Introduction

Informatique théorique

= approche abstraite sur ce qui peut être calculé avec un ordinateur.

Probablement le cours le plus abstrait de toute la formation.

⇒ l'un des plus durs.

Abstrait = approche théorique

cours de mathématiques traitant des propriétés de modèles représentant des traitements informatiques.

⇒ n'aide pas à écrire un programme
⇒ n'aide pas à concevoir un ordinateur
ou à comprendre comment il fonctionne.

Mais alors,à quoi sert ce cours ?

Un des cours les plus fondamentaux pour un informaticien :

- comprendre pourquoi l'informatique est une science.
- se poser des questions fondamentales
  - sur ce que l'on fait.
  - sur ce que l'on peut faire (et ne pas faire).

Exemples de questions auxquelles on va répondre :

- Que signifie "faire un traitement informatique" ?
- Quel genre de choses peut-on calculer ?
- A quelle vitesse peut-on le faire ?
- Combien de mémoire cela prend-il ?

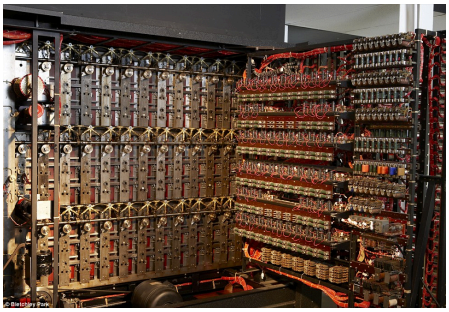
1 Historique

Un peu d'histoire :

Quand les mathématiciens se sont-ils intéressés à ces problèmes ?

-200	**Euclide**	algorithme de calcul du PGCD.
	Archimède	méthode d'exhaustion pour le calcul de π.
	Eratosthène	crible pour rechercher des nombres premiers.
800	**Al-Khwarizmi**	algorithmes utilisant le système de numération Hindou.
1850	**Boole**	algèbre binaire (unification symbolique de la logique et du calcul).
1880	**Fredge**	premier langage formel (ensemble de symboles + règles de manipulation).
1890	**Peano**	axiomatisation des mathématiques comme un langage symbolique.
1910	**Whitehead** **Russell**	poursuite des travaux de Fredge, fondation de la logique moderne.
1931	**Goëdel**	théorème d'incomplétude.
1936	**Church**	lambda calcul (fonctions récursives), existence de problème indécidable.
	Turing	machine de Turing.
1945	**Kleene**	thèse de Church-Turing (base de la calculabilité).

On connaissait donc les limites des ordinateurs avant que les ordinateurs existent ?



1940 : ordinateur électro-magnétique (Bombe de Church)

implémentation physique d'une machine à états.
utilisé pour casser le code enigma allemand (cryptographie)
17576 combinaisons testées en quelques heures
152 de ces machines étaient en service à la fin de la guerre

2 Machines abstraites

Comment va-t-on faire pour répondre à ces questions ?

On va raisonner sur des machines abstraites.

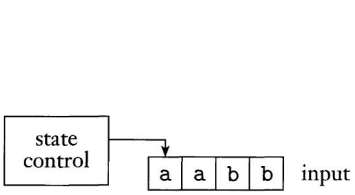
Qu'est-ce-qu'une machine abstraite ?

— modèle extrêmement simplifié (et très idéalisé) d'une machine.
— prend en entrée une chaîne de symboles.
— accepte ou rejette la chaîne d'entrée.
Pour certains modèles, capable de produire une sortie.

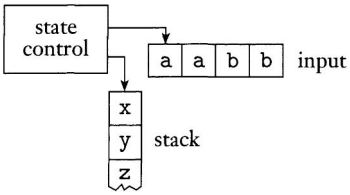
Quel est l'intérêt d'une telle machine ?

— machine pour laquelle on peut donner une définition mathématique simple et précise.
— permet des raisonnements rigoureux et la démonstration de propriétés fondamentales (théorèmes).

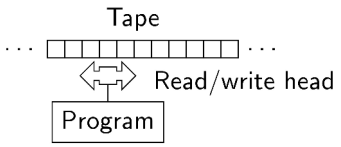
On considérera 3 types de machines abstraites :



automate fini



automate à pile



machine de Turing

Modèles abstraits : (par puissance d'abstraction)
- **automate fini** : machine à états.
- **automate à pile** : machine à états avec mémoire (pile FILO).
- **machine de Turing** : machine à états utilisant une bande en lecture/écriture.

