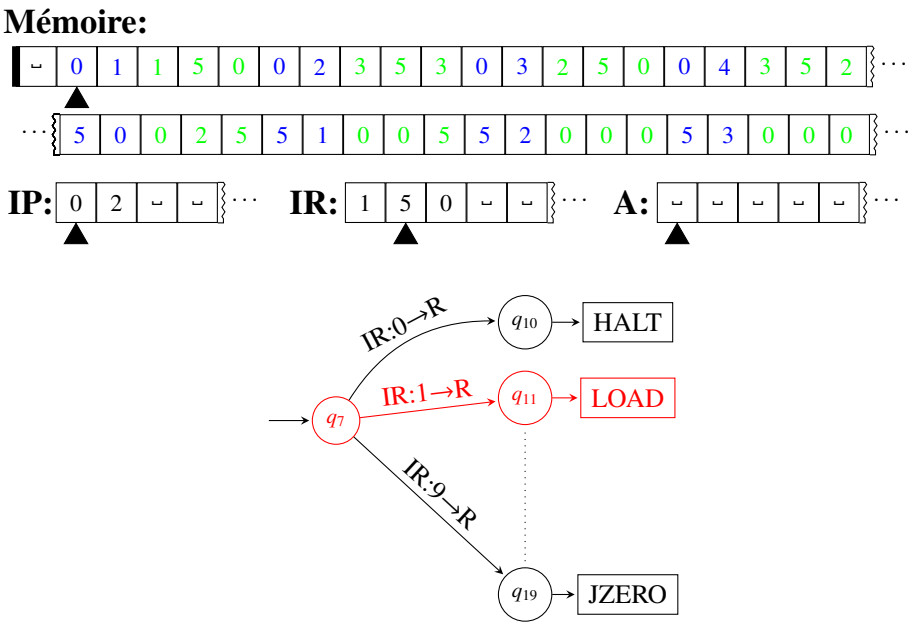
• Chargement du contenu à l'adresse de IP dans l'IR



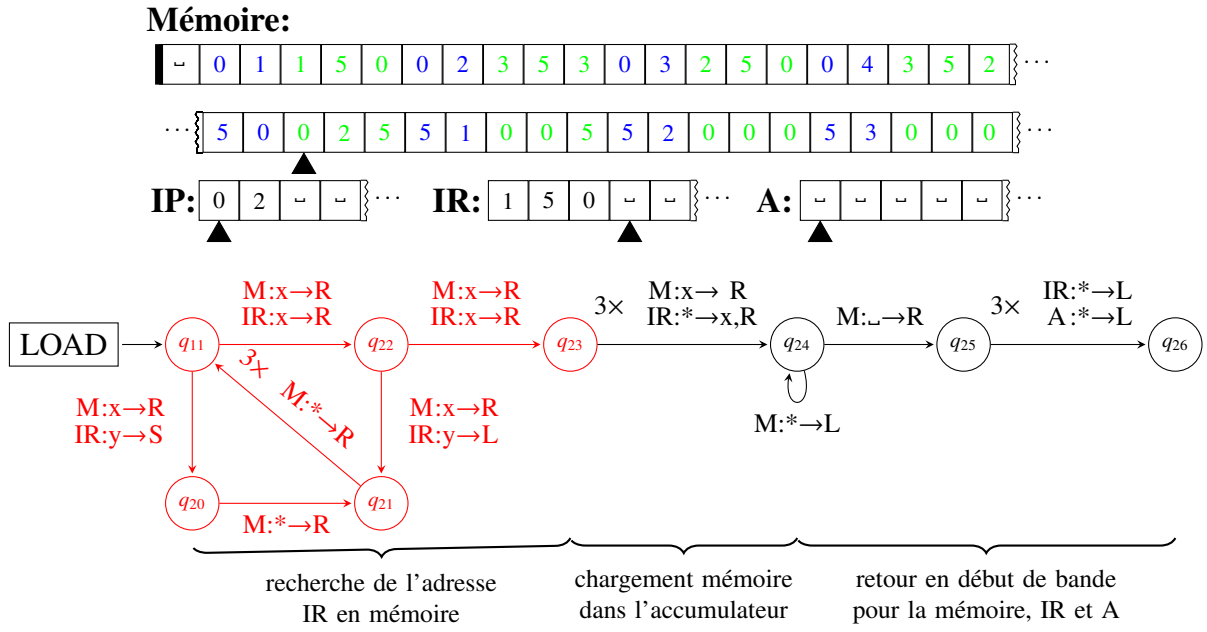
• Réinitialisation position des curseurs et incrémentation de l'IP.



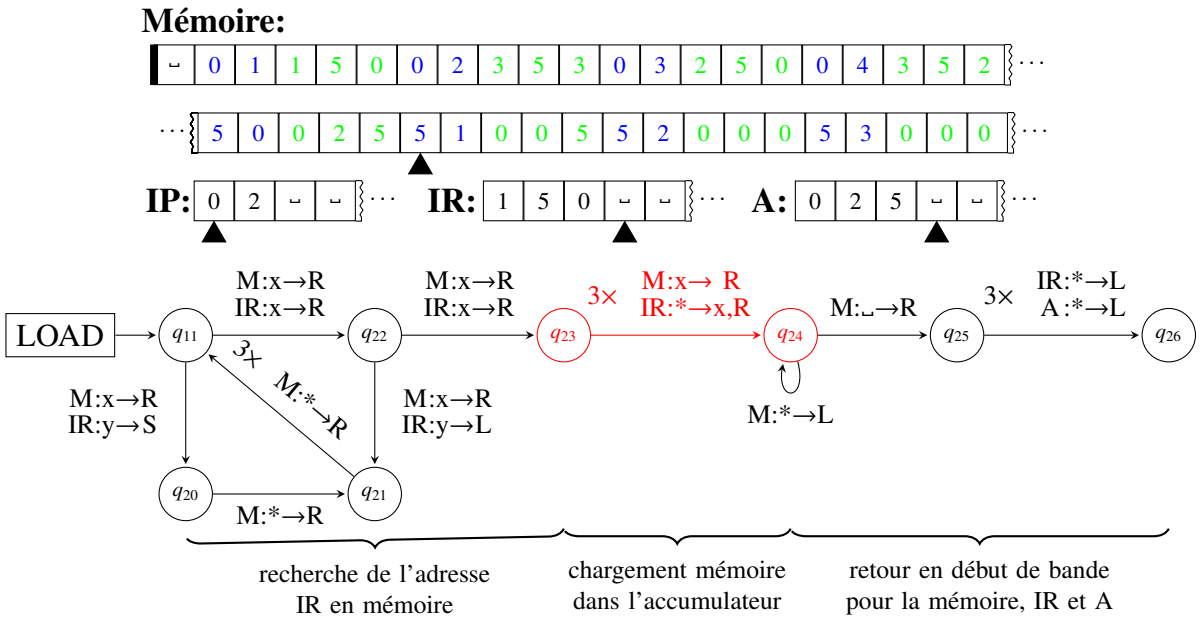
• Décodage de l'instruction à exécuter
code 1 dans la cellule 0 de l'IR = `LOAD`



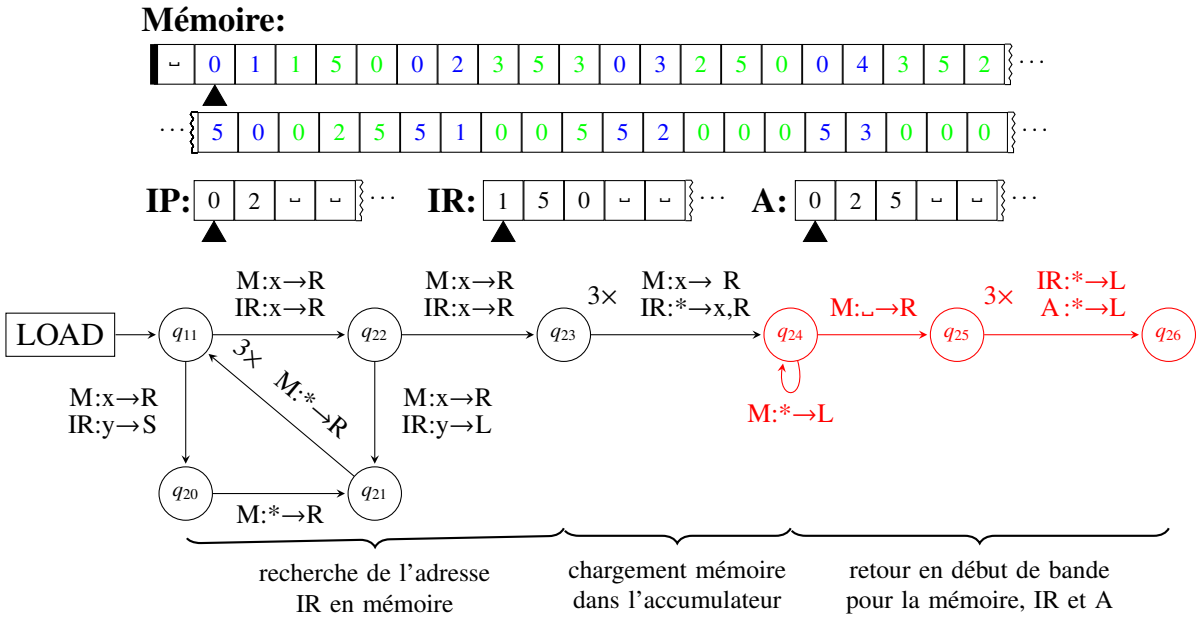
• Exécution de l'instruction décodée (`LOAD`)
recherche de l'adresse (cellule 1 et 2 = 50) dans IR = `LOAD`.



• Exécution de l'instruction décodée (ici `LOAD`)



• Exécution de l'instruction décodée (ici `LOAD`)



et on continue ainsi jusqu'à ce que la machine s'arrête.

6 Résumé

Nous avons vu dans ce cours que :

- la machine de Turing est modèle très puissant pour représenter les algorithmes,
- les langages récursivement énumérables sont les langages pouvant être reconnu par une MT ; les mots hors langages peuvent faire boucler la machine,
- toutes les variations des machines de Turing sont équivalentes.

Chapitre V

Complexité temporelle

Introduction

On se concentre maintenant sur des problèmes décidables ou calculables (à savoir à des algorithmes qui s'arrêtent toujours), à savoir :

- pour une MTD, elle s'arrête toujours en acceptant ou en rejetant.
- pour une MTND, toutes les branches d'exécutions s'arrêtent toujours en acceptant ou en rejetant.

Donc, on ne considère plus que les problèmes pour lesquels il est possible d'écrire des algorithmes qui les **décident**.

On s'intéresse au **temps** nécessaire pour résoudre ces problèmes.

On va donc maintenant introduire :

- faire un rappel sur les asymptotiques,
- se demander comment mesurer le temps d'exécution d'un programme sur une entrée,
- définir le comportement asymptotique du temps d'exécution,
- la notion de complexité temporelle.
- les différentes classes de complexité (**P**, **NP**, ...) ainsi que les propriétés qui leurs sont associées.

1 Asymptotiques

Soit une fonction $f(n)$ qui décrit le comportement d'un certain phénomène en fonction d'un paramètre n. Par exemple, dans notre cas, $f(n)$ serait le temps qu'un programme met pour trier un tableau d'éléments.

Ce temps dépend non seulement de la taille du tableau, mais du tableau lui-même (par exemple, ce dernier peut être partiellement trié, et cette caractéristique rend le tri plus rapide).

L'idée d'asymptotique est de s'intéresser au comportement de la fonction f quand n devient plus grand, en se débarrassant des caractéristiques de f (par exemple, des oscillations) qui gênent l'analyse, et en la comparant à une autre fonction g exempte de ces scories.

Par exemple :

- f et g sont du même ordre de grandeur.
- majorer f par g : f toujours moins grande que g.
- minorer f par g : f toujours plus grande que f.
- f négligeable devant g.

1.1 Majoration asymptotique (grand O)

Cette partie est un rappel. Voir le début du chapitre précédent pour les exemples.

Définition V.1 (grand O)

Soit f et g des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$.
$f(n) = O(g(n))$ si $\exists c > 0, n_0 \in \mathbb{N}^* \mid \forall n \geq n_0, f(n) \leq c.g(n)$
se lit : g est une borne asymptotique supérieure pour f.
se comprend : f ne grandit pas plus vite que g.

Cette définition s'interprète de la manière suivante :

- On borne supérieurement une fonction $f(n)$ par une fonction $c.g(n)$.
- La constante c ne dépend pas de n. Il n'y a aucune contrainte sur son ordre de grandeur (pourrait être 10^9).
- Le n_0 est le n à partir duquel la relation $f(n) \leq c.g(n)$ est vérifiée.
  On tient compte ainsi de la tendance générale quand n devient assez grand.

1.2 Domination asymptotique (petit o)

Définition V.2 (petit o)

On écrit $f(n) = o(g(n))$ si :
$$\forall \varepsilon > 0, \exists n_0, \forall n > n_0, f(n) \leq \varepsilon g(n).$$
se lit : f est dominée asymptotiquement par g.
se comprend : f est négligeable devant g.

La notation o permet donc d'indiquer des termes qui peuvent être négligés car trop petit devant le terme principal.

Une autre façon d'interpréter cette notation est :
$$f(n) = o\left(g(n)\right) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

À savoir, pour n assez grand, $f(n)$ est négligeable devant $g(n)$.

Exemple 55: $f(n) = \log(n)$ est dominée par $g(n) = n$. On a bien $\lim_{n\to\infty} \frac{\log n}{n} = 0$.

2 Référentiel de mesure temporelle

On remarque que :

- sur une machine de Turing, en comptant le nombre de transitions,
- sur un processeur classique, en nombre de cycles ou d'instructions du processeur.
  par exemple, en utilisant un compteur haute résolution, ou un outil de profiling.

Mais comment avoir une mesure qui représente quelque chose ?

Cela dépend sans doute :

- du type de machine utilisée (variation de machines de Turing, type de processeur, architecture, ...)

- du fait que la machine soit déterministe ou pas (comment compter les transitions ?)

Étudions ce problème sur quelques exemples.

2.1 Temps d'exécution sur une machine déterministe

Donnons une première définition du temps d'exécution.

Définition V.3 (Temps d'exécution d'une machine de Turing M sur une entrée w)

Le **temps d'exécution** de $M(w)$ est le nombre de transitions nécessaires à M pour traiter la chaîne d'entrée w.

Noter que ce résultat est seulement valide pour l'algorithme M appliqué sur l'entrée w.

Lors du comptage du nombre de transitions, on utilisera des majorations asymptotiques. A savoir, lorsqu'on indique qu'une étape prend $O(n)$ transitions pour réaliser une tache sur un mot de taille n, cela signifie donc que cette tache nécessite au plus p parcours de la bande (où p ne dépend pas de n ; par exemple $p = 4$).

La majoration asymptotique nous permettra donc de donner l'ordre de grandeur du pire temps pour l'exécution de la tache.

Exemple 56: Soit le langage $A = \{0^k 1^k | k \geq 0\}$
Ce langage est décidable.
Une MT simple bande qui décide A est :

$M_1(\langle w \rangle) =$

```
1 pour chaque cellule de la bande
2 |   si on trouve un 0 à droite d'un 1 alors rejeter
3 tant que il reste des 0 sur la bande
4 |   barrer un 0
5 |   si il reste un 1 alors le barrer sinon rejeter
6 si il reste des 1 alors rejeter sinon accepter
```

Exécutions :

w	00001011
1-2	rejeter

w	00011111
1-2	00011111
3-4	x00x1111
3-4	xx0xx111
3-4	xxxxxx11
5	rejeter

w	00001111
1-2	00001111
3-4	x000x111
3-4	xx00xx11
3-4	xxx0xxx1
3-4	xxxxxxxx
5	accepter

Quel est le temps mis par la MT M_1 pour décider si son entrée $w \in A$?
Pour cette MT,

- ce temps se mesure en nombre de transitions (= de pas).
- ce dernier dépend de la longueur n de l'entrée w ($n = |w|$).

Analyse de M_1 :

1-2 : prend au plus $O(n)$ pas.
3-5 : répétition au plus $n/2$ fois de $O(n)$ pas $\Rightarrow O(n^2)$.
6 : prend $O(n)$ pas.

Donc, M_1 prend $O(n^2)$ pas pour décider si $w \in A$.

Peut-on trouver une autre MT qui décide le même langage plus rapidement ?

Exemple 57: Soit le code suivant d'une machine de Turing :

$M_2(\langle w \rangle) =$

```
1 pour chaque cellule de la bande
2 |   si on trouve un 0 à droite d'un 1 alors rejeter
3 tant que il reste des 0 sur la bande
4 |   si le nombre de 0 et de 1 restant est impair alors rejeter
5 |   barrer un 0 sur deux
6 |   si il reste un 1 alors barrer un 1 sur deux sinon rejeter
7 si il reste des 1 alors rejeter sinon accepter
```

Exécutions :

w	`00001111`
1-2	`00001111`
3-6	`x0x0x1x1`
3-6	`xxxxxxxx`
7	accepter

w	`000000111111`
1-2	`000000111111`
3-6	`x0x0x0x1x1x1`
3-6	`xxx0xxxxx1xx`
3-6	`xxxxxxxxxxxx`
7	accepter

w	`00111111`
1-2	`00111111`
3-6	`x0x1x1x1`
3-6	`xxxxx1xx`
7	rejeter

Pour 0^k1^k, à chaque exécution de la boucle 3,

- pour M_1, un seul 0 et un seul 1 sont barrés.
  $\Rightarrow$ la boucle ligne 3 est exécutée k fois.
- pour M_2, le nombre de 0 et de 1 est divisé par 2.
  $\Rightarrow$ la boucle ligne 3 est exécutée $\log_2 k$ fois.
  Rappel : doubler k fois $\Rightarrow 2^k$ fois plus grand.

