Master Informatique - M1 - UE Complexité
Chapitre 5 : La classe de complexité **P**

Philippe Jégou

Laboratoire d'Informatique et Systèmes - LIS - UMR CNRS 7020

Équipe COALA - COntraintes, ALgorithmes et Applications

(Algorithmique et Complexité de l'Intelligence Artificielle)

Campus de Saint-Jérôme

Département Informatique et Interactions

Faculté des Sciences

Université d'Aix-Marseille

philippe.jegou@univ-amu.fr

28 septembre 2022

En préambule

Ce que l'on va voir :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe

En préambule

Ce que l'on va voir :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe

- **Le modèle de calcul déterministe ?**
  - Machines de Turing Déterministes pour qu'il n'y ait pas imprécision sur
    - la taille de la donnée en entrée : longueur du mot en entrée
    - le temps de calcul : nombre de transitions de la machine

En préambule

Ce que l'on va voir :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe
- **Le modèle de calcul déterministe ?**
  - Machines de Turing Déterministes pour qu'il n'y ait pas imprécision sur
    - la taille de la donnée en entrée : longueur du mot en entrée
    - le temps de calcul : nombre de transitions de la machine
  - Extension aux autres modèles de calcul déterministes
    - pour s'assurer que la définition par MTD n'est pas restrictive
    - pour faciliter le "classement" de problèmes dans la classe **P**

En préambule

Ce que l'on va voir :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe
- **Le modèle de calcul déterministe ?**
  - Machines de Turing Déterministes pour qu'il n'y ait pas imprécision sur
    - la taille de la donnée en entrée : longueur du mot en entrée
    - le temps de calcul : nombre de transitions de la machine
  - Extension aux autres modèles de calcul déterministes
    - pour s'assurer que la définition par MTD n'est pas restrictive
    - pour faciliter le "classement" de problèmes dans la classe **P**
- Pour terminer : **la classe co-P**

Plan

1. Machines de Turing Déterministes : définition
2. MTD : de la reconnaissance de langages à la resolution de problèmes
3. La classe de complexité P définie par MTD
4. Pour conclure sur la classe P : Problèmes complémentaires

Plan

La Machine de Turing

- **Motivations de Turing :**
  - Formaliser la notion de calcul rationnel, "mécanisable"
  - Revient à formaliser la notion d'algorithme

La Machine de Turing

- **Motivations de Turing :**
  - Formaliser la notion de calcul rationnel, "mécanisable"
  - Revient à formaliser la notion d'algorithme
- **Il existe plusieurs modèles de machines de Turing**

  Ici : présentation du modèle *a priori* le plus simple

La Machine de Turing

- **Motivations de Turing :**
  - Formaliser la notion de calcul rationnel, "mécanisable"
  - Revient à formaliser la notion d'algorithme
- **Il existe plusieurs modèles de machines de Turing**
  Ici : présentation du modèle *a priori* le plus simple
- **Idée de base** (avec un opérateur "humain")
  - un opérateur dispose d'un crayon et d'une gomme

La Machine de Turing

- **Motivations de Turing :**
  - Formaliser la notion de calcul rationnel, "mécanisable"
  - Revient à formaliser la notion d'algorithme
- **Il existe plusieurs modèles de machines de Turing**
  Ici : présentation du modèle *a priori* le plus simple
- **Idée de base** (avec un opérateur "humain")
  - un opérateur dispose d'un crayon et d'une gomme
  - il dispose d'instructions prédéfinies simples à exécuter (un programme)

La Machine de Turing

- **Motivations de Turing :**
  - Formaliser la notion de calcul rationnel, "mécanisable"
  - Revient à formaliser la notion d'algorithme
- **Il existe plusieurs modèles de machines de Turing**
  Ici : présentation du modèle *a priori* le plus simple
- **Idée de base** (avec un opérateur "humain")
  - un opérateur dispose d'un crayon et d'une gomme
  - il dispose d'instructions prédéfinies simples à exécuter (un programme)
  - il voit 1 feuille de papier figurant dans une sequence (infinie) de feuilles

La Machine de Turing

- **Motivations de Turing :**
  - Formaliser la notion de calcul rationnel, "mécanisable"
  - Revient à formaliser la notion d'algorithme
- **Il existe plusieurs modèles de machines de Turing**
  Ici : présentation du modèle *a priori* le plus simple
- **Idée de base** (avec un opérateur "humain")
  - un opérateur dispose d'un crayon et d'une gomme
  - il dispose d'instructions prédéfinies simples à exécuter (un programme)
  - il voit 1 feuille de papier figurant dans une sequence (infinie) de feuilles
  - sur chaque feuille est écrit un symbole ou rien (feuille blanche)

La Machine de Turing

- **Motivations de Turing :**
  - Formaliser la notion de calcul rationnel, "mécanisable"
  - Revient à formaliser la notion d'algorithme
- **Il existe plusieurs modèles de machines de Turing**
  Ici : présentation du modèle *a priori* le plus simple
- **Idée de base** (avec un opérateur "humain")
  - un opérateur dispose d'un crayon et d'une gomme
  - il dispose d'instructions prédéfinies simples à exécuter (un programme)
  - il voit 1 feuille de papier figurant dans une sequence (infinie) de feuilles
  - sur chaque feuille est écrit un symbole ou rien (feuille blanche)
  - en fonction des instructions, de l'état courant et ce qu'il voit, il va :
    - éventuellement arrêter le traitement en disant OUI ou alors NON
    - éventuellement gommer le symbole et le remplacer par un autre
    - éventuellement passer à un autre état
    - éventuellement se positionner devant la feuille de gauche, ou de droite

Définition des MTD

Sous forme mécanique, une Machine de Turing est constituée de :

Définition des MTD

Sous forme mécanique, une Machine de Turing est constituée de :

- un système de contrôle d'états (il contient le programme)

Définition des MTD

Sous forme mécanique, une Machine de Turing est constituée de :

- un système de contrôle d'états (il contient le programme)
- un ruban infini constitué de cases pouvant contenir chacune 1 symbole c'est la mémoire de la machine (au début : il contient la donnée)
  (on peut imaginer les cases numérotées : ... , -3, -2, -1, 0, 1, 2, 3,...)

Définition des MTD

Sous forme mécanique, une Machine de Turing est constituée de :

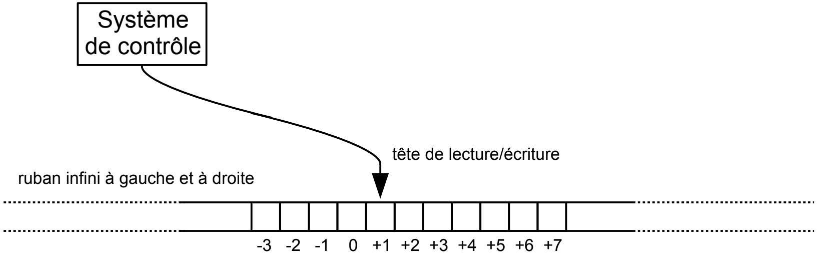
- un système de contrôle d'états (il contient le programme)
- un ruban infini constitué de cases pouvant contenir chacune 1 symbole c'est la mémoire de la machine (au début : il contient la donnée)
  (on peut imaginer les cases numérotées : ... , -3, -2, -1, 0, 1, 2, 3,...)
- une tête de lecture/écriture pointant sur une case du ruban
  (commandée par le système : peut se déplacer d'une case à gauche ou à droite)

Définition des MTD

Sous forme mécanique, une Machine de Turing est constituée de :

- un système de contrôle d'états (il contient le programme)
- un ruban infini constitué de cases pouvant contenir chacune 1 symbole c'est la mémoire de la machine (au début : il contient la donnée) (on peut imaginer les cases numérotées : ... , -3, -2, -1, 0, 1, 2, 3,...)
- une tête de lecture/écriture pointant sur une case du ruban (commandée par le système : peut se déplacer d'une case à gauche ou à droite)

Schématiquement :



Définition des MTD

Un **programme** M est défini par $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ où :

(on assimile 1 programme à 1 machine)

- Q est un ensemble fini d'états
- $q_0 \in Q$ est l'état de départ
- $q_{oui} \in Q$ est l'état terminal d'acceptation (réponse OUI)
- $q_{non} \in Q$ est l'état terminal de rejet (réponse NON)

Définition des MTD

Un **programme** M est défini par $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ où :
(on assimile 1 programme à 1 machine)

- Q est un ensemble fini d'états
- $q_0 \in Q$ est l'état de départ
- $q_{oui} \in Q$ est l'état terminal d'acceptation (réponse *OUI*)
- $q_{non} \in Q$ est l'état terminal de rejet (réponse *NON*)
- Σ est l'alphabet d'entrée : symboles pour le codage des données
- Γ est l'alphabet de ruban avec $\Sigma \subsetneq \Gamma$ (inclusion stricte)
- $\beta \in \Gamma$ est le symbole *blanc* (contenu des cases non utilisées)

Définition des MTD

Un **programme** M est défini par $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ où :

(on assimile 1 programme à 1 machine)

- Q est un ensemble fini d'états
- $q_0 \in Q$ est l'état de départ
- $q_{oui} \in Q$ est l'état terminal d'acceptation (réponse *OUI*)
- $q_{non} \in Q$ est l'état terminal de rejet (réponse *NON*)
- Σ est l'alphabet d'entrée : symboles pour le codage des données
- Γ est l'alphabet de ruban avec $\Sigma \subsetneq \Gamma$ (inclusion stricte)
- $\beta \in \Gamma$ est le symbole *blanc* (contenu des cases non utilisées)
- δ est la fonction de transition :

$$\delta : (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$$

Explications de la fonction de transition...

Définition des MTD

Explications de la fonction de transition...

$$\delta : (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$$

- δ prend en entrée un couple $(q, s) \in (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma$:
  - l'état courant non terminal $q \in (Q \backslash \{q_{oui}, q_{non}\})$
  - le symbole de ruban $s \in \Gamma$ écrit sur la case pointée

Définition des MTD

Explications de la fonction de transition...

$$\delta : (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$$

- δ prend en entrée un couple $(q, s) \in (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma$:
  - l'état courant non terminal $q \in (Q \backslash \{q_{oui}, q_{non}\})$
  - le symbole de ruban $s \in \Gamma$ écrit sur la case pointée
- δ fournit en résultat un triplet $(q', s', \Delta) \in Q \times \Gamma \times \{-1, 0, +1\}$:

Définition des MTD

Explications de la fonction de transition...

$$\delta : (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$$

- δ prend en entrée un couple $(q, s) \in (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma$:
  - l'état courant non terminal $q \in (Q \backslash \{q_{oui}, q_{non}\})$
  - le symbole de ruban $s \in \Gamma$ écrit sur la case pointée
- δ fournit en résultat un triplet $(q', s', \Delta) \in Q \times \Gamma \times \{-1, 0, +1\}$:
  - $q' \in Q$ est le nouvel état courant

Définition des MTD

Explications de la fonction de transition...

$$\delta : (Q\backslash\{q_{oui}, q_{non}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$$

- δ prend en entrée un couple $(q, s) \in (Q\backslash\{q_{oui}, q_{non}\}) \times \Gamma$:
  - l'état courant non terminal $q \in (Q\backslash\{q_{oui}, q_{non}\})$
  - le symbole de ruban $s \in \Gamma$ écrit sur la case pointée
- δ fournit en résultat un triplet $(q', s', \Delta) \in Q \times \Gamma \times \{-1, 0, +1\}$:
  - $q' \in Q$ est le nouvel état courant
  - $s' \in \Gamma$ est écrit par la tête sur la case pointée

Définition des MTD

Explications de la fonction de transition...

$$\delta : (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$$

- δ prend en entrée un couple $(q, s) \in (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma$:
  - l'état courant non terminal $q \in (Q \backslash \{q_{oui}, q_{non}\})$
  - le symbole de ruban $s \in \Gamma$ écrit sur la case pointée
- δ fournit en résultat un triplet $(q', s', \Delta) \in Q \times \Gamma \times \{-1, 0, +1\}$:
  - $q' \in Q$ est le nouvel état courant
  - $s' \in \Gamma$ est écrit par la tête sur la case pointée
  - $\Delta \in \{-1, 0, +1\}$ indique le déplacement de la tête de lecture/écriture :
    - d'une case vers la gauche (-1)
    - ou d'une case vers la droite (+1)
    - ou bien pas de déplacement (0).

Définition des MTD

Explications de la fonction de transition...

$$\delta : (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$$

- δ prend en entrée un couple $(q, s) \in (Q \backslash \{q_{oui}, q_{non}\}) \times \Gamma$:
  - l'état courant non terminal $q \in (Q \backslash \{q_{oui}, q_{non}\})$
  - le symbole de ruban $s \in \Gamma$ écrit sur la case pointée
- δ fournit en résultat un triplet $(q', s', \Delta) \in Q \times \Gamma \times \{-1, 0, +1\}$:
  - $q' \in Q$ est le nouvel état courant
  - $s' \in \Gamma$ est écrit par la tête sur la case pointée
  - $\Delta \in \{-1, 0, +1\}$ indique le déplacement de la tête de lecture/écriture :
    - d'une case vers la gauche (-1)
    - ou d'une case vers la droite (+1)
    - ou bien pas de déplacement (0).

