TLANG – Décidabilité et Complexité

Didier Lime

École Centrale de Nantes – LS2N

Année 2017 – 2018

Introduction

- **Algorithme** : Méthodes de résolution automatique d'un **problème** ;

Introduction

- **Algorithme** : Méthodes de résolution automatique d'un **problème** ;
- Pour un problème donné, **plusieurs** algorithmes de résolution sont possibles ;

Introduction

- **Algorithme** : Méthodes de résolution automatique d'un **problème** ;
- Pour un problème donné, **plusieurs** algorithmes de résolution sont possibles ;
- Comment **comparer** ces solutions ?

Introduction

- **Algorithme** : Méthodes de résolution automatique d'un **problème** ;
- Pour un problème donné, **plusieurs** algorithmes de résolution sont possibles ;
- Comment **comparer** ces solutions ?
- Peut-on en déduire la **difficulté** du problème ?

Introduction

- **Algorithme** : Méthodes de résolution automatique d'un **problème** ;
- Pour un problème donné, **plusieurs** algorithmes de résolution sont possibles ;
- Comment **comparer** ces solutions ?
- Peut-on en déduire la **difficulté** du problème ?
- Peut-on alors en déduire une **hiérarchie** de problèmes ?

Plan

Problèmes algorithmique

- On définit un **problème algorithmique** par :

Problèmes algorithmique

- On définit un **problème algorithmique** par :
  - un ensemble d'entrées : les **données** ;

Problèmes algorithmique

- On définit un **problème algorithmique** par :
  - un ensemble d'entrées : les **données** ;
  - une question (si possible formalisée) ;

Problèmes algorithmique

- On définit un **problème algorithmique** par :
  - un ensemble d'entrées : les **données** ;
  - une question (si possible formalisée) ;
  - un ensemble de sorties répondant à la question : les **résultats**.

Problèmes algorithmique

- On définit un **problème algorithmique** par :
  - un ensemble d'entrées : les **données** ;
  - une question (si possible formalisée) ;
  - un ensemble de sorties répondant à la question : les **résultats**.
- Problème d'**optimisation** : le résultat doit être optimal selon une certaine mesure ;

Problèmes algorithmique

- On définit un **problème algorithmique** par :
  - un ensemble d'entrées : les **données** ;
  - une question (si possible formalisée) ;
  - un ensemble de sorties répondant à la question : les **résultats**.
- Problème d'**optimisation** : le résultat doit être optimal selon une certaine mesure ;
- Problème de **décision** : le résultat est « oui » ou « non ».

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**
  - booléens $\{0, 1\}$;

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**
  - booléens $\{0, 1\}$;
  - entiers $\{0, 1, \ldots, 9\}$;

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**
  - booléens $\{0, 1\}$;
  - entiers $\{0, 1, \ldots, 9\}$;
  - rationnels $\{0, 1, \ldots, 9, /\}$;

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**
  - booléens $\{0, 1\}$;
  - entiers $\{0, 1, \ldots, 9\}$;
  - rationnels $\{0, 1, \ldots, 9, /\}$;
  - caractères $\{a, \ldots, z\}$;

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**
  - booléens $\{0, 1\}$;
  - entiers $\{0, 1, \ldots, 9\}$;
  - rationnels $\{0, 1, \ldots, 9, /\}$;
  - caractères $\{a, \ldots, z\}$;
  - réels : approximation (rationnels) ou purement symbolique $\{0, 1, \ldots, 9, \pi, e, \sqrt{2} \ldots\}$.

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**
  - booléens $\{0, 1\}$;
  - entiers $\{0, 1, \ldots, 9\}$;
  - rationnels $\{0, 1, \ldots, 9, /\}$;
  - caractères $\{a, \ldots, z\}$;
  - réels : approximation (rationnels) ou purement symbolique $\{0, 1, \ldots, 9, \pi, e, \sqrt{2} \ldots\}$.
  - ...

Problèmes algorithmiques : Encodage

- Tout algorithme opère non pas sur les objets eux-mêmes mais sur leurs **représentations** p.ex. entiers et bases ;
- Ces représentations correspondent à des **mots** sur un alphabet **fini**
  - booléens $\{0, 1\}$;
  - entiers $\{0, 1, \ldots, 9\}$;
  - rationnels $\{0, 1, \ldots, 9, /\}$;
  - caractères $\{a, \ldots, z\}$;
  - réels : approximation (rationnels) ou purement symbolique $\{0, 1, \ldots, 9, \pi, e, \sqrt{2} \ldots\}$.
  - ...
- On peut donc se restreindre à des **entrées** et **sorties** sous forme de **mots** (finis) sur un alphabet fini.

Problèmes de décision

- On peut ramener des problèmes **généraux** à des problèmes de décision (mais parfois un nombre **infini** de tels problèmes) :

Problèmes de décision

- On peut ramener des problèmes **généraux** à des problèmes de décision (mais parfois un nombre **infini** de tels problèmes) :

Taille du plus court chemin (Optimisation)

Entrées: Un graphe pondéré G de taille n. Deux nœuds u et v.
Résultat: La taille du plus court chemin allant de u à v dans G.

Problèmes de décision

- On peut ramener des problèmes **généraux** à des problèmes de décision (mais parfois un nombre **infini** de tels problèmes) :

Taille du plus court chemin (Optimisation)

Entrées: Un graphe pondéré G de taille n. Deux nœuds u et v.
Résultat: La taille du plus court chemin allant de u à v dans G.

Taille du plus court chemin (Décision)

Entrées: Un graphe pondéré G de taille n. Deux nœuds u et v.
Résultat: Existe-t-il un chemin allant de u à v en moins de n arêtes ?

Problèmes de décision

- On peut ramener des problèmes **généraux** à des problèmes de décision (mais parfois un nombre **infini** de tels problèmes) :

Taille du plus court chemin (Optimisation)

Entrées: Un graphe pondéré G de taille n. Deux nœuds u et v.
Résultat: La taille du plus court chemin allant de u à v dans G.

Taille du plus court chemin (Décision)

Entrées: Un graphe pondéré G de taille n. Deux nœuds u et v.
Résultat: Existe-t-il un chemin allant de u à v en moins de n arêtes ?

Addition (Décision)

Entrées: Trois entiers relatifs a, b et c.
Résultat: c est-il la somme de a et b ?

Problèmes de décision

Factorisation

Entrées: Un entier a.
Résultat: La liste de ses facteurs premiers.

Problèmes de décision

Factorisation

Entrées: Un entier a.
Résultat: La liste de ses facteurs premiers.

Factorisation (Décision) ? ?

Entrées: Un entier a. Une liste d'entiers premiers $(p_i)_i$ inférieurs à a.
Résultat: Les p_i sont-ils les facteurs premiers de a ?

Problèmes de décision

Factorisation

Entrées: Un entier a.
Résultat: La liste de ses facteurs premiers.

Factorisation (Décision) ? ?

Entrées: Un entier a. Une liste d'entiers premiers $(p_i)_i$ inférieurs à a.
Résultat: Les p_i sont-ils les facteurs premiers de a ?

- Le problème de décision se résoud **facilement** par le calcul de $\prod_i p_i$;

Problèmes de décision

Factorisation

Entrées: Un entier a.
Résultat: La liste de ses facteurs premiers.

Factorisation (Décision) ? ?

Entrées: Un entier a. Une liste d'entiers premiers $(p_i)_i$ inférieurs à a.
Résultat: Les p_i sont-ils les facteurs premiers de a ?

- Le problème de décision se résoud **facilement** par le calcul de $\prod_i p_i$;
- Le problème originel est bien plus compliqué ! (principe de la **cryptographie asymétrique**) ;

Problèmes de décision

Factorisation

Entrées: Un entier a.
Résultat: La liste de ses facteurs premiers.

Factorisation (Décision) ? ?

Entrées: Un entier a. Une liste d'entiers premiers $(p_i)_i$ inférieurs à a.
Résultat: Les p_i sont-ils les facteurs premiers de a ?

- Le problème de décision se résoud **facilement** par le calcul de $\prod_i p_i$;
- Le problème originel est bien plus compliqué ! (principe de la **cryptographie asymétrique**) ;
- **Mais** on peut résoudre le problème originel à l'aide du problème de décision.

Problèmes de décision

Factorisation

Entrées: Un entier a.
Résultat: La liste de ses facteurs premiers.

Factorisation (Décision) ? ?

Entrées: Un entier a. Une liste d'entiers premiers $(p_i)_i$ inférieurs à a.
Résultat: Les p_i sont-ils les facteurs premiers de a ?

- Le problème de décision se résoud **facilement** par le calcul de $\prod_i p_i$;
- Le problème originel est bien plus compliqué ! (principe de la **cryptographie asymétrique**) ;
- **Mais** on peut résoudre le problème originel à l'aide du problème de décision. Comment ?

Problèmes de décision

Factorisation

Entrées: Un entier a.
Résultat: La liste de ses facteurs premiers.

Factorisation (Décision) ? ?

Entrées: Un entier a. Une liste d'entiers premiers $(p_i)_i$ inférieurs à a.
Résultat: Les p_i sont-ils les facteurs premiers de a ?

- Le problème de décision se résoud **facilement** par le calcul de $\prod_i p_i$;
- Le problème originel est bien plus compliqué ! (principe de la **cryptographie asymétrique**) ;
- **Mais** on peut résoudre le problème originel à l'aide du problème de décision. Comment ? il y a un nombre **fini** de listes d'entiers premiers possibles.

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;
  - Se **formalisent** bien.

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;
  - Se **formalisent** bien.
- On peut exprimer tous les problèmes de décision comme un problème d'**appartenance** à un ensemble :

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;
  - Se **formalisent** bien.
- On peut exprimer tous les problèmes de décision comme un problème d'**appartenance** à un ensemble :
  - soit $\mathcal{P}$ un problème de décision ;

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;
  - Se **formalisent** bien.
- On peut exprimer tous les problèmes de décision comme un problème d'**appartenance** à un ensemble :
  - soit $\mathcal{P}$ un problème de décision ;
  - soit X l'ensemble des entrées telles que la réponse est « oui » ;

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;
  - Se **formalisent** bien.
- On peut exprimer tous les problèmes de décision comme un problème d'**appartenance** à un ensemble :
  - soit $\mathcal{P}$ un problème de décision ;
  - soit X l'ensemble des entrées telles que la réponse est « oui » ;
  - alors le problème $\mathcal{P}'$ suivant est équivalent à $\mathcal{P}$:

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;
  - Se **formalisent** bien.
- On peut exprimer tous les problèmes de décision comme un problème d'**appartenance** à un ensemble :
  - soit $\mathcal{P}$ un problème de décision ;
  - soit X l'ensemble des entrées telles que la réponse est « oui » ;
  - alors le problème $\mathcal{P}'$ suivant est équivalent à $\mathcal{P}$:

$\mathcal{P}'$

Entrées: x
Résultat: x appartient-il à X ?