Analyse de M_2 : $n = |w|$

1-2 : prend au plus $O(n)$ pas.
3-6 : répétition au plus $O(\log n)$ fois de $n/2$ pas $\Rightarrow O(n \log n)$.
7 : prend $O(n)$ pas.

Le temps d'exécution de M_2 est $O(n \log n)$.

Peut-on trouver une autre MT qui s'exécute encore plus rapidement en utilisant plusieurs bandes ?

Exemple 58: (version multibande de $A = \{0^k1^k | k \geq 0\}$) Avec une MT à deux bandes :

$M_3(\langle w \rangle) =$

```
1 pour chaque cellule de la bande 1
2 |   si on trouve un 0 à droite d'un 1 alors rejeter
3 pour chaque cellule de la bande 1 contenant un 0
4 |   déplacer le 0 de la bande 1 vers la bande 2
5 pour chaque cellule de la bande 1 contenant un 1
6 |   barrer le 1 sur la bande 1
7 |   si il reste un 0 sur la bande 2 alors le barrer sinon rejeter
8 si il reste des 0 sur la bande 2 alors rejeter sinon accepter
```

Exécutions :

	bande 1	bande 2
w	00001111	________
1-2	00001111	________
3-4	____1111	0000____
5-7	____xxxx	xxxx____
8	accepter	

	bande 1	bande 2
w	00000111	________
1-2	00000111	________
3-4	_____111	00000___
5-7	_____xxx	00xxx___
8	rejeter	

Analyse de M_3 :

1-2 : prend au plus n pas.
3-4 : prend au plus n pas.
5-7 : prend au plus n pas.
8 : prend au plus n pas.

Le temps d'exécution de M_3 est en $O(n)$.
Si la chaîne est acceptée, il n'est évidemment pas possible de faire plus rapide (puisqu'il faut lire l'entrée et que celle-ci est de taille n).

Conclusion :

Les MTs simple bande et multi-bandes ont donc :

- la même puissance en terme de calculabilité.
  ils peuvent résoudre les mêmes problèmes.
- une puissance différente en terme de complexité.
  ils ne les résolvent pas à la même vitesse.

Il faut standardiser les mesures. On utilisera donc le modèle standard simple bande d'une MT pour calculer les vitesses d'exécution.

On obtient donc comme mesure de référence de temps pour un algorithme déterministe (codé dans la machine de Turing M) :

Définition V.4 (Temps d'exécution déterministe d'un algorithme sur une entrée)

Le **temps d'exécution déterministe** de $M(w)$ est le nombre de transitions nécessaires à pour une machine de Turing déterministe M simple bande codant l'algorithme pour traiter la chaîne d'entrée w.

Mais j'ai calculé la vitesse de mon algorithme sur une MT multibande, comment connaître la vitesse équivalente sur une MT classique ?

Théorème V.1 (Pénalité engendrée par l'utilisation d'une MT multibande)

Soit $t(n)$ une fonction telle que $t(n) \geq n$.
Pour toute MT à k-bandes qui s'exécute en temps $t(n)$ sur une entrée de longueur n, il existe une MT à une bande qui s'exécute en temps $O(k^2 t(n)^2)$.

Démonstration:

Soit M_k une MT à k bandes qui s'exécute en temps $t(n)$.
Construisons une MT à une bande qui s'exécute en temps $O(k^2 t(n)^2)$.
On a vu comment simuler M_k sur M :

- M stocke sur sa bande les k bandes de M_k en les séparant par #.
- position du pointeur sur une bande = symbole marqué (une marque par bande).

Simulation de M comme M_k :

- parcourir la bande pour lire les caractères sous chaque pointeur.
- parcourir la bande pour mettre à jour le caractère et la position de chaque pointeur.
- si l'on dépasse l'extrémité, on ajoute un espace en décalant le contenu de la bande.

Portion active des bandes :

- M_k s'exécute en temps $t(n)$, chacune de ses bandes accèdent **au plus** les $t(n)$ premières cellules.
- M utilise donc **au plus** les $k \times t(n) + k + 1 = O(kt(n))$ premières cellules.
  $k + 1$ = les séparateurs de bandes.

M fait **à chaque pas** (dans le pire des cas) :

1. parcourir la bande pour lire les caractères sous chaque pointeur.
   prend au plus un temps $O(k.t(n))$ où $t(n)$ est le temps d'exécution.
2. ajouter un espace à chaque bande
   un ajout de un espace = un parcours de la bande en temps $O(kt(n))$.
   au pire, k espace sont ajoutés.
3. mettre à jour la position de chaque pointeur et du caractère pointé.
   prend au plus un temps $O(kt(n))$.

Comme ceci est réalisé $O(kt(n))$ fois
$\Rightarrow$ l'exécution de M se fait en temps $O(k^2t(n)^2)$. □

Exemple 59: de conversion de temps d'exécution Si un programme met un temps $t(n) = O(n)$ sur une machine à $k = 2$ bandes, ce programme mettra un temps $O(2^2O(n)^2) = O(n^2)$ sur une MT à une bande.

2.2 Temps d'exécution sur une machine non déterministe

Comment définir le temps d'exécution dans le cas d'une machine de Turing non-déterministe ?

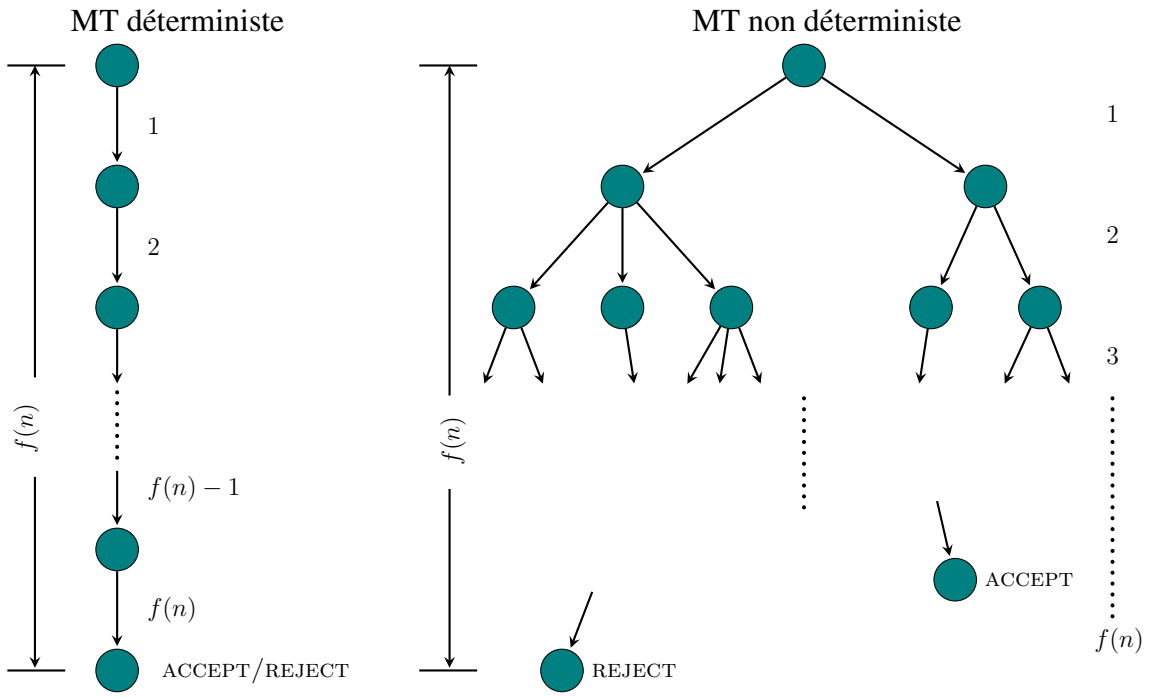
Définition V.5 (Décideur dans le cas d'une MTND (rappel))

une MTND M est un décideur si l'évaluation de toutes ses branches s'arrête pour toutes les entrées.
$\Rightarrow$ M décide $\mathcal{L}(M)$.

Définition V.6 (Temps d'exécution d'une MTND)

le **temps d'exécution** t de M sur l'entrée w est le nombre maximum de transitions que M traverse dans n'importe laquelle de ses branches de calcul sur l'entrée w.
t est le temps d'exécution de la branche la plus longue de $M(\langle w \rangle)$.

Comparaison de la complexité temporelle : ci-dessous, on note n la longueur de l'entrée w et $f(n)$ le temps d'exécution mesuré.



Mais comment comparer un temps d'exécution obtenu sur une machine de Turing non déterministe à celui d'une machine déterministe ?

Théorème V.2 (Pénalité engendrée par l'utilisation d'une MT non déterministe)

Soit $t(n)$ une fonction telle que $t(n) \geq n$. Pour toute MT non déterministe M' qui s'exécute en temps $t(n)$, il existe une MT déterministe M à une bande qui s'exécute en temps $2^{O(t(n))}$.

Démonstration:

Pour M', chaque branche à un temps de calcul au plus $t(n)$. Chaque nœud de M' a au plus b fils (= nb maximum de choix de transitions). Donc, le nombre de feuilles de M' est au plus de $O(b^{t(n)})$. Le temps de traitement d'une branche est au plus de $O(t(n))$. L'ordre de parcours des nœuds importe peu : dans le cas le pire, on passe dans toutes les branches. Donc, le temps total de simulation de M' par M est en temps $O(t(n)b^{t(n)})$. Or $O(t(n)b^{t(n)}) = O(e^{t(n)\log b}) = O(2^{t(n)\log b/\log 2}) = 2^{O(t(n))}$. □

Les MT non déterministes, s'il était possible de les réaliser physiquement, permettrait donc un gain exponentiel par rapport aux MTs classiques.

3 Temps d'exécution d'un algorithme

3.1 Définition

On veut maintenant une mesure de la performance de l'algorithme M indépendamment de son entrée.

Comment faire ?

Il faut avoir une mesure de la taille n du problème que cet algorithme va traiter.

Exemples 60:

- opération sur un tableau de p éléments : $n = p$
  Exemple : recherche du max, tri, ...
- opération sur un matrice de $p \times p$ éléments : $n = p \times p$
  Exemple : transposition, inversion, ...
- opération sur un graphe non orienté de p nœuds : plus compliqué dans ce cas.
  Exemple : recherche d'un chemin entre deux sommets, recherche d'un chemin Hamiltonien, ...
  Il y a au minimum p arêtes et au maximum p^2.
  Clairement, la complexité va aussi dépendre du nombre d'arêtes.
  On prend $n = p + p \times p$.
- ...

Dans tous les cas ci-dessus, la taille du problème à traiter est en fait la taille du codage de l'entrée.

Relativement intuitif : plus la quantité de données mise en entrée d'un algorithme est importante, plus le temps de traitement devrait être important.

Si ce n'est pas le cas, cela signifie qu'une partie des données mise en entrée de l'algorithme n'est pas nécessaire à son exécution.

Exemple 61: Pour un algorithme M de calcul du nombre de sommets dans un graphe $\langle V, E\rangle$, il est inutile de passer la liste des arêtes $\langle E\rangle$.
On exécute $M(\langle V\rangle)$ en ne passant que la liste des sommets.

On définit donc le temps d'exécution d'un algorithme de la manière suivante :

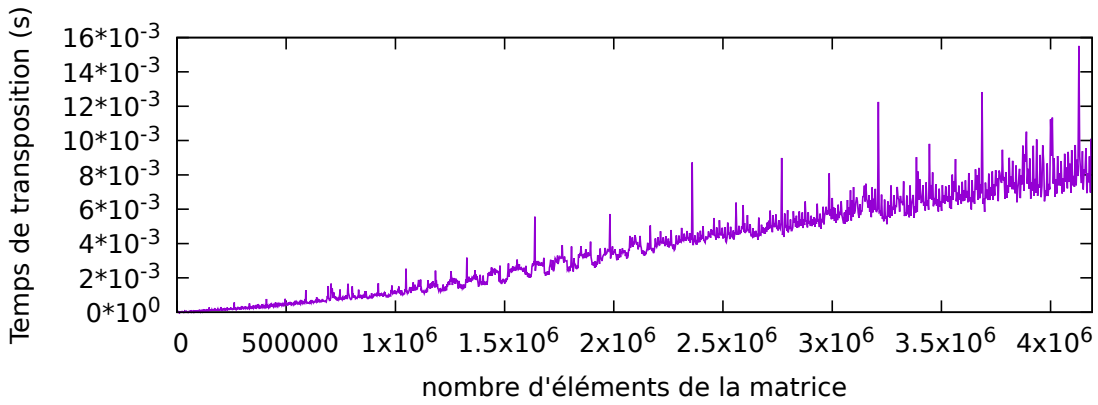
Définition V.7 (Temps d'exécution d'un algorithme M)

Le **temps d'exécution** de M est la fonction $f : \mathbb{N} \to \mathbb{N}$ où $f(n)$ est le nombre maximum de transition (resp. nombre de cycles) nécessaires à M pour traiter une chaîne d'entrée (resp. entrée) de longueur n.

$f(n)$ représente donc le temps **au pire** qui sera nécessaire pour résoudre un problème un problème de taille n (aucune entrée de taille n ne passe plus de temps).

Remarques : on dit alors que l'algorithme M s'exécute en temps $f(n)$ (ou M est à temps $f(n)$).

A noter que les temps d'exécution sur un ordinateur sont souvent perturbés par des mécanismes internes (nombre de registres, mémoire locale sur le CPU ...) et externes (nombre de niveaux de cache, taille des caches, taille des blocs, ...).



L'exemple ci-dessus présente le temps d'exécution de la transposition d'une matrice $p \times p$ en fonction de n.

On constate que :

- la progression est approximativement linéaire (en fonction de la taille de l'entrée $n = p^2$).
- il existe certaines tailles de matrice "maudites" : il y a, à intervalle régulier, des pics correspondants à des cas où les caches font défaut en cascade.

3.2 Comportements asymptotiques

Comme le temps d'exécution $f(n)$ d'un algorithme peut être très irrégulier, on s'intéresse au comportement asymptotique de cette fonction.

Les **ordres de grandeur** asymptotiques les plus courants sont les suivants :

Grand O	vitesse de croissance	exemple
1	constante	10
$\log\log n$	loglogarithmique	
$\log n$	logarithmique	$\log(n^2)$
$\log(n)^c, c > 1$	polylogarithmique	$\log(n)^2$
$n^c, c \in (0,1)$	puissance fractionnaire	$n^{1/2}$
n	linéaire	$3n$
$n \log n$ $\log(n!)$	quasi-linéaire	
n^2	quadratique	$2n^2$
$n^c, c > 1$	polynomiale	$n^{3/2}$
e^n $c^n, c > 1$	exponentielle	2^n
$e^{n^c}, c > 1$ n^n	(poly)exponentielle	$2^{n^2}, n^n$ $n!$

Note : formule de Stirling $n! \sim \sqrt{2\pi n}(n/e)^n$

Exemples d'ordres asymptotiques communs :

- recherche dans une table de correspondance : 1
- recherche dans un arbre binaire équilibré : $\log n$
- recherche séquentielle : n
- tri : naïf = n^2, à bulle = $n \log n < n^2$
- voyageur de commerce : naïf = $n!$, mais possiblement 2^n

Exemple possiblement contre-intuitif :

Considérons la fonction d'addition de deux entiers suivantes :

$\textsc{SillyAdd}(\langle a, b\rangle) =$

```
r=a
POUR i=1 à b : r = r + 1
RETOURNER r
```

Comme déjà indiqué, on utilise un codage raisonnable des entiers a et b non signés. Prenons ici le binaire.

Soit $n = |\langle a, b\rangle|$. En négligeant le séparateur, supposons $n = |\langle a\rangle| + |\langle b\rangle|$. Le maximum de boucles est obtenu pour $|\langle a\rangle| = 0$ et $|\langle b\rangle| = n$.

Donc, si $n = 32$, on est assuré que l'algorithme effectuera $2^{32} - 1 \simeq 4.10^9$ de boucles au pire (nombre maximum de transitions).

A savoir, une entrée de taille n engendre au pire 2^n boucles, soit un nombre exponentiel par rapport à l'entrée. Noter ici que l'on ne compte même pas le coût de l'incrémentation. **L'ordre asymptotique de cet algorithme est exponentiel.**

Évidemment, une addition classique (comme vue dans le chapitre sur les fonctions calculable) est, au pire, à temps $10.n^2$ (= temps $T(n) = n$ sur une MT à $k = 2$ bandes - $5kT(n)^2$ sur une machine à une bande).

Lorsqu'un problème est décidable, si la taille de l'entrée est assez petite, alors, la plupart du temps, il est toujours possible de trouver une solution en un temps raisonnable.

Exemple : pour l'algorithme **SillyAdd** précédent, s'il est envisageable de l'utiliser pour $n = 32$, il devient très rapidement inutilisable lorsque n devient plus grand.

On va donc s'intéresser au **comportement asymptotique** du temps d'exécution d'un algorithme, à savoir à quelle vitesse son temps de calcul augmente lorsque la taille du problème à traiter augmente lui-aussi.

Afin de les quantifier, on va utiliser des ordres asymptotiques (= fonctions de majoration).

Ceci nous permettra de définir des classes particulières de problèmes pour lesquels, même si des algorithmes existent pour les résoudre, ils sont **par nature infaisables** (= prennent trop de temps à calculer) dès que la taille du problème augmente.

