LINFO1123 :
Calculabilité, logique et complexité
Synthèse

Professeur : Yves Deville

Année 2022-2023

Chapitre 2 : Concepts

Ensemble = collection d'objets, sans répétition, appelés les *éléments* de l'ensemble.

Produit Cartésien = Paire de 2 éléments ; où chaque élément de la paire appartient à un ensemble (ex : A x B, premier élément de la paire appartient à l'ensemble A, deuxième appartient à l'ensemble B)

Sous-Ensemble = énumération de tous les sous-ensembles présent dans un ensemble, se note $2^{\text{nom ensemble}}$. (ex : ensemble A = {0,1}, son sous-ensemble est 2^A = {Ø, {0}, {1}, {0,1}})

Complément = énumération de tout les éléments qui ne se trouvent pas l'ensemble de base, se note $\bar{A}$

Chaîne de caractères = séquence finie de symboles juxtaposés. (ex : A = abccbababccc)

- ⇨ Ɛ = chaîne de caractère vide

Alphabet = ensemble de symboles qui compose une chaîne de caractères, se note Σ. (ex pour A : Σ = {a,b,c})

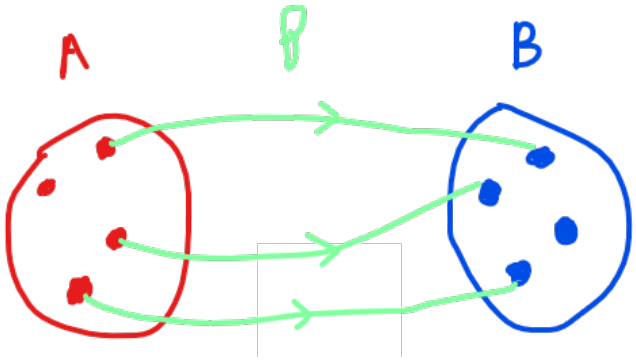
Langage = ensemble de mots constitués de symboles d'un alphabet donné

Si l'on souhaite énumérer toutes les chaînes de caractères possible d'un alphabet Σ, on peut écrire cela par Σ^* (ex : Σ = {0,1}, Σ^* = { Ɛ,0,1,00,01,10,11,000,001,010,100,…}) -> ensemble infini

Une **Relation** est définie sur 2 ensembles, et représente un couple du produit cartésien de 2 éléments de ces 2 ensembles, se note <a,b>

Les fonctions sont un type de relation.

Une **Fonction** *f* de A dans B est une relation telle que pour a ∈ A, il existe au plus un seul b ∈ tel que <a,b> ∈ *f*. Cela représente par la photo ci-dessous pour les 2 ensembles A et B.



Remarque :

- ➔ Une et une seule flèche maximum part d'un élément de A
- ➔ Certains éléments de A n'ont pas de flèche vers B
  Si c'est le cas, on dit que la fonction f(X) est indéfinie

Donc, si a ∈ A et il n'existe pas de b ∈ B tel que f(a) = b, alors

f(a) est *indéfini* et on note f(a) = ⊥ (*bottom*).

Domaine de f = éléments de l'ensemble duquel partent les flèches

Image de f = éléments de l'ensemble qui sont la cible d'une flèche

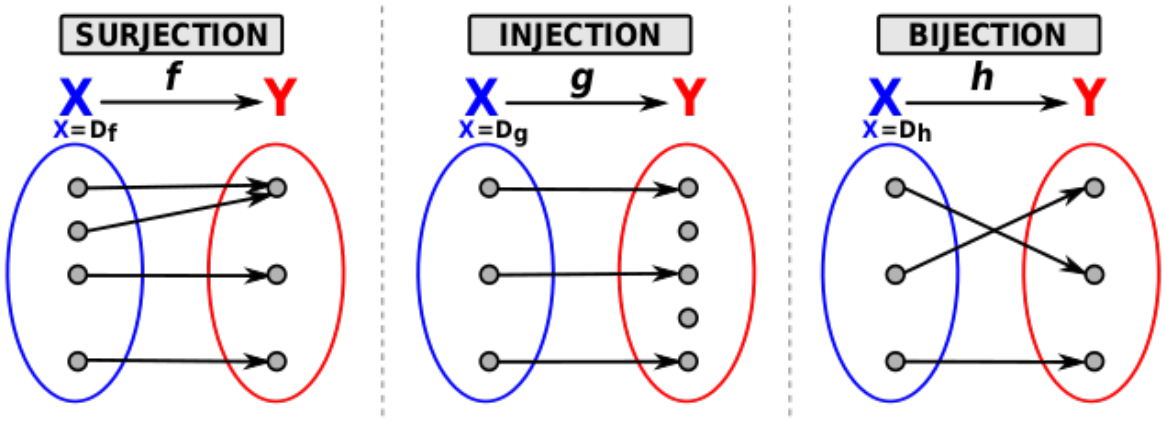
f est une **fonction totale** si le domaine de la fonction est l'ensemble A lui-même (fonction définie pour tout les éléments de A, en gros tout les points de A ont une flèche qui part de leur point).