Problèmes de décision

- Les problèmes de décision sont d'un intérêt tout particulier
  - Plus **simples** ;
  - Ont tous le **même type** de résultat (facilite les mises en relation) ;
  - Se **formalisent** bien.
- On peut exprimer tous les problèmes de décision comme un problème d'**appartenance** à un ensemble :
  - soit $\mathcal{P}$ un problème de décision ;
  - soit X l'ensemble des entrées telles que la réponse est « oui » ;
  - alors le problème $\mathcal{P}'$ suivant est équivalent à $\mathcal{P}$:

$\mathcal{P}'$

Entrées: x
Résultat: x appartient-il à X ?

- On peut donc naturellement confondre $\mathcal{P}$ et X.

Plan

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ est la fonction de **transition** ;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \rightarrow \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \rightarrow \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \rightarrow \mathcal{B}^*$ est la fonction de **sortie** ;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \to \mathcal{B}^*$ est la fonction de **sortie** ;
- L'**exécution** de l'algorithme sur l'entrée w est la suite :

$$\mathcal{E}(w), \mathcal{T}(\mathcal{E}(w)), \mathcal{T}(\mathcal{T}(\mathcal{E})), \ldots$$

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \rightarrow \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \rightarrow \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \rightarrow \mathcal{B}^*$ est la fonction de **sortie** ;
- L'**exécution** de l'algorithme sur l'entrée w est la suite :

$$\mathcal{E}(w), \mathcal{T}(\mathcal{E}(w)), \mathcal{T}(\mathcal{T}(\mathcal{E})), \ldots$$

- Si la suite est **infinie**, l'algorithme ne se **termine pas** sur w ;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \to \mathcal{B}^*$ est la fonction de **sortie** ;
- L'**exécution** de l'algorithme sur l'entrée w est la suite :

$$\mathcal{E}(w), \mathcal{T}(\mathcal{E}(w)), \mathcal{T}(\mathcal{T}(\mathcal{E})), \ldots$$

- Si la suite est **infinie**, l'algorithme ne se **termine pas** sur w ;
- Si la suite est **finie** (disons n termes), l'algorithme se termine sur w et **produit** $\mathcal{S}(\mathcal{T}^n(\mathcal{E}(w)))$;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \to \mathcal{B}^*$ est la fonction de **sortie** ;
- L'**exécution** de l'algorithme sur l'entrée w est la suite :

$$\mathcal{E}(w), \mathcal{T}(\mathcal{E}(w)), \mathcal{T}(\mathcal{T}(\mathcal{E})), \ldots$$

- Si la suite est **infinie**, l'algorithme ne se **termine pas** sur w ;
- Si la suite est **finie** (disons n termes), l'algorithme se termine sur w et **produit** $\mathcal{S}(\mathcal{T}^n(\mathcal{E}(w)))$;
- Cette définition impose qu'un algorithme soit **séquentiel** ;

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \to \mathcal{B}^*$ est la fonction de **sortie** ;
- L'**exécution** de l'algorithme sur l'entrée w est la suite :

$$\mathcal{E}(w), \mathcal{T}(\mathcal{E}(w)), \mathcal{T}(\mathcal{T}(\mathcal{E})), \ldots$$

- Si la suite est **infinie**, l'algorithme ne se **termine pas** sur w ;
- Si la suite est **finie** (disons n termes), l'algorithme se termine sur w et **produit** $\mathcal{S}(\mathcal{T}^n(\mathcal{E}(w)))$;
- Cette définition impose qu'un algorithme soit **séquentiel** ;
- Reste à **restreindre** $\mathcal{E}, \mathcal{S}$ et $\mathcal{T}$.

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \ldots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \to \mathcal{B}^*$ est la fonction de **sortie** ;
- L'**exécution** de l'algorithme sur l'entrée w est la suite :

$$\mathcal{E}(w), \mathcal{T}(\mathcal{E}(w)), \mathcal{T}(\mathcal{T}(\mathcal{E})), \ldots$$

- Si la suite est **infinie**, l'algorithme ne se **termine pas** sur w ;
- Si la suite est **finie** (disons n termes), l'algorithme se termine sur w et **produit** $\mathcal{S}(\mathcal{T}^n(\mathcal{E}(w)))$;
- Cette définition impose qu'un algorithme soit **séquentiel** ;
- Reste à **restreindre** $\mathcal{E}, \mathcal{S}$ et $\mathcal{T}$. Pourquoi ?

Algorithmes et configurations

Soit $\mathcal{P}$ un problème algorithmique. On **encode** les entrées sur l'alphabet A, les sorties sur l'alphabet B.

- Soit $X = \{x_1, x_2, \dots\}$ un ensemble de **variables** ;
- Une **valuation** sur X est appelée **configuration**. $\mathcal{C}$ est l'ensemble des configurations sur X ;
- On définit un algorithme par la donnée de trois fonctions :
  - $\mathcal{E} : \mathcal{A}^* \to \mathcal{C}$ est la fonction d'**entrée** ;
  - $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ est la fonction de **transition** ;
  - $\mathcal{S} : \mathcal{C} \to \mathcal{B}^*$ est la fonction de **sortie** ;
- L'**exécution** de l'algorithme sur l'entrée w est la suite :

$$\mathcal{E}(w), \mathcal{T}(\mathcal{E}(w)), \mathcal{T}(\mathcal{T}(\mathcal{E})), \dots$$

- Si la suite est **infinie**, l'algorithme ne se **termine pas** sur w ;
- Si la suite est **finie** (disons n termes), l'algorithme se termine sur w et **produit** $\mathcal{S}(\mathcal{T}^n(\mathcal{E}(w)))$;
- Cette définition impose qu'un algorithme soit **séquentiel** ;
- Reste à **restreindre** $\mathcal{E}, \mathcal{S}$ et $\mathcal{T}$. Pourquoi ? Décider V : $\mathcal{C} = V, \mathcal{E} = id, \mathcal{T} = \emptyset, \mathcal{S} = \chi(V)$ (fonction indicatrice de V).

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).
- $\mathcal{A}$ **semi-décide** P si :

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).
- $\mathcal{A}$ **semi-décide** P si :
  - son alphabet d'entrée inclut A ;

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).
- $\mathcal{A}$ **semi-décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{1\}$;

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).
- $\mathcal{A}$ **semi-décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).
- $\mathcal{A}$ **semi-décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ ne se termine pas.

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).
- $\mathcal{A}$ **semi-décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ ne se termine pas.
- P est **décidable** (resp. **semi-décidable**) s'il existe un algorithme qui le décide (resp. le semi-décide) ;

Décidabilité

Définition

Soit A un **alphabet**, P une partie de A^* (donc un problème de décision), et $\mathcal{A}$ un **algorithme**.

- $\mathcal{A}$ **décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{0, 1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ produit 0 (à partir de w).
- $\mathcal{A}$ **semi-décide** P si :
  - son alphabet d'entrée inclut A ;
  - son alphabet de sortie inclut $\{1\}$;
  - pour tout mot $w \in P$, $\mathcal{A}$ produit 1 (à partir de w) ;
  - pour tout mot $w \notin P$, $\mathcal{A}$ ne se termine pas.
- P est **décidable** (resp. **semi-décidable**) s'il existe un algorithme qui le décide (resp. le semi-décide) ;
- Si P n'est pas décidable, on dit qu'il est **indécidable**.

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N
- L'évolution de q (la prochaine instruction) ne dépend que de sa valeur actuelle et de la lettre courante (suivant une fonction $f : [1..N] \times A \to [1..N]$).

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N
- L'évolution de q (la prochaine instruction) ne dépend que de sa valeur actuelle et de la lettre courante (suivant une fonction $f : [1..N] \times A \to [1..N]$).
- Le résultat (booléen) ne dépend que de la valeur de q une fois le mot lu en entier (selon une fonction $F : [1..N] \to \{0, 1\}$).

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N
- L'évolution de q (la prochaine instruction) ne dépend que de sa valeur actuelle et de la lettre courante (suivant une fonction $f : [1..N] \times A \to [1..N]$).
- Le résultat (booléen) ne dépend que de la valeur de q une fois le mot lu en entier (selon une fonction $F : [1..N] \to \{0, 1\}$).

Plus formellement :

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N
- L'évolution de q (la prochaine instruction) ne dépend que de sa valeur actuelle et de la lettre courante (suivant une fonction $f : [1..N] \times A \rightarrow [1..N]$).
- Le résultat (booléen) ne dépend que de la valeur de q une fois le mot lu en entier (selon une fonction $F : [1..N] \rightarrow \{0, 1\}$).

Plus formellement :

- $X = \{s, i, q\}$ // Variables de l'algorithme ;

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N
- L'évolution de q (la prochaine instruction) ne dépend que de sa valeur actuelle et de la lettre courante (suivant une fonction $f : [1..N] \times A \to [1..N]$).
- Le résultat (booléen) ne dépend que de la valeur de q une fois le mot lu en entier (selon une fonction $F : [1..N] \to \{0, 1\}$).

Plus formellement :

- $X = \{s, i, q\}$ // Variables de l'algorithme ;
- $\forall w, \mathcal{E}(w) = (w, 0, 0)$ // Fonction d'entrée ;

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N
- L'évolution de q (la prochaine instruction) ne dépend que de sa valeur actuelle et de la lettre courante (suivant une fonction $f : [1..N] \times A \to [1..N]$).
- Le résultat (booléen) ne dépend que de la valeur de q une fois le mot lu en entier (selon une fonction $F : [1..N] \to \{0, 1\}$).