Si le nouvel état courant est :

- q_{oui} : la machine (le programme) s'arrête avec la réponse OUI
- q_{non} : la machine (le programme) s'arrête avec la réponse NON
- un autre état que q_{oui} ou q_{non} alors la machine poursuit son exécution

Fonctionnement d'une MTD

Conditions initiales en début de traitement :

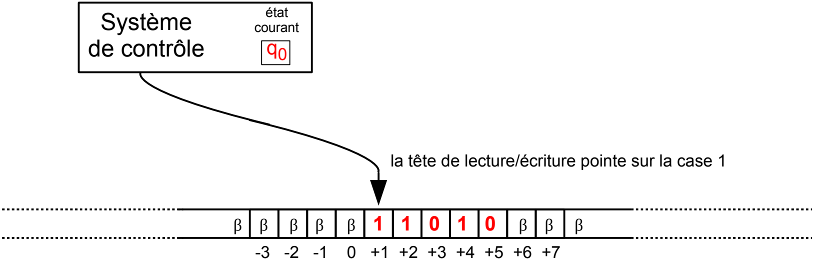
- donnée en entrée: un mot $m \in \Sigma^*$
  m écrit sur le ruban de la case indicée 1 à la case indicée $|m|$
  (toutes les autres cases du ruban sont marquées à blanc avec β)
- la tête de lecture/écriture pointe sur la case indicée 1
- l'état courant initial est q_0

Fonctionnement d'une MTD

Conditions initiales en début de traitement :

- donnée en entrée: un mot $m \in \Sigma^*$
  m écrit sur le ruban de la case indicée 1 à la case indicée $|m|$
  (toutes les autres cases du ruban sont marquées à blanc avec β)
- la tête de lecture/écriture pointe sur la case indicée 1
- l'état courant initial est q_0

Schématiquement :



Ici avec comme donnée le mot $m = 11010$ défini sur $\Sigma = \{0, 1\}$

Fonctionnement d'une MTD

À chaque étape :

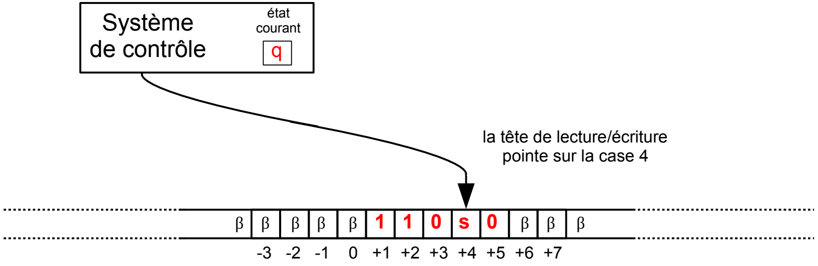
- machine dans un état courant $q \in Q$
- tête de lecture/écriture pointant sur une case contenant $s \in \Gamma$
- le système de contrôle opère une transition définie par la fonction δ

Fonctionnement d'une MTD

À chaque étape :

- machine dans un état courant $q \in Q$
- tête de lecture/écriture pointant sur une case contenant $s \in \Gamma$
- le système de contrôle opère une transition définie par la fonction δ

Schématiquement :

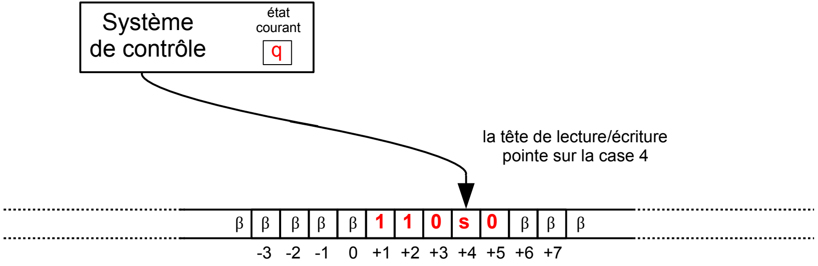


Fonctionnement d'une MTD

À chaque étape :

- machine dans un état courant $q \in Q$
- tête de lecture/écriture pointant sur une case contenant $s \in \Gamma$
- le système de contrôle opère une transition définie par la fonction δ

Schématiquement :



Transition $\delta(q, s) = (q', s', \Delta)$ avec $\Delta = +1$

Fonctionnement d'une MTD

À chaque étape, avec une transition $\delta(q, s) = (q', s', \Delta)$ avec $\Delta = +1$

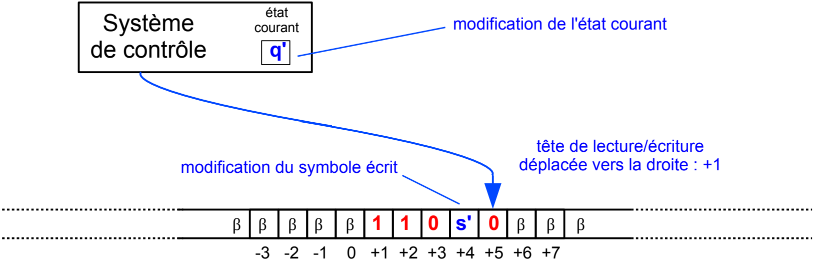
- le nouvel état est q'
- la tête de lecture/écriture remplace le symbole s par s'
- la tête de lecture/écriture se déplace d'une case à droite car $\Delta = +1$

Fonctionnement d'une MTD

À chaque étape, avec une transition $\delta(q, s) = (q', s', \Delta)$ avec $\Delta = +1$

- le nouvel état est q'
- la tête de lecture/écriture remplace le symbole s par s'
- la tête de lecture/écriture se déplace d'une case à droite car $\Delta = +1$

Schématiquement :



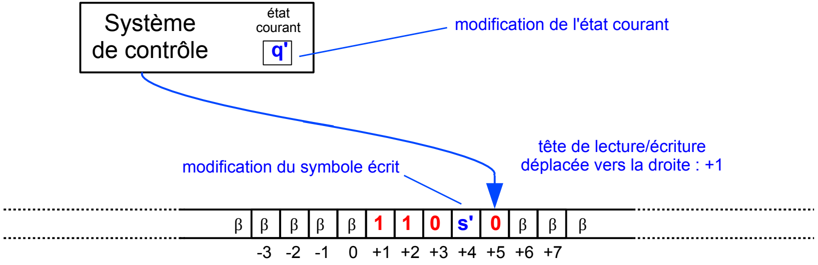
Transition $\delta(q, s) = (q', s', \Delta)$ avec $\Delta = +1$

Fonctionnement d'une MTD

Après la transition $\delta(q, s) = (q', s', \Delta)$ avec $\Delta = +1$

- le nouvel état est q'
- la tête de lecture/écriture a remplacé le symbole s par s'
- la tête de lecture/écriture s'est déplacée à droite car $\Delta = +1$

Schématiquement :



Si $q' = q_{oui}$ **ou** $q' = q_{non}$ **alors arrêt, sinon le calcul se poursuit**

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16{+}8{+}0{+}2{+}0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16{+}8{+}0{+}2{+}0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Algorithme : va chercher et tester le bit de poids le plus faible qui indique la parité

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16{+}8{+}0{+}2{+}0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Algorithme : va chercher et tester le bit de poids le plus faible qui indique la parité

- **Si** le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Algorithme : va chercher et tester le bit de poids le plus faible qui indique la parité

- **Si** le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement
- Déplacer la tête de la gauche vers la droite :
  - jusqu'au premier symbole blanc (situé juste après le mot)
  - la case située à sa gauche contient alors le bit de poids le plus faible

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16{+}8{+}0{+}2{+}0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Algorithme : va chercher et tester le bit de poids le plus faible qui indique la parité

- **Si** le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement
- Déplacer la tête de la gauche vers la droite :
  - jusqu'au premier symbole blanc (situé juste après le mot)
  - la case située à sa gauche contient alors le bit de poids le plus faible
- Déplacer la tête d'une case de la droite vers la gauche

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Algorithme : va chercher et tester le bit de poids le plus faible qui indique la parité

- **Si** le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant) **Sinon** poursuite du traitement
- Déplacer la tête de la gauche vers la droite :
  - jusqu'au premier symbole blanc (situé juste après le mot)
  - la case située à sa gauche contient alors le bit de poids le plus faible
- Déplacer la tête d'une case de la droite vers la gauche
- Il suffit alors de comparer le symbole écrit dans cette case avec 0 :
  - S'il est égal à 0, alors le nombre est pair et donc arrêt sur q_{oui}
  - S'il est égal à 1, alors le nombre est impair et donc arrêt sur q_{non}

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci

  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement en transitant vers la droite et sur q_1

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16{+}8{+}0{+}2{+}0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci

  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement en transitant vers la droite et sur q_1

- Il faut un état de travail q_1 pour traiter ceci

  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci

  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement en transitant vers la droite et sur q_1

- Il faut un état de travail q_1 pour traiter ceci

  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc

- Il faut un état de travail q_2 pour traiter ceci

  Déplacer la tête d'une case de la droite vers la gauche

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Codage binaire sur $\Sigma = \{0, 1\}$, poids forts à gauche et sans bit redondant

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

β	β	β	β	β	**1**	**1**	**0**	**1**	**0**	β	β	β
	-3	-2	-1	0	+1	+2	+3	+4	+5	+6	+7	

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci

  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement en transitant vers la droite et sur q_1

- Il faut un état de travail q_1 pour traiter ceci

  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc

- Il faut un état de travail q_2 pour traiter ceci

  Déplacer la tête d'une case de la droite vers la gauche

- Arrivé sur la dernière case du mot dans l'état q_2, il faut tester la case

Fonctionnement d'une MTD

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci
  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement en transitant vers la droite et sur q_1
- Il faut un état de travail q_1 pour traiter ceci
  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc
- Il faut un état de travail q_2 pour traiter ceci
  Déplacer la tête d'une case de la droite vers la gauche
- Arrivé sur la dernière case du mot dans l'état q_2, il faut tester la case

Fonctionnement d'une MTD

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci
  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement en transitant vers la droite et sur q_1
- Il faut un état de travail q_1 pour traiter ceci
  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc
- Il faut un état de travail q_2 pour traiter ceci
  Déplacer la tête d'une case de la droite vers la gauche
- Arrivé sur la dernière case du mot dans l'état q_2, il faut tester la case

On définit donc le programme $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ où :

- On a $Q = \{q_0, q_1, q_2, q_{oui}, q_{non}\}$, $\Sigma = \{0, 1\}$ et $\Gamma = \{0, 1, \beta\}$

Fonctionnement d'une MTD

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci
  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement en transitant vers la droite et sur q_1
- Il faut un état de travail q_1 pour traiter ceci
  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc
- Il faut un état de travail q_2 pour traiter ceci
  Déplacer la tête d'une case de la droite vers la gauche
- Arrivé sur la dernière case du mot dans l'état q_2, il faut tester la case

On définit donc le programme $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ **où :**

- On a $Q = \{q_0, q_1, q_2, q_{oui}, q_{non}\}$, $\Sigma = \{0, 1\}$ et $\Gamma = \{0, 1, \beta\}$
- δ est la fonction de transition :

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Avec q_{oui} ou q_{non}, écriture et déplacement sans importance, cf. **"?"**

Fonctionnement d'une MTD

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci
  Si le premier symbole = 0 **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement
- Il faut un état de travail q_1 pour traiter ceci
  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc
- Il faut un état de travail q_2 pour traiter ceci
  Déplacer la tête d'une case de la droite vers la gauche
- Arrivé sur la dernière case du mot dans l'état q_2, il faut tester la case

On définit donc le programme $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ **où :**

- On a $Q = \{q_0, q_1, q_2, q_{oui}, q_{non}\}$, $\Sigma = \{0, 1\}$ et $\Gamma = \{0, 1, \beta\}$
- δ est la fonction de transition :

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Avec q_{oui} ou q_{non}, écriture et déplacement sans importance, cf. "?"

Fonctionnement d'une MTD

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci
  Si le premier symbole $=0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement
- Il faut un état de travail q_1 pour traiter ceci
  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc
- Il faut un état de travail q_2 pour traiter ceci
  Déplacer la tête d'une case de la droite vers la gauche
- Arrivé sur la dernière case du mot dans l'état q_2, il faut tester la case

On définit donc le programme $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ **où :**

- On a $Q = \{q_0, q_1, q_2, q_{oui}, q_{non}\}$, $\Sigma = \{0, 1\}$ et $\Gamma = \{0, 1, \beta\}$
- δ est la fonction de transition :

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Avec q_{oui} ou q_{non}, écriture et déplacement sans importance, cf. "?"