Fonctionnement plus simple qu'une machine réelle pourtant avec des capacités de "calcul" similaires.

Que signifie calculer dans ce cours ?
= accepter ou refuser l'entrée de la machine

Un peu plus formellement,
- entrée w = suite finie de symboles (exemple : $\{0, 1\}$).
- l'entrée w est reconnue par la machine M si M accepte w, et rejette sinon.

Peut-on écrire une machine qui reconnaît :

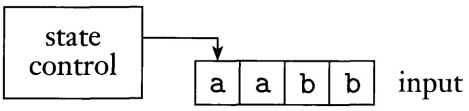
1. une chaîne de symboles binaires qui se termine par 0.
   oui, avec un automate fini.
2. une chaîne si elle représente un code en langage C légal.
   oui, avec un automate à pile.
3. une chaîne si elle représente un programme qui n'entre jamais dans une boucle infinie pour n'importe quelle entrée.
   non, il n'est pas possible d'écrire un tel programme.

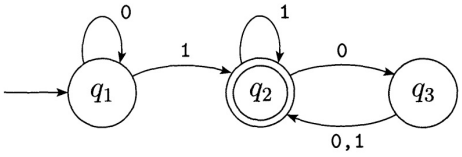
On va prouver ceci indépendamment du langage ou de la machine utilisée en utilisant des machines abstraites.

2.1 Automate fini

Qu'est ce qu'un automate fini ?

- le modèle le plus simple de machine abstraite.
- lié à la représentation théorique d'un contrôleur.





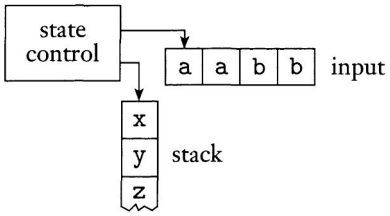
Applications :
- modélisation d'un circuit (porte automatique, ...).
- modélisation de protocole.

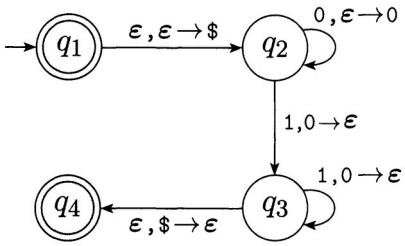
— vérification de modèle.
— utile pour le traitement de symboles.
— pour trouver des "motifs" dans des chaînes de symboles (expression régulière).

2.2 Automate à pile

Qu'est ce qu'un automate à pile ?

— automate fini avec mémoire de type pile FILO.
— étroitement lié aux langages libres du contexte.





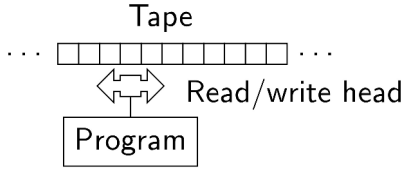
Applications :

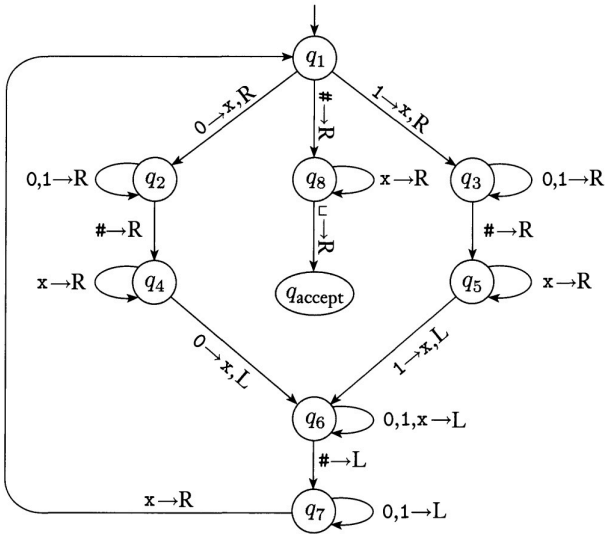
— rôle important dans les compilateurs.
— conception de langage de programmation (syntaxe).
— étude du langage naturel.
— écriture sous forme d'une grammaire (XML, ...)
— traduction automatique

2.3 Machine de Turing

Caractéristiques :

— la machine abstraire la plus puissante.
— même capacité de calcul qu'un ordinateur moderne.
et que tout futur ordinateur (une version quantique existe !).





Applications :

- aucune application pratique !
- tout type de machine peut être simulé avec cette machine.
- utilisée pour effectuer des démonstrations sur les propriétés des machines abstraites ou réelles.