Plus formellement :

- $X = \{s, i, q\}$ // Variables de l'algorithme ;
- $\forall w, \mathcal{E}(w) = (w, 0, 0)$ // Fonction d'entrée ;
- $\forall (s, i, q) \in \mathcal{C}, \mathcal{T}((s, i, q)) = (s, i + 1, f(q, s[i]))$ // Transitions ;

Exemple : Automates finis

Une classe **très simple** d'algorithmes :

- Une mémoire finie en lecture seule lue de gauche à droite : on lit le mot d'entrée **lettre par lettre** ;
- Un **programme fini** avec un **compteur de programme** : une variable entière q entre 1 et N
- L'évolution de q (la prochaine instruction) ne dépend que de sa valeur actuelle et de la lettre courante (suivant une fonction $f : [1..N] \times A \to [1..N]$).
- Le résultat (booléen) ne dépend que de la valeur de q une fois le mot lu en entier (selon une fonction $F : [1..N] \to \{0, 1\}$).

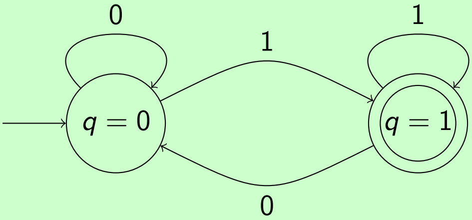
Plus formellement :

- $X = \{s, i, q\}$ // Variables de l'algorithme ;
- $\forall w, \mathcal{E}(w) = (w, 0, 0)$ // Fonction d'entrée ;
- $\forall (s, i, q) \in \mathcal{C}, \mathcal{T}((s, i, q)) = (s, i + 1, f(q, s[i]))$ // Transitions ;
- $\forall c = (w, |w| + 1, q), \mathcal{S}(c) = F(q)$ // Fonction de sortie.

Exemple : Automates finis

On peut représenter un algorithme de cette classe graphiquement :

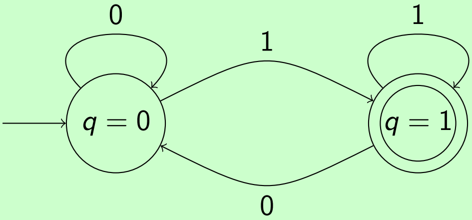
Exemple



Exemple : Automates finis

On peut représenter un algorithme de cette classe graphiquement :

Exemple

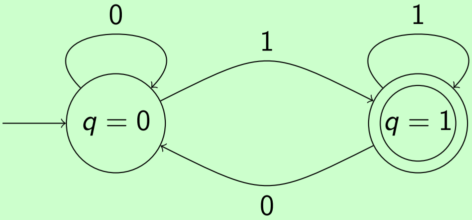


w (écrit en base 2) est-il **impair** ?

Exemple : Automates finis

On peut représenter un algorithme de cette classe graphiquement :

Exemple



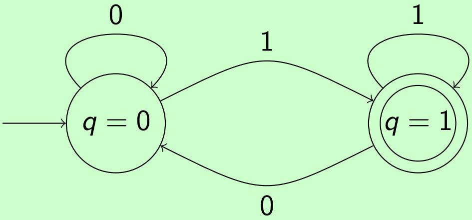
w (écrit en base 2) est-il **impair** ?

L'ensemble $2\mathbb{N}$ et son complémentaire dans $\mathbb{N}$ sont donc ≪ **décidables** par automate fini ≫ (AF-décidable).

Exemple : Automates finis

On peut représenter un algorithme de cette classe graphiquement :

Exemple



w (écrit en base 2) est-il **impair** ?

L'ensemble $2\mathbb{N}$ et son complémentaire dans $\mathbb{N}$ sont donc « **décidables** par automate fini » (AF-décidable).

Exercice

Donner un automate qui teste si l'entrée (écrite en base 2) est nulle.

Exemple : Automates finis

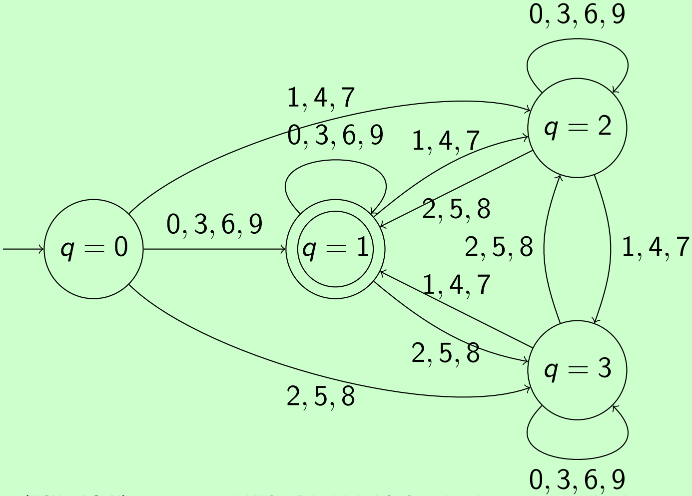
Exemple

$3\mathbb{N}$ est-il AF-décidable ?

Exemple : Automates finis

Exemple

3ℕ est-il AF-décidable ?



Exemple : Automates finis

- Tout automate se traduit aisément en **algorithme**.
  Donc AF-décidabilité implique décidabilité ;

Exemple : Automates finis

- Tout automate se traduit aisément en **algorithme**.
  Donc AF-décidabilité implique décidabilité ;
- Et même en langage de programmation :

Exemple : Automates finis

- Tout automate se traduit aisément en **algorithme**.
  Donc AF-décidabilité implique décidabilité ;
- Et même en langage de programmation :

Automate fini en C

```
q=0;
for (i=0; i < strlen(w); ++i)
    q=f(q,w[i]);
return F(q);
```

Exemple : Automates finis

- Tout automate se traduit aisément en **algorithme**.
  Donc AF-décidabilité implique décidabilité ;
- Et même en langage de programmation :

Automate fini en C

```
q=0;
for (i=0; i < strlen(w); ++i)
    q=f(q,w[i]);
return F(q);
```

- Et dans l'autre sens ?

Exemple : Automates finis

- Tout automate se traduit aisément en **algorithme**.
  Donc AF-décidabilité implique décidabilité ;
- Et même en langage de programmation :

Automate fini en C

```
q=0;
for (i=0; i < strlen(w); ++i)
    q=f(q,w[i]);
return F(q);
```

- Et dans l'autre sens ?
  $\{0^n 1^n | n \in \mathbb{N}\}$ n'est pas AF-décidable

Exemple : Automates finis

- Tout automate se traduit aisément en **algorithme**.
  Donc AF-décidabilité implique décidabilité ;
- Et même en langage de programmation :

Automate fini en C

```
q=0;
for (i=0; i < strlen(w); ++i)
    q=f(q,w[i]);
return F(q);
```

- Et dans l'autre sens ?
  $\{0^n1^n | n \in \mathbb{N}\}$ n'est pas AF-décidable
- Il nous faut un modèle plus **expressif**.

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :
  - On ne peut pas mettre un tel programme dans une mémoire **finie** ;

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :
  - On ne peut pas mettre un tel programme dans une mémoire **finie** ;
  - **Tout** ensemble est décidable par automate **infini**.

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :
  - On ne peut pas mettre un tel programme dans une mémoire **finie** ;
  - **Tout** ensemble est décidable par automate **infini**.
    Décider X : $Q = A^*$, $q_0 = \epsilon$, $f(q, a) = qa$, $F(q) = 1$ ssi $q \in X$.

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :
  - On ne peut pas mettre un tel programme dans une mémoire **finie** ;
  - **Tout** ensemble est décidable par automate **infini**.
    Décider X : $Q = A^*$, $q_0 = \epsilon$, $f(q, a) = qa$, $F(q) = 1$ ssi $q \in X$.
- Mais, il faut un programme **plus long** que le nombre de lettres de l'entrée !

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :
  - On ne peut pas mettre un tel programme dans une mémoire **finie** ;
  - **Tout** ensemble est décidable par automate **infini**.
    Décider X : $Q = A^*$, $q_0 = \epsilon$, $f(q, a) = qa$, $F(q) = 1$ ssi $q \in X$.
- Mais, il faut un programme **plus long** que le nombre de lettres de l'entrée !
- On **désynchronise** la lecture de l'entrée et les instructions ;

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :
  - On ne peut pas mettre un tel programme dans une mémoire **finie** ;
  - **Tout** ensemble est décidable par automate **infini**.
    Décider X : $Q = A^*$, $q_0 = \epsilon$, $f(q, a) = qa$, $F(q) = 1$ ssi $q \in X$.
- Mais, il faut un programme **plus long** que le nombre de lettres de l'entrée !
- On **désynchronise** la lecture de l'entrée et les instructions ;
- On rend la mémoire **infinie** et on permet sa **modification**
  mais seule une partie finie sera utilisée si le programme se termine.

				1	0	0	1		

$\uparrow$

Machines de Turing (1935)

- Pour étendre notre classe d'algorithmes, supprimer le caractère fini du programme (nombre d'états) n'est pas un bonne idée :
  - On ne peut pas mettre un tel programme dans une mémoire **finie** ;
  - **Tout** ensemble est décidable par automate **infini**.
    Décider X : $Q = A^*$, $q_0 = \epsilon$, $f(q, a) = qa$, $F(q) = 1$ ssi $q \in X$.
- Mais, il faut un programme **plus long** que le nombre de lettres de l'entrée !
- On **désynchronise** la lecture de l'entrée et les instructions ;
- On rend la mémoire **infinie** et on permet sa **modification**
  mais seule une partie finie sera utilisée si le programme se termine.

				1	0	0	1		

- $\widetilde{A} = A \cup \{\square\}$

Machine de Turing

Définition

Une **machine de Turing** sur un alphabet A est une paire $(Q, \to)$ t.q. :

- Q est un ensemble **fini** d'états. On suppose que Q contient au moins les trois états suivant :

Machine de Turing

Définition

Une **machine de Turing** sur un alphabet A est une paire $(Q, \rightarrow)$ t.q. :

- Q est un ensemble **fini** d'états. On suppose que Q contient au moins les trois états suivant :
  - init est l'**état initial** ;

Machine de Turing

Définition

Une **machine de Turing** sur un alphabet A est une paire $(Q, \rightarrow)$ t.q. :

- Q est un ensemble **fini** d'états. On suppose que Q contient au moins les trois états suivant :
  - init est l'**état initial** ;
  - accept est l'**état acceptant** ;

Machine de Turing

Définition

Une **machine de Turing** sur un alphabet A est une paire $(Q, \rightarrow)$ t.q. :

- Q est un ensemble **fini** d'états. On suppose que Q contient au moins les trois états suivant :
  - init est l'**état initial** ;
  - accept est l'**état acceptant** ;
  - reject est l'**état refusant** ;

Machine de Turing

Définition

Une **machine de Turing** sur un alphabet A est une paire $(Q, \rightarrow)$ t.q. :

- Q est un ensemble **fini** d'états. On suppose que Q contient au moins les trois états suivant :
  - init est l'**état initial** ;
  - accept est l'**état acceptant** ;
  - reject est l'**état refusant** ;
- $\rightarrow \in Q \times \widetilde{A} \times \widetilde{A} \times \{-1, 0, +1\} \times Q$ est la **relation de transition**.