3.3 Ordres de grandeur

Temps de calcul (taille du problème en fonction de la complexité)

On suppose qu'un instruction s'exécute en $1\mu s = 10^{-6}s$ (rappel : $1ms = 10^{-3}s$)

ordre de grandeur \ $n =$	10	20	30	40	50	60
n	10μs	20μs	30μs	40μs	50μs	60μs
n^2	100μs	400μs	900μs	1.6m	2.5ms	36ms
n^3	1ms	8ms	27ms	64ms	125ms	216ms
n^5	100ms	$3,2$s	$24,3$s	1,7 min	5,2 min	13 min
2^n	$1ms$	$1s$	$17,6$ min	$12,7$ jours	$35,7$ ans	$36,6$ Kan
3^n	$59ms$	$58'$	$6,5$ ans	$385,5$ Kan	22.7 Gan	$1,3$ Tan

Kan = 1000 ans = un millénaire.
Man = 1.000.000 ans = un million d'années.
Gan = 1.000.000.000 ans = un milliard d'années.
Tan = 1.000.000.000.000 ans = mille milliard d'années.

Rappel : on estime que l'univers n'a "que" 13.8 milliard d'années.

Pour les ordres de grandeur :

- de n^k, le temps d'exécution est raisonnable.
- de k^n, le temps d'exécution devient rapidement trop long dès que la taille du problème augmente.

L'illusion de la puissance :

Attendez ! Donnez-moi un Cray Titan à 17.59 Pflops et je vous le fais.

Rappel : 1 PFlops = 1 petaflops = 1 million de Gflops.

Si la puissance de mon ordinateur est multipliée par ...

Pour un même temps de calcul, la taille n du problème peut augmenter de :

ordre de grandeur	$n =$	10	10^2	10^3	10^6	10^9
	n	$\times 10$	$\times 10^2$	$\times 10^3$	$\times 10^6$	$\times 10^9$
	n^2	$\times 3,16$	$\times 10$	$\times 31,6$	$\times 1000$	$\times 31623$
	n^3	$\times 2,15$	$\times 4,64$	$\times 10$	$\times 100$	$\times 1000$
	n^5	$\times 1,58$	$\times 2,51$	$\times 3,98$	$\times 15,8$	$\times 63$
	2^n	$+3,32$	$+6,64$	$+9,96$	$+19,9$	$+29,89$
	3^n	$+2,09$	$+4,19$	$+6,29$	$+12,57$	$+18,89$

Pour les ordres de grandeur :

- de n^k, la taille du problème peut être multipliée.
- de k^n, la taille du problème progresse de quelques unités.

Autrement dit, dans ce dernier cas, l'augmentation de la puissance de calcul ne permet pas d'augmenter significativement la taille du problème qu'il est possible de traiter.

3.4 Temps d'exécution des opérations standards

On s'intéresse maintenant l'ordre de grandeur du temps que mettent les opérations arithmétiques de base à s'exécuter.

Soit deux entiers x et y tels que $x \leq y$, et on note $n = \log(x)$ et $m = \log(y)$.

Le logarithme est pris dans la base dans laquelle les opérations sont effectuées (exemple : en binaire, $\log = \log_2$). Les explications sont données en binaire mais se généralisent à toute base.

Exemples 62: si on veut effectuer $140 + 65 = 205$

- en binaire, on effectue : $10001100 + 1000001 = 11001101$. On a $n = \lceil \log_2(143) \rceil = 8$ et $m = \lceil \log_2(65) \rceil = 6$. L'addition se fait bit à bit. On a besoin de n opérations avec gestion de retenue.
- en base 16, on effectue : $8c + 41 = cd.$. On a $n = \lceil \log_{16}(143) \rceil = 2$ et $m = \lceil \log_{16}(65) \rceil = 2$. L'addition se fait en hexadécimal. On a besoin de deux opérations avec gestion de retenue.

- sur une ALU (Unité de calcul logique et arithmétique d'un processeur) de 64 bits, les calculs s'effectuent par paquets de 64 bits, donc en base 2^{64}. On retrouve bien que le coût de l'addition de deux entiers de 64 bits est 1.

Les opérations sont bien définies à un facteur multiplicatif près suivant la base utilisée.

On a pour les opérations arithmétiques de base :

- **addition :** $x + y$ prend un temps $O(\max(n, m))$ (donc au pire $O(n)$).
  Résultat sur $n + 1$ bits au plus.
- **soustraction :** idem addition (en prenant le complément à deux + 1).
- **multiplication :** $x \times y$ en temps $O(n \times m)$), au pire $O(n^2)$.
  Résultat sur $n + m + 1$ bits au plus.
  Multiplier par 2^i = faire i décalage à gauche. Donc $x \times y$ = sommer $\log(y)$ fois (au plus) un nombre de $\log(x)$ bits décalé à gauche.
- **division entière :** x/y en temps $O(m \times (n - m + 1))$.
  Résultat sur $n - m$ bits.
  Même idée que la multiplication (sauf que l'on fait des soustractions, et que l'on arrête dès que le reste est inférieur au numérateur : d'où le $n - m + 1$).
  cas le pire : $m = n/2$, on est en $O(n^2)$.

Le calcul modulo p permet d'affirmer que tout calcul ci-dessus sera toujours effectué au maximum sur p bits (donc $n = m = p$), avec réduction préalable modulo p si n ou m est plus grand que p.
Pour l'algorithme d'Euclide : $O(n)$, en version étendue : $O(n \times m)$. Résultat sur m bits au plus.

Exemple 63: algorithme d'exponentiation rapide de x^y Étudions ce cas en binaire. Remarquons tout d'abord que si $x = 2^n$ et $y = 2^m$, alors $x^y = (2^n)^{2^m} = 2^{n2^m}$

En conséquence, x^y possède $n2^m$ bits (à un près), soit un **nombre exponentiel de bits**.

L'algorithme d'exponentiation rapide : l'algorithme répète m fois, une élévations au carré (à chaque fois) et une multiplications par x (pour chaque bit de y à 1).

Le nombre total d'opérations est en $O(n^2 \times 2^p)$. La difficulté consiste à tenir compte de l'augmentation progressive du nombre de bits nécessaires pour stocker le résultat de chaque opération, ce qui augmente la complexité des opérations suivantes.

Remarque 28:

Des algorithmes plus rapides existent mais ils ne deviennent efficaces que pour n assez grand (asymptotique) ; autrement dit pour les grands nombres (par exemple 4096 bits).

Pour la multiplication de deux nombres à n bits :

- classique = $O(n^2)$,
- Karatsuba (1962) = $O(n^{\log_2 3})$,
- Schönhage-Strassen (1971) = $O(n \log(n) \log\log(n))$
- Harvey-van der Hoeven (2019) = $O(n \log(n))$,

On pense que ce dernier résultat ne peut pas être amélioré (*i.e.* le gain ne se fera pas sur l'ordre de grandeur).

Mais pourquoi ne serait-il pas possible de faire encore mieux ?

4 Faisabilité

Intéressons-nous maintenant à la faisabilité des problèmes, et donnons une première définition qui semble raisonnable :

Définition V.8 (Problème infaisable)

Un problème est **infaisable** si le comportement asymptotique du temps d'exécution du meilleur algorithme connu pour le résoudre est exponentiel ou subexponentiel.

Mais que signifie subexponentiel ?

Définition V.9 (fonction subexponentielle)

Une fonction $f(x)$ est subexponentielle si :

- $\forall \alpha > 0, \lim_{x\to\infty} f(x)/x^\alpha = +\infty$
- $\lim_{x\to\infty} \log(f(x))/x = 0$

à savoir elle croit beaucoup plus vite que tout polynôme, mais moins vite qu'une exponentielle.

Exemple 64: $f(x) = x^{\log x}$ est une fonction subexponentielle.
La majorité des méthodes de cryptographie moderne ont des difficultés d'attaque d'ordre subexponentiel.

Pour ce type d'algorithme :

- on peut l'exécuter pour une taille d'entrée petite.
- dès que la taille du problème augmente, le temps d'exécution devient tel qu'il devient inutile de lancer l'exécution :
  - il est déraisonnable d'attendre le temps nécessaire (et d'assurer les ressources qui permettre d'assurer le programme de s'exécuter),
  - avec le rythme d'évolution (exponentiel) des puissances des machines (s'il ne change pas), il est plus avantageux de lancer le programme sur une future machine qui n'existe pas encore, que sur une machine actuelle.

    En clair, attendre que l'évolution de la technologie fasse progresser la valeur du n correspondant à un temps d'attente raisonnable.
- La paradigme quantique implique que certains problèmes considérés actuellement comme infaisable pourront se révéler faisable.

Exemple 65: Un problème a un comportement asymptotique en 2^n.
Pour $n = 40$, j'ai un résultat en moins de 13 jours, ce qui est raisonnable.
Je veux maintenant faire un calcul pour $n = 60$. La table précédente nous indique que le temps de calcul sera de l'ordre de 36.600 ans, ce qui l'est beaucoup moins.
L'une des expressions de la loi de Moore dit que la puissance des ordinateurs double tous les 2 ans. Supposons que cette loi reste vraie dans le futur (par amélioration de la technologie, ou augmentation du nombre de transistor, des fréquences, etc ...)
En conséquence, la puissance de calcul progresse de 2^p en $2p$ années. Donc, si j'attends $2 \times 20 = 40$ ans (*i.e.* $p = 60 - 40$), je pourrais exécuter mon calcul avec $n = 20$ en 13 jours.
On notera que pour $n = 1000$, le problème restera infaisable encore longtemps.

Remarque 29:
La progression de la puissance de calcul devrait finir par se heurter à des murs (taille des circuits, fréquences, énergétique, ressources, ...). De nombreux problèmes resteront donc très probablement infaisables.

Pour simplifier :

- un problème est exponentiel s'il est infaisable à long terme,
- un problème est subexponentiel s'il est infaisable à court ou à moyen terme.

Mais alors, quand est-ce qu'un problème est faisable avec un ordinateur ?

Définition V.10 (Problème faisable)
Un problème est considéré comme **faisable** si le comportement asymptotique du temps d'exécution du meilleur algorithme connu pour le résoudre est au plus polynomial.

Afin d'étudier plus précisément les comportements asymptotiques des problèmes, nous allons définir des classes de complexité.

Exercice 37 (Factorielle)**.** *Soit la fonction calculable $f(p) = p!$.*

1. *Écrire le programme (en pseudo-code) qui calcule cette fonction.*
2. *En utilisant la formule de Stirling $n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, calculer l'ordre asymptotique du nombre de bits dans $n!$.*
3. *Lorsque je multiplie $(n-1)!$ par n, de combien de bits supplémentaires dois-je disposer afin de stocker le résultat ?*
4. *L'algorithme naïf de multiplication de deux entiers de p bits est de l'ordre de $O(p^2)$ (à noter que l'algorithme de Fürer (2007) est en temps $O(p \log p)$. Quel est le coût de calcul de $n!$ connaissant $(n-1)!$?*
5. *Sachant que pour k grand, $\sum_{i=1}^{k} i^2 \log^2 i = O(k^3 \log^2 k)$, calculer la complexité de $n!$.*
6. *En déduire la complexité de calcul de la factorielle si n est un nombre à q bits.*

5 Théorème de Hiérarchie

5.1 Complexité inhérente

Une hypothèse envisageable serait qu'il n'existe pas vraiment une complexité inhérente à un problème.

Par exemple, s'il existe un problème P pour lequel je connais un algorithme permettant de le résoudre en temps 2^n, il y a peut-être un algorithme plus compliqué qui le résout plus vite en $2^{\sqrt{n}}$, et un autre plus compliqué encore plus vite en $n^{\log n}$ ou en n^{1024}, ... jusqu'à $O(n)$.

Évidemment, plus l'algorithme est complexe, plus son code devient long, possiblement tendant vers l'infini à mesure que l'on s'approche de $O(n)$.

Cette hypothèse vous semble absurde ?

Pour le problème de la multiplication de deux matrices :

- "force brute" en $O(n^3)$ avec la définition du produit matriciel.
- algorithme de Strassen (1969) en $O(n^{2.807})$
- algorithme de Coppersmith-Winograd (1980) en $O(n^{2.5})$
- nouvel algorithme de Coppersmith-Winograd (1987) en $O(n^{2.376})$

- algorithme de le Gall (2014) en $O(n^{2.3728639})$
- algorithme de Alman-Williams (2021) en $O(n^{2.3728596})$
- algorithme de Fawzi (2022) utilisant de l'apprentissage pour l'adapter à la taille de la matrice et à la machine (CPU,GPU,...)
- algorithme FBHHRBNRSSSHK (2023) montrant que l'on peut encore améliorer, pour une matrice 5×5, le record établit par Fawzi de 98 multiplications à 96 dans $\mathbb{Z}^2$.
- ...?

Cet exemple montre donc que, même pour un problème classique, on continue à trouver de nouveaux algorithmes qui permettent d'améliorer les temps de calcul.

5.2 Théorème d'accélération

Définissons tout d'abord une mesure de complexité :

Définition V.11 (mesure de complexité de Blum (simplifiée))

Une mesure de complexité de Blum est une paire (ϕ, Φ) telle que :

- ϕ est une énumération de l'ensemble des fonctions calculables qui, comme les machines de Turing, est un ensemble dénombrable. On notera ϕ_i la $i^{\text{ème}}$ fonction calculable, $\phi_i(w)$ l'exécution de ϕ_i sur l'entrée w.
- Φ est un ensemble de fonctions calculables associé à ϕ représentant l'ensemble de mesures de complexité où $\Phi_i(w)$ représente la complexité de $\phi_i(w)$

Par exemple, Φ peut représenter le nombre de transitions lors de l'exécution. Tout type de mesure peut être choisi : le nombre de cellules de la bande utilisés, le nombre de cycles sur le processeur considéré, ...

Théorème V.3 (d'accélération de Blum (1967))

Pour toute fonction totalement calculable r, il existe une fonction f totalement calculable telle que $\phi_i = f$ implique qu'il existe j tel que $\phi_j = f$ et $\Phi_i(w) > r(w, \Phi_j(w))$ presque partout.

Interprétation : pour n'importe quelle accélération r, il existe une fonction totalement calculable f (dont le code est ϕ_i et la complexité est Φ_i) pour laquelle je peux trouver un autre code ϕ_j dont la complexité est inférieure pour presque tous les w.

1. si on prend par exemple une fonction exponentielle pour r, on a $\Phi_i(w) > 2^{\Phi_j(w))}$, à savoir $\phi_j(w)$ va exponentiellement plus vite que $\phi_i(w)$.
2. "presque partout" signifie que pour un nombre fini d'entrées w, l'algorithme ϕ_j n'accélère pas la fonction f.
3. la dépendance de r à w permet de faire dépendre l'accélération à w.

Le théorème est en fait encore plus général et s'applique aux fonctions partiellement calculables.

Il affirme donc l'existence de fonctions que l'on peut "infiniment" optimiser (en se débarrassant possiblement des cas les plus problématiques).

5.3 Théorème de hiérarchie

Montrons maintenant qu'il existe effectivement une hiérarchie des complexités, à savoir :

- il existe des problèmes qui ne peuvent pas être calculé avec une complexité plus faible,

- en conséquence, ces problèmes peuvent être optimisé jusqu'à, potentiellement, leur complexité minimale.

Mais avant de pouvoir le démontrer, une définition et quelques rappels.

5.3.1 Fonction constructible en temps

Définition V.12 (Fonction constructible en temps)

Une fonction $f(n) \leq n$, non décroissante ($f(n) \leq f(n+1)$) est dite constructible en temps si il existe une fonction calculable F qui calcule $F : 1^n \mapsto 1^{f(n)}$ en temps $f(n)$.

à savoir, si il existe une MT M qui prend en entrée $w = 1^n$ et s'arrête en $O(f(n))$ transitions avec $1^{f(n)}$ sur sa bande.

Remarques 30:

- toutes les fonctions classiques n, n^k, 2^n sont constructibles en temps si $f(n) \geq c.n$ où $c = O(1)$ (sinon on n'a pas suffisamment de temps pour lire le contenu de la bande).
- Inversement, toute fonction en $o(n)$ n'est pas constructible en temps (même raison).
- si f_1 et f_2 sont constructibles en temps, alors $f_1 + f_2$ et $f_1.f_2$ aussi. En conséquence, tout polynôme de $\mathbb{N}[X]$ est constructible en temps.

Exemple 66: Considérons $f(n) = n^3$ en se plaçant sur un transducteur :

$M(w) =$
compter le nombre n de symbole 1 sur la bande
calculer n^3
écrire n^3 fois le symbole 1 sur la bande

Chacune de ses opérations est au plus en n^3 :

- **comptage des 1 :** n incrémentations sur $\log(n)$ bits au plus $= O(n \log n)$
- **calcul de n^3 :** calcul n^2 en $O(\log^2(n))$, calcul de n^3 en $O(\log(n)\log(n^2)) = O(\log^2(n))$. Donc le tout en $O(\log^2(n))$.
- **écriture de 1^{n^3} sur la bande de sortie :** en $O(n^3)$ (décrémentation du compteur n^3 en $O(n^3 \log n^3)$)

Tous les termes sont dominés par la dernière étape en $O(n^3) = O(f(n))$.

La dernière étape nécessairement un temps $f(n) = n^3$ puisqu'il faut écrire n^3 fois un 1 sur la bande de sortie.

Donc $f(n)$ est constructible en temps.

5.3.2 Temps d'exécution d'une MT universelle

On rappelle que :

Théorème V.4 (temps d'exécution d'une MT universelle)

Soit une machine de Turing M.
Notons :

- $T_M(w)$ le temps d'exécution de M sur l'entrée w.
- $poly(n)$ une fonction polynomiale de variable n.