Fonctionnement d'une MTD

Traduction de l'algorithme en programme pour Machine de Turing :

- Il faut que l'état initial q_0 serve à traiter ceci
  Si le premier symbole $= 0$ **Alors** arrêt sur q_{non} (cf. pas de bit redondant)
  Sinon poursuite du traitement
- Il faut un état de travail q_1 pour traiter ceci
  Déplacer la tête de la gauche vers la droite jusqu'au premier symbole blanc
- Il faut un état de travail q_2 pour traiter ceci
  Déplacer la tête d'une case de la droite vers la gauche
- **Arrivé sur la dernière case du mot dans l'état q_2, il faut tester la case**

On définit donc le programme $M = (Q, q_0, q_{oui}, q_{non}, \Sigma, \Gamma, \beta, \delta)$ où :

- On a $Q = \{q_0, q_1, q_2, q_{oui}, q_{non}\}$, $\Sigma = \{0, 1\}$ et $\Gamma = \{0, 1, \beta\}$
- δ est la fonction de transition :

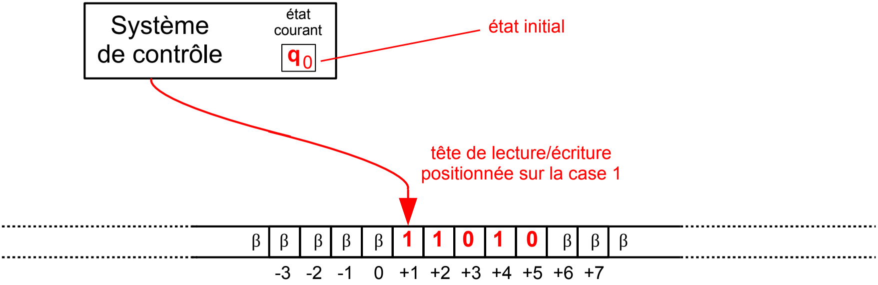
δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Avec q_{oui} ou q_{non}, écriture et déplacement sans importance, cf. "?"

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)



Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

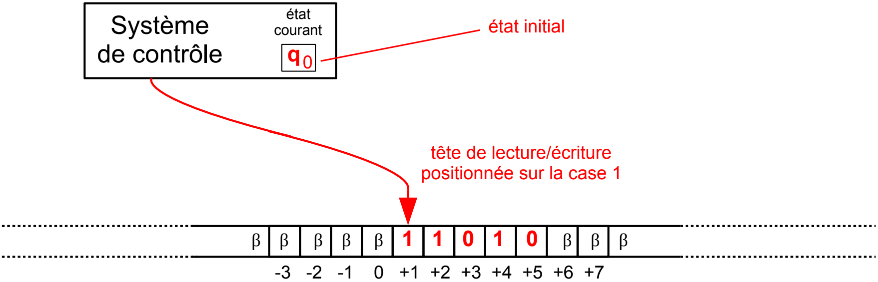


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

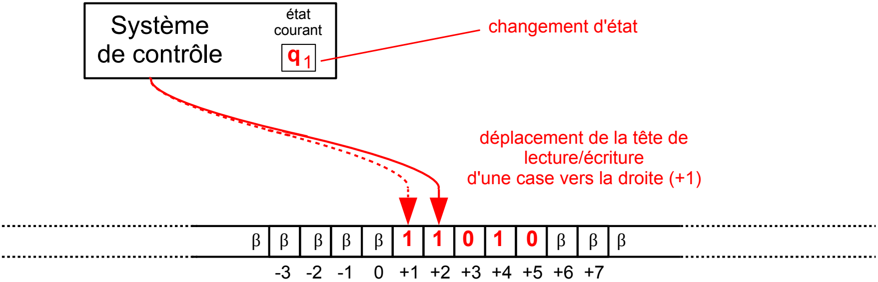


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

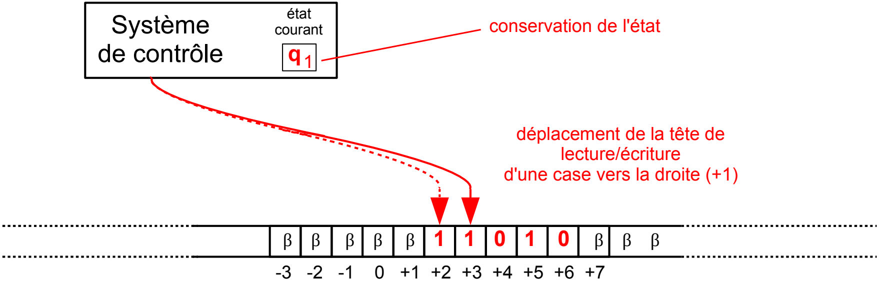


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

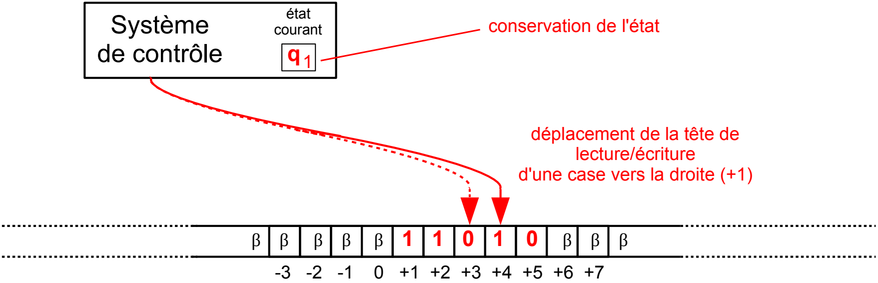


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

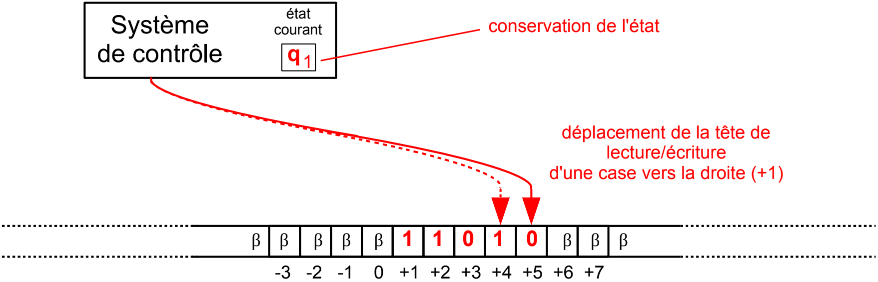


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

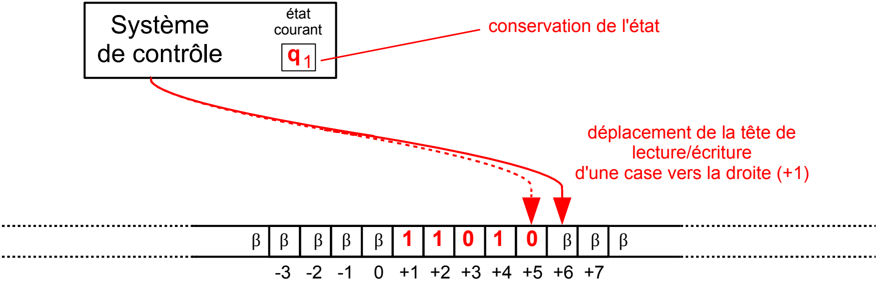


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

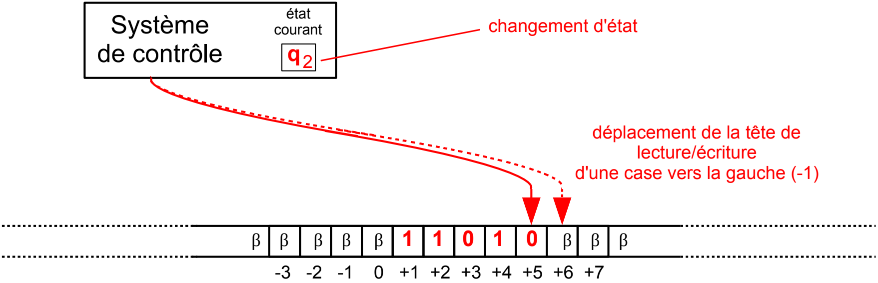


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

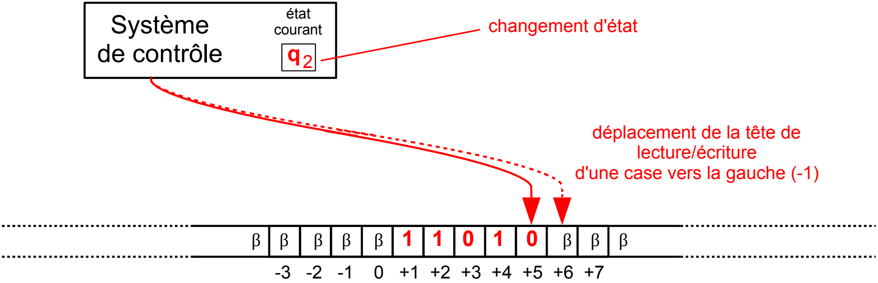


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

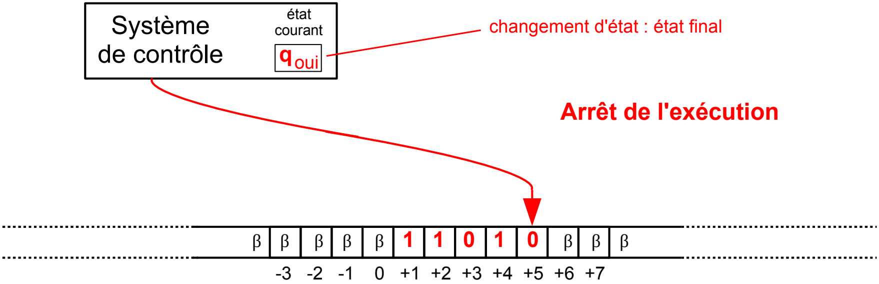


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 27$ le mot en entrée est $m = 11011$ (cf. 27 = 16+8+0+2+1)

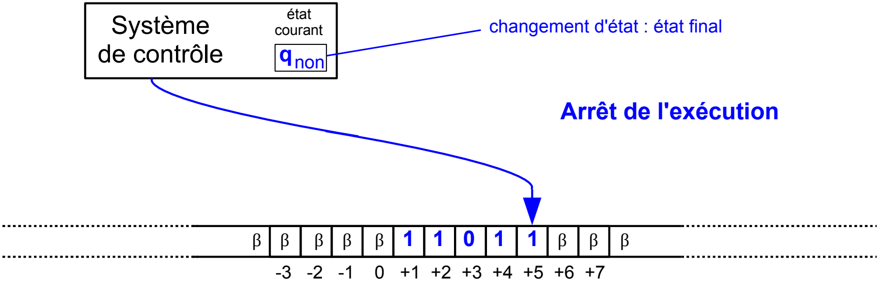


Table de la fonction de transition δ

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_2, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD : **avec un bug !**

Fonctionnement d'une MTD : **avec un bug !**

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

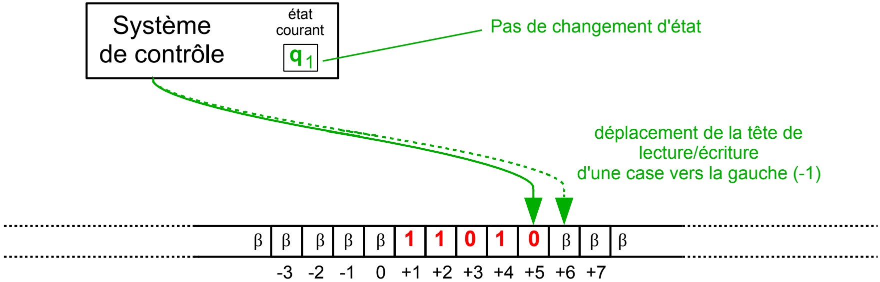


Table de la fonction de transition δ **avec erreur de programmation !**

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_1, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)

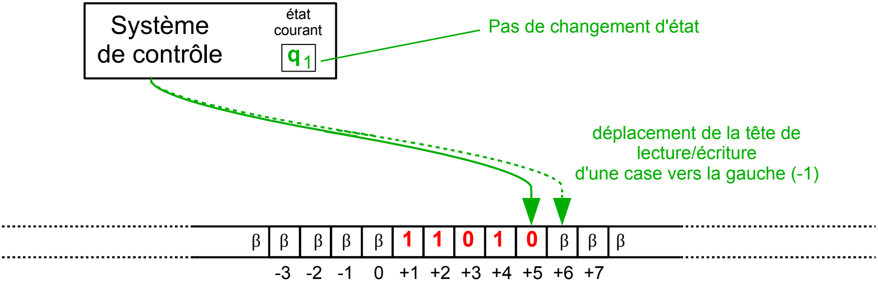


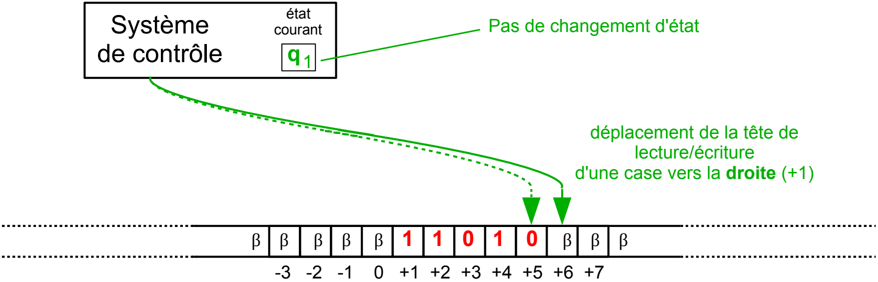
Table de transition δ avec erreur de programmation : ici bouble infinie!