f est une **fonction partielle** si elle n'est pas totale (fonction indéfinie pour au moins un élément). Une fonction totale est aussi une fonction partielle.

f est **injective** lorsque chaque élément du domaine est lié à un élément de l'image.

f est **surjective** lorsque chaque élément du domaine relie AU MOINS un élément de l'image. Deux éléments du domaine peuvent pointer vers le même élément de l'image.

f est **bijective** lorsque chaque élément du domaine relie chaque élément de l'image. Aucun élément du domaine ou de l'image n'est pas relié.



On parle également **d'extension de fonctions** dans les propriétés de fonctions, qui consiste à rajouter des éléments à une fonction, mais uniquement là où la fonction n'est pas définie. La fonction de base a la même valeur que la nouvelle fonction partout où la fonction était définie.

Pour définir une fonction, on peut utiliser la **table** = texte fini déterminant sans contradiction ni ambiguïté le contenu de la fonction. Il n'est pas nécessaire de décrire ou connaitre un moyen de calculer la fonction pour pouvoir la définir via sa table.

Combien il y a-t-il de programmes informatiques différents ?

Objectif : comparer la taille de 2 ensembles

Si les ensembles sont finis, c'est simple on compte leurs éléments distinctifs et on voit si on obtient le même chiffre. Mais s'ils sont infinis, comment comparer leur taille ?

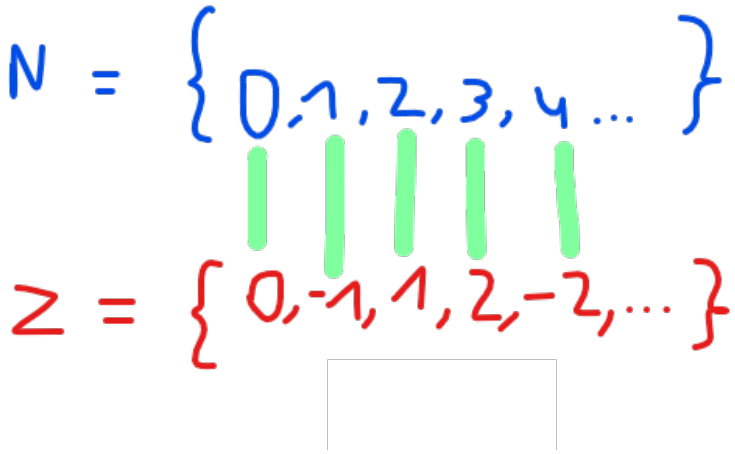
⇨ Déterminer s'il est possible de mettre les 2 ensembles en BIJECTION l'un avec l'autre. Si c'est le cas on dira qu'ils ont la même taille, le même **cardinal** (on utilise cardinal et pas taille ici pour les ensembles infinis pour éviter l'ambiguïté).

Pour nommer le nombre d'éléments dans un ensemble :

Si l'ensemble est fini, on appelle cela la **taille**

Si l'ensemble est infini, on appelle cela le **cardinal**

Un ensemble est **énumérable** (ou dénombrable) s'il est fini OU il a le même cardinal que N (être mit en bijection avec les entiers naturels). Un ensemble infini est énumérable s'il existe une liste infinie de tous ses éléments (c'est en quelque sorte une bijection avec les nombres naturels puisque dans cette liste, chaque élément de la liste infinie est en lien avec un nombre entier naturel, son indice dans la liste).



⇨ Les 2 ensembles ont le même cardinal, Z est un ensemble infini et N, qui est une liste de tous les entiers naturels positifs, est aussi infini. Pour chaque élément de Z, on peut faire correspondre un élément de N (son indice dans le tableau Z tout simplement)

Rationnel = rapport entre 2 entiers différents de 0

L'ensemble des programmes Java est-t-il un ensemble énumérable ?

Un Programme Java c'est une chaine de caractères qui est finie, et l'alphabet est donné (ex : code ASCII). Si j'ai un programme Java, celui-ci a une longueur particulière (*n* caractères, chaque caractère est composé de 8 bits). Finalement un programme Java peut être vu comme un grand nombre binaire de 8 * *n* bits. On peut voir donc un programme Java comme un grand nombre entier, ce qui veut dire que l'ensemble des programmes Java est énumérable, chaque programme Java peut correspondre à un nombre entier différent.