Machine de Turing

Définition

- Une **configuration** d'une machine de Turing $(Q, \to)$ est un triplet (q, f, i) où :

Machine de Turing

Définition

- Une **configuration** d'une machine de Turing $(Q, \rightarrow)$ est un triplet (q, f, i) où :
  - $q \in Q$ est l'**état** de la machine ;

Machine de Turing

Définition

- Une **configuration** d'une machine de Turing $(Q, \rightarrow)$ est un triplet (q, f, i) où :
  - $q \in Q$ est l'**état** de la machine ;
  - $f : \mathbb{Z} \rightarrow \widetilde{A}$ est le contenu de la **bande** ;

Machine de Turing

Définition

- Une **configuration** d'une machine de Turing $(Q, \to)$ est un triplet (q, f, i) où :
  - $q \in Q$ est l'**état** de la machine ;
  - $f : \mathbb{Z} \to \widetilde{A}$ est le contenu de la **bande** ;
  - $i \in \mathbb{Z}$ est la position de la **tête de lecture**.

Machine de Turing

Définition

- Une **configuration** d'une machine de Turing $(Q, \rightarrow)$ est un triplet (q, f, i) où :
  - $q \in Q$ est l'**état** de la machine ;
  - $f : \mathbb{Z} \rightarrow \widetilde{A}$ est le contenu de la **bande** ;
  - $i \in \mathbb{Z}$ est la position de la **tête de lecture**.
- La configuration **initiale** de la machine sur l'entrée w est $(\text{init}, f_0, 0)$ avec :

$$f_0(i) = \begin{cases} w(i), \text{ si } i \in [0..|w| - 1], \\ \square \text{ sinon.} \end{cases}$$

Machine de Turing

- Une machine de Turing $(Q, \rightarrow)$ **passe** de la configuration (q, f, i) à la configuration (q', f', i') ssi il existe $(q, a, b, x, q') \in \rightarrow$ t.q. :

Machine de Turing

- Une machine de Turing $(Q, \rightarrow)$ **passe** de la configuration (q, f, i) à la configuration (q', f', i') ssi il existe $(q, a, b, x, q') \in\rightarrow$ t.q. :
  - $a = f(i)$; Si a est inscrit sous la tête de lecture...

Machine de Turing

- Une machine de Turing $(Q, \rightarrow)$ **passe** de la configuration (q, f, i) à la configuration (q', f', i') ssi il existe $(q, a, b, x, q') \in \rightarrow$ t.q. :
  - $a = f(i)$; Si a est inscrit sous la tête de lecture...
  - $b = f'(i)$; Écrire b à la place ;

Machine de Turing

- Une machine de Turing $(Q, \rightarrow)$ **passe** de la configuration (q, f, i) à la configuration (q', f', i') ssi il existe $(q, a, b, x, q') \in \rightarrow$ t.q. :
  - $a = f(i)$; Si a est inscrit sous la tête de lecture...
  - $b = f'(i)$; Écrire b à la place ;
  - $\forall j \neq i, f(j) = f'(j)$;

Machine de Turing

- Une machine de Turing $(Q, \rightarrow)$ **passe** de la configuration (q, f, i) à la configuration (q', f', i') ssi il existe $(q, a, b, x, q') \in \rightarrow$ t.q. :
  - $a = f(i)$; Si a est inscrit sous la tête de lecture...
  - $b = f'(i)$; Écrire b à la place ;
  - $\forall j \neq i, f(j) = f'(j)$;
  - $i' = i + x$; Déplacer éventuellement la tête de lecture.

Machine de Turing

- Une machine de Turing $(Q, \rightarrow)$ **passe** de la configuration (q, f, i) à la configuration (q', f', i') ssi il existe $(q, a, b, x, q') \in \rightarrow$ t.q. :
  - $a = f(i)$; Si a est inscrit sous la tête de lecture...
  - $b = f'(i)$; Écrire b à la place ;
  - $\forall j \neq i, f(j) = f'(j)$;
  - $i' = i + x$; Déplacer éventuellement la tête de lecture.
- Si la machine atteint l'état accept, elle **s'arrête** et renvoie ≪ oui ≫ ;

Machine de Turing

- Une machine de Turing $(Q, \rightarrow)$ **passe** de la configuration (q, f, i) à la configuration (q', f', i') ssi il existe $(q, a, b, x, q') \in \rightarrow$ t.q. :
  - $a = f(i)$; Si a est inscrit sous la tête de lecture...
  - $b = f'(i)$; Écrire b à la place ;
  - $\forall j \neq i, f(j) = f'(j)$;
  - $i' = i + x$; Déplacer éventuellement la tête de lecture.
- Si la machine atteint l'état accept, elle **s'arrête** et renvoie « oui » ;
- Si la machine atteint l'état reject, elle **s'arrête** et renvoie « non ».

Machines de Turing : Exemple

Exemple

Construire une machine qui teste si l'entrée est **nulle** (ou vide).

Machines de Turing : Exemple

Exemple

Construire une machine qui teste si l'entrée est **nulle** (ou vide).

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;

Machines de Turing : Exemple

Exemple

Construire une machine qui teste si l'entrée est **nulle** (ou vide).

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;
- $\rightarrow = \left\{ \begin{array}{l} (\text{init}, 0, 0, +1, \text{init}), \\ (\text{init}, 1, 1, +1, \text{reject}), \\ (\text{init}, \square, \square, 0, \text{accept}) \end{array} \right\}$

Machines de Turing : Exemple

Exemple

Construire une machine qui teste si l'entrée est **nulle** (ou vide).

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;
- $\rightarrow = \left\{ \begin{array}{l} (\text{init}, 0, 0, +1, \text{init}), \\ (\text{init}, 1, 1, +1, \text{reject}), \\ (\text{init}, \square, \square, 0, \text{accept}) \end{array} \right\}$

On dit que $\{0\}$ est « **décidable** par machine de Turing » (MT-décidable).

Machines de Turing : Exemple

Exemple

Construire une machine qui teste si l'entrée est **nulle** (ou vide).

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;
- $\rightarrow = \left\{ \begin{array}{l} (\text{init}, 0, 0, +1, \text{init}), \\ (\text{init}, 1, 1, +1, \text{reject}), \\ (\text{init}, \square, \square, 0, \text{accept}) \end{array} \right\}$

On dit que $\{0\}$ est « **décidable** par machine de Turing » (MT-décidable).

Exercice

Construire une machine de Turing qui teste si l'entrée est **impaire**.

Machines de Turing : Exemple

Exemple

Construire une machine qui teste si l'entrée est **nulle** (ou vide).

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;
- $\rightarrow = \left\{ \begin{array}{l} (\text{init}, 0, 0, +1, \text{init}), \\ (\text{init}, 1, 1, +1, \text{reject}), \\ (\text{init}, \Box, \Box, 0, \text{accept}) \end{array} \right\}$

On dit que $\{0\}$ est « **décidable** par machine de Turing » (MT-décidable).

Exercice

Construire une machine de Turing qui teste si l'entrée est **impaire**.

Exercice

Construire une machine de Turing qui teste si deux chaînes (l'une de 0 et l'autre de 1 p.ex.) sont de même longueur.

Machines de Turing : Algorithme associé

On peut facilement construire un **algorithme** à partir d'une machine de Turing :

- Les configurations de l'algorithme sont celles de la machine ;

Machines de Turing : Algorithme associé

On peut facilement construire un **algorithme** à partir d'une machine de Turing :

- Les configurations de l'algorithme sont celles de la machine ;
- La fonction d'entrée est : $\mathcal{E}(w) = (\text{init}, f_0, 0)$;

Machines de Turing : Algorithme associé

On peut facilement construire un **algorithme** à partir d'une machine de Turing :

- Les configurations de l'algorithme sont celles de la machine ;
- La fonction d'entrée est : $\mathcal{E}(w) = (\mathrm{init}, f_0, 0)$;
- La fonction de transition est celle de la machine ;

Machines de Turing : Algorithme associé

On peut facilement construire un **algorithme** à partir d'une machine de Turing :

- Les configurations de l'algorithme sont celles de la machine ;
- La fonction d'entrée est : $\mathcal{E}(w) = (\text{init}, f_0, 0)$;
- La fonction de transition est celle de la machine ;
- La fonction de sortie est définie par : $\forall i, \mathcal{S}(\text{accept}, f, i) = 1$ et $\forall i, \mathcal{S}(\text{reject}, f, i) = 0$.

MT-décidabilité et décidabilité

- Toute machine de Turing se **traduit** trivialement en algorithme.
  Donc MT-décidabilité implique décidabilité.

MT-décidabilité et décidabilité

- Toute machine de Turing se **traduit** trivialement en algorithme.
  Donc MT-décidabilité implique décidabilité.
- Comme pour les automates on peut **programmer** facilement en C ou autre langage de programmation, une machine de Turing (cf. TP) ;

MT-décidabilité et décidabilité

- Toute machine de Turing se **traduit** trivialement en algorithme.
  Donc MT-décidabilité implique décidabilité.
- Comme pour les automates on peut **programmer** facilement en C ou autre langage de programmation, une machine de Turing (cf. TP) ;
- On se convainc facilement qu'un automate est un **cas particulier** de machine de Turing.
  Donc AF-décidabilité implique MT-décidabilité ;

MT-décidabilité et décidabilité

- Toute machine de Turing se **traduit** trivialement en algorithme.
  Donc MT-décidabilité implique décidabilité.
- Comme pour les automates on peut **programmer** facilement en C ou autre langage de programmation, une machine de Turing (cf. TP) ;
- On se convainc facilement qu'un automate est un **cas particulier** de machine de Turing.
  Donc AF-décidabilité implique MT-décidabilité ;
- Existe-t-il des problèmes décidables qui ne sont **pas** MT-décidables ?

Thèse de Church-Turing

Thèse de Church-Turing

Tout ensemble **décidable** est **MT-décidable**.

Thèse de Church-Turing

Thèse de Church-Turing

Tout ensemble **décidable** est **MT-décidable**.