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_1, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)



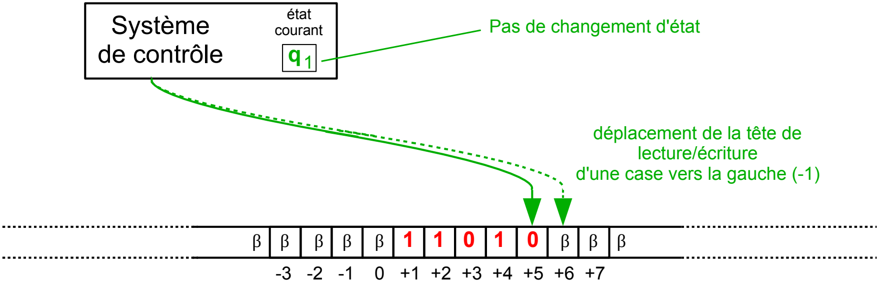
Bouble infinie car la machine reste dans q_1

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_1, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16+8+0+2+0$)



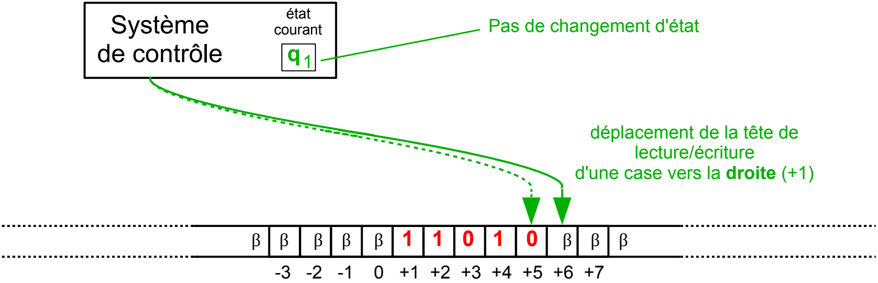
Bouble infinie car la machine reste dans q_1

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_1, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Exemple : vérification de la parité d'un entier $n \in \mathbb{N}^*$

Avec l'entier $n = 26$ le mot en entrée est $m = 11010$ (cf. $26 = 16{+}8{+}0{+}2{+}0$)



Bouble infinie car la machine reste dans q_1

δ	**0**	**1**	β
q_0	$(q_{non}, ?, ?)$	$(q_1, 1, +1)$	$(q_{non}, ?, ?)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(q_1, \beta, -1)$
q_2	$(q_{oui}, ?, ?)$	$(q_{non}, ?, ?)$	*impossible*

Fonctionnement d'une MTD

Quelques remarques :

- **Machine Déterministe ?**

  Car dans une configuration (q, s) une seule transition est possible

Fonctionnement d'une MTD

Quelques remarques :

- **Machine Déterministe ?**

  Car dans une configuration (q, s) une seule transition est possible

- **Modèle présenté : MTD pour la décision**

  Sert à la reconnaissance de langage :

  "Est-ce qu'un mot appartient à un langage donné ?"

  (dans l'exemple les mots sur $\Sigma = \{0, 1\}$ commençant par 1 et finissant par 0)

Fonctionnement d'une MTD

Quelques remarques :

- **Machine Déterministe ?**

  Car dans une configuration (q, s) une seule transition est possible

- **Modèle présenté : MTD pour la décision**

  Sert à la reconnaissance de langage :

  "Est-ce qu'un mot appartient à un langage donné ?"

  (dans l'exemple les mots sur $\Sigma = \{0, 1\}$ commençant par 1 et finissant par 0)

- **On peut définir des machines autres que pour la décision :**

  Pour faire tous les calculs qu'un ordinateur peut réaliser !

  (cf. cours de Calculabilité de L3 Informatique...)

  On le rappelle sur un exemple.

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$
- **Le programme :**

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$
- **Le programme :**
  1. la machine va rechercher le symbole "$+$"

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$
- **Le programme :**
  1. la machine va rechercher le symbole "$+$"
  2. elle remplace le "$+$" par le symbole "1"

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$
- **Le programme :**
  1. la machine va rechercher le symbole "$+$"
  2. elle remplace le "$+$" par le symbole "1"
  3. elle recherche le dernier "1" de y (le plus à droite)

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$
- **Le programme :**
  1. la machine va rechercher le symbole "$+$"
  2. elle remplace le "$+$" par le symbole "1"
  3. elle recherche le dernier "1" de y (le plus à droite)
  4. elle remplace le dernier "1" par le symbole blanc β

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$
- **Le programme :**
  1. la machine va rechercher le symbole "$+$"
  2. elle remplace le "$+$" par le symbole "1"
  3. elle recherche le dernier "1" de y (le plus à droite)
  4. elle remplace le dernier "1" par le symbole blanc β
  5. en fin de calcul le ruban contient le mot $m' = 11111111$
     m' composé de 8 symboles 1 est le codage de la somme de $3 + 5$

Fonctionnement d'une MTD

Un machine de Turing pour additionner des entiers :

- **Le résultat sera écrit sur le ruban**
- **Pour faire simple :** codage unaire sur $\Sigma = \{1, +\}$

  codage de 2 entiers x et y à additionner :

  par un mot $m = 1^x + 1^y$ (le symbole "$+$" sert de séparateur)

  pour additionner $x = 3$ et $y = 5$, le codage de l'entrée est $m = 111 + 11111$
- **Le programme :**
  1. la machine va rechercher le symbole "$+$"
  2. elle remplace le "$+$" par le symbole "1"
  3. elle recherche le dernier "1" de y (le plus à droite)
  4. elle remplace le dernier "1" par le symbole blanc β
  5. en fin de calcul le ruban contient le mot $m' = 11111111$
     m' composé de 8 symboles 1 est le codage de la somme de $3 + 5$
- **Bien sûr, on peut utiliser des codages raisonnables pour faire des calculs...**

Plan

1. Machines de Turing Déterministes : définition
2. MTD : de la reconnaissance de langages à la resolution de problèmes
3. La classe de complexité P définie par MTD
4. Pour conclure sur la classe P : Problèmes complémentaires

Reconnaissance de langage

Une Machine de Turing Déterministe peut **reconnaître des langages :**

Définition

Etant donné un programme M, défini sur un alphabet d'entrée Σ, trois langages disjoints peuvent être définis sur Σ :

Reconnaissance de langage

Une Machine de Turing Déterministe peut **reconnaître des langages :**

Définition

Etant donné un programme M, défini sur un alphabet d'entrée Σ, trois langages disjoints peuvent être définis sur Σ :

- $Accepte(M) = \{m \in \Sigma^* :\ M$ s'arrête sur q_{oui} avec m en entrée $\}$

Reconnaissance de langage

Une Machine de Turing Déterministe peut **reconnaître des langages :**

Définition

Etant donné un programme M, défini sur un alphabet d'entrée Σ, trois langages disjoints peuvent être définis sur Σ :

- $Accepte(M) = \{m \in \Sigma^* : M$ s'arrête sur q_{oui} avec m en entrée $\}$
- $Rejet(M) = \{m \in \Sigma^* : M$ s'arrête sur q_{non} avec m en entrée $\}$

Reconnaissance de langage

Une Machine de Turing Déterministe peut **reconnaître des langages :**

Définition

Etant donné un programme M, défini sur un alphabet d'entrée Σ, trois langages disjoints peuvent être définis sur Σ :

- $Accepte(M) = \{m \in \Sigma^* : M$ s'arrête sur q_{oui} avec m en entrée $\}$
- $Rejet(M) = \{m \in \Sigma^* : M$ s'arrête sur q_{non} avec m en entrée $\}$
- $Boucle(M) = \{m \in \Sigma^* : M$ ne s'arrête pas avec m en entrée $\}$.

Reconnaissance de langage

Une Machine de Turing Déterministe peut **reconnaître des langages :**

Définition

Etant donné un programme M, défini sur un alphabet d'entrée Σ, trois langages disjoints peuvent être définis sur Σ :

- $Accepte(M) = \{m \in \Sigma^* : M$ s'arrête sur q_{oui} avec m en entrée $\}$
- $Rejet(M) = \{m \in \Sigma^* : M$ s'arrête sur q_{non} avec m en entrée $\}$
- $Boucle(M) = \{m \in \Sigma^* : M$ ne s'arrête pas avec m en entrée $\}$.

$L(M) = Accepte(M)$ est le langage reconnu par le programme M.

Reconnaissance de langage

Une Machine de Turing Déterministe peut **reconnaître des langages :**

Définition

Etant donné un programme M, défini sur un alphabet d'entrée Σ, trois langages disjoints peuvent être définis sur Σ :

- $Accepte(M) = \{m \in \Sigma^* :\ M$ s'arrête sur q_{oui} avec m en entrée $\}$
- $Rejet(M) = \{m \in \Sigma^* :\ M$ s'arrête sur q_{non} avec m en entrée $\}$
- $Boucle(M) = \{m \in \Sigma^* :\ M$ ne s'arrête pas avec m en entrée $\}$.

$L(M) = Accepte(M)$ est le langage reconnu par le programme M.

Pour le programme M défini dans la section précédente, on a:

$$L(M) = Accepte(M) = \{m \in \{0,1\}^* :\ m \text{ débute par 1 et termine par 0 } \}$$

Reconnaissance de langage

Une Machine de Turing Déterministe peut **reconnaître des langages :**

Définition

Etant donné un programme M, défini sur un alphabet d'entrée Σ, trois langages disjoints peuvent être définis sur Σ :

- $Accepte(M) = \{m \in \Sigma^* :$ M s'arrête sur q_{oui} avec m en entrée $\}$
- $Rejet(M) = \{m \in \Sigma^* :$ M s'arrête sur q_{non} avec m en entrée $\}$
- $Boucle(M) = \{m \in \Sigma^* :$ M ne s'arrête pas avec m en entrée $\}$.

$L(M) = Accepte(M)$ est le langage reconnu par le programme M.

Pour le programme M défini dans la section précédente, on a:

$$L(M) = Accepte(M) = \{m \in \{0,1\}^* : m \text{ débute par 1 et termine par 0 } \}$$

Mais quelle est la capacité de calcul des Machines de Turing ?

MTD : un modèle de calcul universel

Une question cruciale en Théorie de la Calculabilité :

Est-ce que le modèle de calcul des MTD est suffisant pour résoudre tous les problèmes qu'un ordinateur peut résoudre ?

MTD : un modèle de calcul universel

Une question cruciale en Théorie de la Calculabilité :

Est-ce que le modèle de calcul des MTD est suffisant pour résoudre tous les problèmes qu'un ordinateur peut résoudre ?

Définition

Un langage L est dit **récursivement énumérable** si et seulement si il existe un programme M pour MTD tel que $L = L(M)$.

MTD : un modèle de calcul universel

Une question cruciale en Théorie de la Calculabilité :

Est-ce que le modèle de calcul des MTD est suffisant pour résoudre tous les problèmes qu'un ordinateur peut résoudre ?

Définition

Un langage L est dit **récursivement énumérable** si et seulement si il existe un programme M pour MTD tel que $L = L(M)$.

Remarque : pour les langages récursivement énumérables, il peut y avoir des entrées pour lesquelles le programme de reconnaissance peut boucler.

MTD : un modèle de calcul universel

Une question cruciale en Théorie de la Calculabilité :

Est-ce que le modèle de calcul des MTD est suffisant pour résoudre tous les problèmes qu'un ordinateur peut résoudre ?

Définition

Un langage L est dit **récursivement énumérable** si et seulement si il existe un programme M pour MTD tel que $L = L(M)$.

Remarque : pour les langages récursivement énumérables, il peut y avoir des entrées pour lesquelles le programme de reconnaissance peut boucler.

d'où une définition plus restrictive : les **langages récursifs**.

Définition

Un langage L est dit **récursif** si et seulement s'il existe un programme M pour MTD tel que $L = L(M)$ et $Boucle(M) = \emptyset$.

MTD : un modèle de calcul universel

Une question naturelle : *Le modèle des MTD est-il* **universel** *?*

En reformulant : *Est-il aussi puissant que les autres modèles de calcul ?*

MTD : un modèle de calcul universel

Une question naturelle : *Le modèle des MTD est-il* **universel** *?*

En reformulant : *Est-il aussi puissant que les autres modèles de calcul ?*

Un théorème fondamental de la **Théorie de la Calculabilité** répond à cette question :

Théorème

Une langage L est récursif si et seulement s'il est reconnaissable sur un modèle de calcul quelconque, i.e. les modèles de calcul définis par les langages de programmation comme Pascal, C, Java, Python, Machines de Von Neumann, avec l'hypothèse d'une mémoires de taille infiinie.

MTD : un modèle de calcul universel

Une question naturelle : *Le modèle des MTD est-il* **universel** *?*

En reformulant : *Est-il aussi puissant que les autres modèles de calcul ?*

Un théorème fondamental de la **Théorie de la Calculabilité** répond à cette question :

Théorème

Une langage L est récursif si et seulement s'il est reconnaissable sur un modèle de calcul quelconque, i.e. les modèles de calcul définis par les langages de programmation comme Pascal, C, Java, Python, Machines de Von Neumann, avec l'hypothèse d'une mémoires de taille infiinie.

Et au-delà, la célèbre **"Thèse de Church"**...