Il existe une MT universelle capable de simuler une machine M sur l'entrée w telle que :

$$T_U(\langle M, w\rangle) \leq poly(|\langle M\rangle|).T_M(w)\log T_M(w)$$

Nous avons déjà vu une variation de ce théorème avec une pénalité en $T \log T$ (cf machine de Turing universelle efficace).

Celui-ci précise que la constante multiplicative peut être majorée par un polynôme qui ne dépend que de la longueur de l'encodage de la machine.

5.3.3 Théorème de hiérarchie temporelle

Théorème V.5 (Hiérarchie temporelle)

Soient :

- G une fonction constructible en temps
- g une autre fonction constructible en temps telle que $g(n) \log g(n) = o(G(n))$

Il existe des langages en temps $G(n)$ qui ne sont reconnus par aucune machine en temps $g(n)$.

Démonstration:

Soient deux fonctions $g(n)$ et $G(n)$, telle que :

- $\lim_{n\to\infty} \frac{g(n)\log g(n)}{G(n)} = 0$ (à savoir $g(n) \log g(n) = o(G(n))$).
- $g(n)$ et $G(n)$ sont constructibles en temps,

Soit l'ensemble :

$$E_G = \{M \in \mathcal{R} \mid \mathcal{L}(M) \text{ en temps } G(n)\}.$$

E_G est l'ensemble des machines décidables en temps $G(n)$.

Donc, E_G contient aussi l'ensemble des machines en temps $g(n)$ (i.e. $E_g \subseteq E_G$).

On veut montrer que cette inclusion est stricte ($E_g \subsetneq E_G$), à savoir qu'il existe des langages qui ne sont pas décidable par n'importe quelle machine en temps $g(n)$.

Considérons la MT D suivante qui utilise la MTU U :

$D(\langle M \rangle) =$

```
CALCULER n = |⟨M⟩|
SIMULER U(⟨M, ⟨M⟩⟩) sur au plus G(n) transitions
SI elle s'est arrêtée
ALORS RETOURNER l'opposé de la décision
SINON ACCEPTER // peu importe pour la démonstration
```

Comme $G(n)$ est constructible en temps (i.e. $G(n)$ peut être calculé à partir d'une entrée de taille n qui est utilisé pour compter les transitions), et que U exécute au plus $G(n)$ transitions, D **est en temps** $G(n)$.

Prenons maintenant une machine de Turing R en temps $g(n)$.

Quel temps prend $D(\langle R \rangle)$ à s'exécuter ? Comme R en temps $g(n)$, la simulation est faite en temps $poly(|\langle R \rangle|)(g(n) \log(g(n)))$

Le terme $poly(|\langle R \rangle|)$ est ennuyeux puisqu'il ajoute un facteur multiplicatif en $poly(n)$, ce qui se produit parce qu'on exécute R sur sa propre description (*i.e.* $n = |\langle R \rangle|$).

En conséquence, la taille de l'entrée (dont dépend le temps d'exécution) est proportionnelle à la description de la machine.

On voudrait pouvoir regarder avoir un comportement asymptotique du temps d'exécution de R.

Remarquons que $\langle R \rangle$ et $\langle R \rangle 10^k$ représentent la même machine de Turing (puisque le padding est ignoré).

Donc pour une entrée de $n = |\langle R \rangle 10^k| = |\langle R \rangle| + k + 1$, et pour n assez grand, $k/n \simeq 1$, on a $poly(|\langle R \rangle 10^k|) = poly(|\langle R \rangle|)$ car seul compte l'encodage effectif de la machine (voir V.4).

Pour toute machine R en temps $g(n)$, il existe donc une constante k_R, telle que $poly(|\langle R \rangle|)(g(n)\log(g(n))) < G(n)$ où $poly(|\langle R \rangle|)$ est une constante "indépendante" de n.

En conséquence, pour tout R en temps $g(n)$, il existe un mot $w_R = \langle R \rangle 10^k$ tel que $D(w_R)$ simule $U(R, \langle R \rangle 10^k)$ en temps $g(n)\log(g(n))$.

Nous pouvons maintenant utiliser l'argument de diagonalisation.

Nous avons construit une MT $D(w)$ où, pour toute machine R en temps $g(n)$, il existe un mot w_R tel que :

- la simulation $U(R, \langle w_R \rangle)$ s'arrête en temps $g(n)\log(g(n)) = o(G(n))$.
- $D(w_R)$ prend la décision opposée de $R(w_R)$.

Toute machine R en temps $g(n)$ reconnaît un langage différent de D, car la décision de D sur w_R est l'opposée de celle de R (diagonalisation).

Comme D est en temps $G(n)$, il existe donc bien dans des langages en temps $G(n)$ qui ne peuvent être reconnus par aucun langage en temps $g(n)$. □

Nous venons donc de démontrer qu'il existe bien des problèmes qui ont une complexité minimale (ils ne peuvent pas être résolus en moins de temps).

Évidemment, on pense que la plupart des problèmes sont de cette nature. Le théorème d'accélération démontre, quand à lui, que ce n'est pas toujours le cas.

6 Classes de complexité en temps déterministe

Grâce au théorème de Hiérarchie (voir théorème V.5), nous savons qu'il y a un sens à classifier les problèmes en fonction de leurs complexités.

On peut donc définir des classes de complexité.

6.1 Définition

On rappelle qu'une machine de Turing M est à temps $t(n)$ si, pour toute entrée w de longueur n, le nombre de transitions que prend M pour décider w est en $O(t(n))$.

A savoir, $t(n)$ est la borne asymptotique supérieure de son temps d'exécution.

Définition V.13 (classe de complexité temporelle)

Soit $t : \mathbb{N} \to \mathbb{R}^+$ une fonction.
La classe de complexité temporelle $\texttt{TIME}(t(n))$ est l'ensemble de tous les langages décidables par une MT déterministe à temps $O(t(n))$.

Exemple 67: $\texttt{TIME}(n^2)$ est contient l'ensemble des langages (*i.e.* des problèmes) décidables par un algorithme s'exécutant sur une machine de Turing en temps $O(n^2)$ pour une entrée de taille n.

On notera que :

- On donne une borne supérieure asymptotique pour le temps d'exécution d'une machine de Turing M. Elle peut être sur-évaluée (majoration trop large).
- De la même façon, si un langage $L \in \mathbf{TIME}(n^3)$, cela signifie qu'il existe une MT M en temps n^3 qui reconnaît L.

  Néanmoins, il existe peut-être un autre algorithme M' qui reconnaît L de manière plus efficace (par exemple en temps $n^2 \log n$).
- Pour un langage A (voir par l'exemple 56), y-a-t-il une contradiction entre $A \in \texttt{TIME}(n^2)$ et $A \in \texttt{TIME}(n \log n)$?

  A appartient aux deux. Il est toujours possible de faire plus lent (par exemple en faisant autre chose).

Nous verrons par la suite que, souvent, il n'est pas nécessaire d'obtenir la meilleure borne possible avec le meilleur algorithme possible pour se faire une idée de la complexité d'un problème.

On remarquera que la notion de complexité est en général lié à un algorithme, et non directement à un problème.

Lorsque que l'on dit qu'un problème possède une certaine complexité, on signifie par là :

- soit qu'il a été démontré (mathématiquement) qu'il n'existe pas d'algorithme plus rapide pour ce problème,
- soit qu'il s'agit de la complexité du meilleur algorithme **connu**, impliquant que sa complexité réelle est peut-être plus faible.

6.2 Stabilité du temps polynomial

Définition V.14 (Temps polynomial)

si un MT M s'exécute en temps $t(n) = O(n^c)$ avec $c > 1$,
alors on dit que M s'exécute en temps polynomial.

Corollaire V.1 (complexité d'une MT multi-bandes)

Toute MT M_k à k-bandes à temps polynomial possède
une MT M à une bande équivalente à temps polynomial.

Démonstration:

Si M_k s'exécute en temps $t_k(n) = O(n^c)$, alors M s'exécute en temps $t(n) = O(k^2 t_k(n)^2) = O(k^2 n^{2c}) = O(n^{c'})$ où $c' > 2c > 1$.
Donc, M s'exécute aussi en temps polynomial. □

Conséquence : tout algorithme s'exécutant en temps polynomial sur une MT à k-bandes s'exécute aussi en temps polynomial sur un MT à une bande.

Plus généralement : tout algorithme s'exécutant en temps polynomial sur toute variation d'une MT déterministe s'exécute aussi en temps polynomial sur un MT à une bande. Mieux, la pénalité est au plus quadratique.

6.3 Complexité polynomiale déterministe

On voit donc que des langages polynomiaux (donc décidés par un algorithme en temps polynomial) sont fondamentaux pour l'étude de la complexité.

On définit donc :

Définition V.15 (classe de complexité **P**)

P est la classe des langages qui sont décidables en temps polynomial par une MT à simple bande.
Autrement dit : $\mathbf{P} = \bigcup_k \texttt{TIME}(n^k)$

Remarques 31:

- comme vu précédemment, **P** correspond à la classe des problèmes qui sont solvables sur un ordinateur en un temps réaliste (= faisable).
- **P** est invariant pour tous les modèles de machine que l'on peut simuler sur une MT simple bande en un temps polynomial.
  si un modèle de machine M' résout un problème en temps polynomial et que la simulation de M' sur une MT M se fait en temps polynomial, **alors** M peut résoudre le même problème en temps polynomial.

Conséquence : comment déterminer si un algorithme est à temps polynomial ?

- exprimer les entrées de l'algorithme, et voir comment celles-ci participent à la taille n de l'entrée.
- écrire l'algorithme en pseudo-code
- pour chaque étape, donner une borne supérieure de sa complexité en fonction de n.
- pour des étapes successives, les complexités s'additionnent.
- la complexité d'une boucle est la complexité du corps de boucle multiplié par le nombre de répétitions.
  on s'assurera que le nombre de boucles est toujours fini (condition d'arrêt).
- on conservera que le terme dominant du cumul des complexités.

Exemple 68: Pour un code séquentiel où l'on aura trouvé les complexités n^5, n^2 et 4.
$O(n^5 + n^2 + 4) = O(n^5)$ car $\lim_{n\to\infty} n^5 + n^2 + 4 = \lim_{n\to\infty} n^5(1 + 1/n^3 + 4/n^5) = \lim_{n\to\infty} n^5$. Le terme en n^5 domine donc tous les autres.

L'algorithme s'exécute alors en temps polynomial (car tout ordinateur classique peut être simulé sur une machine de Turing avec une pénalité au plus d'ordre polynomiale).

Attention : l'affirmation précédente n'est valide que pour les machines physiquement réalisables [1] à l'exception des machines quantiques.

Donnons maintenant quelques exemples.

1. ce qui exclut les machines de Turing non déterministes

Exemple : langage PATH

Définition V.16 (langage PATH)

PATH = $\{\langle G, s, t\rangle \mid G$ est un graphe avec un chemin de s vers $t\}$

Donc $\langle G, s, t\rangle \in$ PATH ssi il existe un chemin entre les sommets s et t dans le graphe $G = (V, E)$, ce qui ne peut être le cas que si $(s, t) \in V^2$.

Proposition V.1

PATH $\in$ **P**

Démonstration: PATH $\in$ **P**

Montrons qu'il existe un décideur M à temps polynomial pour PATH.

$M(\langle G, s, t\rangle) =$

```
   // les marquages se font dans la liste des sommets de G
1  marquer le sommet s
2  répéter
3  |   pour chaque arête de G
4  |   |   si un seul sommet est marqué alors marquer l'autre
5  jusqu'à ce que plus aucun nouveau sommet ne soit marqué
6  si t est marqué alors accepter sinon rejeter
```

Temps d'exécution de M : soit m le nombre de nœuds dans G.

- Les lignes 1 et 6 parcourent l'entrée une fois en temps $O(m)$.
- Les lignes 2-5 s'exécutent au plus m fois (temps maximum de propagation entre s et t). Chaque exécution a au plus $O(m^2)$ étapes (chaque nœud est lié avec au plus $m - 1$ autre nœud), et s'exécute en temps $O(m^3)$.
- $m = O(n)$ où $n =$ longueur de la chaîne d'entrée.

Dont M s'exécute en temps $O(n + n^3 + n) = O(n^3)$.
Donc, PATH $\in$ TIME$(n^3) \subset$ **P**. □

Exemple : langage RELPRIME

Définition V.17 (langage RELPRIME)

RELPRIME = $\{\langle x, y\rangle \mid x$ et y sont entiers et PGCD$(x, y) = 1\}$

Donc $\langle x, y\rangle \in$ RELPRIME ssi (x, y) sont des entiers premiers entre eux.

Proposition V.2

RELPRIME $\in$ **P**

Démonstration: RELPRIME $\in$ **P**

Montrons qu'il existe un décideur M à temps polynomial pour RELPRIME.
On utilise l'algorithme d'Euclide.

$M(\langle x, y\rangle) =$

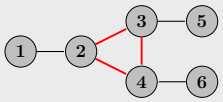
```
1  si x < y alors échanger x et y
2  répéter
3  |   x = x  mod y
4  |   échanger x et y
5  jusqu'à jusqu'à ce que y = 0
6  si x = 1 alors accepter sinon rejeter
```

Temps d'exécution de M :

- Les lignes 1 et 5 sont exécutées une seule fois.
- Chaque exécution de la ligne 3, réduit la valeur de x par aux moins 2. Donc le nombre d'exécutions est au plus de $O(z)$ où $z = \log_2 x + \log_2 y$.
- Toute opération arithmétique (comparaison, modulo) peut être exécutée en temps polynomial du codage des entrées, donc polynomial en z.

Au total, M est polynomial en z.
Comme $z = O(n)$, RELPRIME $\in$ TIME$(n) \subset \mathbf{P}$. □

Exercice 38 (Triangle)**.** *Dans un graphe non orienté, un triangle est un ensemble de 3 arêtes connectées entre elles.*



Soit : $\texttt{TRIANGLE} = \{\langle G\rangle \mid G \text{ est un graphe non orienté contenant un triangle}\}$
Montrer que $\texttt{TRIANGLE} \in \boldsymbol{P}$.

Exercice 39 (Fermeture de **P** par les opérateurs réguliers)**.** *Montrer que* ***P*** *est fermé par :*

1. *union.*
2. *concaténation.*
3. *opérateur de Kleene *.*

Exercice 40 (CONNECTED)**.**
Soit $\texttt{CONNECTED} = \{\langle G\rangle \mid G$ *est un graphe tels que toutes paires de sommets est connectée par une arête*$\}$
Montrer que $\texttt{CONNECTED} \in \boldsymbol{P}$.

Exercice 41 (BIPARTITE)**.**
Un graphe bipartite est un graphe tel que les sommets peuvent être partitionnés en deux ensembles A et B tels que toute arête relie un sommet dans A à un sommet dans B (il n'y a pas d'arête entre deux sommets de A ou entre deux sommets de B).
$\texttt{BIPARTITE} = \{\langle G\rangle \mid G \text{ est un graphe bipartite}\}$
Montrer que $\texttt{BIPARTITE} \in \boldsymbol{P}$.

Exercice 42 (TREE)**.**
Un graphe est un arbre s'il est connecté et ne contient aucun cycle.
Alternativement, un graphe G est un arbre si tout couple $(u, v) \in G$ *est connecté par un chemin simple (= sans répétition de sommets).*
$\texttt{TREE} = \{\langle G\rangle \mid G \text{ est un arbre}\}$
Montrer que $\texttt{TREE} \in \boldsymbol{P}$.

6.4 Complexité subexponentiel et exponentiel déterministe

Les classes de complexité au-delà de **P** sont :

Définition V.18 (classe de complexité **SUBEXP**)

SUBEXP est la classe des langages qui sont décidables en temps subexponentiel par une MT à simple bande.
Autrement dit :**SUBEXP** = $\texttt{TIME}(2^{o(n)})$

Définition V.19 (classe de complexité **EXP**)

EXP est la classe des langages qui sont décidables en temps exponentiel par une MT à simple bande.
Autrement dit : **EXP** = $\bigcup_k \texttt{TIME}(2^{n^k})$

Nous avons vu que les problèmes subexponentiels ou exponentiels étaient généralement infaisables.

Mais, serait-il possible de définir une sous-classe de problèmes pour laquelle la résolution du problème pourrait paraître envisageable ?

7 Classes de complexité en temps non déterministe

On sait déjà que les machines déterministes sont capables de résoudre des problèmes exponentiels en des temps bien inférieurs à une machine déterministe (une MTND en temps $f(n)$ se simule sur une MT déterministe en temps $2^{O(f(n))}$).

On voudrait créer une classe de complexité juste au-dessus de **P** mais moins compliquées que **EXP**.

La complexité d'un problème pour laquelle on cherche une solution peut venir de plusieurs facteurs, notamment de la complexité :

1. à trouver une solution candidate,
2. à vérifier qu'un candidat solution est bien solution du problème.

A priori, les problèmes pour lesquels il est simple de vérifier qu'une solution est bien une solution est plus simple, car il localise la complexité sur la recherche de solution.

Tentons de formaliser cette notion.

7.1 Vérificateur et certificat

Avant de pouvoir parler de la classe **NP**, nous devons aborder la notion de vérificateur.

Définition V.20 (vérificateur)

Un **vérificateur** V pour un langage A est un algorithme tel que :
$A = \{w \mid \exists c; V(\langle w, c\rangle) \text{ accepte}\}$

Remarques 32:

- V utilise une information c supplémentaire pour vérifier que $w \in A$.
  c s'appelle un certificat (ou preuve) d'appartenance à A.
- le temps d'exécution du vérificateur se mesure en terme de longueur de w
  i.e. la longueur de c ne compte pas dans la longueur de l'entrée.