- Ne peut pas être **prouvée** (pas de notion complètement formelle d'algorithme) ;

Thèse de Church-Turing

Thèse de Church-Turing

Tout ensemble **décidable** est **MT-décidable**.

- Ne peut pas être **prouvée** (pas de notion complètement formelle d'algorithme) ;
- Très largement acceptée comme **vraie** ;

Thèse de Church-Turing

Thèse de Church-Turing

Tout ensemble **décidable** est **MT-décidable**.

- Ne peut pas être **prouvée** (pas de notion complètement formelle d'algorithme) ;
- Très largement acceptée comme **vraie** ;
- Le calcul par machine de Turing **coïncide** avec d'autres notions proches : λ-**calcul** et **fonctions récursives**.

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;
  - Un ensemble fini de **instructions** (quintuplets (q, a, b, i, q')) ;

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;
  - Un ensemble fini de **instructions** (quintuplets (q, a, b, i, q')) ;
  - Un **alphabet** fini.

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;
  - Un ensemble fini de **instructions** (quintuplets (q, a, b, i, q')) ;
  - Un **alphabet** fini.
- On peut **encoder** toute machine M avec un alphabet fini, p.ex :

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;
  - Un ensemble fini de **instructions** (quintuplets (q, a, b, i, q')) ;
  - Un **alphabet** fini.
- On peut **encoder** toute machine M avec un alphabet fini, p.ex :
  - Les **états** :

$$e(\text{init}) = 1, e(\text{accept}) = 11, e(\text{reject}) = 111, \ldots$$

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;
  - Un ensemble fini de **instructions** (quintuplets (q, a, b, i, q')) ;
  - Un **alphabet** fini.
- On peut **encoder** toute machine M avec un alphabet fini, p.ex :
  - Les **états** :

$$e(\text{init}) = 1, e(\text{accept}) = 11, e(\text{reject}) = 111, \ldots$$

  - Les **symboles** :

$$e(0) = 1, e(-1) = 11, e(1) = 111, e(\square) = 1111, \ldots$$

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;
  - Un ensemble fini de **instructions** (quintuplets (q, a, b, i, q')) ;
  - Un **alphabet** fini.
- On peut **encoder** toute machine M avec un alphabet fini, p.ex :
  - Les **états** :

$$e(\text{init}) = 1, e(\text{accept}) = 11, e(\text{reject}) = 111, \ldots$$

  - Les **symboles** :

$$e(0) = 1, e(-1) = 11, e(1) = 111, e(\square) = 1111, \ldots$$

  - Les **instructions** :

$$e((q, a, b, i, q')) = e(q)0e(a)0e(b)0e(i)0e(q')0$$

Encodage de machines de Turing

- Une machine de Turing est **complètement** décrite par un ensemble **fini** de données :
  - Un ensemble fini d'**états** ;
  - Un ensemble fini de **instructions** (quintuplets (q, a, b, i, q')) ;
  - Un **alphabet** fini.
- On peut **encoder** toute machine M avec un alphabet fini, p.ex :
  - Les **états** :

$$e(\text{init}) = 1, e(\text{accept}) = 11, e(\text{reject}) = 111, \ldots$$

  - Les **symboles** :

$$e(0) = 1, e(-1) = 11, e(1) = 111, e(\Box) = 1111, \ldots$$

  - Les **instructions** :

$$e((q, a, b, i, q')) = e(q)0e(a)0e(b)0e(i)0e(q')0$$

  - M d'instructions $I_1, \ldots, I_n$,

$$e(M) = e(\rightarrow) = 00e(I_1)e(I_2)\ldots e(I_n)0$$

Encodage de machines de Turing

Exemple

La machine M, qui décide $\{0\}$, est définie par :

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;

Encodage de machines de Turing

Exemple

La machine M, qui décide $\{0\}$, est définie par :

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;
- $\rightarrow = \left\{ \begin{array}{l} (\text{init}, 0, 0, +1, \text{init}), \\ (\text{init}, 1, 1, +1, \text{reject}), \\ (\text{init}, \square, \square, 0, \text{accept}) \end{array} \right\}$

Encodage de machines de Turing

Exemple

La machine M, qui décide $\{0\}$, est définie par :

- $Q = \{\text{init}, \text{accept}, \text{reject}\}$;
- $\rightarrow = \left\{ \begin{array}{l} (\text{init}, 0, 0, +1, \text{init}), \\ (\text{init}, 1, 1, +1, \text{reject}), \\ (\text{init}, \square, \square, 0, \text{accept}) \end{array} \right\}$

M peut être **encodée** par :

$$e(M) = 0010101011101010111011101110111010111011110101100$$

On note souvent $\langle M \rangle$ l'encodage de M pour un encodage non spécifié.

Théorème de Rice

- On peut donc construire des machines qui vérifient des **propriétés** sur les **machines**

Théorème de Rice

- On peut donc construire des machines qui vérifient des **propriétés** sur les **machines**
- Cependant soit A un alphabet et S un ensemble de parties de A^* (différent de 2^{A^*} et de $\emptyset$). Soit le problème $\mathcal{P}$ suivant :

Théorème de Rice

- On peut donc construire des machines qui vérifient des **propriétés** sur les **machines**
- Cependant soit A un alphabet et S un ensemble de parties de A^* (différent de 2^{A^*} et de $\emptyset$). Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Vérification de propriétés

Entrées:	une machine de Turing M
Résultat:	L'ensemble des mots acceptés par M est-il dans S ?

Théorème de Rice

- On peut donc construire des machines qui vérifient des **propriétés** sur les **machines**
- Cependant soit A un alphabet et S un ensemble de parties de A^* (différent de 2^{A^*} et de $\emptyset$). Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Vérification de propriétés

Entrées: une machine de Turing M
Résultat: L'ensemble des mots acceptés par M est-il dans S ?

Théorème (Théorème de Rice)

$\mathcal{P}$ est ***indécidable****.*

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w
Résultat: M s'arrête-t-elle sur l'entrée w ?

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w
Résultat: M s'arrête-t-elle sur l'entrée w ?

Théorème

Le problème $\mathcal{P}$ est **indecidable**.

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w

Résultat: M s'arrête-t-elle sur l'entrée w ?

Théorème

Le problème $\mathcal{P}$ est **indecidable**.

- On peut le prouver par le théorème de Rice (et inversement) ;

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w

Résultat: M s'arrête-t-elle sur l'entrée w ?

Théorème

Le problème $\mathcal{P}$ est **indecidable**.

- On peut le prouver par le théorème de Rice (et inversement) ;
- Autre preuve : on suppose qu'il existe H telle que H accepte $(\langle M\rangle, w)$ si M s'arrête sur w, et refuse sinon . Soit la machine H' telle que :

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w
Résultat: M s'arrête-t-elle sur l'entrée w ?

Théorème

Le problème $\mathcal{P}$ est **indecidable**.

- On peut le prouver par le théorème de Rice (et inversement) ;
- Autre preuve : on suppose qu'il existe H telle que H accepte $(\langle M \rangle, w)$ si M s'arrête sur w, et refuse sinon . Soit la machine H' telle que :
  - H' accepte s si H refuse (s, s) ;

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w

Résultat: M s'arrête-t-elle sur l'entrée w ?

Théorème

Le problème $\mathcal{P}$ est **indecidable**.

- On peut le prouver par le théorème de Rice (et inversement) ;
- Autre preuve : on suppose qu'il existe H telle que H accepte $(\langle M \rangle, w)$ si M s'arrête sur w, et refuse sinon . Soit la machine H' telle que :
  - H' accepte s si H refuse (s, s) ;
  - H' rentre dans une boucle infinie si H accepte (s, s).

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w
Résultat: M s'arrête-t-elle sur l'entrée w ?

Théorème

Le problème $\mathcal{P}$ est **indecidable**.

- On peut le prouver par le théorème de Rice (et inversement) ;
- Autre preuve : on suppose qu'il existe H telle que H accepte $(\langle M \rangle, w)$ si M s'arrête sur w, et refuse sinon . Soit la machine H' telle que :
  - H' accepte s si H refuse (s, s) ;
  - H' rentre dans une boucle infinie si H accepte (s, s).

  Que dire de l'exécution de H' sur l'entrée $\langle H' \rangle$?

Problème de l'arrêt d'une machine de Turing

Soit le problème $\mathcal{P}$ suivant :

$\mathcal{P}$: Arrêt d'une machine de Turing

Entrées: Une machine M, un mot w

Résultat: M s'arrête-t-elle sur l'entrée w ?

Théorème

Le problème $\mathcal{P}$ est **indecidable**.

- On peut le prouver par le théorème de Rice (et inversement) ;
- Autre preuve : on suppose qu'il existe H telle que H accepte $(\langle M \rangle, w)$ si M s'arrête sur w, et refuse sinon . Soit la machine H' telle que :
  - H' accepte s si H refuse (s, s) ;
  - H' rentre dans une boucle infinie si H accepte (s, s).

  Que dire de l'exécution de H' sur l'entrée $\langle H' \rangle$?
- Il est donc **impossible** de faire un programme qui dit pour tout programme si celui-ci va s'**arrêter**.

Machines universelles

- En **encodant** une machine de Turing comme précédemment, on peut essayer de construire une **machine programmable**

Machines universelles

- En **encodant** une machine de Turing comme précédemment, on peut essayer de construire une **machine programmable**

Définition (Machine universelle)

Une machine de Turing U est **universelle** sur l'alphabet A si pour un encodage e sur A, toute machine de Turing M sur A et toute entrée $w \in A^*$:

U accepte (resp. refuse) $e(M)w$ ssi M accepte (resp. refuse) w.

Machines universelles

- En **encodant** une machine de Turing comme précédemment, on peut essayer de construire une **machine programmable**

Définition (Machine universelle)

Une machine de Turing U est **universelle** sur l'alphabet A si pour un encodage e sur A, toute machine de Turing M sur A et toute entrée $w \in A^*$:

U accepte (resp. refuse) $e(M)w$ ssi M accepte (resp. refuse) w.

Théorème

Pour tout alphabet A, il **existe** *une machine de Turing* **universelle** *sur A.*

Machines universelles

- En **encodant** une machine de Turing comme précédemment, on peut essayer de construire une **machine programmable**

Définition (Machine universelle)

Une machine de Turing U est **universelle** sur l'alphabet A si pour un encodage e sur A, toute machine de Turing M sur A et toute entrée $w \in A^*$:

U accepte (resp. refuse) $e(M)w$ ssi M accepte (resp. refuse) w.