MTD : un modèle de calcul universel

Une question naturelle : *Le modèle des MTD est-il* ***universel*** *?*

En reformulant : *Est-il aussi puissant que les autres modèles de calcul ?*

Un théorème fondamental de la **Théorie de la Calculabilité** répond à cette question :

Théorème

Une langage L est récursif si et seulement s'il est reconnaissable sur un modèle de calcul quelconque, i.e. les modèles de calcul définis par les langages de programmation comme Pascal, C, Java, Python, Machines de Von Neumann, avec l'hypothèse d'une mémoires de taille infiinie.

Et au-delà, la célèbre **"Thèse de Church"**... d'un point de vue pratique :

MTD : un modèle de calcul universel

Une question naturelle : *Le modèle des MTD est-il* **universel** *?*

En reformulant : *Est-il aussi puissant que les autres modèles de calcul ?*

Un théorème fondamental de la **Théorie de la Calculabilité** répond à cette question :

Théorème

Une langage L est récursif si et seulement s'il est reconnaissable sur un modèle de calcul quelconque, i.e. les modèles de calcul définis par les langages de programmation comme Pascal, C, Java, Python, Machines de Von Neumann, avec l'hypothèse d'une mémoires de taille infiinie.

Et au-delà, la célèbre **"Thèse de Church"**... d'un point de vue pratique :

"Tous les modèles de calcul sont aussi puissants que les MTD"

$\Rightarrow$ il serait illusoire de vouloir créer des modèles de calcul plus puissants (c'est-à-dire capables de reconnaître plus de langages)

Reconnaissance par MTD et résolution de problèmes

Le lien est direct entre :

- résolution de problèmes de décision et
- reconnaissance de langages :

Reconnaissance par MTD et résolution de problèmes

Le lien est direct entre :

- résolution de problèmes de décision et
- reconnaissance de langages :

Pour rappel (chapitre 4) :

À tout problème de décision π codé via un système S (avec Σ) on associe le langage $L(\pi, S)$ des instances positives de π :

$$L(\pi, S) = \{ \ m \in \Sigma^* : m \text{ code par le système } S \text{ une instance } I \in V_\pi \ \}$$

Reconnaissance par MTD et résolution de problèmes

Le lien est direct entre :

- résolution de problèmes de décision et
- reconnaissance de langages :

Pour rappel (chapitre 4) :

À tout problème de décision π codé via un système S (avec Σ) on associe le langage $L(\pi, S)$ des instances positives de π :

$$L(\pi, S) = \{\ m \in \Sigma^* : m \text{ code par le système } S \text{ une instance } I \in V_\pi\ \}$$

Définition

On dira qu'**un programme** M **pour MTD résout un problème de décision** π sous un système de codage S, si M s'arrête pour toutes les entrées $m \in \Sigma^*$ et si $L(M) = L(\pi, S)$.

En d'autres termes : un programme M pour MTD **résout un problème de décision** π sous le système de codage S si M **reconnaît le langage des instances positives** de π et **refuse sur toutes les autres entrées**

Reconnaissance par MTD et résolution de problèmes

La notion de problème **résoluble** (qui peut être résolu) s'énonce, en termes de calculabilité par la notion de **décidabilité** (cf. problèmes de décisions)

Définition

Un problème π est dit **décidable** si et seulement s'il existe un codage de π et un programme M pour MTD tels que :

- $Accepte(M) = \{m \in \Sigma^* : \ m$ code une donnée positive de π $\}$
- $Rejet(M) = \{m \in \Sigma^* : \ m$ code une donnée négative de π $\}$
- $Boucle(M) = \emptyset$

Reconnaissance par MTD et résolution de problèmes

La notion de problème **résoluble** (qui peut être résolu) s'énonce, en termes de calculabilité par la notion de **décidabilité** (cf. problèmes de décisions)

Définition

Un problème π est dit **décidable** si et seulement s'il existe un codage de π et un programme M pour MTD tels que :

- $Accepte(M) = \{m \in \Sigma^* : m \text{ code une donnée positive de } \pi \}$
- $Rejet(M) = \{m \in \Sigma^* : m \text{ code une donnée négative de } \pi \}$
- $Boucle(M) = \emptyset$

Analogie avec les langages (cf. langage récursivement énumérable) :

Définition

Un problème π est dit **semi-décidable** si et seulement s'il existe un codage de π et un programme M pour MTD tel que

$$Accepte(M) = \{m \in \Sigma^* : m \text{ code une donnée positive de } \pi \}$$

mais sans garantie d'arrêt sur m ne codant pas une donnée positive

Avant de revenir à la Théorie de la Complexité...

Retour sur les Machines de Turing...

Il existe différents modèles de machines de Turing, parmi lesquels :

- modèle identique mais avec un demi-ruban infini (début à la case 1)
- modèle multi-têtes : il y a 1 ruban et k têtes ; les transitions sont réalisées en fonction de l'état courant et des symboles écrits sous chacune des têtes
- modèle multi-rubans et multi-têtes: il y a k rubans et k têtes (par exemple, 1 ruban pour la donnée, un pour le résultat, et $k-2$ rubans pour les calculs)
- modèle des Machines de Turing Non-Déterministes sera étudié dans le chapitre 6
- etc.

Avant de revenir à la Théorie de la Complexité...

Retour sur les Machines de Turing...

Il existe différents modèles de machines de Turing, parmi lesquels :

- modèle identique mais avec un demi-ruban infini (début à la case 1)
- modèle multi-têtes : il y a 1 ruban et k têtes ; les transitions sont réalisées en fonction de l'état courant et des symboles écrits sous chacune des têtes
- modèle multi-rubans et multi-têtes: il y a k rubans et k têtes (par exemple, 1 ruban pour la donnée, un pour le résultat, et $k-2$ rubans pour les calculs)
- modèle des Machines de Turing Non-Déterministes sera étudié dans le chapitre 6
- etc.

Ce qu'il faut retenir de ces différents modèles de machines :

- ils sont tous équivalents pour ce qui concerne l'étendue de leur capacité de calcul = l'ensemble des fonctions calculables est le même
- concernant leur temps de calcul, cela ne sera généralement pas le cas

Plan

Mesure du temps et complexité sur MTD

Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Mesure du temps et complexité sur MTD

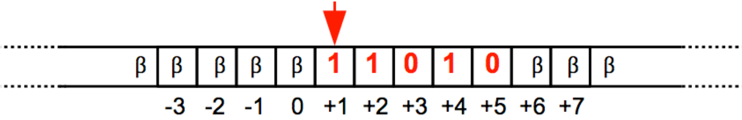
Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Sur un exemple : retour sur le test de parité

- en entrée : un mot $m \in \{0,1\}^*$ avec $|m| = n$ et tête sur la case 1



Mesure du temps et complexité sur MTD

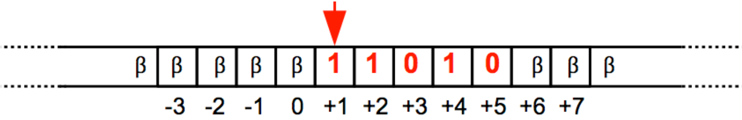
Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Sur un exemple : retour sur le test de parité

- en entrée : un mot $m \in \{0,1\}^*$ avec $|m| = n$ et tête sur la case 1



- la tête se déplace sur les cases jusqu'à la case d'indice $n+1$

Mesure du temps et complexité sur MTD

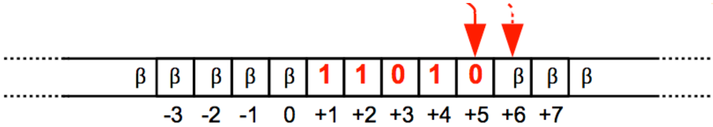
Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Sur un exemple : retour sur le test de parité

- arrivée jusqu'à la case d'indice $n + 1$, la tête revient sur la case d'indice n



Mesure du temps et complexité sur MTD

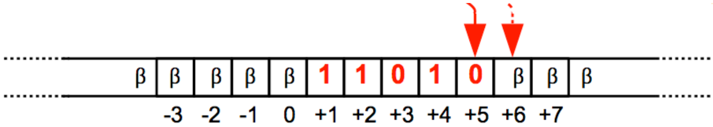
Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Sur un exemple : retour sur le test de parité

- arrivée jusqu'à la case d'indice $n+1$, la tête revient sur la case d'indice n



- coût (avec pire des cas quand la donnée n'est pas erronée) : $n+2$ transitions

Mesure du temps et complexité sur MTD

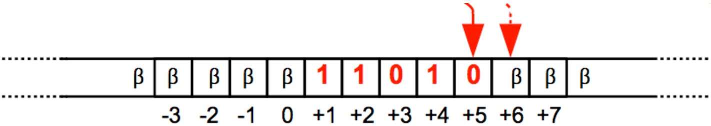
Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Sur un exemple : retour sur le test de parité

- arrivée jusqu'à la case d'indice $n+1$, la tête revient sur la case d'indice n



- coût (avec pire des cas quand la donnée n'est pas erronée) : $n+2$ transitions
  - de la case 1 à la case $n+1$: n transitions

Mesure du temps et complexité sur MTD

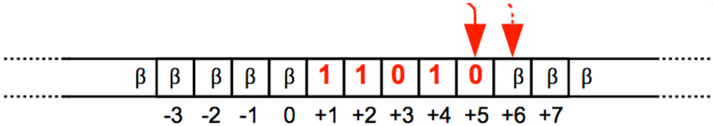
Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Sur un exemple : retour sur le test de parité

- arrivée jusqu'à la case d'indice $n + 1$, la tête revient sur la case d'indice n



- coût (avec pire des cas quand la donnée n'est pas erronée) : $n + 2$ transitions
  - de la case 1 à la case $n + 1$: n transitions
  - de la case $n + 1$ vers la case n : 1 transition

Mesure du temps et complexité sur MTD

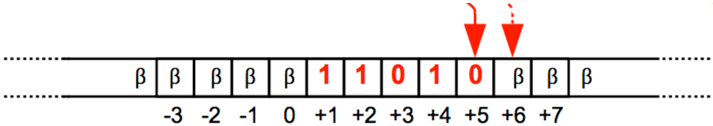
Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Sur un exemple : retour sur le test de parité

- arrivée jusqu'à la case d'indice $n+1$, la tête revient sur la case d'indice n



- coût (avec pire des cas quand la donnée n'est pas erronée) : $n+2$ transitions
  - de la case 1 à la case $n+1$: n transitions
  - de la case $n+1$ vers la case n : 1 transition
  - test final de la case n : 1 transition

Mesure du temps et complexité sur MTD

Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Mesure du temps et complexité sur MTD

Machine de Turing : permet de définir précisément le temps d'exécution

Définition

Soient M un programme pour MTD, et $m \in \Sigma^*$, une entrée ; le temps requis par M pour traiter m est égal au nombre de transitions opérées jusqu'à un état d'arrêt.

Et donc la complexité des programmes :

Définition

Soit M un programme pour MTD qui s'arrête pour toute entrée $m \in \Sigma^*$; la **fonction de complexité en temps du programme** M est notée T_M et est définie de $\mathbb{N} \rightarrow \mathbb{N}$ par $T_M(n) =$

$$max\{t : \exists m \in \Sigma^*, |m| = n \text{ et } t \text{ est le temps de calcul de } M \text{ avec } m\ \}$$

Le "*max*" de la définition : pour prendre en compte le pire des cas

Définition de la classe de complexité **P**

On peut ainsi définir la notion de programme polynomial pour MTD :

Définition

Un programme M pour MTD est dit **polynomial** s'il existe un polynôme q tel que $\forall n \in \mathbb{N}$, $T_M(n) \leq q(n)$, i.e. $T_M(n) \in O(q(n))$.

Définition de la classe de complexité **P**

On peut ainsi définir la notion de programme polynomial pour MTD :

Définition

Un programme M pour MTD est dit **polynomial** s'il existe un polynôme q tel que $\forall n \in \mathbb{N}$, $T_M(n) \leq q(n)$, i.e. $T_M(n) \in O(q(n))$.

Et donc la classe de complexité **P** comme un ensemble de langages :

Définition

La **classe de langages P** est définie par :

$$\mathbf{P} = \{\ L : \exists \text{ un programme } M \text{ pour MTD polynomial tel que } L = L(M)\}$$

Définition de la classe de complexité **P**

On peut ainsi définir la notion de programme polynomial pour MTD :

Définition

Un programme M pour MTD est dit **polynomial** s'il existe un polynôme q tel que $\forall n \in \mathbb{N}$, $T_M(n) \leq q(n)$, i.e. $T_M(n) \in O(q(n))$.