Conclusion :

- ➔ Un Programme Informatique est une chaîne finie de caractères
- ➔ Quel que soit le langage utilisé, il y a une infinité énumérable de programmes informatiques
- ➔ L'ensemble des programmes Informatiques est énumérable
- ➔ L'univers des programmes informatiques est donc de la même taille que l'ensemble *n*

➔ Dès que l'on parle de calculabilité et de programme informatique, nous sommes dans l'univers de l'énumérable, l'univers des entiers.

Existe-t-il des ensembles infinis non-énumérables (impossible de les mettre en bijection avec les entiers) ?

Théorème de Cantor :

L'ensemble des nombres réels compris entre 0 et 1 n'est pas énumérable, il y a plus de réels entre 0 et 1 que de nombres d'entiers.

⇨ Preuve : On suppose E énumérable, il existe donc une énumération des éléments de E : $x_0, x_1,\ldots,x_n,\ldots$

On peut alors construire une table, où une ligne va représenter un réel compris entre 0 et 1. Les différentes colonnes correspondent à la position d'un chiffre dans la décimale. Un élément de mon tableau sera donc le *i-ème* digit de l'extension décimale de ce réel compris entre 0 et 1. Un élément de ma table sera donc un nombre compris entre 0 et 9.

$x_k = 0.\, x_{k0}\, x_{k1} \ldots x_{kk} \ldots$ ex : 0,12....

	1 digit	2 digit	3 digit	...	k+1 digit	...
x_0	x_{00}	x_{01}	x_{02}	...	x_{0k}	...
x_1	x_{10}	x_{11}	x_{12}	...	x_{1k}	...
x_2	x_{20}	x_{21}	x_{22}	...	x_{2k}	...
:	:	:	:	:	:	:
x_k	x_{k0}	x_{k1}	x_{k2}	...	x_{kk}	...
:	:	:	:	:	:	:

On va ensuite sélectionner la diagonale *d* de cette table, et on obtient un nouveau nombre réel compris entre 0 et 1, constitués de tous les digits sur la diagonale. On va ensuite modifier ce nombre obtenu :

- Si le digit est différent de 5, on le met à 5
- S'il est égal à 5, on le met à 6

On obtient un nouveau nombre *d'* composé de 5 et de 6 compris entre 0 et 1. Ce nombre est donc dans la table que l'on a créée auparavant, puisqu'on a fait l'hypothèse que cette table contenait tous les réels entre 0 et 1.

	1 digit	2 digit	3 digit	...	k+1 digit	...
x_0	x_{00}	x_{01}	x_{02}	...	x_{0k}	...
x_1	x_{10}	x_{11}	x_{12}	...	x_{1k}	...
x_2	x_{20}	x_{21}	x_{22}	...	x_{2k}	...
:	:	:	:	:	:	:
x_k	x_{k0}	x_{k1}	x_{k2}	...	x_{kk}	...
:	:	:	:	:	:	:

On arrive à une contradiction. Si le nombre est dans la table, il se trouve à un endroit bien précis. On va dire qu'il se trouve sur la ligne *p*. Sur cette ligne *p*, je vais regarder le nombre qui est sur la diagonale d'auparavant. On a donc la valeur x_{pp}. *d'* n'est pas dans l'énumération car si $d' = x_p$,

- Alors $d' = 0.x_{p0}\, x_{p1} \ldots x_{pp} \ldots$
- $= 0.x'_{00}\, x'_{11} .. x'_{pp} ..$

Ce qui est impossible car $\underline{x_{pp} \neq x'_{pp}}$

Rappel : *d* = diagonale de la table

d' = nouveau nombre obtenu après modification avec les 5 et 6
p = ligne sur lequel se trouve le nombre d'
x_p = nouvel entier qui se trouve sur la ligne p du tableau
x_{pp} = nombre qui se trouve sur la ligne p et sur la diagonale d
x'_{pp} = nombre qui se trouve sur la ligne p et qui appartient à l'entier d'

Autre explication :

Le théorème de Cantor énonce qu'il n'existe pas de correspondance bijective entre l'ensemble des nombres naturels (1, 2, 3, 4, ...) et l'ensemble des nombres réels.

Supposons qu'il existe une correspondance bijective complète entre les nombres naturels et les nombres réels (Preuve par l'absurde). Cela signifierait qu'il est possible d'organiser tous les nombres réels dans une liste infinie.