Théorème

Pour tout alphabet A, il **existe** *une machine de Turing* **universelle** *sur A.*

On utilise trois bandes : une pour $e(M)w$, une pour l'état courant de M et une pour la bande de n

Machines universelles

- En **encodant** une machine de Turing comme précédemment, on peut essayer de construire une **machine programmable**

Définition (Machine universelle)

Une machine de Turing U est **universelle** sur l'alphabet A si pour un encodage e sur A, toute machine de Turing M sur A et toute entrée $w \in A^*$:

$$U \text{ accepte (resp. refuse) } e(M)w \text{ ssi } M \text{ accepte (resp. refuse) } w.$$

Théorème

Pour tout alphabet A, il **existe** *une machine de Turing* **universelle** *sur A.*

On utilise trois bandes : une pour $e(M)w$, une pour l'état courant de M et une pour la bande de n

- Ce résultat a fortement influencé l'**architecture** proposée par **John Von Neumann** et la notion de **programme mémorisé.**

Plan

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :
  - La **complexité temporelle** est le nombre d'étapes faites par la machine de Turing pour **terminer** ;

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :
  - La **complexité temporelle** est le nombre d'étapes faites par la machine de Turing pour **terminer** ;
  - La **complexité spatiale** est le nombre **maximum** de cases de la bande utilisées (différentes de $\square$) **simultanément** par la machine (jusqu'à la terminaison).

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :
  - La **complexité temporelle** est le nombre d'étapes faites par la machine de Turing pour **terminer** ;
  - La **complexité spatiale** est le nombre **maximum** de cases de la bande utilisées (différentes de $\square$) **simultanément** par la machine (jusqu'à la terminaison).
- On peut regarder (en fonction de l'entrée) :

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :
  - La **complexité temporelle** est le nombre d'étapes faites par la machine de Turing pour **terminer** ;
  - La **complexité spatiale** est le nombre **maximum** de cases de la bande utilisées (différentes de $\square$) **simultanément** par la machine (jusqu'à la terminaison).
- On peut regarder (en fonction de l'entrée) :
  - la complexité au **meilleur** cas ;

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :
  - La **complexité temporelle** est le nombre d'étapes faites par la machine de Turing pour **terminer** ;
  - La **complexité spatiale** est le nombre **maximum** de cases de la bande utilisées (différentes de □) **simultanément** par la machine (jusqu'à la terminaison).
- On peut regarder (en fonction de l'entrée) :
  - la complexité au **meilleur** cas ;
  - la complexité en **moyenne** ;

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :
  - La **complexité temporelle** est le nombre d'étapes faites par la machine de Turing pour **terminer** ;
  - La **complexité spatiale** est le nombre **maximum** de cases de la bande utilisées (différentes de $\square$) **simultanément** par la machine (jusqu'à la terminaison).
- On peut regarder (en fonction de l'entrée) :
  - la complexité au **meilleur** cas ;
  - la complexité en **moyenne** ;
  - la complexité au **pire** cas.

Complexité

- On a modèle **rigoureux** pour la notion d'**algorithme** ;
- On se place dans le cadre de problèmes **décidables** ;
- On peut définir naturellement deux critères de **comparaison** :
  - La **complexité temporelle** est le nombre d'étapes faites par la machine de Turing pour **terminer** ;
  - La **complexité spatiale** est le nombre **maximum** de cases de la bande utilisées (différentes de $\square$) **simultanément** par la machine (jusqu'à la terminaison).
- On peut regarder (en fonction de l'entrée) :
  - la complexité au **meilleur** cas ;
  - la complexité en **moyenne** ;
  - la complexité au **pire** cas.
- On étudie ici la complexité au **pire** cas.

Classes de complexité

- Soit une fonction $T : \mathbb{N} \to \mathbb{N}$ croissante.

Classes de complexité

- Soit une fonction $T : \mathbb{N} \to \mathbb{N}$ croissante.
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** temporelle $\mathbf{DTIME}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en moins de $T(n)$ étapes ;

Classes de complexité

- Soit une fonction $T : \mathbb{N} \to \mathbb{N}$ croissante.
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** temporelle $\mathbf{DTIME}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en moins de $T(n)$ étapes ;
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** spatiale $\mathbf{DSPACE}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en utilisant moins de $T(n)$ cases de bande.

Classes de complexité

- Soit une fonction $T : \mathbb{N} \to \mathbb{N}$ croissante.
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** temporelle $\mathbf{DTIME}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en moins de $T(n)$ étapes ;
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** spatiale $\mathbf{DSPACE}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en utilisant moins de $T(n)$ cases de bande.
- D pour déterministe, et 1 pour une seule bande.

Classes de complexité

- Soit une fonction $T : \mathbb{N} \to \mathbb{N}$ croissante.
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** temporelle $\mathbf{DTIME}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en moins de $T(n)$ étapes ;
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** spatiale $\mathbf{DSPACE}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en utilisant moins de $T(n)$ cases de bande.
- D pour déterministe, et 1 pour une seule bande.
- $DTIME \subseteq DSPACE$

Classes de complexité

- Soit une fonction $T : \mathbb{N} \to \mathbb{N}$ croissante.
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** temporelle $\mathbf{DTIME}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en moins de $T(n)$ étapes ;
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** spatiale $\mathbf{DSPACE}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en utilisant moins de $T(n)$ cases de bande.
- D pour déterministe, et 1 pour une seule bande.
- $DTIME \subseteq DSPACE$

Exemple

$$\{0\} \in \mathbf{DTIME}_1^{\{0,1\}}(n), \{0\} \in \mathbf{DSPACE}_1^{\{0,1\}}(n)$$

$$2\mathbb{N} \in \mathbf{DTIME}_1^{\{0,1\}}(n), 2\mathbb{N} \in \mathbf{DSPACE}_1^{\{0,1\}}(n)$$

$$\{0^n1^n | n \in \mathbb{N}\} \in \mathbf{DTIME}_1^{\{0,1\}}(n^2), \{0^n1^n | n \in \mathbb{N}\} \in \mathbf{DSPACE}_1^{\{0,1\}}(n)$$

Classes de complexité

- Soit une fonction $T : \mathbb{N} \to \mathbb{N}$ croissante.
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** temporelle $\mathbf{DTIME}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en moins de $T(n)$ étapes ;
  - Le problème $\mathcal{P}$ appartient à la **classe de complexité** spatiale $\mathbf{DSPACE}_1^A(T(n))$ s'il existe une machine M qui décide toute entrée de $\mathcal{P}$ de longueur n en utilisant moins de $T(n)$ cases de bande.
- D pour déterministe, et 1 pour une seule bande.
- $DTIME \subseteq DSPACE$

Exemple

$$\{0\} \in \mathbf{DTIME}_1^{\{0,1\}}(n), \{0\} \in \mathbf{DSPACE}_1^{\{0,1\}}(n)$$

$$2\mathbb{N} \in \mathbf{DTIME}_1^{\{0,1\}}(n), 2\mathbb{N} \in \mathbf{DSPACE}_1^{\{0,1\}}(n)$$

$$\{0^n1^n | n \in \mathbb{N}\} \in \mathbf{DTIME}_1^{\{0,1\}}(n^2), \{0^n1^n | n \in \mathbb{N}\} \in \mathbf{DSPACE}_1^{\{0,1\}}(n)$$

En fait, $\exists c$ constante t.q. $\{0^n1^n | n \in \mathbb{N}\} \in \mathbf{DTIME}_1^{\{0,1\}}(cnlog_2(n))$

Simulation de machines de Turing

- Soient $M_1 = (Q_1, \rightarrow_1)$ et $M_2 = (Q_2, \rightarrow_2)$ deux machines de Turing. Soit $\mathcal{R}$ une relation sur les configurations de M_1 et M_2. $\mathcal{R}$ est une **simulation** de M_2 par M_1 si :

Simulation de machines de Turing

- Soient $M_1 = (Q_1, \to_1)$ et $M_2 = (Q_2, \to_2)$ deux machines de Turing. Soit $\mathcal{R}$ une relation sur les configurations de M_1 et M_2. $\mathcal{R}$ est une **simulation** de M_2 par M_1 si :
  - $\text{init}_1 \mathcal{R} \text{init}_2$;

Simulation de machines de Turing

- Soient $M_1 = (Q_1, \rightarrow_1)$ et $M_2 = (Q_2, \rightarrow_2)$ deux machines de Turing. Soit $\mathcal{R}$ une relation sur les configurations de M_1 et M_2. $\mathcal{R}$ est une **simulation** de M_2 par M_1 si :
  - $\mathrm{init}_1 \mathcal{R} \mathrm{init}_2$;
  - si $q_2 \rightarrow_2 q_2'$ et $q_1 \mathcal{R} q_2$ alors il existe $q_1^1, \ldots, q_1^n$ t.q. $q_1 \rightarrow_1 q_1^1 \rightarrow_1 \cdots \rightarrow_1 q_1^n$ et $q_2' \mathcal{R} q_1^n$;

Simulation de machines de Turing

- Soient $M_1 = (Q_1, \rightarrow_1)$ et $M_2 = (Q_2, \rightarrow_2)$ deux machines de Turing. Soit $\mathcal{R}$ une relation sur les configurations de M_1 et M_2. $\mathcal{R}$ est une **simulation** de M_2 par M_1 si :
  - $\text{init}_1 \mathcal{R} \text{init}_2$;
  - si $q_2 \rightarrow_2 q_2'$ et $q_1 \mathcal{R} q_2$ alors il existe $q_1^1, \dots, q_1^n$ t.q. $q_1 \rightarrow_1 q_1^1 \rightarrow_1 \cdots \rightarrow_1 q_1^n$ et $q_2' \mathcal{R} q_1^n$;
  - si $q_1 \mathcal{R} \text{accept}_2$ alors $q_1 = \text{accept}_1$.

Simulation de machines de Turing

- Soient $M_1 = (Q_1, \rightarrow_1)$ et $M_2 = (Q_2, \rightarrow_2)$ deux machines de Turing. Soit $\mathcal{R}$ une relation sur les configurations de M_1 et M_2. $\mathcal{R}$ est une **simulation** de M_2 par M_1 si :
  - $\mathrm{init}_1 \mathcal{R} \mathrm{init}_2$;
  - si $q_2 \rightarrow_2 q_2'$ et $q_1 \mathcal{R} q_2$ alors il existe $q_1^1, \ldots, q_1^n$ t.q. $q_1 \rightarrow_1 q_1^1 \rightarrow_1 \cdots \rightarrow_1 q_1^n$ et $q_2' \mathcal{R} q_1^n$;
  - si $q_1 \mathcal{R} \mathrm{accept}_2$ alors $q_1 = \mathrm{accept}_1$.
  - si $q_1 \mathcal{R} \mathrm{reject}_2$ alors $q_1 = \mathrm{reject}_1$.