Et donc la classe de complexité **P** comme un ensemble de langages :

Définition

La **classe de langages P** est définie par :

$$\mathbf{P} = \{\ L : \exists \text{ un programme } M \text{ pour MTD polynomial tel que } L = L(M)\}$$

Classe de complexité **P** :

Ensemble des langages L reconnaissables par MTD en temps polynomial

Définition de la classe de complexité **P**

La classe de complexité **P** définie comme un ensemble de langages :

Définition

La **classe de langages P** est définie par :

$$\mathbf{P} = \{\ L : \exists \text{ un programme } M \text{ pour MTD polynomial tel que } L = L(M)\}$$

Définition de la classe de complexité **P**

La classe de complexité **P** définie comme un ensemble de langages :

Définition

La **classe de langages P** est définie par :

$$\mathbf{P} = \{\ L : \exists \text{ un programme } M \text{ pour MTD polynomial tel que } L = L(M)\}$$

Et comme :

reconnaissance de langages $\Leftrightarrow$ résolution de problèmes de décision

Définition de la classe de complexité **P**

La classe de complexité **P** définie comme un ensemble de langages :

Définition

La **classe de langages P** est définie par :

$$\mathbf{P} = \{\ L : \exists \text{ un programme } M \text{ pour MTD polynomial tel que } L = L(M)\}$$

Et comme :

reconnaissance de langages $\Leftrightarrow$ résolution de problèmes de décision

Définition

On dit qu'un problème de décision π appartient à la classe **P** sous le système de codage S si $L(\pi, S) \in \mathbf{P}$, c'est-à-dire, s'il existe un programme M pour MTD qui résout le problème π sous le système de codage S en un temps polynomial (le problème π est dit polynomial).

Définition de la classe de complexité **P**

La classe de complexité **P** définie comme un ensemble de langages :

Définition

La **classe de langages P** est définie par :

$$\mathbf{P} = \{ \; L : \exists \text{ un programme } M \text{ pour MTD polynomial tel que } L = L(M) \}$$

Et comme :

reconnaissance de langages $\Leftrightarrow$ résolution de problèmes de décision

Définition

On dit qu'un problème de décision π appartient à la classe **P** sous le système de codage S si $L(\pi, S) \in \mathbf{P}$, c'est-à-dire, s'il existe un programme M pour MTD qui résout le problème π sous le système de codage S en un temps polynomial (le problème π est dit polynomial).

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Définition de la classe de complexité **P**

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Définition de la classe de complexité **P**

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Mais : **Le modèle de calcul des MTD est-il le plus pertinent ?**

Définition de la classe de complexité **P**

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Mais : **Le modèle de calcul des MTD est-il le plus pertinent ?**

- L'emploi des MTD n'est il pas trop restrictif ?

  Le procédé des MTD s'avère lent (cf. temps pour trier n valeurs avec MTD ?)

Définition de la classe de complexité **P**

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Mais : **Le modèle de calcul des MTD est-il le plus pertinent ?**

- L'emploi des MTD n'est il pas trop restrictif ?

  Le procédé des MTD s'avère lent (cf. temps pour trier n valeurs avec MTD ?)
- S'agit-il du modèle de calcul le plus approprié pour définir la classe des problèmes polynomiaux ?

Définition de la classe de complexité **P**

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Mais : **Le modèle de calcul des MTD est-il le plus pertinent ?**

- L'emploi des MTD n'est il pas trop restrictif ?

  Le procédé des MTD s'avère lent (cf. temps pour trier n valeurs avec MTD ?)
- S'agit-il du modèle de calcul le plus approprié pour définir la classe des problèmes polynomiaux ?
- N'existe-t-il pas un autre modèle de calcul déterministe qui permette d'élargir la classe **P** ?

Définition de la classe de complexité **P**

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Mais : **Le modèle de calcul des MTD est-il le plus pertinent ?**

- L'emploi des MTD n'est il pas trop restrictif ?
  Le procédé des MTD s'avère lent (cf. temps pour trier n valeurs avec MTD ?)
- S'agit-il du modèle de calcul le plus approprié pour définir la classe des problèmes polynomiaux ?
- N'existe-t-il pas un autre modèle de calcul déterministe qui permette d'élargir la classe **P** ?
- Est-ce que certains problèmes de décision seraient exponentiels sur MTD et polynomiaux avec un autre modèle de calcul déterministe ?

Définition de la classe de complexité **P**

Classe de complexité P :

Problèmes de décision résolubles par MTD en temps polynomial

Mais : **Le modèle de calcul des MTD est-il le plus pertinent ?**

- L'emploi des MTD n'est il pas trop restrictif ?

  Le procédé des MTD s'avère lent (cf. temps pour trier n valeurs avec MTD ?)
- S'agit-il du modèle de calcul le plus approprié pour définir la classe des problèmes polynomiaux ?
- N'existe-t-il pas un autre modèle de calcul déterministe qui permette d'élargir la classe **P** ?
- Est-ce que certains problèmes de décision seraient exponentiels sur MTD et polynomiaux avec un autre modèle de calcul déterministe ?

Réponse : on ne connaît pas à ce jour de modèles de calcul déterministe "exponentiellement plus rapide" pour cela car il existe une "équivalence polynomiale" entre les différents modèles de calcul déterministes réalistes définis jusqu'à présent.

La classe de complexité **P** au-delà des MTD

On ne connaît pas à ce jour de modèle de calcul déterministe "exponentiellement plus rapide" car il existe une "équivalence polynomiale" entre les différents modèles de calcul déterministes réalistes définis jusqu'à présent.

La classe de complexité **P** au-delà des MTD

On ne connaît pas à ce jour de modèle de calcul déterministe "exponentiellement plus rapide" car il existe une "équivalence polynomiale" entre les différents modèles de calcul déterministes réalistes définis jusqu'à présent.

Il faut maintenant le prouver !

La classe de complexité **P** au-delà des MTD

On ne connaît pas à ce jour de modèle de calcul déterministe "exponentiellement plus rapide" car il existe une "équivalence polynomiale" entre les différents modèles de calcul déterministes réalistes définis jusqu'à présent.

Il faut maintenant le prouver !

On va le montrer (en fait dire que cela a été montré...)

La classe de complexité **P** au-delà des MTD

On ne connaît pas à ce jour de modèle de calcul déterministe "exponentiellement plus rapide" car il existe une "équivalence polynomiale" entre les différents modèles de calcul déterministes réalistes définis jusqu'à présent.

Il faut maintenant le prouver !

On va le montrer (en fait dire que cela a été montré...)

Idée de la preuve :

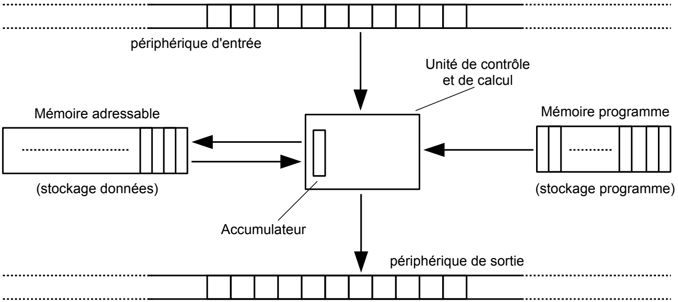
En étudiant le rapport d'efficacité relative existant entre les MTD et le **modèle RAM** "à la Von Neumann" qui constitue un modèle de calcul (architecture d'ordinateur) très proche des ordinateurs usuels.

(RAM pour **Random Access Machine** ou machine à mémoire adressable)

On les définit :

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur



Éléments de la machine :

- périphérique d'entrée
- périphérique de sortie
- unité de contrôle et de calcul (avec compteur ordinal)
- programme stocké dans une mémoire
- mémoire adressable avec nombre infini de registres $c(i)$ où $i \in \mathbb{N}$, et $c(0)$ est un accumulateur

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read / write`

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read` / `write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read` / `write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read / write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read / write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`
- structure de contrôle :
  - pour débuter : `start` et pour arrêter l'exécution `stop`

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read / write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`
- structure de contrôle :
  - pour débuter : `start` et pour arrêter l'exécution `stop`
  - pour affecter la valeur du compteur ordinal à n : `goto n`

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read / write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`
- structure de contrôle :
  - pour débuter : `start` et pour arrêter l'exécution `stop`
  - pour affecter la valeur du compteur ordinal à n : `goto n`
  - pour affecter la valeur du compteur ordinal à n au cas où la valeur dans l'accumulateur est positive, sinon incrémente le compteur ordinal : `if >= 0 then n`

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read` / `write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`
- structure de contrôle :
  - pour débuter : `start` et pour arrêter l'exécution `stop`
  - pour affecter la valeur du compteur ordinal à n : `goto n`
  - pour affecter la valeur du compteur ordinal à n au cas où la valeur dans l'accumulateur est positive, sinon incrémente le compteur ordinal : `if >= 0 then n`

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read / write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`
- structure de contrôle :
  - pour débuter : `start` et pour arrêter l'exécution `stop`
  - pour affecter la valeur du compteur ordinal à n : `goto n`
  - pour affecter la valeur du compteur ordinal à n au cas où la valeur dans l'accumulateur est positive, sinon incrémente le compteur ordinal : `if >= 0 then n`

Ce dispositif "matériel" doté de ce langage

- permet d'écrire des programmes et de les exécuter : c'est comme de l'**assembleur**, donc proche du C, du C++, etc.

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read` / `write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`
- structure de contrôle :
  - pour débuter : `start` et pour arrêter l'exécution `stop`
  - pour affecter la valeur du compteur ordinal à n : `goto n`
  - pour affecter la valeur du compteur ordinal à n au cas où la valeur dans l'accumulateur est positive, sinon incrémente le compteur ordinal : `if >= 0 then n`

Ce dispositif "matériel" doté de ce langage

- permet d'écrire des programmes et de les exécuter : c'est comme de l'**assembleur**, donc proche du C, du C++, etc.
- constitue une architecture réaliste, et surtout...

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur et des programmes définis sur la base du jeu d'instruction suivant :

- pour les entrées / sorties : `read / write`
- pour les transferts vers l'accumulateur : `load`
  - mode immédiat `load-imm i` $\equiv$ charger i
  - mode direct `load-dir i` $\equiv$ charger $c(i)$
  - mode indirect `load-ind i` $\equiv$ charger $c(c(i))$
- pour les transferts depuis l'accumulateur : `store`
  - mode direct `store i` $\equiv$ recopier dans le i^{eme} registre
  - mode indirect `store-ind i` $\equiv$ recopier dans le $c(i)^{eme}$ registre
- opérations arithmétiques de base : `add, sub, mult, div`
- structure de contrôle :
  - pour débuter : `start` et pour arrêter l'exécution `stop`
  - pour affecter la valeur du compteur ordinal à n : `goto n`
  - pour affecter la valeur du compteur ordinal à n au cas où la valeur dans l'accumulateur est positive, sinon incrémente le compteur ordinal : `if >= 0 then n`

Ce dispositif "matériel" doté de ce langage

- permet d'écrire des programmes et de les exécuter : c'est comme de l'**assembleur**, donc proche du C, du C++, etc.
- constitue une architecture réaliste, et surtout...

Cela définit un modèle de calcul théorique... et réaliste !

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur qui

- permet d'écrire des programmes et de les exécuter
- constitue une architecture réaliste, et surtout...
- définit un modèle de calcul !

Mais qu'en est-il du temps ?

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur qui

- permet d'écrire des programmes et de les exécuter
- constitue une architecture réaliste, et surtout...
- définit un modèle de calcul !

Mais qu'en est-il du temps ?

Hypothèse temporelle pour les RAM :

"toutes les opérations (adressage compris) s'exécutent en temps constant"

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur qui

- permet d'écrire des programmes et de les exécuter
- constitue une architecture réaliste, et surtout...
- définit un modèle de calcul !

Mais qu'en est-il du temps ?

Hypothèse temporelle pour les RAM :

"toutes les opérations (adressage compris) s'exécutent en temps constant"

Pour savoir si les MTD définissent correctement la classe **P** :

- on doit être sûr que tout problème polynomial pour RAM...
- peut aussi se résoudre en temps polynomial sur MTD
  (vous allez programmer la réciproque ou presque en TP...)

La classe de complexité **P** au-delà des MTD

Modèle RAM (à la Von Neumann) : une architecture d'ordinateur qui

- permet d'écrire des programmes et de les exécuter
- constitue une architecture réaliste, et surtout...
- définit un modèle de calcul !

Mais qu'en est-il du temps ?

Hypothèse temporelle pour les RAM :

"toutes les opérations (adressage compris) s'exécutent en temps constant"

Pour savoir si les MTD définissent correctement la classe **P** :

- on doit être sûr que tout problème polynomial pour RAM...
- peut aussi se résoudre en temps polynomial sur MTD
  (vous allez programmer la réciproque ou presque en TP...)

Sinon il y aurait des problèmes polynomiaux sur RAM et pas sur MTD.

Et la classe P serait incomplètement définie avec les MTD !

La classe de complexité **P** au-delà des MTD

Pour être sûr que la classe P est correctement définie avec les MTD

on doit montrer que pour tout problème de décision π

Théorème

Il existe un programme pour MTD qui résout π en temps polynomial

si et seulement si

il existe un programme pour RAM qui résout π en temps polynomial

La classe de complexité **P** au-delà des MTD

Pour être sûr que la classe P est correctement définie avec les MTD

on doit montrer que pour tout problème de décision π

Théorème

Il existe un programme pour MTD qui résout π en temps polynomial

si et seulement si

il existe un programme pour RAM qui résout π en temps polynomial

Si on montre cela, on aura aussi démontré que c'est valable pour ce qui est polynomial en C, en Java, en Python, etc.