Définition V.21 (vérification)

- un **vérificateur à temps polynomial** est un vérificateur qui s'exécute en temps polynomial sur la longueur de w.
- un langage A est un **langage polynomialement vérifiable** si il possède un vérificateur à temps polynomial.

Exemple 69: de vérificateur
Soit $S = \{s_1, s_2, \ldots, s_n\}$ un ensemble d'entiers et t un entier.
Existe-t-il un sous-ensemble de S tel que la somme de ses entiers soit égal à t ?
On définit le langage associé : SUBSET-SUM= $\left\{\langle S, t\rangle \mid \exists S' \subset S, \sum_{s_i \in S'} s_i = t\right\}$
Exemples de certificats c **pour** $S = \{5, 3, 4, 2, 1, 6\}$ et $t = 13$
$\{6, 4, 3\}$, $\{5, 4, 3, 1\}$, ...
ce sont toutes des valeurs possibles pour c.

Reformulation :

un vérificateur est donc un algorithme qui :

- $\forall w \in A$, il existe (au moins) un certificat c tel que $V(w, c)$ accepte.
- $\forall w \notin A$, il n'existe aucun certificat c tel que $V(w, c)$ accepte.

On peut considérer c comme la solution au problème w,
V est une façon de vérifier que cette solution est valide.

7.2 Classe NP

7.2.1 Définition

Définition V.22 (Classe **NP**)
La classe **NP** est la classe des langages qui sont polynomialement vérifiables.

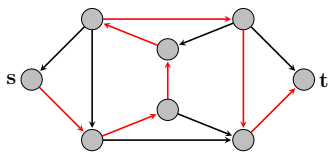
Remarques 33:

- ne provient pas de la complexité de vérification qu'une solution est acceptable.
- mais bien du problème lui-même.

7.2.2 Exemples

Exemples 1 : HAMPATH et $\overline{\text{HAMPATH}}$

Un chemin Hamiltonien dans un graphe orienté est un chemin orienté qui passe exactement une seule fois par tous les sommets :
HAMPATH = $\{\langle G, s, t\rangle \mid G$ graphe orienté avec un chemin Hamiltonien de s à $t\}$



HAMPATH est polynomialement vérifiable.
Un chemin Hamiltonien C d'un graphe $G = (V, E)$ est une liste de sommets consécutifs de C.
Comment vérifier qu'un chemin C est un graphe Hamiltonien de G ?
On utilise le décideur suivant :

$M(\langle G, C\rangle) =$

```
   // vérifie que C contient tous les sommets de G
1  pour chaque sommet s de C
2  |  si s ∈ V alors marquer s dans V sinon rejeter
3  pour chaque sommet s de V
4  |  si s n'est pas marqué alors rejeter
   // vérifie que le chemin dans C existe dans E
5  pour chaque paire de sommets (u, v) consécutive de C
6  |  si (u, v) ∉ E alors rejeter
7  accepter
```

Sa complexité est en $O(n^2)$ où $n = |\langle G, C\rangle|$, donc polynomialement vérifiable.
En revanche, trouver une solution est beaucoup plus difficile :

- complexité de résolution par force brute : $n! = O(n^n)$ (test de tous les chemins possibles).
- pas d'algorithme connu résolvant HAMPATH en temps polynomial.

Définition V.23 (Langage "graphe non Hamiltonien")

$\overline{\texttt{HAMPATH}}$ = graphes sans chemin Hamiltonien.
non polynomialement vérifiable : impossible de valider sans tout vérifier.

Proposition V.3

HAMPATH $\in$ **NP**.

Démonstration:

Définissons un vérificateur V à temps polynomial. Il faut montrer que :

- $\forall \langle G\rangle \in$ HAMPATH, alors $\exists c \mid V(\langle G, c\rangle)$ accepte,
- $\forall c$ tel que $V(\langle G, c\rangle)$ accepte, alors $\langle G\rangle \in$ HAMPATH

Définissons un vérificateur V comme suit :

$V(\langle G, c\rangle) =$

```
1  si c est un chemin hamiltonien alors accepter sinon rejeter
```

V s'exécute en temps polynomial (vérifier que chaque transition du chemin est dans la liste des arêtes, puis vérifier que chaque sommet n'est traversé qu'une seule fois).
Soit H $= \{\langle G\rangle \mid V(\langle G, c\rangle)$ accepte$\}$.

- $\forall \langle G\rangle \in$ H, il existe c qui accepte $\langle G, c\rangle$. Ceci implique que c est un chemin hamiltonien. Donc, $\langle G\rangle$ est un graphe Hamiltonien et H $\subseteq$ HAMPATH.
- Pour tout $\langle G\rangle \in$ HAMPATH, soit c le cycle Hamiltonien dans ce graphe, alors V accepte $\langle G, c\rangle$. Donc, HAMPATH $\subseteq$ H.

Donc, HAMPATH $\in$ **NP**. □

Exemple 2 : COMPOSITES

Définition V.24 (Langage "composite")

Un entier est composite s'il est le produit de deux entiers plus grands que 1 :
On définit le langage :
$\texttt{COMPOSITES} = \{x \mid x = pq, \text{ pour deux entiers } p, q > 1\}$

Proposition V.4

$\texttt{COMPOSITES} \in \mathbf{NP}$.

Démonstration:

Soit le vérificateur V défini par :

$V(\langle x, c\rangle) =$

```
1 si (c n'est pas 1 ou x) et (c divise x) alors accepter sinon rejeter
```

V s'exécute en temps polynomial $O(|\langle x\rangle|^2)$ (division en n^2).
Soit $\texttt{C} = \{\langle x\rangle \mid V(\langle x, c\rangle) \text{ accepte}\}$.

- $\forall \langle x\rangle \in \texttt{C}$, il existe c tel que V accepte $\langle x, c\rangle$. Donc, x est un nombre composite, et $\texttt{C} \subseteq \texttt{COMPOSITES}$
- $\forall \langle x\rangle \in \texttt{COMPOSITES}$, soit c un diviseur de x avec $1 < c < x$. Alors V accepte $\langle x, c\rangle$, $\texttt{COMPOSITES} \subseteq \texttt{C}$.

Donc, $\texttt{COMPOSITES} \in \mathbf{NP}$. □

7.2.3 Caractérisation la classe NP

Théorème V.6 (caractérisation de la classe **NP**)

(Un langage A est dans **NP**) si et seulement si (il est décidé par une MT non-déterministe en temps polynomial).

Autrement dit, s'il existe un vérificateur à temps polynomial qui décide A.

Démonstration: ($A \in \mathbf{NP}$) $\Rightarrow$ (A **décidé par une MTND en temps polynomial)**

Si A est dans **NP**, alors il existe un vérificateur V à temps polynomial qui permet de le vérifier.
Soit n^k le temps d'exécution de V.
Soit N, la MT non-déterministe suivante :

$N(\langle w\rangle) =$

```
1 choix non déterministe d'une chaîne c de longueur au plus n^k
2 exécuter V(⟨w, c⟩)
3 si V accepte alors accepter sinon rejeter
```

Alors, sur la MTND N :

1. générer l'ensemble des chaînes de longueur n^k se fait en temps polynomial $O(n^k)$ (arbre avec $|\Sigma|$ fils par nœud et de profondeur n^k). On obtient ainsi tous les certificats possibles.
2. vérifier chaque certificat avec $V(\langle w, c\rangle)$ prend lui-aussi un temps polynomial (revient à poursuivre chaque feuille de l'arbre précédent et l'accepter ou la rejeter).
3. acceptation des branches. $O(1)$ sur une MTND.

V accepte si l'une quelconque des branches accepte. Donc, si un certificat c existe pour w, alors N le trouve nécessairement.
N permet donc de décider pour tout w de A en temps polynomial. □

Démonstration: (*A* décidé par une MTND en temps polynomial) ⇒ (*A* ∈ NP)

Soit A un langage accepté par une MTND N à temps polynomial. On construit un vérificateur à temps polynomial de la façon suivante :

$V(\langle w, c\rangle) =$

1 **simuler** $N(\langle w\rangle)$ en utilisant c comme la description du choix non-déterministe de N à chaque étape.
2 **si** *la branche d'exécution de N accepte* **alors** accepter **sinon** rejeter

Si un MTND est à temps polynomial, alors chacune de ses branches s'exécute en temps polynomial. Or, le certificat c pris est celui qui permet de choisir l'une des branches acceptante à chaque nœud de la MTND. Donc, l'exécution de V ne revient à évaluer qu'une seule branche de la MTND. Par conséquent, V s'exécute en temps polynomial. □

Définition V.25 (NTIME($t(n)$))

NTIME($t(n)$) = ensemble des langages qui peuvent être décidés par une MT non-déterministe à temps $O(t(n))$.

Corollaire V.2

NP = $\bigcup_k$ NTIME(n^k)

Démonstration:

conséquence directe du théorème précédent. □

Nous avons donc 2 façons de démontrer qu'un problème appartient à **NP** :

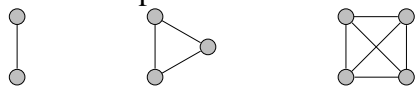
- en utilisant un vérificateur.
- en utilisant une MT non-déterministe à temps polynomial.

NP a, en fait, pour signification **P**olynomial en temps **N**on déterministe, et non pas **N**on **P**olynomial. Maintenant, application pour montrer l'appartenance de nouveaux langages à **NP**.

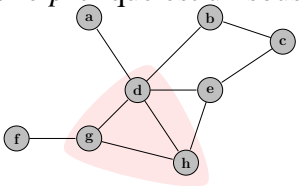
Exemple 1 : CLIQUE

Définition V.26 (Langage Clique)

Un graphe complet K_p est un graphe qui contient p noœuds, et tel que chaque nœud soit connecté au $p-1$ autres nœuds par une arête.



Une p-clique est un sous-graphe complet K_p d'un graphe G non orienté.



Le graphe ci-contre contient deux 3-cliques : $\{g, h, d\}$ et $\{d, e, h\}$.

On définit le langage :

CLIQUE = $\{\langle G, k\rangle \mid G$ est un graphe avec une k-clique$\}$

Proposition V.5

CLIQUE ∈ **NP**.

DÉMONSTRATION:

Méthode 1 : en utilisant un vérificateur.

Le certificat c du vérificateur V sont les nœuds constituant une k-clique.

$V(\langle G, k, c\rangle) =$

1 **tester si** c est un sous-ensemble de k nœuds de G.
2 **tester si** chaque couple de nœuds de c est connecté dans G.
3 **si** *les deux tests sont vrais* **alors** accepter **sinon** rejeter

Trivialement, V accepte tout $\langle G, k\rangle \in$ `CLIQUE` en temps polynomial.

Donc, `CLIQUE` est polynomialement vérifiable et `CLIQUE` $\in$ **NP**. □

Méthode 2 : en utilisant une MT non-déterministe à temps polynomial qui accepte `CLIQUE`.

La MTND N suivante permet de décider `CLIQUE` :

$N(\langle G, k\rangle) =$

1 **choix non-déterministe** de c contenant k-nœuds de G.
2 **tester si** chaque couple de nœuds de c est connecté.
3 **si** *le test est vrai* **alors** accepter **sinon** rejeter

Toutes les étapes de N s'exécutent en temps polynomial.

Donc, `CLIQUE` $\in$ **NP**. □

EXEMPLE 2 : `SUBSET-SUM`

Proposition V.6

`SUBSET-SUM` $\in$ **NP**.

DÉMONSTRATION:

Méthode 1 : en utilisant un vérificateur

Le certificat c du vérificateur V sont des entiers de S dont la somme est t.

$V(\langle S, t, c\rangle) =$

1 **tester si** c est un sous-ensemble d'entiers de S.
2 **tester si** la somme des nombres de c fait t.
3 **si** *les deux tests sont vrais* **alors** accepter **sinon** rejeter

Trivialement, V accepte tout $\langle S, t\rangle \in$ `SUBSET-SUM` en temps polynomial.

Donc, `SUBSET-SUM` est polynomialement vérifiable et `SUBSET-SUM` $\in$ **NP**. □

Méthode 2 :en utilisant une MT non-déterministe à temps polynomial qui accepte `SUBSET-SUM`.

La MTND N suivante permet de décider `SUBSET-SUM` :

$N(\langle G, k\rangle) =$

1 **choix non-déterministe** d'un sous-ensemble c de S.
2 **tester si** la somme des nombres de c fait t.
3 **si** *le test est vrai* **alors** accepter **sinon** rejeter

Toutes les étapes de N s'exécutent en temps polynomial.

Donc, `SUBSET-SUM` $\in$ **NP**. □

Exercice 43 (Fermeture de **NP**). *Montrer que **NP** est fermé par union :*

1. *avec un vérificateur à temps polynomial.*
2. *avec une machine de Turing non déterministe à temps polynomial.*

Exercice 44 (SUBSET_SUM). *Montrer que SUBSET_SUM est dans **NP** :*

1. *avec un vérificateur à temps polynomial.*
2. *avec une machine de Turing non déterministe à temps polynomial.*

Exercice 45 (UNARY_SUM).
Soit : $\texttt{UNARY\_SUM} = \Big\{\langle S, t\rangle$ *où* $S = \{x_1, \ldots, x_k\}$ *sont des entiers stockés en unaire et* $\exists S' \subseteq S$ *tel que* $\sum_{x_i \in S'} x_i = t\Big\}$

1. *Montrer que* $\texttt{UNARY\_SUM} \in \boldsymbol{P}$.
2. *Serait-il possible de trouver un algorithme équivalent pour SUBSET_SUM montrer qu'il est dans **NP** ?*

7.2.4 Autres propriétés en temps non déterministe

Le théorème de Hiérarchie existe aussi en temps non déterministe :

Théorème V.7 (Hiérarchie en temps non-déterministe)

Si $g(n)$ est croissante et constructible en temps, et $f(n+1)\log f(n+1) = o(g(n))$ alors $\texttt{NTIME}(f(n)) \subsetneq \texttt{NTIME}(g(n))$.

Nous ne démontrerons pas ce résultat.

Théorème V.8

Pour toute fonction $t : \mathbb{N} \to \mathbb{N}$, $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n)) \subseteq \texttt{TIME}\left(2^{O(t(n))}\right)$

Démonstration:

- $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n))$: trivial, car toute machine déterministe est une machine non déterministe avec une seule branche.
- $\texttt{NTIME}(t(n)) \subseteq \texttt{TIME}(2^{O(t(n))})$: l'ensemble des branches de calcul d'une machine non déterministe en $t(n)$ est en $2^{O(t(n))}$. Comme chaque branche prend un temps, au plus de $t(n)$, cela peut être simulé en temps $t(n).2^{O(t(n))} = 2^{O(t(n))}$ sur une machine déterministe.

□

7.3 Temps non-déterministe exponentiel

De manière similaire à **EXP**, on peut définir l'équivalent pour le temps non déterministe :

Définition V.27 (classe de complexité **NEXP**)

NEXP est la classe des langages qui sont décidables en temps exponentiel par une MT à simple bande.

Autrement dit : $\textbf{NEXP} = \bigcup_k \texttt{NTIME}(2^{n^k})$

8 NP-complétude

8.1 Comparaison entre P et NP

Si on compare les caractéristiques de **P** et de **NP** :

P classe des algorithmes qui peuvent être décidés "rapidement".
NP classe des algorithmes qui peuvent être vérifiés "rapidement".

où "rapidement" = en temps polynomial.

Réflexions :

La puissance de la vérifiabilité à temps polynomial
(i.e. d'une MT non-déterministe à temps polynomial)
semble plus importante que la décidabilité à temps polynomial
(i.e. d'une MT déterministe à temps polynomial)
Autrement dit, que $\mathbf{P} \subset \mathbf{NP}$
Mais ...

Malheureusement,

- personne n'a jamais réussi à démontrer si $\mathbf{P} = \mathbf{NP}$
  un des grands problèmes d'informatique/mathématiques
- beaucoup pensent que $\mathbf{P} \neq \mathbf{NP}$
  comme cela n'a jamais été démontré, cela est peut-être faux.

En 1971, Cook & Levin démontrent séparément que :

- Il existe des problèmes caractéristiques de la classe **NP** (dit problèmes **NP**-complets).
- Pour un problème **NP**-complets,
  si **n'importe lequel** d'entre eux est décidable par un MT déterministe en temps polynomial,
  alors **tous** les problèmes de **NP** peuvent être décidés par une MT déterministe en temps polynomial.
- **Reformulation** :
  il existe certains langages L tels que si $L \in \mathbf{P}$, alors $\mathbf{P} = \mathbf{NP}$.

et ils trouvent un exemple de problème **NP**-complet.

8.2 Réduction en temps polynomial

Rappel

Si un problème A peut être réduit en un autre problème B, alors :

- si B est décidable, alors A est décidable.
- si B est énumérable, alors A est énumérable.

On remarquera qu'une réduction est une transformation en un **nombre fini de pas**.

Définition V.28 (fonction calculable en temps polynomial)

Une fonction $f : \Sigma^* \to \Sigma^*$ est une **fonction calculable en temps polynomial** s'il existe une MT M à temps polynomial qui s'arrête avec $f(w)$ sur sa bande lorsque son entrée est w.

Définition V.29 (langage réductible en temps polynomial)

Un langage A est **réductible en temps polynomial** en un langage B, s'il existe une fonction f calculable en temps polynomial telle que : $w \in A \Leftrightarrow f(w) \in B$

Notation : $A \leq_{\mathbf{P}} B$

On appelle f une réduction en temps polynomial de A à B

Théorème V.9 (Réduction en temps polynomial à **P**)

Soit $A \leq_P B$ et $B \in \mathbf{P}$.
Alors $A \in \mathbf{P}$.