Cantor a utilisé une astuce ingénieuse pour montrer qu'une telle liste ne peut pas exister. Il a proposé de construire un nouveau nombre réel qui ne figure pas dans la liste. Cantor a utilisé une méthode de construction spéciale : il a considéré les décimales de chaque nombre réel de la liste.

Imaginons que nous ayons une liste infinie de nombres réels :

0,14159...
0,71828...
0,57781...
0,62345...
...

Cantor décida de créer de nouveaux nombres réels en choisissant des décimales différentes pour chaque position de décimale. En continuant avec la liste de ci-dessus :

0,23594…

Cantor a procédé en construisant un nombre réel dont les décimales diffèrent des décimales correspondantes des nombres réels de la liste.

Pour illustrer cela, il a choisi de prendre la première décimale après la virgule de chaque nombre réel et de construire un nouveau nombre réel dont la première décimale diffère de celle de tous les nombres de la liste. Par exemple, si la première décimale du premier nombre est 1, Cantor choisit une autre valeur pour la première décimale de son nombre construit, disons 2.

Ensuite, Cantor passe à la deuxième décimale après la virgule et choisit une valeur différente pour cette décimale, encore une fois, afin que son nombre construit diffère de tous les nombres réels de la liste. Ce processus se répète pour chaque décimale.

Conclusion : E n'est pas énumérable, les réels ne sont pas énumérables.

Quelques Exemples :

Ensembles énumérables	*Ensembles non-énumérables*
- Ensemble des entiers - Ensemble de nombres pairs - Ensemble de paires, triplets d'entiers - Ensemble des rationnels	- Ensemble des Réels R - Ensemble des sous-ensembles de N - Ensemble des fonctions de N dans N

L'univers des programmes informatiques est une infinité énumérable de programmes informatique pour résoudre un problème (problème = fonction)

L'univers des problèmes informatiques est l'ensemble des fonctions de N dans N (non-énumérables). Par rapport aux programmes informatiques, il y a trop de fonctions. Nous savons déjà que la toute grande partie des fonctions ne pourront pas être calculée par un programme.

Ensemble des programmes : énumérables, et l'ensemble des problèmes : non-énumérables.

Tout sous-ensemble (infini ou non) d'un ensemble énumérable est aussi énumérable.

Une fonction dont la table est infinie peut être décrite de manière finie. Exemple : la fonction identité f(x) = x. Sa table est infinie mais elle peut être facilement décrite de manière finie par l'expression f(x) = x

Tout ensemble non énumérable peut être mis en bijection avec l'ensemble des réels. Il existe des ensembles qui ont encore plus d'éléments que R et qui ne peuvent pas être mis en bijection avec R. Exemple : l'ensemble 2^R est non énumérable et ne peut pas être mit en bijection avec R.

L'énumérabilité des programmes Java et la non énumérabilité des fonctions de N vers N est une preuve de l'existence de fonctions non calculable. Cela indique qu'il n'y a pas assez de programmes Java par rapport au nombre de fonctions de N vers N. Il est donc impossible de représenter ou de calculer chaque fonction par un programme.

Chapitre 3 : Résultats Fondamentaux

Algorithme = procédure (ensemble d'instructions) ayant pour but de produire un résultat. Cette procédure peut s'appliquer à n'importe quelle données.

Attention, un algorithme n'est PAS une fonction. L'algorithme va calculer qu'une et une seule fonction à la fois.

On va considérer dans ce cours que programme == algorithme. Dans notre vision de ce qu'est un programme ou un algorithme, nous n'allons pas considérer de limites :

- Quant à la taille des données
- Quant à la taille des instructions
- Quant à la taille de mémoire disponible (même si on sait que la mémoire sera finie)

Dans nos univers, nous avons l'univers des programmes informatiques, et l'univers des fonctions.

Univers des Programmes	**Univers des Fonctions**

Choix d'un langage particulier (ici Java) et on va considérer que la calculabilité est le fait de pouvoir être calculé par un programme Java. Énumérables	On se limite aux fonctions de N dans N (N -> N, donc 1 input et 1 résultat). Non-énumérables

Qu'est-ce qu'une fonction calculable ?

Une fonction *f* de N -> N est **Calculable** s'il existe un programme Java pour la calculer. Ce programme va :

1) Si f(x) est définie : donner comme résultat la valeur de f(x)
2) Si f(x) n'est pas définie : ne va pas se terminer ou lancer un message d'erreur f(x) = ⊥

⇨ Attention, cette définition se base sur le fait que l'on ait un programme Java à la base. Une fonction peut être calculable sans que l'on sache comment l'écrire sous forme de programme.
⇨ Lorsqu'une fonction est non-calculable, cela veut dire qu'il n'existe pas de programme pour la calculer !
⇨ Parmi les fonctions calculables, il existe des **Fonctions Partielles Calculable**, cela veut dire que dans certains cas il n'y a pas de résultats. Et il existe également des **Fonctions Totales Calculable**, qui produisent un résultat sans aucune exception, dans tous les cas.