La classe de complexité **P** au-delà des MTD

Pour être sûr que la classe P est correctement définie avec les MTD

on doit montrer que pour tout problème de décision π

Théorème

Il existe un programme pour MTD qui résout π en temps polynomial

si et seulement si

il existe un programme pour RAM qui résout π en temps polynomial

Si on montre cela, on aura aussi démontré que c'est valable pour ce qui est polynomial en C, en Java, en Python, etc.

Mais comment démontrer cela ?

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

$\Rightarrow$ Comme dans le TP 1 (exercice 3) : par simulation de machine !

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

$\Rightarrow$ Comme dans le TP 1 (exercice 3) : par simulation de machine !

Il a été démontré que les rapports de temps entre :

- l'exécution d'un programme en $O(T(n))$ sur une machine B

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

$\Rightarrow$ Comme dans le TP 1 (exercice 3) : par simulation de machine !

Il a été démontré que les rapports de temps entre :

- l'exécution d'un programme en $O(T(n))$ sur une machine B
- et le temps requis par une machine A pour simuler la machine B

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

$\Rightarrow$ Comme dans le TP 1 (exercice 3) : par simulation de machine !

Il a été démontré que les rapports de temps entre :

- l'exécution d'un programme en $O(T(n))$ sur une machine B
- et le temps requis par une machine A pour simuler la machine B

est possible dans les temps exprimés à l'intersection des lignes et des colonnes :

	Machines A	**Machines A**	**Machines A**
Machines B	**MTD**	**MTD k rubans**	**RAM**
MTD	$O(T(n))$	$O(T(n))$	$O(T(n).log(T(n)))$
MTD k rubans	$O(T(n)^2)$	$O(T(n))$	$O(T(n).log(T(n)))$
RAM	$O(T(n)^3)$	$O(T(n)^2)$	$O(T(n))$

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

$\Rightarrow$ Comme dans le TP 1 (exercice 3) : par simulation de machine !

Il a été démontré que les rapports de temps entre :

- l'exécution d'un programme en $O(T(n))$ sur une machine B
- et le temps requis par une machine A pour simuler la machine B

est possible dans les temps exprimés à l'intersection des lignes et des colonnes :

	Machines A	**Machines A**	**Machines A**
Machines B	**MTD**	**MTD k rubans**	**RAM**
MTD	$O(T(n))$	$O(T(n))$	$O(T(n).log(T(n)))$
MTD k rubans	$O(T(n)^2)$	$O(T(n))$	$O(T(n).log(T(n)))$
RAM	$O(T(n)^3)$	$O(T(n)^2)$	$O(T(n))$

Deux exemples :

1. Une machine A qui est une RAM peut simuler en $O(T(n).log(T(n)))$ une machine B qui est une MTD et qui s'exécute en $O(T(n))$
2. Une machine A qui est une MTD peut simuler en $O(T(n)^3)$ une machine B qui est une RAM et qui s'exécute en $O(T(n))$

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

⇒ Comme dans le TP 1 (exercice 3) : par simulation de machine !

Il a été démontré que les rapports de temps entre :

- l'exécution d'un programme en $O(T(n))$ sur une machine B
- et le temps requis par une machine A pour simuler la machine B

est possible dans les temps exprimés à l'intersection des lignes et des colonnes :

	Machines A	**Machines A**	**Machines A**
Machines B	**MTD**	**MTD k rubans**	**RAM**
MTD	$O(T(n))$	$O(T(n))$	$O(T(n).log(T(n)))$
MTD k rubans	$O(T(n)^2)$	$O(T(n))$	$O(T(n).log(T(n)))$
RAM	$O(T(n)^3)$	$O(T(n)^2)$	$O(T(n))$

Deux exemples :

1. Une machine A qui est une RAM peut simuler en $O(T(n).log(T(n)))$ une machine B qui est une MTD et qui s'exécute en $O(T(n))$
2. Une machine A qui est une MTD peut simuler en $O(T(n)^3)$ une machine B qui est une RAM et qui s'exécute en $O(T(n))$

La classe de complexité **P** au-delà des MTD

Comment démontrer cela ?

$\Rightarrow$ Comme dans le TP 1 (exercice 3) : par simulation de machine !

Il a été démontré que les rapports de temps entre :

- l'exécution d'un programme en $O(T(n))$ sur une machine B
- et le temps requis par une machine A pour simuler la machine B

est possible dans les temps exprimés à l'intersection des lignes et des colonnes :

	Machines A	**Machines A**	**Machines A**
Machines B	**MTD**	**MTD k rubans**	**RAM**
MTD	$O(T(n))$	$O(T(n))$	$O(T(n).log(T(n)))$
MTD k rubans	$O(T(n)^2)$	$O(T(n))$	$O(T(n).log(T(n)))$
RAM	$O(T(n)^3)$	$O(T(n)^2)$	$O(T(n))$

Deux exemples :

1. Une machine A qui est une RAM peut simuler en $O(T(n).log(T(n)))$ une machine B qui est une MTD et qui s'exécute en $O(T(n))$, et surtout
2. **Une machine A qui est une MTD peut simuler en $O(T(n)^3)$ une machine B qui est une RAM et qui s'exécute en $O(T(n))$**

La classe de complexité **P** au-delà des MTD

Puisqu'il est établi que :

1. Une machine A qui est une RAM peut simuler en $O(T(n).log(T(n)))$ une machine B qui est une MTD et qui s'exécute en $O(T(n))$, et surtout
2. **Une machine A qui est une MTD peut simuler en $O(T(n)^3)$ une machine B qui est une RAM et qui s'exécute en $O(T(n))$**

on a bien pour tout problème de décision π :

Théorème

Il existe un programme pour MTD qui résout π en temps polynomial
si et seulement si
il existe un programme pour RAM qui résout π en temps polynomial

La classe de complexité **P** au-delà des MTD

Puisqu'il est établi que :

1. Une machine A qui est une RAM peut simuler en $O(T(n).log(T(n)))$ une machine B qui est une MTD et qui s'exécute en $O(T(n))$, et surtout
2. **Une machine A qui est une MTD peut simuler en $O(T(n)^3)$ une machine B qui est une RAM et qui s'exécute en $O(T(n))$**

on a bien pour tout problème de décision π :

Théorème

Il existe un programme pour MTD qui résout π en temps polynomial
si et seulement si
il existe un programme pour RAM qui résout π en temps polynomial

Donc si un programme en langage C est polynomial en $O(p(n))$, il restera polynomial sur une MTD, au pire en $O(p(n)^3)$ et donc :

La classe de complexité **P** au-delà des MTD

Puisqu'il est établi que :

1. Une machine A qui est une RAM peut simuler en $O(T(n).log(T(n)))$ une machine B qui est une MTD et qui s'exécute en $O(T(n))$, et surtout
2. **Une machine A qui est une MTD peut simuler en $O(T(n)^3)$ une machine B qui est une RAM et qui s'exécute en $O(T(n))$**

on a bien pour tout problème de décision π :

Théorème

Il existe un programme pour MTD qui résout π en temps polynomial
si et seulement si
il existe un programme pour RAM qui résout π en temps polynomial

Donc si un programme en langage C est polynomial en $O(p(n))$, il restera polynomial sur une MTD, au pire en $O(p(n)^3)$ et donc :

La classe P est correctement définie avec les MTD

La classe de complexité **P** : le point

La classe P est correctement définie avec les MTD

et donc

La classe P contient l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par un ordinateur (sous l'hypothèse d'une mémoire de taille non bornée)

La classe de complexité **P** : le point

La classe P est correctement définie avec les MTD

et donc

La classe P contient l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par un ordinateur (sous l'hypothèse d'une mémoire de taille non bornée)

Deux remarques :

1. Généralement, la classe **P** est assimilée à la classe des problèmes "faciles puisqu'ils sont tous de complexité polynomiale... mais attention :

La classe de complexité **P** : le point

La classe P est correctement définie avec les MTD

et donc

La classe P contient l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par un ordinateur (sous l'hypothèse d'une mémoire de taille non bornée)

Deux remarques :

1. Généralement, la classe **P** est assimilée à la classe des problèmes "faciles puisqu'ils sont tous de complexité polynomiale... mais attention :
   "Est-ce qu'un .problème de décision de complexité $\Theta(n^{100})$ *est facile ?*

La classe de complexité **P** : le point

La classe P est correctement définie avec les MTD

et donc

La classe P contient l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par un ordinateur (sous l'hypothèse d'une mémoire de taille non bornée)

Deux remarques :

1. Généralement, la classe **P** est assimilée à la classe des problèmes "faciles puisqu'ils sont tous de complexité polynomiale... mais attention :
   "Est-ce qu'un .problème de décision de complexité $\Theta(n^{100})$ *est facile ?*
2. À ce jour :
   - on ne connaît que peu de problèmes "réels" pour lesquels on a démontré qu'ils ne sont pas dans la classe **P**

La classe de complexité **P** : le point

La classe P est correctement définie avec les MTD

et donc

La classe P contient l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par un ordinateur (sous l'hypothèse d'une mémoire de taille non bornée)

Deux remarques :

1. Généralement, la classe **P** est assimilée à la classe des problèmes "faciles puisqu'ils sont tous de complexité polynomiale... mais attention :

   "Est-ce qu'un .problème de décision de complexité $\Theta(n^{100})$ *est facile ?*

2. À ce jour :
   - on ne connaît que peu de problèmes "réels" pour lesquels on a démontré qu'ils ne sont pas dans la classe **P**
   - mais on connaît beaucoup de problèmes "réels" pour lesquels on ne connaît pas d'algorithme polynomial (on n'a pas montré qu'ils $\in$ **P**)

Plan

Problèmes complémentaires

Notion de problème complémentaire

D'abord à partir d'un exemple :

CLIQUE
Donnée : Un graphe non-orienté $G = (S, A)$ et un entier k
Question : G possède-t-il une clique de taille k ou plus ?
(une clique de G est un sous-graphe complet de G)

Problèmes complémentaires

Notion de problème complémentaire

D'abord à partir d'un exemple :

CLIQUE
Donnée : Un graphe non-orienté $G = (S, A)$ et un entier k
Question : G possède-t-il une clique de taille k ou plus ?
(une clique de G est un sous-graphe complet de G)

Le problème complémentaire de CLIQUE : **co-CLIQUE**

co-CLIQUE
Donnée : Un graphe non-orienté $G = (S, A)$ et un entier k
Question : Est-ce que G ne possède aucune clique de taille k ou plus ?

Problèmes complémentaires

Notion de problème complémentaire

D'abord à partir d'un exemple :

CLIQUE
Donnée : Un graphe non-orienté $G = (S, A)$ et un entier k
Question : G possède-t-il une clique de taille k ou plus ?
(une clique de G est un sous-graphe complet de G)

Le problème complémentaire de CLIQUE : **co-CLIQUE**

co-CLIQUE
Donnée : Un graphe non-orienté $G = (S, A)$ et un entier k
Question : Est-ce que G ne possède aucune clique de taille k ou plus ?

Définition dont l'intérêt semble poser question...

... car il semble que résoudre **CLIQUE** revient à résoudre **co-CLIQUE**

Problèmes complémentaires

Notion de problème complémentaire

D'abord à partir d'un exemple :

CLIQUE
Donnée : Un graphe non-orienté $G = (S, A)$ et un entier k
Question : G possède-t-il une clique de taille k ou plus ?
(une clique de G est un sous-graphe complet de G)

Le problème complémentaire de CLIQUE : **co-CLIQUE**

co-CLIQUE
Donnée : Un graphe non-orienté $G = (S, A)$ et un entier k
Question : Est-ce que G ne possède aucune clique de taille k ou plus ?

Définition dont l'intérêt semble poser question...

... car il semble que résoudre **CLIQUE** revient à résoudre **co-CLIQUE**

Mais la communauté scientifique n'a pas (encore ?) démontré que ces problèmes ont la même complexité !!!

Problèmes complémentaires

Plus généralement :

Définition

Étant donné un problème de décision π, on appelle **problème de décision complémentaire** de π, le problème de décision noté co-π tel que :

- $D_{co-\pi} = D_\pi$ est l'ensemble de ses instances
- $V_{co-\pi} = D_\pi \backslash V_\pi$ est l'ensemble de ses instances positives

Problèmes complémentaires

Plus généralement :

Définition

Étant donné un problème de décision π, on appelle **problème de décision complémentaire** de π, le problème de décision noté co-π tel que :

- $D_{co-\pi} = D_\pi$ est l'ensemble de ses instances
- $V_{co-\pi} = D_\pi \backslash V_\pi$ est l'ensemble de ses instances positives

En clair :

- Question posée par un problème de décision π :

  Étant donnée $I \in D_\pi$, est-ce que $I \in V_\pi$, i.e. I est-elle une instance positive de π ?

Problèmes complémentaires

Plus généralement :

Définition

Étant donné un problème de décision π, on appelle **problème de décision complémentaire** de π, le problème de décision noté co-π tel que :

- $D_{co-\pi} = D_\pi$ est l'ensemble de ses instances
- $V_{co-\pi} = D_\pi \backslash V_\pi$ est l'ensemble de ses instances positives

En clair :

- Question posée par un problème de décision π :

  Étant donnée $I \in D_\pi$, est-ce que $I \in V_\pi$, i.e. I est-elle une instance positive de π ?

- La question posée par son problème complémentaire co-π est inversée

  Étant donnée $I \in D_{co-\pi}$, est-ce que $I \in D_\pi \backslash V_\pi$, i.e. I est-elle une instance négative de π ?