Démonstration:

Soit un algorithme M (= une MT) qui décide B en temps polynomial.
Soit f une réduction en temps polynomial de A vers B.
Soit N l'algorithme suivant :

$N(\langle w \rangle) =$

1 **calculer** $f(w)$.
2 **exécuter** $M(< f(w) >)$
3 **décider** comme M

N décide B car M accepte $f(w)$ pour tout $w \in A$, puisque f est une réduction de A dans B et M un décideur pour A.
N s'exécute en temps polynomial puisque la réduction f et le décideur M s'exécutent consécutivement, chacun en temps polynomial.

□

Maintenant, un exemple de réduction en temps polynomial.

8.3 Rappel de logique des propositions

Tables de vérité : opérateurs logiques OU $\vee$, ET $\wedge$ et NON $\neg$

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

x	$\overline{x}$
0	1
1	0

Définition :

littéral = variable booléenne (1/vrai ou 0/faux).
clause = plusieurs littéraux connectés par $\vee$
exemple : $v_1 \vee \overline{v_2} \vee \overline{v_3} \vee v_4$
forme normale conjonctive = plusieurs clauses connectées par $\wedge$
exemple : $(v_1 \vee \overline{v_2}) \wedge (\overline{v_3} \vee v_4)$
note 1 : aussi nommée FNC.
note 2 : FNCk = FNC où chacune des clauses a k littéraux.

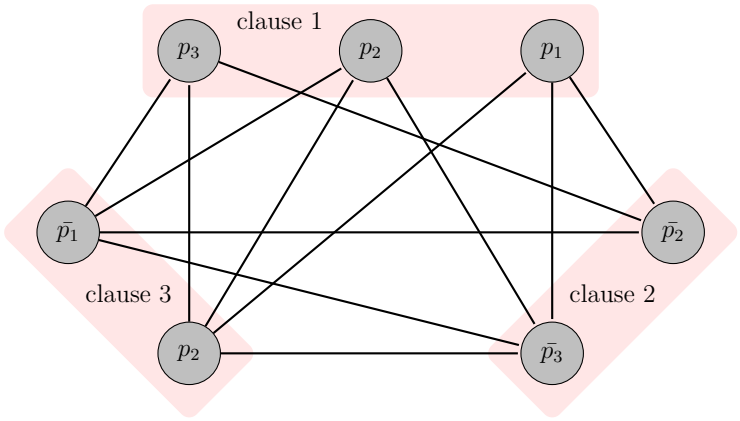
Forme normale conjonctive satisfiable :

Une FNC F est satisfiable s'il existe une combinaison de littéraux telle que F soit vrai.
exemple : $F = (v_1 \vee \overline{v_2}) \wedge (\overline{v_3} \vee v_4)$ est satisfiable ($F = 1$)
avec $v_1 = 1$, $v_2 = 1$, $v_3 = 0$ et $v_4 = 0$.

Graphe associé à une FNC :

- chaque littéral représente un nœud pour chacun des littéraux d'une clause.
- chaque clause représente un ensemble de nœuds non connectés entre eux.
- chaque littéral x d'une clause est connecté à tous les littéraux de toutes les autres clauses, sauf si ce littéral est $\overline{x}$.

Exemple : $(p_1 \vee p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \wedge (\overline{p_1} \vee p_2)$



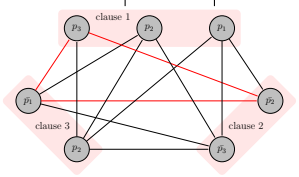
La FNC représente des "blocs" de relations similaires entre nœuds d'un graphe.

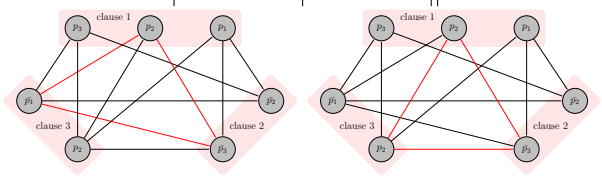
Lien entre FNC et clique

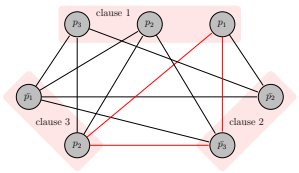
Un FNC à k clauses peut être transformé en un graphe. Si on trouve une k-cliques dans ce graphe, alors la FNC est satisfiable.

Exemple : $(p_1 \vee p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \wedge (\overline{p_1} \vee p_2)$ (donc ici $k = 3$)

p_1	p_2	p_3	$p_1 \vee p_2 \vee p_3$	$\overline{p_2} \vee \overline{p_3}$	$\overline{p_1} \vee p_2$	p
faux	faux	faux	faux	vrai	vrai	faux
faux	faux	vrai	vrai	vrai	vrai	vrai
faux	vrai	faux	vrai	vrai	vrai	vrai
faux	vrai	vrai	vrai	faux	vrai	faux
vrai	faux	faux	vrai	vrai	faux	faux
vrai	faux	vrai	vrai	vrai	faux	faux
vrai	vrai	faux	vrai	vrai	vrai	vrai
vrai	vrai	vrai	vrai	faux	vrai	faux


$p_1 = p_2 = 0, p_3 = 1$


$p_1 = 0, p_2 = 1, p_3 = 0$


$p_1 = p_2 = 1, p_3 = 0$

Rappel

- $\texttt{SAT}_3 = \{\langle F \rangle \mid F$ est une $\texttt{FNC}_3$ satisfiable$\}$
- $\texttt{CLIQUE} = \{\langle G, q \rangle \mid G$ est un graphe avec une q-clique$\}$

Proposition V.7

$\texttt{SAT}_3 \leq_P \texttt{CLIQUE}$.

Démonstration:

Définissons la fonction f qui construit un graphe G associé à une FNC comme vu dans l'exemple précédent.

① Pour tout $F \in \texttt{SAT}_3$, alors $f(F) \in \texttt{CLIQUE}$, puisque pour une FNC à n clauses, $f(F) \in \texttt{CLIQUE}_n$ et que $\texttt{CLIQUE}_n \subset \texttt{CLIQUE}$.

② Inversement, la construction associant un littéral par nœud, f est clairement inversible. Donc, $f(F) \in \texttt{CLIQUE} \Rightarrow F \in \texttt{SAT}_3$.

③ La construction du graphe se fait en temps polynomial (générer les sommets = $O(3n)$, générer les arêtes = $O(3n^2)$).

Donc f est bien une réduction de $\texttt{SAT}_3$ vers $\texttt{CLIQUE}$ calculable en temps polynomial. □

Conséquence :

- on a vu que : $\texttt{SAT}_3 \leq_P \texttt{CLIQUE}$
- et que si $A \leq_P B$ et $B \in \mathbf{P}$ alors $A \in \mathbf{P}$.

Donc, si $\texttt{CLIQUE} \in \mathbf{P}$ alors $\texttt{SAT}_3 \in \mathbf{P}$ aussi.

Mais peut-on dire quelque chose si $\texttt{CLIQUE} \in \mathbf{NP}$?

8.4 Définition et premières propriétés

Définition V.30 (langage **NP**-complet)

Un langage est B est **NP**-complet si :

1. $B \in \mathbf{NP}$
2. $\forall A \in \mathbf{NP}$, $A \leq_P B$ (= B est **NP**-difficile)

Remarque 34:

Les langages **NP**-complets sont les langages les plus difficiles de **NP**.

Théorème V.10

Si B est **NP**-complet et $B \in \mathbf{P}$ alors $\mathbf{P} = \mathbf{NP}$.

Démonstration:

Si B est **NP**-complet, alors pour tout $A \in \mathbf{NP}$, il existe une réduction en temps polynomial vers B.
Or si $B \in \mathbf{P}$, B est décidable en temps polynomial.
Décider A = réduire à B + décider B ; faisable en temps polynomial. □

Théorème V.11

Si B est **NP**-complet et $B \leq_P C$ et $C \in \mathbf{NP}$ alors $\mathbf{C}$ est **NP**-complet.

Démonstration:

si B est **NP**-complet, alors $\forall A \in \mathbf{NP}$, $A \leq_P B$. Or $B \leq_P C$ implique $A \leq_P C$ (les deux réductions consécutives sont effectuées en temps polynomial). Donc, si $C \in \mathbf{NP}$, alors C est **NP**-complet puisque $\forall A \in \mathbf{NP}$, $A \leq_P C$. □

8.5 Théorème de Cook-Levin

Avant de passer au théorème, considérons le langage suivant :

Définition V.31

$\texttt{SAT} = \{\langle F \rangle \mid F \text{ est une expression logique satisfiable}\}$

Proposition V.8

$\texttt{SAT} \in \mathbf{NP}$.

Démonstration:

- **avec un vérificateur à temps polynomial** : prendre comme certificat c une chaîne contenant la valeur de vérité de chaque littéral unique de l'expression logique. L'évaluation de l'expression logique à partir des valeurs des littéraux se fait en temps polynomial $O(n)$.

- **avec une MT non déterministe** : choix non déterministe de la valeur de vérité de chaque littéral unique, puis évaluation de l'expression logique. Chaque branche s'exécute trivialement en temps polynomial.

□

Problème principal : nous n'avons pas de problème **NP**-complet. Ceci est résolu avec le théorème suivant.

Théorème V.12 (Cook-Levin)

SAT est **NP**-complet.

DÉMONSTRATION:

Il faut montrer que :

- SAT $\in$ **NP** (fait : voir théorème précédent)
- $\forall A \in$ **NP**, $A \leq_P$ SAT
  Il faut montrer que tout langage de **NP** peut être réduit en temps polynomial en SAT.

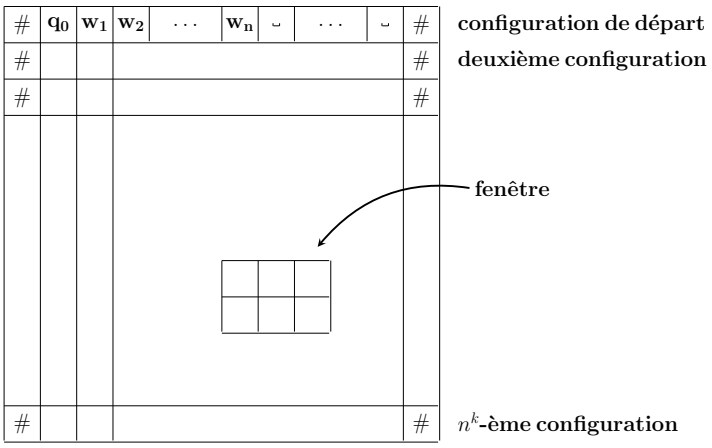
Soit N une MT non déterministe qui décide le langage A.
On veut construire une réduction f telle que :
N accepte $w \Leftrightarrow F$ est satisfiable.
$N \in$ **NP** signifie qu'une branche d'exécution de N s'exécute en temps n^k pour une entrée de taille n, donc :

- une branche d'exécution de N passe par n^k configuration différente
  Rappel : configuration = bande + état courant à sa position.
- N visite au plus n^k cases : une configuration fait au plus n^k symboles.

On définit un tableau (voir page suivante) :

- de taille $n^k \times n^k$ qui représente une branche d'exécution de N sur w.
- qui utilise un alphabet étendu $C = \Gamma \cup Q \cup \{\#\}$ (alphabet de la bande, états et marqueur d'extrémité).

Note : devrait être $n^k + 3$ avec # au début et à la fin + état courant.



La branche d'exécution de N est stockée comme suit dans le tableau :

- La première ligne représente la configuration de depart.
- Les lignes suivantes représentent la suite des configurations dans l'ordre d'exécution de la branche. Chaque ligne peut être déduite de la précédente par les règles de transition de N.
- Si n'importe quelle ligne du tableau est une configuration acceptante, alors le tableau est acceptant.

Donc, un tableau accepte w si N accepte w. Ainsi,

- décider si N accepte w équivaut à décider s'il existe un tableau acceptant pour l'exécution de $N(\langle w \rangle)$.
- il faut trouver une formule logique F permettant de trouver si un tableau acceptant existe.

On veut construire une expression logique F qui permet de décider si un tableau est acceptant lorsque F est satisfiable. F doit garantir que toutes les conditions suivantes sont vraies :

1. Chaque cellule est occupée par exactement 1 symbole (ϕ_{cell}).
2. Le tableau est dans une configuration acceptante (ϕ_{accept}).
3. La première ligne est la configuration d'entrée avec w (ϕ_{start}).
4. La suite des configurations suivantes est cohérente avec une exécution de N (ϕ_{move}).

Autrement dit, F s'écrit sous la forme :

$F = \phi_{\text{cell}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}}$

On définit le littéral $x_{i,j,s}$ de la façon suivante.

$x_{i,j,s} = 1$ ssi la cellule (i, j) du tableau stocke le symbole $s \in C$ et 0 sinon.

Expression de ϕ_{cell} :

$f_{i,j,1}$ = la cellule (i, j) contient **au moins** un symbole = $\bigvee_{s \in C} x_{i,j,s}$

$f_{i,j,2}$ = la cellule (i, j) contient **au plus** un symbole = $\bigwedge_{s,t \in C, s \neq t} (\overline{x}_{i,j,s} \vee \overline{x}_{i,j,t})$

Ces conditions doivent être vérifiées pour toutes les cellules du tableau : $\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} f_{i,j,1} \wedge f_{i,j,2} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t \in C, s \neq t} (\overline{x}_{i,j,s} \vee \overline{x}_{i,j,t}) \right) \right]$.

Expression de ϕ_{accept} :

Si il y a l'état q_{accept} n'importe où dans la table.

$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}$

Expression de ϕ_{start} :

Si la première ligne commence par #, suivi par q_0, puis $w = w_1 w_2 \dots w_n$, et complété par des blancs ␣, et fini par #.

$\phi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \dots \wedge x_{1,n+2,w_n}$
$\wedge x_{1,n+3,␣} \wedge \dots \wedge x_{1,n^k-1,␣} \wedge x_{1,n^k,\#}$

Expression de ϕ_{move} :

Fenêtre : définie par sa position (i, j) et sa taille 2×3.

Fenêtre légale : transformation possible par N sur la fenêtre.

a	q_1	b
q_2	a	c

légale si N a une transition $\delta(q_1, b) = (q_2, c, L)$

a	q_1	b
a	a	q_2

légale si N a une transition $\delta(q_1, b) = (q_2, a, R)$

a	a	q_1
a	a	b

légale si N a une transition $\delta(q_1, c) = (q_2, b, R)$

#	a	b
#	a	b

légale (pas de transition dans cette fenêtre)

a	b	a
a	b	q_2

légale si N a une transition $\delta(q_1, b) = (q_2, c, L)$

a	a	a
b	a	a

légale si N a une transition $\delta(q_1, a) = (q_2, b, L)$

a	b	b
a	a	b

non légale (la transition devrait être dans la fenêtre)

a	q_1	b
q_2	a	q_2

non légale (2 transitions sur la 2ème ligne)

a	q_1	a
q_2	c	b

non légale (transition à gauche, caractère à droite modifié)

Expression de ϕ_{move} :

Validité du tableau : si toutes les fenêtres sont légales.

Soit le prédicat Legal(i, j) qui est vrai si la fenêtre (i, j) est légale.

$$\text{Legal}(i, j) = \bigvee_{\{a_1,\ldots,a_6\} \text{ est légale}} x_{i,j,a_1} \wedge x_{i,j,a_2} \wedge \ldots \wedge x_{i,j,a_6}$$

Autrement dit, Legal(i, j) est l'union de toutes les configurations légales sur une fenêtre 3×2 en accord avec le fonctionnement de N sur deux configurations consécutives.

En conséquence, $\phi_{\text{move}} = \bigwedge_{1 \leq i,j \leq n^k} \text{Legal}(i, j)$

Note : raison pour laquelle l'application du prédicat ϕ_{move} produit une suite de configuration valide correspondant à l'exécution d'une MT :

- la première ligne du tableau est une ligne valide (entrée w de la MT assurée par ϕ_{start}).
- la fenêtre 3×2 permet de produire (par récurrence) une configuration valide (la suivante) à partir de la configuration précédente (valide pour i=1).

Conséquence de cette construction :

$\boxed{\Rightarrow}$ F satisfiable

$\Rightarrow$ F représente l'affectation d'un tableau acceptant.

$\Rightarrow$ le tableau acceptant représente une branche d'exécution de N sur w.

$\Rightarrow$ N accepte l'entrée w.

$\boxed{\Leftarrow}$ Inversement, si N accepte l'entrée w,

$\Rightarrow$ la suite des configurations se représente comme un tableau acceptant.

$\Rightarrow$ la formule F équivalente est satisfiable.

$\Rightarrow F$ satisfiable.

Donc, pour tout MT N non déterministe à temps polynomial :
N accepte $w \Leftrightarrow F$ est satisfiable
Cette construction est donc une réduction de tout langage de **NP** vers SAT.

Montrons maintenant que cette réduction peut être faite en temps polynomial.
Notons tous d'abord que :

- l'alphabet étendu C utilisé pour coder le tableau (on notera $c = \#C$).
- la MT N

ne dépendent pas de n.
Alors, le temps de calcul est de l'ordre de :

ϕ_{cell} : $f_{i,j,1}$ et $f_{i,j,2}$ ne dépendent que de c. Donc ϕ_{cell} est en $O(n^{2k})$.
ϕ_{accept} : Trivialement de l'ordre de $O(n^{2k})$.
ϕ_{start} : Trivialement de l'ordre de $O(n^k)$.
ϕ_{move} : Legal(i, j) ne dépend que de c. Donc ϕ_{move} est en $O(n^{2k})$.