3 Décidabilité et complexité

On définit :

- **problème infaisable** : problème qu'aucun programme n'est capable de résoudre rapidement.
- **problème indécidable** : problème qu'aucun programme n'est capable de résoudre (du tout).

Questions :

- y-a-t-il des problèmes infaisables ?
- y-a-t-il des problèmes indécidables ?

Pourquoi se poser ces questions ?

⇒ Pour des raisons pragmatiques :

- utiliser des algorithmes efficaces (lorsque c'est possible).
- être capable d'identifier la difficulté d'un problème.
- éviter les problèmes intraitables ou impossibles.

3.1 Décidabilité

Problème indécidable

En informatique théorique, ce problème est formulé ainsi :

- Tester la validité d'une expression (mathématique).
  i.e. si elle est vraie ou fausse.
- Cette vérification est-elle toujours possible ?

Résultats que nous allons voir :

— Certains problèmes fondamentaux ne peuvent pas être résolus par un ordinateur.
— Il existe des expressions pour lesquelles on ne pourra jamais déterminer si elles sont vraies ou fausses.

Ces résultats utilisent un modèle théorique d'ordinateur :

⇒ aucun ordinateur n'en sera jamais capable.

Exemples de problèmes indécidables :

— $10^{\text{ème}}$ problème de Hilbert : est-ce qu'une équation à plusieurs variables et des coefficients entiers a une solution entière ?
exemple : $5x + 15y = 12$
— quelle est la compression optimale d'une chaîne x de symboles ?

De très nombreux problèmes informatiques sont indécidables :

— **est-ce qu'un programme s'arrête ?**
à savoir, s'arrête-t-il toujours qu'importe son entrée.
impossible d'écrire un programme qui vérifie si un programme s'arrête dans tous les cas.
— **est-ce qu'un programme est correct ?**
impossible d'écrire un programme qui vérifie si un programme produit le résultat qui est attendu.
— **est-ce que deux programmes sont équivalents ?**
impossible d'écrire un programme qui vérifie si deux programmes font la même chose.
— **est-ce qu'un programme est optimal ?**
impossible d'écrire un programme qui vérifie qu'aucun autre programme n'est meilleur (au sens choisi) à celui qui est fourni.

Attention : dans tous les exemples précédents

Il est possible d'écrire des programmes (ou des démonstrations) qui vont donner des réponses :

— soit dans des cas particuliers,
i.e. pour une certaine classe de programme.
— soit des réponses partielles
i.e. qui vont marcher dans certains cas mais pas d'autres.

mais jamais **aucun** ne sera capable de le faire dans le cas général.

Conséquence importante :

— un algorithme **résout un problème** s'il le résout **dans tous les cas**.
⇒ doit être capable de marcher pour toute entrée.
— si un algorithme résout un problème partiellement (sur une partie du domaine), il faut vérifier que l'entrée appartient au domaine de résolution.

Exemple : un algorithme d'inversion de matrice fonctionne pour toute matrice dont le déterminant n'est pas nul.

3.2 Calculabilité

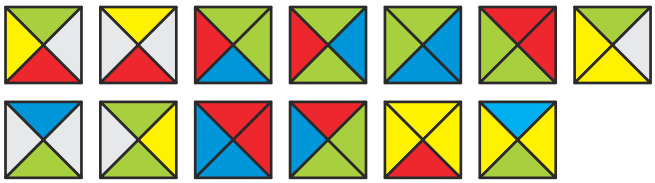
Pour la notion de décidabilité, on s'intéressait à résoudre des problèmes dont la réponse était binaire (accepter/rejeter).

Une machine de Turing permet également de définir une fonction f quelconque de la manière suivante :

— elle débute avec x écrit sur sa bande,
— elle s'arrête avec $f(x)$ écrit sur sa bande.

Une fonction peut être :
- totalement calculable (si elle est calculable pour tout x),
- partiellement calculable (si $f(x)$ n'est pas calculable, alors la machine boucle).

Question : existe-t-il des fonctions qui ne sont pas calculables ?
Soit l'ensemble des pavés suivants :



Objectif : utiliser ces pavés pour couvrir un plan infini (à savoir, trouver une méthode permettant de calculer le pavé de n'importe quel point d'un plan).