Classes de problèmes complémentaires

On étend cette notion aux classes de problèmes :

Définition

Si C est une classe (ensemble) de problèmes de décision, alors co-C est sa classe complémentaire, i.e. l'ensemble de ses problèmes de décision complémentaires : $\text{co-}C = \{\text{co-}\pi : \pi \in C\}$

Classes de problèmes complémentaires

On étend cette notion aux classes de problèmes :

Définition

Si C est une classe (ensemble) de problèmes de décision, alors co-C est sa classe complémentaire, i.e. l'ensemble de ses problèmes de décision complémentaires : $\text{co-}C = \{\text{co-}\pi : \pi \in C\}$

Intuitivement :

- Si les problèmes d'une classe de complexité C sont faciles, on peut penser que ceux de sa classe de complexité complémentaire co-C le sont aussi

Classes de problèmes complémentaires

On étend cette notion aux classes de problèmes :

Définition

Si C est une classe (ensemble) de problèmes de décision, alors co-C est sa classe complémentaire, i.e. l'ensemble de ses problèmes de décision complémentaires : $\text{co-}C = \{\text{co-}\pi : \pi \in C\}$

Intuitivement :

- Si les problèmes d'une classe de complexité C sont faciles, on peut penser que ceux de sa classe de complexité complémentaire co-C le sont aussi
- Mais si les problèmes d'une classe de complexité C sont difficiles, qu'en est-il de ceux de sa classe complémentaire co-C ?

Classes de problèmes complémentaires

On étend cette notion aux classes de problèmes :

Définition

Si C est une classe (ensemble) de problèmes de décision, alors co-C est sa classe complémentaire, i.e. l'ensemble de ses problèmes de décision complémentaires : $\text{co-}C = \{\text{co-}\pi : \pi \in C\}$

Intuitivement :

- Si les problèmes d'une classe de complexité C sont faciles, on peut penser que ceux de sa classe de complexité complémentaire co-C le sont aussi
- Mais si les problèmes d'une classe de complexité C sont difficiles, qu'en est-il de ceux de sa classe complémentaire co-C ?
  - Pour un problème $\pi \in C$, il faut savoir pour une instance $I \in D_\pi$ s'il existe une solution (on prouve qu'il en existe une et le problème est résolu)

Classes de problèmes complémentaires

On étend cette notion aux classes de problèmes :

Définition

Si C est une classe (ensemble) de problèmes de décision, alors co-C est sa classe complémentaire, i.e. l'ensemble de ses problèmes de décision complémentaires : $\text{co-}C = \{\text{co-}\pi : \pi \in C\}$

Intuitivement :

- Si les problèmes d'une classe de complexité C sont faciles, on peut penser que ceux de sa classe de complexité complémentaire co-C le sont aussi
- Mais si les problèmes d'une classe de complexité C sont difficiles, qu'en est-il de ceux de sa classe complémentaire co-C ?
  - Pour un problème $\pi \in C$, il faut savoir pour une instance $I \in D_\pi$ s'il existe une solution (on prouve qu'il en existe une et le problème est résolu)
  - Mais pour son problème complémentaire co-π, il faut montrer qu'il n'existe aucune solution... et peut-être explorer tout son espace de recherche.

Classes de problèmes complémentaires

On étend cette notion aux classes de problèmes :

Définition

Si C est une classe (ensemble) de problèmes de décision, alors co-C est sa classe complémentaire, i.e. l'ensemble de ses problèmes de décision complémentaires : $\text{co-}C = \{\text{co-}\pi : \pi \in C\}$

Intuitivement :

- Si les problèmes d'une classe de complexité C sont faciles, on peut penser que ceux de sa classe de complexité complémentaire co-C le sont aussi
- Mais si les problèmes d'une classe de complexité C sont difficiles, qu'en est-il de ceux de sa classe complémentaire co-C ?
  - Pour un problème $\pi \in C$, il faut savoir pour une instance $I \in D_\pi$ s'il existe une solution (on prouve qu'il en existe une et le problème est résolu)
  - Mais pour son problème complémentaire co-π, il faut montrer qu'il n'existe aucune solution... et peut-être explorer tout son espace de recherche.

Il peut exister une véritable dissymétrie entre π et co-π, donc entre C et co-C

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Preuve. Il suffit de montrer que : $\forall\ \Pi \in \mathbf{P}$ alors co-$\Pi \in \mathbf{P}$.

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Preuve. Il suffit de montrer que : $\forall\ \Pi \in \mathbf{P}$ alors co-$\Pi \in \mathbf{P}$.

Soit $\Pi \in \mathbf{P}$

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Preuve. Il suffit de montrer que : $\forall\ \Pi \in \mathbf{P}$ alors co-$\Pi \in \mathbf{P}$.

Soit $\Pi \in \mathbf{P} \Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Preuve. Il suffit de montrer que : $\forall\ \Pi \in \mathbf{P}$ alors co-$\Pi \in \mathbf{P}$.

Soit $\Pi \in \mathbf{P} \Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

$\Rightarrow$ M s'arrêtera pour toute mot m en entrée codant une instance de Π :

- sur q_{oui} si $m \in L(\pi, S)$ (m code une instance positive de Π)
- sur q_{non} si $m \notin L(\pi, S)$ (m code une instance négative de Π)

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

P = co-**P**

Preuve. Il suffit de montrer que : $\forall\ \Pi \in$ **P** alors co-$\Pi \in$ **P**.

Soit $\Pi \in$ **P** $\Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

$\Rightarrow M$ s'arrêtera pour toute mot m en entrée codant une instance de Π :

- sur q_{oui} si $m \in L(\pi, S)$ (m code une instance positive de Π)
- sur q_{non} si $m \notin L(\pi, S)$ (m code une instance négative de Π)

Soit le programme M' identique à M sauf pour les états terminaux q_{oui} et q_{non} qui sont permutés : q_{non} devient q_{oui} et q_{oui} devient q_{non}.

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Preuve. Il suffit de montrer que : $\forall\ \Pi \in \mathbf{P}$ alors co-$\Pi \in \mathbf{P}$.

Soit $\Pi \in \mathbf{P} \Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

$\Rightarrow$ M s'arrêtera pour toute mot m en entrée codant une instance de Π :

- sur q_{oui} si $m \in L(\pi, S)$ (m code une instance positive de Π)
- sur q_{non} si $m \notin L(\pi, S)$ (m code une instance négative de Π)

Soit le programme M' identique à M sauf pour les états terminaux q_{oui} et q_{non} qui sont permutés : q_{non} devient q_{oui} et q_{oui} devient q_{non}.

M' est un programme polynomial pour MTD qui résout co-Π, car M' s'arrête toujours :

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

P = co-**P**

Preuve. Il suffit de montrer que : $\forall\ \Pi \in$ **P** alors co-$\Pi \in$ **P**.

Soit $\Pi \in$ **P** $\Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

$\Rightarrow M$ s'arrêtera pour toute mot m en entrée codant une instance de Π :

- sur q_{oui} si $m \in L(\pi, S)$ (m code une instance positive de Π)
- sur q_{non} si $m \notin L(\pi, S)$ (m code une instance négative de Π)

Soit le programme M' identique à M sauf pour les états terminaux q_{oui} et q_{non} qui sont permutés : q_{non} devient q_{oui} et q_{oui} devient q_{non}.

M' est un programme polynomial pour MTD qui résout co-Π, car M' s'arrête toujours :

- sur q_{non} avec m en entrée si $m \in L(\pi, S)$ (m code une instance négative de co-Π)

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

P = co-**P**

Preuve. Il suffit de montrer que : $\forall\ \Pi \in$ **P** alors co-$\Pi \in$ **P**.

Soit $\Pi \in$ **P** $\Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

$\Rightarrow M$ s'arrêtera pour toute mot m en entrée codant une instance de Π :

- sur q_{oui} si $m \in L(\pi, S)$ (m code une instance positive de Π)
- sur q_{non} si $m \notin L(\pi, S)$ (m code une instance négative de Π)

Soit le programme M' identique à M sauf pour les états terminaux q_{oui} et q_{non} qui sont permutés : q_{non} devient q_{oui} et q_{oui} devient q_{non}.

M' est un programme polynomial pour MTD qui résout co-Π, car M' s'arrête toujours :

- sur q_{non} avec m en entrée si $m \in L(\pi, S)$ (m code une instance négative de co-Π)
- sur q_{oui} avec m en entrée si $m \notin L(\pi, S)$ (m code une instance positive de co-Π)

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Preuve. Il suffit de montrer que : $\forall\ \Pi \in \mathbf{P}$ alors co-$\Pi \in \mathbf{P}$.

Soit $\Pi \in \mathbf{P} \Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

$\Rightarrow$ M s'arrêtera pour toute mot m en entrée codant une instance de Π :

- sur q_{oui} si $m \in L(\pi, S)$ (m code une instance positive de Π)
- sur q_{non} si $m \notin L(\pi, S)$ (m code une instance négative de Π)

Soit le programme M' identique à M sauf pour les états terminaux q_{oui} et q_{non} qui sont permutés : q_{non} devient q_{oui} et q_{oui} devient q_{non}.

M' est un programme polynomial pour MTD qui résout co-Π, car M' s'arrête toujours :

- sur q_{non} avec m en entrée si $m \in L(\pi, S)$ (m code une instance négative de co-Π)
- sur q_{oui} avec m en entrée si $m \notin L(\pi, S)$ (m code une instance positive de co-Π)

Donc on a bien co-$\Pi \in \mathbf{P}$.

Classes de problèmes complémentaires

Le cas particulier de la classes de complexité **P** :

Théorème

$\mathbf{P} = \text{co-}\mathbf{P}$

Preuve. Il suffit de montrer que : $\forall\ \Pi \in \mathbf{P}$ alors co-$\Pi \in \mathbf{P}$.

Soit $\Pi \in \mathbf{P} \Leftrightarrow \exists\ M$ programme polynomial pour MTD qui résout Π

$\Rightarrow M$ s'arrêtera pour toute mot m en entrée codant une instance de Π :

- sur q_{oui} si $m \in L(\pi, S)$ (m code une instance positive de Π)
- sur q_{non} si $m \notin L(\pi, S)$ (m code une instance négative de Π)

Soit le programme M' identique à M sauf pour les états terminaux q_{oui} et q_{non} qui sont permutés : q_{non} devient q_{oui} et q_{oui} devient q_{non}.

M' est un programme polynomial pour MTD qui résout co-Π, car M' s'arrête toujours :

- sur q_{non} avec m en entrée si $m \in L(\pi, S)$ (m code une instance négative de co-Π)
- sur q_{oui} avec m en entrée si $m \notin L(\pi, S)$ (m code une instance positive de co-Π)

Donc on a bien co-$\Pi \in \mathbf{P}$.

Remarque : Dans le cas général, rien ne garantit que $C = \text{co-}C$.

En guise de conclusion, ce que l'on a vu :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe

En guise de conclusion, ce que l'on a vu :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe
- **Le modèle de calcul déterministe ?**
  - MTD pour définir rigoureusement la classe
    (cf. taille de la donnée en entrée et temps de calcul)

En guise de conclusion, ce que l'on a vu :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe
- **Le modèle de calcul déterministe ?**
  - MTD pour définir rigoureusement la classe
    (cf. taille de la donnée en entrée et temps de calcul)
  - Extension à tous les modèles de calcul déterministes "réalistes" :
    Les "ordinateurs" donc !

En guise de conclusion, ce que l'on a vu :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe
- **Le modèle de calcul déterministe ?**
  - MTD pour définir rigoureusement la classe
    (cf. taille de la donnée en entrée et temps de calcul)
  - Extension à tous les modèles de calcul déterministes "réalistes " :
    Les "ordinateurs" donc !
- **Classer un problèmes de décision π dans P ?**

  C'est montrer que $\pi \in$ **P** en montrant qu'il existe un algorithme sur modèle de calcul déterministe qui résout π en temps polynomial.

En guise de conclusion, ce que l'on a vu :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe
- **Le modèle de calcul déterministe ?**
  - MTD pour définir rigoureusement la classe
    (cf. taille de la donnée en entrée et temps de calcul)
  - Extension à tous les modèles de calcul déterministes "réalistes" :
    Les "ordinateurs" donc !
- **Classer un problèmes de décision π dans P ?**

  C'est montrer que $\pi \in \mathbf{P}$ en montrant qu'il existe un algorithme sur modèle de calcul déterministe qui résout π en temps polynomial.
- **Et après ?**

En guise de conclusion, ce que l'on a vu :

- **Définition de la classe de complexité P :**
  - C'est un ensemble de problèmes de décision
  - Tous de complexité polynomiale
  - Sur modèle de calcul déterministe
- **Le modèle de calcul déterministe ?**
  - MTD pour définir rigoureusement la classe
    (cf. taille de la donnée en entrée et temps de calcul)
  - Extension à tous les modèles de calcul déterministes "réalistes" :
    Les "ordinateurs" donc !
- **Classer un problèmes de décision π dans P ?**

  C'est montrer que $\pi \in$ **P** en montrant qu'il existe un algorithme sur modèle de calcul déterministe qui résout π en temps polynomial.
- **Et après ? La classe NP !**