Par conséquent la réduction s'effectue en temps polynomial.
Donc, $\forall A \in \mathbf{NP}$, $A \leq_P$ SAT $\Rightarrow$ SAT.
On en déduit que SAT est **NP**-complet. □

8.6 Autres problèmes NP-complets

8.6.1 FNC-SAT

Définition V.32
FNC-SAT = $\{\langle F \rangle \mid F$ est une FNC satisfiable$\}$

Théorème V.13
FNC-SAT est **NP**-complet.

Démonstration:

- FNC-SAT $\in$ **NP** Même ligne de preuve que pour SAT.
- $\forall A \in$ **NP**, $A \leq_P$ FNC-SAT. la preuve du théorème du Cook-Levin peut être directement réutilisée car la réduction utilise des FNCs.

Donc, FNC-SAT est **NP**-complet. □

Définition V.33
$\text{SAT}_3 = \{\langle F \rangle \mid F$ est une FNC_3 satisfiable$\}$

Proposition V.9
SAT_3 est **NP**-complet.

Démonstration:

- $\text{SAT}_3 \in$ **NP**. Même ligne de preuve que pour SAT.
- $\forall A \in$ **NP**, $A \leq_P \text{SAT}_3$. Pour se faire, on réduit FNC-SAT à SAT_3. Soit $F = \bigwedge_i C_i$ où C_i sont les clauses de F. On a alors 3 cas :

— C_i **a moins de 3 littéraux** : il suffit de dupliquer l'un des littéraux.
Exemple : $C_i = L_1 \Rightarrow C'_i = L_1 \vee L_1 \vee L_1$
— C_i **a 3 littéraux** : on le laisse tel quel.
— C_i **a plus de 3 littéraux** : notons $C_i = L_1 \vee L_2 \vee \ldots \vee L_m$.
Introduire des nouveaux littéraux z_i de la façon suivante :

$$C'_i = (L_1 \vee L_2 \vee z_1) \wedge (\overline{z}_1 \vee L_3 \vee z_2) \wedge (\overline{z}_2 \vee L_4 \vee z_3) \wedge \ldots \wedge (\overline{z}_{m-3} \vee L_{m-1} \vee L_m)$$

Si C_i est satisfiable, alors il existe $(z_1, z_2, \ldots, z_{m-3})$ tels que C'_i soit aussi satisfiable.
Exemple :

$C'_i = (L_1 \vee L_2 \vee z_1) \wedge (\overline{z}_1 \vee L_3 \vee z_2) \wedge (\overline{z}_2 \vee L_4 \vee L_5)$
Si C_i n'est pas satisfiable, alors il n'existe aucune combinaison de (z_1, z_2) qui rende C'_i satisfiable, sinon il existe une combinaison de (z_1, z_2) qui rend C'_i satisfiable.

Soit $F \in$ FNC-SAT satisfiable. Soit F' construit à partir de F par la méthode indiquée. Alors $F' \in \mathtt{SAT}_3$ est satisfiable. Inverse évident car $\mathtt{SAT}_3 \subset$ FNC-SAT. Comme FNC-SAT $\leq_P \mathtt{SAT}_3$ et FNC-SAT **NP**-complet, alors $\mathtt{SAT}_3$ aussi.

□

8.6.2 CLIQUE

Rappel : CLIQUE $= \{\langle G, k\rangle \mid G$ est une $k -$ CLIQUE $\}$

Proposition V.10
CLIQUE est **NP**-complet.

Démonstration:

- CLIQUE $\in$ **NP** : déjà démontré.
- $\forall A \in$ **NP**, $A \leq_P$ CLIQUE. On a déjà montré que $\mathtt{SAT}_3 \leq_P$ CLIQUE. Comme $\mathtt{SAT}_3$ **NP**-complet, alors CLIQUE aussi.

□

On voit que l'on dispose d'une méthode facile pour montrer qu'un problème K est **NP**-complet.

- trouver un autre problème K' qui soit **NP**-complet et "proche" de K
- montrer que $K \in$ **NP**.
- montrer qu'il existe une réduction en temps polynomial de K' vers K.
- en déduire que K est **NP**-complet.

8.6.3 SUBSET-SUM

Rappel : SUBSET-SUM $= \{\langle S, t\rangle \mid \exists S' \subseteq S = \{s_1, s_2, \ldots, s_n\} / \Sigma_{s_i \in S'} s_i = t\}$

Proposition V.11
SUBSET-SUM est **NP**-complet.

DÉMONSTRATION:

- SUBSET-SUM $\in$ **NP** : déjà démontré.
- Par réduction en temps polynomial de $\mathtt{SAT}_3$ vers SUBSET-SUM.

  Soit $F = \bigwedge_{i=1\ldots n} C_i$ une expression logique sous forme de conjonction de clauses C_i, chaque clause étant composée de la disjonction de 3 littéraux parmi $x_1, x_2, \ldots, x_p$ ou de leurs négations.

  On cherche une transformation telle que :

  — chaque littéral x_j possède une valeur unique (vrai ou faux).

  — chaque clause C_i soit vraie.

  Construisons un ensemble d'entiers à $p + n$ chiffres où l'on affecte une propriété à chaque chiffre.

Le rôle de chaque chiffre est le suivant :

- les p premiers chiffres représentent les littéraux x_j. Le $j^{\text{ème}}$ chiffre représente le littéral x_j.
- les n chiffres suivants représentent les clauses C_i. Le $(p+i)^{\text{ème}}$ chiffre représente la clause C_i.

On va donc créer un premier ensemble de $2p$ entiers à $n + p$ chiffres, où tous leurs chiffres sont à 0, sauf pour :

a_j (associé à x_j) son $j^{\text{ème}}$ chiffre à 1 si x_j est **vrai**.
et partout où x_j apparaît dans C_i, son $(p + i)^{\text{ème}}$ chiffre est à 1.

$\overline{a}_j$ (associé à $\overline{x}_j$) son $j^{\text{ème}}$ chiffre à 1 si x_j est **faux**.
et partout où $\overline{x}_j$ apparaît dans C_i, son $(p + i)^{\text{ème}}$ chiffre est à 1.

Quelle valeur de t faut-il alors choisir ?
Il faut nécessairement que chaque a_j ou (exclusif) $\overline{a}_j$ soit dans S'.
$\Rightarrow$ les n premiers chiffres de t sont 1.
$\Rightarrow$ les p chiffres suivants de t sont supérieurs à 1.

Exemple : $F = C_1 \wedge C_2 = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3 \vee x_4)$

	x_1	x_2	x_3	x_4	C_1	C_2
a_1	1	0	0	0	1	0
$\overline{a}_1$	1	0	0	0	0	0
a_2	0	1	0	0	0	1
$\overline{a}_2$	0	1	0	0	1	0
a_3	0	0	1	0	1	0
$\overline{a}_3$	0	0	1	0	0	1
a_4	0	0	0	1	0	1
$\overline{a}_4$	0	0	0	1	0	0

Exemples de choix :

$a_1 + \overline{a}_2 + a_3 + a_4 =$	1	1	1	1	3	1	valide / F vrai
$a_1 + a_2 + \overline{a}_3 + \overline{a}_4 =$	1	1	1	1	1	2	valide / F vrai
$\overline{a}_1 + a_2 + \overline{a}_3 + a_4 =$	1	1	1	1	0	3	valide / F faux
$a_1 + \overline{a_1} + a_3 =$	2	0	1	0	2	0	invalide / F indéfini

Pour la partie attribut : le n premiers chiffres sont 1.
Pour la partie clause : Chaque chiffre est entre 1 et 3.
Pas de t unique si on fait comme cela.
Solution : Choisir $t = 1 \ldots 13 \ldots 3$ en ajoutant 2 variables identiques supplémentaires par clause b_j et b'_j (donc $2p$ variables) définies par le $(n+j)^{\text{ème}}$ chiffre est à 1 et les autres à 0.
Il y a alors 3 cas différents pour la $(n+j)^{\text{ème}}$ colonne :

$= 3$ pas de problème (les 3 littéraux sont vrais).

$= 2$ (resp. 1), alors ajouter b_j **ou** b'_j (resp. b_j **et** b'_j) pour arriver à 3.

$= 0$ alors même en ajoutant b_j et b'_j, impossible de faire 3.

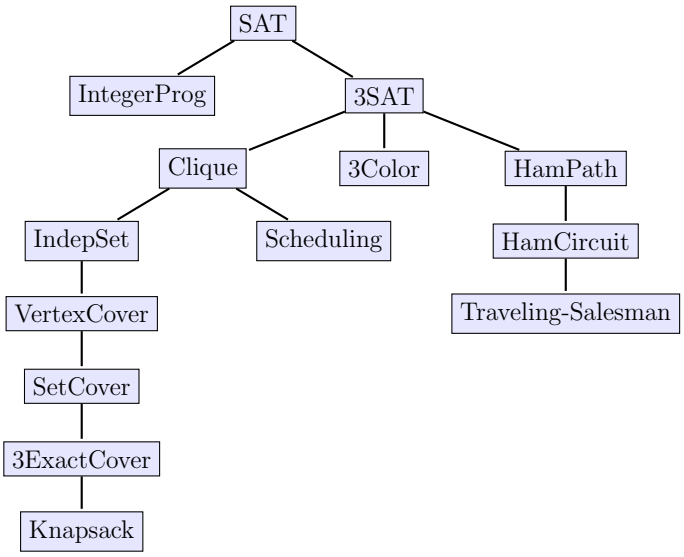
Donc, avec $S = \{a_1,\ldots,a_n,\bar{a}_1,\ldots,\bar{a}_n,b_1,\ldots,b_p,b'_1,\ldots,b'_p\}$ et $t = \underbrace{1\ldots1}_{n \text{ fois}}\underbrace{3\ldots3}_{p \text{ fois}}$.
On a donc trivialement ($\langle F\rangle \in \texttt{SAT}_3 \Leftrightarrow f(\langle F\rangle) \in \texttt{SUBSET-SUM}$).
La réduction f s'effectue clairement en temps polynomial. En conséquence, $\texttt{SAT}_3 \leq_P \texttt{SUBSET-SUM}$ et $\texttt{SAT}_3$ **NP**-complet implique `SUBSET-SUM` **NP**-complet.

□

8.7 Hiérarchies des problèmes NP-complet

De nombreux problèmes **NP**-complets ont été trouvés à ce jour.

Ils constituent une hiérarchie formée par les réductions en temps polynomial pour passer de l'un à l'autre.



Exercice 46 (`SET_PARTITION`)**.**
Soit :

$$\texttt{SET\_PARTITION} = \left\{\langle S\rangle \mid \exists A \subset S \ tq \ \sum_{x_i \in A} x_i = \sum_{x_j \in \overline{A}} x_j\right\} \text{ où } S = \{x_1, \ldots x_n\}$$

est un ensemble d'entiers et $\overline{A} = S \setminus A$.

Montrer que `SET_PARTITION` *est* ***NP****-complet.*

indice : *on pourra utiliser une réduction à* `SUBSET_SUM`*.*

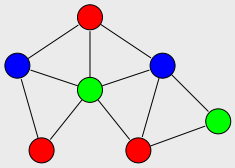
Exercice 47 (DOUBLE-SAT). *Soit DOUBLE-SAT = {⟨Φ⟩ | Φ est une formule booléenne avec 2 assignations vraies }.*

À savoir soit $B = \{b_1, \ldots, b_p\}$ les littéraux de Φ alors il existe 2 assignations de Φ (par exemple $B_1 = \{1, 0, \ldots, 1\}$ et $B_2 = \{0, 1, \ldots, 1\}$) telles ques $\Phi(B_1)$ et $\Phi(B_2)$ soient vraies.

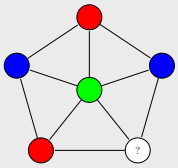
*Montrer que DOUBLE-SAT est **NP**-complet.*

Exercice 48 (3COLOR). *Soit 3COLOR = {⟨G⟩ | G est un graphe dont les nœuds peuvent être colorés avec 3 couleurs tel que aucun couple de nœuds reliés par une arête ne soit de la même couleur }.*

Exemple :



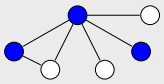
coloriage en 3 couleurs



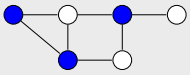
coloriage impossible

*Montrer que 3COLOR est **NP**-complet.*
***indice :** on pourra utiliser une réduction de 3SAT à 3COLOR.*

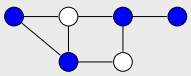
Exercice 49 (VERTEX-COVER). *Une couverture de sommets dans un graphe est un sous-ensemble de sommets tel que chaque arête touche l'un de ses sommets. **Exemple :***



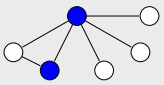
couverture



couverture



couverture minimale



couverture minimale

Soit VERTEX-COVER = {⟨G, k⟩ | G est un graphe non orienté qui a une couverture à k sommets}.

1. *Montrer que VERTEX-COVER ∈ **NP**.*
   ***a)** avec un vérificateur à temps polynomial.*
   ***b)** avec une machine de Turing non déterministe à temps polynomial.*
2. *Montrer que 3SAT $\leq_{\mathbb{P}}$ VERTEX-COVER.*
3. *En déduire que VERTEX-COVER est **NP**-complet.*

8.8 *co*NP-complétude

On peut également définir les ensembles complémentaires pour les ensembles **NP**.

Définition V.34 (classe *co***NP**)

Un langage C appartient à *co***NP** si $\overline{C} \in$ **NP**.

Définition V.35 (*co***NP**-complétude)

Un langage C est *co***NP**-complet si :

- $\overline{C} \in$ **NP**.
- $\forall D \in$ *co***NP**, $D \leq_p C$.

9 Liens entre classes de complexité

Nous donnons maintenant différentes relations qui peuvent être déduites des propriétés précédentes.

Proposition V.12

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$$

Démonstration:

Conséquences du théorème V.8 :

- $\mathbf{P} \subseteq \mathbf{NP}$: par le théorème V.8, $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n))$, et l'utiliser avec $t(n) = n^k$, $\forall k$. Conclure avec $\cup_k \texttt{TIME}(n^k) \subseteq \cup_k \texttt{NTIME}(n^k) \Leftrightarrow \mathbf{P} \subseteq \mathbf{NP}$.
- $\mathbf{NP} \subseteq \mathbf{EXP}$: par le théorème V.8, $\texttt{NTIME}(t(n)) \subseteq \texttt{TIME}(2^{O(t(n))})$, et l'utiliser avec $t(n) = n^k$, $\forall k$. Conclure avec $\cup_k \texttt{NTIME}(n^k) \subseteq \cup_k \texttt{TIME}(2^{n^k}) \Leftrightarrow \mathbf{NP} \subseteq \mathbf{EXP}$.
- $\mathbf{EXP} \subseteq \mathbf{NEXP}$: par le théorème V.8, $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n))$, et l'utiliser avec $t(n) = 2^{n^k}$, $\forall k$. Conclure avec $\cup_k \texttt{TIME}(2^{n^k}) \subseteq \cup_k \texttt{NTIME}(2^{n^k}) \Leftrightarrow \mathbf{EXP} \subseteq \mathbf{NEXP}$.

□

Nous donnons maintenant différentes relations qui peuvent être déduites des propriétés précédentes.

Proposition V.13

$$\mathbf{P} \subsetneq \mathbf{EXP}$$

Démonstration:

Conséquences du théorème V.8 : On a $\mathbf{P} \subseteq \texttt{TIME}(2^n)$. Par le théorème de Hiérarchie, on a $\texttt{TIME}(2^n) \subsetneq \texttt{TIME}(2^{n^2})$ (avec $f(n) = 2^n$ et $g(n) = 2^{n^2}$). Comme $\texttt{TIME}(2^{n^2}) \subseteq \mathbf{EXP}$, on en déduit que $\mathbf{P} \subsetneq \mathbf{EXP}$. □

Proposition V.14

$$\mathbf{NP} \subsetneq \mathbf{NEXP}$$

Démonstration:

Conséquences du théorème V.8 : On a $\mathbf{NP} \subseteq \texttt{NTIME}(2^n)$. Par le théorème de Hiérarchie non déterministe, on a $\texttt{NTIME}(2^n) \subsetneq \texttt{NTIME}(2^{n^2})$ (avec $f(n) = 2^n$ et $g(n) = 2^{n^2}$). Comme $\texttt{NTIME}(2^{n^2}) \subseteq \mathbf{NEXP}$, on en déduit que $\mathbf{NP} \subsetneq \mathbf{NEXP}$. □

Proposition V.15

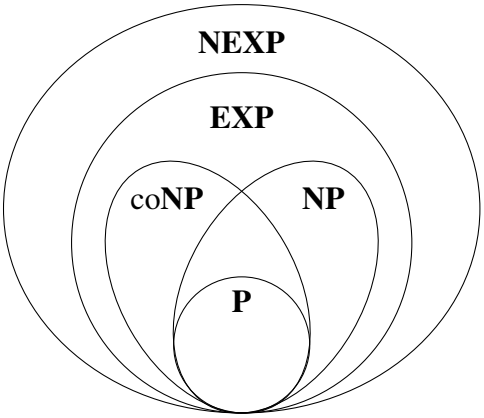
$$\mathbf{P} \subset \mathbf{NP} \cap co\mathbf{NP}$$

Démonstration:

On sait que $\mathbf{P} \subset \mathbf{NP}$ et que $co\mathbf{P} \subset co\mathbf{NP}$. Comme $\mathbf{P} = co\mathbf{P}$, on en déduit $\mathbf{P} \subset \mathbf{NP} \cap co\mathbf{NP}$. □

Ces nouveaux ensembles conduisent à la définition du monde de la complexité algorithmique tel qu'on le pense actuellement, c'est à dire sous les hypothèses que :

- $\mathbf{P} \neq \mathbf{NP}$
- $\mathbf{EXP} \neq \mathbf{NEXP}$
- $\mathbf{NP} \neq co\mathbf{NP}$


NEXP
EXP
coNP
NP
P

Chapitre VI

Complexité spatiale

On se pose le même problème que pour la complexité temporelle, mais, cette fois-ci, avec la quantité de mémoire (= le nombre de cases utilisées sur la bande) nécessaire à la résolution d'un problème.