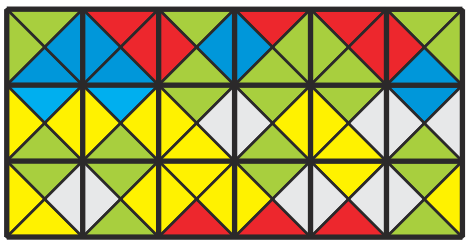
à savoir un arrangement qui puisse être répéter à l'infini sur le plan (donc avoir une méthode constructive).

exemple : pour faire un damier infini, prendre ce pavé  et le répéter à l'infini sur le plan.

Règle : deux pavés adjacents ont la même couleur sur leurs arêtes communes.

Idée : essayer de trouver un pavage périodique

Ce problème n'est pas calculable.
Exemple de pavage partiel :



Raison : on a pu démontrer (mathématiquement) qu'un tel pavage ne peut être qu'apériodique.

i.e. on ne peut pas créer un motif ou une combinaison de motifs permettant un pavage constructif du plan.

Conséquence : le pavage du plan (infini) ne s'arrête jamais.

l'algorithme ne s'arrête donc jamais.

Question : est-il possible de reconnaître un programme qui ne s'arrête jamais ?

ou susceptible de ne jamais s'arrêter ?

3.3 Complexité

Pourquoi étudier la complexité ?

Soit un problème (informatique) que l'on cherche à résoudre.

On cherche une méthode de résolution

une "recette", un algorithme, une méthode mathématique, ...

qui fonctionne **dans tous les cas**.

On souhaite évidemment une méthode efficace :
— la plus rapide possible,
— si possible économique en mémoire,
Efficace = dont la complexité est la plus faible possible

Comme pour la décidabilité, on peut montrer que certains problèmes sont infaisables :
— ils ont une complexité telle qu'ils ne peuvent être résolus en temps raisonnable.
— bien qu'il soit possible de trouver une solution lorsque la taille du problème est raisonnable.

Définition de la complexité d'un algorithme

Fonction $f(n)$, dépendant de la taille n du problème, proportionnelle à l'utilisation d'une ressource pendant l'exécution de l'algorithme sur ce problème.

On définit 2 types de complexités :
— **complexité temporelle** : ≃ temps de calcul.
proportionnelle au nombre d'opérations à effectuer.
— **complexité spatiale** : ≃ mémoire nécessaire.
proportionnelle au nombre de cases de la bande utilisées.
sur une machine de Turing.

On souhaite borner cette complexité :

donc donner la complexité maximale de l'algorithme.

borner $f(n)$ = trouver une fonction $g(n)$ telle que pour n assez grand, il existe une constante c telle que $|f(n)| \leq c.|g(n)|$.

Donc,
— **borner la complexité temporelle** = avoir une idée du temps nécessaire à un facteur multiplicatif près.
— **borner la complexité spatiale** = avoir une idée de la quantité de mémoire nécessaire à un facteur multiplicatif près.

à un facteur multiplicatif près =
résultat indépendant de la puissance ou de la quantité de mémoire dont dispose la machine.

Ceci permet la définition de **classes de complexité** :
— **complexité polynomiale** : bornée par $g(n) = n^k$ avec k constant.
— **complexité exponentielle** : non bornée par un polynôme.
donc, non polynomiale, par exemple : $n!$, $n^{\log n}$, 3^n, ...
où n est la taille du problème à résoudre.

Exemple 1: tri d'une liste

Problème : soit une liste de n entiers, les trier par ordre croissant.

Idée d'un algorithme naïf : itérer n fois : à la $i^{\text{ème}}$ itération, rechercher le plus petit élément dans les $n + 1 - i$ derniers éléments de la liste, et le permuter avec le $i^{\text{ème}}$ élément.

Compléxité : parcours de longueur n, puis $n - 1, \ldots, 1$; soit un total de $n.(n - 1)/2$.

La complexité de cet algorithme de tri est donc polynomiale (bornée par n^2).

On sait que pour des listes quelconques, le meilleur algorithme connu est en $n \log n$.

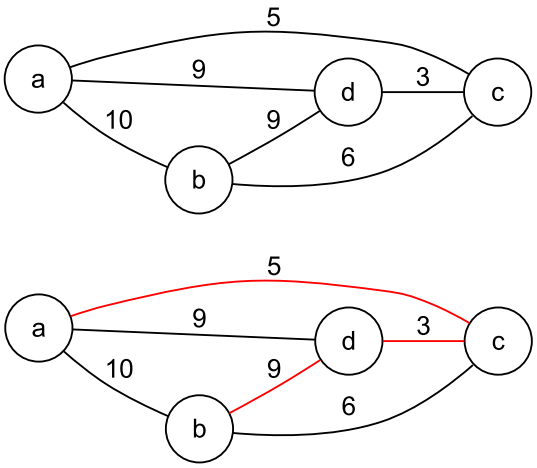
Exemple 2: problème du voyageur de commerce

Trouver le chemin le plus court :
- traversant toutes les villes (nœuds a, b, c, d).
- en restant sur les routes (arêtes avec distance).