Simulation de machines de Turing

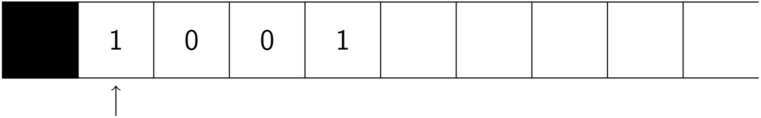
- Soient $M_1 = (Q_1, \rightarrow_1)$ et $M_2 = (Q_2, \rightarrow_2)$ deux machines de Turing. Soit $\mathcal{R}$ une relation sur les configurations de M_1 et M_2. $\mathcal{R}$ est une **simulation** de M_2 par M_1 si :
  - $\text{init}_1 \mathcal{R} \text{init}_2$;
  - si $q_2 \rightarrow_2 q_2'$ et $q_1 \mathcal{R} q_2$ alors il existe $q_1^1, \ldots, q_1^n$ t.q. $q_1 \rightarrow_1 q_1^1 \rightarrow_1 \cdots \rightarrow_1 q_1^n$ et $q_2' \mathcal{R} q_1^n$;
  - si $q_1 \mathcal{R} \text{accept}_2$ alors $q_1 = \text{accept}_1$.
  - si $q_1 \mathcal{R} \text{reject}_2$ alors $q_1 = \text{reject}_1$.
- S'il existe une simulation de M_2 par M_1, on dit que M_1 **simule** M_2.

Simulation de machines de Turing

- Soient $M_1 = (Q_1, \rightarrow_1)$ et $M_2 = (Q_2, \rightarrow_2)$ deux machines de Turing. Soit $\mathcal{R}$ une relation sur les configurations de M_1 et M_2. $\mathcal{R}$ est une **simulation** de M_2 par M_1 si :
  - $\text{init}_1 \mathcal{R} \text{init}_2$;
  - si $q_2 \rightarrow_2 q_2'$ et $q_1 \mathcal{R} q_2$ alors il existe $q_1^1, \ldots, q_1^n$ t.q. $q_1 \rightarrow_1 q_1^1 \rightarrow_1 \cdots \rightarrow_1 q_1^n$ et $q_2' \mathcal{R} q_1^n$;
  - si $q_1 \mathcal{R} \text{accept}_2$ alors $q_1 = \text{accept}_1$.
  - si $q_1 \mathcal{R} \text{reject}_2$ alors $q_1 = \text{reject}_1$.
- S'il existe une simulation de M_2 par M_1, on dit que M_1 **simule** M_2.
- Si M_1 simule M_2 et M_2 **décide** X alors M_1 également.

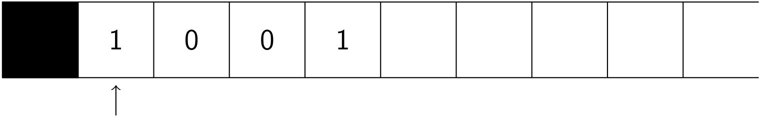
Changement du nombre de bandes

- Une **demi-bande** :



Changement du nombre de bandes

- Une **demi-bande** :

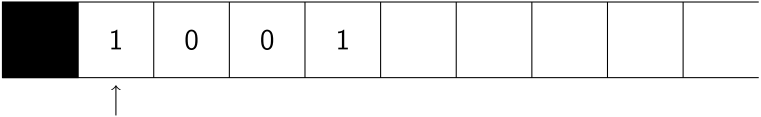


Théorème

Pour toute machine de Turing sur A à une bande, il existe une machine à une demi-bande sur A qui la simule.

Changement du nombre de bandes

- Une **demi-bande** :



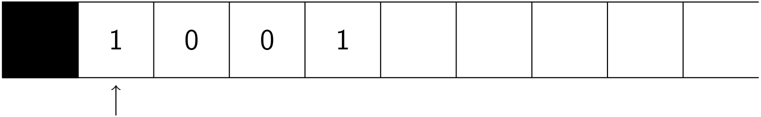
Théorème

Pour toute machine de Turing sur A à une bande, il existe une machine à une demi-bande sur A qui la simule.

Idée de la preuve : les indices pairs de la demi-bande codent les entiers négatifs et les impairs les positifs

Changement du nombre de bandes

- Une **demi-bande** :



Théorème

Pour toute machine de Turing sur A à une bande, il existe une machine à une demi-bande sur A qui la simule.

Idée de la preuve : les indices pairs de la demi-bande codent les entiers négatifs et les impairs les positifs

Théorème

$$\mathbf{DTIME}^A_{1/2}(T(n)) \subseteq \mathbf{DTIME}^A_1(T(n)) \subseteq \mathbf{DTIME}^A_{1/2}(3T(n) + 8n^2)$$

$$\mathbf{DSPACE}^A_{1/2}(T(n)) \subseteq \mathbf{DSPACE}^A_1(T(n)) \subseteq \mathbf{DSPACE}^A_{1/2}(2T(n) + 2n + 1)$$

Changement du nombre de bandes

- $k \geq 2$ bandes (à chaque transition les k têtes de lecture peuvent bouger **simultanément**) :

Changement du nombre de bandes

- $k \geq 2$ bandes (à chaque transition les k têtes de lecture peuvent bouger **simultanément**) :

Théorème

Pour toute machine de Turing sur A à $k \geq 2$ bandes, il existe une machine à une bande sur A qui la simule. **on utilise une case sur k pour chaque bande**

Changement du nombre de bandes

- $k \geq 2$ bandes (à chaque transition les k têtes de lecture peuvent bouger **simultanément**) :

Théorème

Pour toute machine de Turing sur A à $k \geq 2$ bandes, il existe une machine à une bande sur A qui la simule.

Théorème

$$\mathbf{DTIME}_1^A(T(n)) \subseteq \mathbf{DTIME}_k^A(T(n)) \subseteq \mathbf{DTIME}_1^A(6k^2(T(n)+n^2))$$

$$\mathbf{DSPACE}_1^A(T(n)) \subseteq \mathbf{DSPACE}_k^A(T(n)) \subseteq \mathbf{DSPACE}_1^A(2kT(n))$$

Changement du nombre de bandes

- Les complexités obtenues en faisant varier le nombre de bandes sont relativement **proches** ;

Changement du nombre de bandes

- Les complexités obtenues en faisant varier le nombre de bandes sont relativement **proches** ;
- Il est souvent commode d'utiliser **plusieurs** bandes ;

Changement du nombre de bandes

- Les complexités obtenues en faisant varier le nombre de bandes sont relativement **proches** ;
- Il est souvent commode d'utiliser **plusieurs** bandes ;
- On s'intéresse donc plutôt aux **classes de complexité** :

$$\mathbf{DTIME}^A(T(n)) = \bigcup_k \mathbf{DTIME}_k^A(T(n))$$

$$\mathbf{DSPACE}^A(T(n)) = \bigcup_k \mathbf{DSPACE}_k^A(T(n))$$

Changement du nombre de bandes

- Les complexités obtenues en faisant varier le nombre de bandes sont relativement **proches** ;
- Il est souvent commode d'utiliser **plusieurs** bandes ;
- On s'intéresse donc plutôt aux **classes de complexité** :

$$\mathbf{DTIME}^A(T(n)) = \bigcup_k \mathbf{DTIME}_k^A(T(n))$$

$$\mathbf{DSPACE}^A(T(n)) = \bigcup_k \mathbf{DSPACE}_k^A(T(n))$$

Théorème

Si n est négligeable devant $T(n)$, alors pour toute constante c,

$$\mathbf{DTIME}^A(cT(n)) = \mathbf{DTIME}^A(T(n))$$

Machines non déterministes

- Il y a des problèmes ≪ difficiles ≫ pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est ≪ facile ≫

Machines non déterministes

- Il y a des problèmes « difficiles » pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est « facile »
  Factorisation en nombres premiers, équations algébriques, ...

Machines non déterministes

- Il y a des problèmes « difficiles » pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est « facile »
  Factorisation en nombres premiers, équations algébriques, ...
- On veut **hiérarchiser** la complexité de ces problèmes ;

Machines non déterministes

- Il y a des problèmes « difficiles » pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est « facile »
  Factorisation en nombres premiers, équations algébriques, ...
- On veut **hiérarchiser** la complexité de ces problèmes ;
- Il faut étendre les machines de Turing pour qu'elles puissent « deviner » la solution ;

Machines non déterministes

- Il y a des problèmes « difficiles » pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est « facile »
  Factorisation en nombres premiers, équations algébriques, ...
- On veut **hiérarchiser** la complexité de ces problèmes ;
- Il faut étendre les machines de Turing pour qu'elles puissent « deviner » la solution ;
- On peut ajouter un **oracle** qui va modifier la bande de façon non-déterministe ;

Machines non déterministes

- Il y a des problèmes ≪ difficiles ≫ pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est ≪ facile ≫
  Factorisation en nombres premiers, équations algébriques, ...
- On veut **hiérarchiser** la complexité de ces problèmes ;
- Il faut étendre les machines de Turing pour qu'elles puissent ≪ deviner ≫ la solution ;
- On peut ajouter un **oracle** qui va modifier la bande de façon non-déterministe ;
- De façon **équivalente**, on peut autoriser **plusieurs** transitions depuis une **même** configuration (machines de Turing non-déterministes (NMT)).

Machines non déterministes

- Il y a des problèmes « difficiles » pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est « facile »
  Factorisation en nombres premiers, équations algébriques, ...
- On veut **hiérarchiser** la complexité de ces problèmes ;
- Il faut étendre les machines de Turing pour qu'elles puissent « deviner » la solution ;
- On peut ajouter un **oracle** qui va modifier la bande de façon non-déterministe ;
- De façon **équivalente**, on peut autoriser **plusieurs** transitions depuis une **même** configuration (machines de Turing non-déterministes (NMT)).
- La machine **accepte** si l'une des configurations atteinte est dans l'état accept.