Ce type de complexité est appelée la complexité spatiale.

Ceci va nous amener à poser la définition de nouvelles classes de complexité.

Puis, à rechercher les propriétés de ces classes entre-elles, et avec les classes de complexité temporelle.

1 Définitions

Définition : complexité spatiale pour une MT déterministe.

Soit M une MT déterministe qui s'arrête sur toutes ses entrées.

La complexité spatiale de M est une fonction $f : \mathbb{N} \to \mathbb{N}$ où $f(n)$ est le nombre maximum de cases différentes sur lesquelles M visitées sur n'importe quelle entrée de longueur n.

Définition : complexité spatiale pour une MT non déterministe.

Soit N une MT non-déterministe que s'arrête sur toutes ses entrées.

La complexité spatiale de N est la fonction $f(n)$ représentant le nombre de cases maximales utilisé lors de l'exécution de n'importe quelle branche de N pour n'importe quelle entrée de longueur n.

Remarques :

- Si la complexité spatiale de M est $f(n)$, on dit M s'exécute en espace $f(n)$.
- Défini ici en nombre de cases utilisées par la MT.
  Défini en octets sur un processeur.

Soit $f : \mathbb{N} \to \mathbb{R}$ une fonction.

On définit alors :

Définition VI.1 ($\texttt{SPACE}(f(n))$)

$\texttt{SPACE}(f(n))$ est l'ensemble des langages décidés par une MTD M qui s'exécute en espace $f(n)$.

Définition VI.2 ($\texttt{NSPACE}(f(n))$)

$\texttt{NSPACE}(f(n))$ est l'ensemble des langages décidés par une MTND M qui s'exécute en espace $f(n)$.

Note :

Dans un premier temps, on considérera que $f(n) \geq n$, car la bande d'entrée contient l'entrée qui elle-même est de taille n. Donc, si $f(n) < n$ cela signifierait que l'entrée n'a pas été lue en entier.

2 Premières propriétés

Théorème VI.1 (Savitch)

soit une fonction $f : \mathbb{N} \to \mathbb{R}$ telle que $f(n) \geq n$, alors $\texttt{NSPACE}(f(n)) \subseteq \texttt{SPACE}(f(n)^2)$.

Autrement dit, toute MTND qui s'exécute en espace $f(n)$ peut s'exécuter en espace $\texttt{SPACE}(f(n)^2)$ sur une MTD.

Démonstration: (idée principale)

On peut simuler une MTND en espace $f(n)$ sur une MTD en parcourant l'arbre d'exécution de manière préfixe (toutes les branches s'arrêtent). Une branche d'espace $f(n)$ s'exécute au plus en temps $f(n)$ (on ne visite pas plus de case que l'on a de temps). On a alors besoin de $f(n)$ espace de stockage sur une profondeur d'exécution $f(n)$, d'où une simulation en espace déterministe au plus de $f(n)^2$. □

Définition VI.3 (PSPACE)

PSPACE est la classe des langages qui sont décidables en espace polynomial par une MTD, à savoir : **PSPACE** $= \cup_k \texttt{SPACE}(n^k)$

Définition VI.4 (NPSPACE)

NPSPACE est la classe des langages qui sont décidables en espace polynomial par une MTND, à savoir : **NPSPACE** $= \cup_k \texttt{NSPACE}(n^k)$

Théorème VI.2

PSPACE = **NPSPACE**

Démonstration:

Par le théorème de Savitch, $\texttt{NSPACE}(f(n)) \subseteq \texttt{SPACE}(f(n)^2)$. Si $f(n)$ est un polynôme, $f(n)^2$ aussi. □

Proposition VI.1

SPACE = *co***SPACE**

Démonstration:

$\forall f(n), \texttt{SPACE}(f(n)) = \texttt{coSPACE}(f(n))$. Même idée que pour **P** = *co***P**. □

Proposition VI.2

PSPACE = co**NPSPACE**

Démonstration:

PSPACE = **NPSPACE**, donc *co***PSPACE** = *co***NPSPACE**. Conclure avec **SPACE** = *co***SPACE**. □

On a donc : **PSPACE** = co**PSPACE** = **NPSPACE** = co**NPSPACE**

3 Lien avec la complexité temporelle

Théorème VI.3
P ⊆ PSPACE

DÉMONSTRATION:

Si un langage est décidé par une MTD M en temps $f(n)$, alors M visite au plus $f(n)$ cellules différentes. Donc, $\texttt{TIME}(f(n)) \subseteq \texttt{SPACE}(f(n))$, et par conséquent **P** ⊆ **PSPACE**. □

Théorème VI.4
NP ⊆ PSPACE

DÉMONSTRATION:

De la même façon, $\texttt{NTIME}(f(n)) \subseteq \texttt{NSPACE}(f(n))$. Donc, **NP** ⊆ **NPSPACE**. Or, on a montré que **PSPACE** = **NPSPACE**. En conséquence, **NP** ⊆ **PSPACE**. □

Lemme VI.1
une MTD M en espace $f(n)$ a au plus $2^{O(f(n))}$ configurations différentes.

DÉMONSTRATION:

Notons $|Q|$ le nombre d'états de M et $|\Gamma|$ le nombre de symboles de son alphabet (de bande). Si M est en espace $f(n)$, le nombre de bandes différentes est $|\Gamma|^{f(n)}$ et le nombre de couples (position,état) différents sur la bande est $f(n).|Q|$.
En conséquence, le nombre de configurations différentes est donc de : $f(n).|Q|.|\Gamma|^{f(n)} = f(n).2^{O(f(n))} = 2^{O(f(n))}$. □

Théorème VI.5
PSPACE ⊆ EXPTIME

DÉMONSTRATION:

Une MT qui ne boucle pas avec $2^{O(f(n))}$ configurations différentes est au plus en temps $2^{O(f(n))}$. Ceci et le lemme précédent implique $\texttt{SPACE}(f(n)) \subseteq \texttt{TIME}(2^{O(f(n))})$. D'où **PSPACE** ⊆ **EXPTIME**. □

Donc, un programme qui utilise une taille de mémoire polynomiale s'exécute au pire en temps exponentiel.

4 Espace logarithmique

Pour l'instant nous avons toujours considéré que $f(n) \geq n$ à savoir, la bande doit au moins contenir l'entrée.

En revanche, il pourrait être intéressant de savoir ce qu'un décideur utilise comme mémoire **en plus** de l'entrée w. Il devrait alors être possible de passer en sub-linéaire.

On considère à partir de maintenant les exécutions sur la MT suivante à 2 bandes :

- la première bande est la bande d'entrée (contient l'entrée de la MT et est en lecture seule)
- la seconde bande est la bande de travail (en lecture/écriture)

Afin que :

- Seul l'espace utilisé sur la bande de travail est compté.
- Une **configuration** contient l'état de la MT, la bande de travail, les positions des pointeurs sur chaque bande.

Remarque : cette approche n'a pas de sens dans le cadre de la complexité temporelle (temps de lecture de la bande + accès séquentiel aux données).

Définition VI.5 (classe **L**)

L est la classe des langages qui sont décidables en espace logarithmique sur une MTD.
$\mathbf{L} = \texttt{SPACE}(\log n)$

Définition VI.6 (classe **NL**)

NL est la classe des langages qui sont décidables en espace logarithmique sur une MTND.
$\mathbf{NL} = \texttt{NSPACE}(\log n)$

Ces espaces nous intéressent à plusieurs titres :

- ils sont assez gros pour permettre la résolution de problèmes intéressants.
- ils sont robustes (indépendant du modèle de machines ou de la méthode de codage de l'entrée, pour peu qu'elle reste raisonnable).

Pour faire court, les langages à espace logarithmique sont ceux pour lesquels l'algorithme que le reconnaît peut fonctionner avec $O(n)$ variables binaires contenant $O(\log(n))$ (= un nombre fixe de variables numériques de $k.\log_2 n$ bits).
Le langage $A = \left\{0^k 1^k \mid k > 0\right\} \in \mathbf{L}$. En effet, après vérification de syntaxe (en $SPACE(1)$), utiliser deux variables n_0 et n_1 pour calculer et stocker le nombre de 0 et de 1 (en $2.\texttt{SPACE}(\log_2 n)$) pour une entrée de taille n). Comparer n_0 et n_1 et conclure.

Le langage $\texttt{PATH} \in \mathbf{NL}$. En effet, Utiliser deux variables : l'identificateur du sommet courant x et la longueur i du chemin de la manière suivante :

$N(\langle G, s, t\rangle) =$

```
1 Initialisation : x = s, i = 0, nSommets = nombre de sommets de G
2 tant que i < nSommets
3 |   choix non déterministe d'une arête (x, u) dans G
4 |   si u = t alors accepter sinon x = u
5 |   incrémenter i
6 rejeter
```

On utilise en espace 3 variables $\log_2 n$ bits et un algorithme non déterministe. Donc on est en $\texttt{NSPACE}(\log_2 n)$, donc dans **NL**.

De façon similaire à la question de savoir si **P** = **NP**, il se pose la question de savoir si **L** = **NL**.
Pour ce faire, on veut définir une **NL**-complétude et si l'un des langages *NL*-complet est dans *L*, alors on pourra conclure **L** = **NL**.
On a besoin de définir trois chosess :

- les fonctions calculables en espace logarithmique
- une réduction en espace logarithmique
- la **NL**-complétude

On utilisera esp.log. pour espace logarithmique.

Définition VI.7 (transducteur)

Un transducteur est une MT avec une bande d'entrée en lecture seule, une bande de travail et une bande de sortie en écriture seule.

Définition VI.8 (fonction calculable en espace logarithmique)

Une fonction $f(w)$ calculable en esp.log. est :

- un transducteur qui prend en entrée $\langle w \rangle$, avec $n = |\langle w \rangle|$.
- dont la bande de travail contient au plus $O(\log n)$ symboles,
- et qui renvoie la valeur stockée sur sa bande de sortie au moment où le transducteur s'arrête.

Définition VI.9 (réduction en espace logarithmique)

Un langage A est dit réductible en esp.log. à un langage B si il existe une réduction de A à B par une fonction calculable en esp.log..
On note : $A \leq_{\mathbf{L}} B$
Rappel : réduction = fonction calculable tq $\forall w, w \in A \Leftrightarrow f(w) \in B$.

Définition VI.10 (**NL**-complétude)

Un langage B est dit **NL**-complet si :

- $B \in \mathbf{NL}$
- $\forall A \in \mathbf{NL}$, $A \leq_{\mathbf{L}} B$, à savoir tout autre langage de **NL** peut se réduire à B.

Théorème VI.6

Si $A \leq_{\mathbf{L}} B$ et $B \in \mathbf{L}$ alors $A \in \mathbf{L}$.

Démonstration:
Pour décider A en esp.log., effectuer la transduction de A vers B (donc en esp.log.), décider B (en esp.log.). □

Corollaire VI.1

si n'importe quel langage **NL**-complet est dans **L**, alors **L** = **NL**

Démonstration:
si B est **NL**-complet alors $\forall A \in \mathbf{NL}$, $A \leq_{\mathbf{L}} B$.
L'existence de cette réduction implique que $\forall w, w \in A \Leftrightarrow f(w) \in B$.
Si $B \in \mathbf{L}$, le décideur de B permet donc de décider $f(A)$en espace L.
Donc, $A \in \mathbf{L}$ (et pour tout $A \in \mathbf{NL}$). D'où **L** = **NL**. □

Par ailleurs, on peut démontrer que :

Proposition VI.3

- PATH est **NL**-complet.
- $\overline{\texttt{PATH}}$ est **NL**-complet.

Ce qui pour conséquence que :

Théorème VI.7

NL = co**NL**

DÉMONSTRATION:

1. $\forall x \in \mathbf{NL}$, nous avons $x \leq_{\mathbf{L}} \texttt{PATH}$ puisque PATH est **NL**-complet. Alors, par la même fonction de réduction, nous avons $\overline{x} \leq_{\mathbf{L}} \overline{\texttt{PATH}}$. Donc, $\overline{x} \in \mathbf{NL}$ puisque $\overline{\texttt{PATH}} \in \mathbf{NL}$. En conséquence, $\mathbf{NL} \subseteq \texttt{co}\mathbf{NL}$.
2. $\forall x \in \texttt{co}\mathbf{NL}$, on a $\overline{x} \in \mathbf{NL}$. De façon similaire, $\overline{x} \leq_{\mathbf{L}} \texttt{PATH}$ et $x \leq_{\mathbf{L}} \overline{\texttt{PATH}}$. Encore une fois, puisque $\overline{\texttt{PATH}} \in \mathbf{NL}$, on a aussi $x \in \mathbf{NL}$. Donc, $\texttt{co}\mathbf{NL} \subseteq \mathbf{NL}$.

□

Corollaire VI.2

NL $\subseteq$ **P**

DÉMONSTRATION:

- **démonstration 1 :** le nombre de configuration d'un MTD sur une bande de taille $f(n)$ a au plus $c(n) = 2^{O(f(n))}$ configurations différentes. En eps.log., $f(n) = O(\log(n))$, d'où $\exists k$, $c(n) \leq 2^{log(n^k)} = n^k$. En conséquence, cette MTD est en temps $\texttt{TIME}(n^k) \subset \mathbf{P}$.
- **démonstration 2 :** PATH est **NL**-complet. Avec le même argument que la démonstration précédente, la réduction en esp.log. vers PATH est dans **P**. Comme $\texttt{PATH} \in \mathbf{P}$ et PATH **NL**-complet, tout langage de **NL** peut être décidé par PATH en temps polynomial.

□

Définition VI.11 (Fonction constructible en espace)

Une fonction $f(n) \geq n$, non décroissante, est constructible en espace si il existe une fonction calculable F qui calcule $F : 1^n \mapsto 1^{f(n)}$ en espace $O(f(n))$.

Théorème VI.8 (hiérarchie en espace)

Si g est constructible en espace et $f(n) = o(g(n))$, alors $\texttt{SPACE}(f(n)) \subsetneq \texttt{SPACE}(g(n))$.

Corollaire VI.3

NL $\subsetneq$ **PSPACE**

DÉMONSTRATION:

Par le théorème de hiérarchie en espace, on a **NL** $\subsetneq$ **NPSPACE**. Or **PSPACE** = **NPSPACE**. D'où **NL** $\subsetneq$ **PSPACE**. □

5 Synthèse des espaces de complexité

$$\mathbf{L} \subseteq \mathbf{NL} = \mathbf{coNL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$$

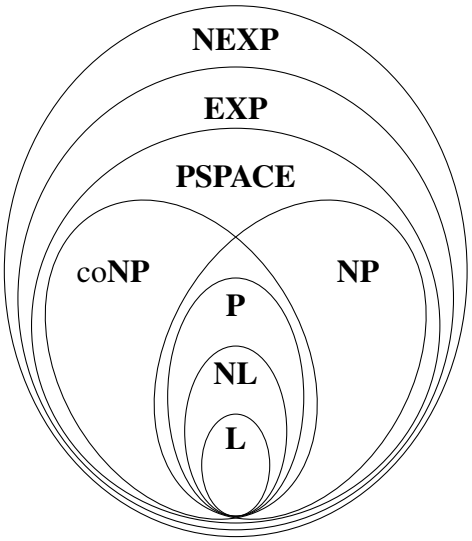
En conséquence,

- soit $\mathbf{coNL} \subsetneq \mathbf{P}$.
- soit $\mathbf{P} \subsetneq \mathbf{PSPACE}$.

Malheureusement, on ne sait pas laquelle des deux est vraie ou si les deux sont vraies (on pense qu'il s'agit de ce dernier cas).
En résumé,

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME}$$

Donc, au moins une des inclusions est stricte.



Remarques :

- actuellement, on ne sait pas laquelle est stricte.
- beaucoup pensent qu'elles le sont toutes.

Bibliographie

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity : a modern approach*. Cambridge University Press, Cambridge, New York (N.Y.), 2009.

[DK00] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity*. Wiley-Interscience, 2000.

[DW83] Martin Davis and Elaine J. Weyuker. *Computability, complexity, and languages : fundamentals of theoretical computer science*. Computer science and applied mathematics. Academic Press, New York, 1983.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractability : A guide to the theory of NP-completeness*. A series of books in the mathematical sciences / Victor Klee, editor. W. H. Freeman, New York, N.Y, 1979. Réimpressions : 1999, 2000 (22e), 2002 (23e), 2003 (24e), 200 ? (25e), 2008 (27e).

[HUM01] John E. Hopcroft, Jeffrey D. Ullman, and Rajeev Motwani. *Introduction to automata theory, languages, and computation*. Addison-Wesley, Boston, San Francisco, 2001.

[Jon97] Neil Jones. *Computability and complexity : from a programming perspective*. Foundations of computing. MIT Press, Cambridge (Mass.), 1997.

[Pap94] Christos Papadimitriou. *Computational complexity*. Addison-Wesley, Reading (Mass.), 1994.

[Sip06] Michael Sipser. *Introduction to the theory of computation*. Thomson Course Technology, Boston, 2006.