On peut montrer que sa complexité temporelle est :

NON POLYNOMIALE

Sa borne est :
pour l'algorithme naïf = $n!$
pour l'algorithme optimal $\simeq 2^n$



Exemples de complexité :
- recherche dans une table de correspondance : 1
- recherche dans un arbre binaire équilibré : $\log n$
- recherche séquentielle : n
- tri : naïf = n^2, à bulle = $n \log n < n^2$
- voyageur de commerce : naïf = $n!$, borne probable = 2^n

Temps de calcul (taille du problème/complexité)

	10	20	30	40	50	60
n	0,00001"	0,00002"	0,00003"	0,00004"	0,00005"	0,00006"
n^2	0,0001"	0,0004"	0,0009"	0,0016"	0,0025"	0,0036"
n^3	0,001"	0,008"	0,027"	0,064"	0,125"	0,216"
n^5	0,1"	3,2"	24,3"	1,7'	5,2'	13,0'
2^n	0,001"	1,0"	17,9'	12,7 jours	35,7 ans	36,6 Kan
3^n	0,059"	58'	6,5 ans	385,5 Kan	22.7 Gan	1,3 Tan

Kan = 1000 ans = un millénaire.
Man = 1.000.000 ans = un million d'années.
Gan = 1.000.000.000 ans = un milliard d'années.
Tan = 1.000.000.000.000 ans = mille milliard d'années.

L'illusion de la puissance :

Attendez ! Donnez-moi un Cray Titan à 17.59 Pflops et je vous le fais.

Rappel : 1 PFlops = 1 petaflops = 1 million de Gflops.

Si la puissance de mon ordinateur est multipliée par ...

Pour un même temps de calcul, la taille n du problème peut augmenter de :

	10	10^2	10^3	10^6	10^9
n	$\times 10$	$\times 10^2$	$\times 10^3$	$\times 10^6$	$\times 10^9$
n^2	$\times 3,16$	$\times 10$	$\times 31,6$	$\times 1000$	$\times 31623$
n^3	$\times 2,15$	$\times 4,64$	$\times 10$	$\times 100$	$\times 1000$
n^5	$\times 1,58$	$\times 2,51$	$\times 3,98$	$\times 15,8$	$\times 63$
2^n	$+3,32$	$+6,64$	$+9,96$	$+19,9$	$+29,89$
3^n	$+2,09$	$+4,19$	$+6,29$	$+12,57$	$+18,89$

Pour les complexités :
- **polynomiales**, la taille du problème peut être multipliée.

— **exponentielles**, la taille du problème progresse de quelques unités.

Autrement dit :

— **si la complexité d'un problème est polynomiale** :
Le problème peut être résolu, y compris pour les gros problèmes.
Il suffit d'y passer suffisamment de temps.
— **si la complexité d'un problème est exponentielle** :
La solution pour des problèmes de petite taille peut être trouvée,
mais les problèmes de grande taille sont infaisables.

Conclusion :

— **Un problème est infaisable** si sa complexité est exponentielle.
— Bien qu'il soit possible d'écrire un algorithme qui le résolve lorsque le problème est de petite taille.

Conséquences

Ce n'est pas parce qu'il est possible d'écrire un algorithme pour résoudre un problème sur un ordinateur que le problème peut être résolu.

Le problème peut :

— **être indécidable** : on ne sait pas si une solution sera trouvée, et s'il en existe, quand elle sera trouvée.
⇒ problème de l'arrêt d'un algorithme.
— **avoir complexité non polynomiale** : intraitable sinon pour des tailles de problèmes très petites.
⇒ problème de la complexité algorithmique.

Objectif de l'informatique théorique :

être en mesure de distinguer les différentes typologies des problèmes et connaître les limites d'un ordinateur.

4 Résumé

Résumé du programme du cours :

— modèle abstrait de la machine de Turing
  — que peut-on reconnaitre ?
  — quelles sont ses limites ?
  — qu'apportent des modèles de machines abstraites plus complexes ?
  — modèle de calcul informatique
— décidabilité
— calculabilité
— complexité temporelle
— complexité spatiale (pas le temps en général)