Machines non déterministes

- Il y a des problèmes ≪ difficiles ≫ pour lesquels **vérifier** qu'un candidat donné est une solution (ou pas) est ≪ facile ≫
  Factorisation en nombres premiers, équations algébriques, ...
- On veut **hiérarchiser** la complexité de ces problèmes ;
- Il faut étendre les machines de Turing pour qu'elles puissent ≪ deviner ≫ la solution ;
- On peut ajouter un **oracle** qui va modifier la bande de façon non-déterministe ;
- De façon **équivalente**, on peut autoriser **plusieurs** transitions depuis une **même** configuration (machines de Turing non-déterministes (NMT)).
- La machine **accepte** si l'une des configurations atteinte est dans l'état accept.
- La machine **rejette** si toutes les configurations atteintes sont dans l'état reject.

Machines non déterministes

Exemple

SAT « very light »

Entrées: Une formule booléenne $x_1 \wedge x_2 \cdots \wedge x_n$
Résultat: Une valeur des variables qui rend la formule vraie

On encode la formule par $x \wedge x \wedge x \wedge \cdots \wedge x$ n fois.

- $Q = \{\text{init}, \text{accept}, \text{reject}, \mathit{check}\}$;

Machines non déterministes

Exemple

SAT « very light »

Entrées: Une formule booléenne $x_1 \wedge x_2 \cdots \wedge x_n$
Résultat: Une valeur des variables qui rend la formule vraie

On encode la formule par $x \wedge x \wedge x \wedge \cdots \wedge x$ n fois.

- $Q = \{\text{init}, \text{accept}, \text{reject}, \mathit{check}\}$;
- $$\rightarrow = \left\{ \begin{array}{l} (\text{init}, x, 0, +1, \text{init}), \\ (\text{init}, x, 1, +1, \text{init}), \\ (\text{init}, \wedge, \wedge, +1, \text{init}), \\ (\text{init}, \square, \square, -1, \mathit{check}), \\ (\mathit{check}, 0, 0, 0, \text{reject}), \\ (\mathit{check}, \wedge, \wedge, -1, \mathit{check}), \\ (\mathit{check}, 1, 1, -1, \mathit{check}), \\ (\mathit{check}, \square, \square, 0, \text{accept}) \end{array} \right\}$$

Machines non déterministes

- On peut définir les **classes de complexité** **NTIME** et **NSPACE** de la même façon que précédemment ;

Machines non déterministes

- On peut définir les **classes de complexité** **NTIME** et **NSPACE** de la même façon que précédemment ;
- On a $\mathbf{DTIME}(T(n)) \subseteq \mathbf{NTIME}(T(n))$ et $\mathbf{DSPACE}(T(n)) \subseteq \mathbf{NSPACE}(T(n))$ car une machine déterministe est un **cas particulier** de machine non déterministe.

Machines non déterministes

- On peut définir les **classes de complexité** **NTIME** et **NSPACE** de la même façon que précédemment ;
- On a $\mathbf{DTIME}(T(n)) \subseteq \mathbf{NTIME}(T(n))$ et $\mathbf{DSPACE}(T(n)) \subseteq \mathbf{NSPACE}(T(n))$ car une machine déterministe est un **cas particulier** de machine non déterministe.

Théorème

Pour toute machine **non déterministe***, il existe une machine* **déterministe** *qui la* **simule***.*
Mais en un temps exponentiel !

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_{k} \mathbf{DTIME}(n^k)$$

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;
- Les problèmes **polynomiaux** en espace : **PSPACE** ;

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;
- Les problèmes **polynomiaux** en espace : **PSPACE** ;
- Les problèmes à vérification **polynomiale** en espace : **NPSPACE** ;

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;
- Les problèmes **polynomiaux** en espace : **PSPACE** ;
- Les problèmes à vérification **polynomiale** en espace : **NPSPACE** ;
- Les problèmes **exponentiels** : **EXPTIME** ;

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;
- Les problèmes **polynomiaux** en espace : **PSPACE** ;
- Les problèmes à vérification **polynomiale** en espace : **NPSPACE** ;
- Les problèmes **exponentiels** : **EXPTIME** ;
- Les problèmes à vérification **exponentielle** : **NEXPTIME**

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;
- Les problèmes **polynomiaux** en espace : **PSPACE** ;
- Les problèmes à vérification **polynomiale** en espace : **NPSPACE** ;
- Les problèmes **exponentiels** : **EXPTIME** ;
- Les problèmes à vérification **exponentielle** : **NEXPTIME**
- Les problèmes **exponentiels** en espace : **EXPSPACE** ;

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;
- Les problèmes **polynomiaux** en espace : **PSPACE** ;
- Les problèmes à vérification **polynomiale** en espace : **NPSPACE** ;
- Les problèmes **exponentiels** : **EXPTIME** ;
- Les problèmes à vérification **exponentielle** : **NEXPTIME**
- Les problèmes **exponentiels** en espace : **EXPSPACE** ;
- Les problèmes à vérification **exponentielle** en espace : **NEXPSPACE**

Grandes classes de complexité

- Les problèmes **polynomiaux** : **PTIME** (ou **P**) :

$$\mathbf{P} = \bigcup_k \mathbf{DTIME}(n^k)$$

- Les problèmes à vérification **polynomiale** : **NPTIME** (ou **NP**) ;
- Les problèmes **polynomiaux** en espace : **PSPACE** ;
- Les problèmes à vérification **polynomiale** en espace : **NPSPACE** ;
- Les problèmes **exponentiels** : **EXPTIME** ;
- Les problèmes à vérification **exponentielle** : **NEXPTIME**
- Les problèmes **exponentiels** en espace : **EXPSPACE** ;
- Les problèmes à vérification **exponentielle** en espace : **NEXPSPACE**
- Les problèmes **élémentaires** : **ELEMENTARY** :

$$\mathbf{ELEMENTARY} = \mathbf{DTIME}(2^n) \cup \mathbf{DTIME}(2^{2^n}) \cup \mathbf{DTIME}(2^{2^{2^n}}) \cup \cdots$$

Grandes classes de complexité

- Les classes précédentes donnent la complexité pour répondre « oui » au problème ;

Grandes classes de complexité

- Les classes précédentes donnent la complexité pour répondre « oui » au problème ;
- Pour chaque classe $\mathcal{C}$ on définit la classe « $co - \mathcal{C}$ » qui donne la complexité pour répondre « non ».

Grandes classes de complexité

- Les classes précédentes donnent la complexité pour répondre « oui » au problème ;
- Pour chaque classe $\mathcal{C}$ on définit la classe « $co - \mathcal{C}$ » qui donne la complexité pour répondre « non ».
- si on peut ramener par une **réduction polynomiale** (logarithmique pour **P**) tout problème de la classe $\mathcal{C}$ au problème $\mathcal{P}$ alors $\mathcal{P}$ est dans la classe « $\mathcal{C}$-**difficile** » ;

Grandes classes de complexité

- Les classes précédentes donnent la complexité pour répondre « oui » au problème ;
- Pour chaque classe $\mathcal{C}$ on définit la classe « $co - \mathcal{C}$ » qui donne la complexité pour répondre « non ».
- si on peut ramener par une **réduction polynomiale** (logarithmique pour **P**) tout problème de la classe $\mathcal{C}$ au problème $\mathcal{P}$ alors $\mathcal{P}$ est dans la classe « $\mathcal{C}$**-difficile** » ;
- si $\mathcal{P}$ est également dans $\mathcal{C}$ alors $\mathcal{P}$ est dans « $\mathcal{C}$**-complet** »

Hierarchie des grandes classes de complexité

- On a :

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{EXSPACE}$$

Hierarchie des grandes classes de complexité

- On a :

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{EXSPACE}$$

- L'une des trois premières inclusions est **stricte** car $\mathbf{P} \subset \mathbf{EXPTIME}$!

Hierarchie des grandes classes de complexité

- On a :

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{EXSPACE}$$

- L'une des trois premières inclusions est **stricte** car $\mathbf{P} \subset \mathbf{EXPTIME}$!
- La question à **1 million de dollars** est $\mathbf{P} = \mathbf{NP}$? (probablement pas).

Conclusion

- Une théorie **fondamentale** pour l'informatique :

Conclusion

- Une théorie **fondamentale** pour l'informatique :
  - **Indécidabilité** de l'arrêt des algortihmes ;

Conclusion

- Une théorie **fondamentale** pour l'informatique :
  - **Indécidabilité** de l'arrêt des algortihmes ;
  - **Indécidabilité** de toutes les propriétés **non triviales** ;

Conclusion

- Une théorie **fondamentale** pour l'informatique :
  - **Indécidabilité** de l'arrêt des algortihmes ;
  - **Indécidabilité** de toutes les propriétés **non triviales** ;
  - Notion de **complexité** pour **comparer** les algorithmes et **hiérarchiser** les problèmes.

Conclusion

- Une théorie **fondamentale** pour l'informatique :
  - **Indécidabilité** de l'arrêt des algortihmes ;
  - **Indécidabilité** de toutes les propriétés **non triviales** ;
  - Notion de **complexité** pour **comparer** les algorithmes et **hiérarchiser** les problèmes.
- Une théorie qui a vu le jour **avant** l'invention des ordinateurs ($\sim$1935) ! ;

Conclusion

- Une théorie **fondamentale** pour l'informatique :
  - **Indécidabilité** de l'arrêt des algortihmes ;
  - **Indécidabilité** de toutes les propriétés **non triviales** ;
  - Notion de **complexité** pour **comparer** les algorithmes et **hiérarchiser** les problèmes.
- Une théorie qui a vu le jour **avant** l'invention des ordinateurs ($\sim$1935) ! ;
- Une théorie **liée** à d'autres grands problèmes mathématiques : λ-calcul, récursion, incomplétude des théories logiques ;

Conclusion

- Une théorie **fondamentale** pour l'informatique :
  - **Indécidabilité** de l'arrêt des algortihmes ;
  - **Indécidabilité** de toutes les propriétés **non triviales** ;
  - Notion de **complexité** pour **comparer** les algorithmes et **hiérarchiser** les problèmes.
- Une théorie qui a vu le jour **avant** l'invention des ordinateurs ($\sim$1935) ! ;
- Une théorie **liée** à d'autres grands problèmes mathématiques : λ-calcul, récursion, incomplétude des théories logiques ;
- Principaux **acteurs** : Alan Turing, Alonzo Church, Stephen Kleene et Kurt Gödel.

Bibliographie

- P. Dehornoy, *Complexité et Décidabilité*, Springer-Verlag, 1993.
- M. Sipser, *Introduction to the Theory of Computation*, PWS Pub. Co., 1996.
- http ://www.wikipedia.org