Exemple

On souhaite construire une fonction f(x) qui me donne les 6 numéros gagnants du Loto. C'est une fonction constante (= on ne les connait pas les 6 numéros gagnants aujourd'hui mais quand ils seront connus, le résultat sera unique)
Cette fonction est-elle calculable ? (Aujourd'hui, sans attendre le résultat du loto)

⇨ OUI, *f* est calculable car ce que l'on peut faire, c'est écrire plusieurs programmes *p1*, qui imprime les 6 numéros. On fait ensuite le programme *p2*, qui imprime une autre combinaison. On fait *n* programmes jusqu'à avoir le programme p_n qui affiche la bonne combinaison.
⇨ Il y a de l'ordre de 228 millions de programmes (combinaisons possibles) au total. Un de ces programmes calcule ma fonction *f*. Donc la fonction est calculable.

Récursivité

Un Ensemble A est **Récursif** s'il existe un programme Java qui reçoit comme donnée un nombre naturel *x*, qui va donner un résultat :

- 1 si $x \in A$
- 0 si $x \notin A$

⇨ Programme se termine toujours avec une réponse (0 ou 1)
⇨ Tout ensemble qui est fini est récursif
⇨ Un ensemble récursif est d'office énumérable
⇨ Un ensemble est récursif si et seulement si la fonction est une fonction totale calculable.

"Récursif" et "récursivité" sont deux termes qui sont étroitement liés, mais ils ont des significations légèrement différentes.

Un **Algorithme Récursif** c'est un algo qui s'appelle lui-même

Un **Ensemble Récursif** c'est un ensemble dont tu peux déterminer si un élément est, ou non, dans l'ensemble. C'est donc un ensemble pour lequel il existe un programme qui dit 1 si l'élément est dans l'ensemble et 0 sinon.

Un Ensemble A est **Récursivement Énumérable** s'il existe un programme Java qui reçoit comme donnée un nombre naturel *x*, qui va donner un résultat :

- 1 si $x \in A$
- Ne se termine pas ou retourne autre chose que 1 si $x \notin A$

⇨ Récursivement énumérable = version amoindrie d'être récursif

⇨ Un ensemble est récursivement énumérable si et seulement s'il existe une fonction *f* calculable tel que l'ensemble est égal au domaine de la fonction, ou vide, ou est égal à l'image de la fonction.

Différence entre ensemble récursif & ensemble récursivement énumérable

Ensemble récursif	Ensemble récursivement énumérable
Existe s'il existe un programme qui prend en input *x* et qui renvoie : - 1 si $x \in A$ - 0 si $x \notin A$	Existe s'il existe un programme qui prend en input *x* et qui renvoie : - 1 si $x \in A$ - Autre résultat, ou ne se termine pas

Propriétés

1. Si un ensemble est récursif, l'ensemble est récursivement énumérable et son complément est récursif + récursivement énumérable.
2. Si un ensemble et son complément sont récursivement énumérables, l'ensemble est récursif
3. Si un ensemble est fini, cet ensemble est récursif
4. Si un complément d'ensemble est fini, son ensemble est récursif

Thèse de Church-Turing

La thèse de Church-Turing vient de la question : « Que signifie calculable ? »

Alonzo Church et Alan Turing ont fondé la **thèse de Church-Turing** :

1. **Toutes les définitions formelles de la calculabilité connues à ce jour sont équivalentes.**
   S'il existe un programme Java pour calculer une fonction, il existera un programme Python, une machine de Turing, une machine de lambda calcul pour calculer cette même fonction.
2. **Tous les formalisations de la calculabilité établies par la suite seront équivalentes aux définitions connues.**
   Si d'autres formalismes de calculabilité venaient à être inventé dans le futur, ceux-ci ne seraient pas plus puissant que ce qui existe aujourd'hui.

3. **Aucun modèle de la notion de la fonction calculable n'est plus puissant que les Machines de Turing.**
   Il n'y a pas d'autres modèles plus puissant que la machine de Turing.

4. **Toute fonction calculable (sens intuitif) est calculable par une machine de Turing.**
   L'intuition de ce qui est calculable, est en fait, cette intuition est capturée par les machines de Turing ou le langage Java.

Aucune thèse jusqu'à aujourd'hui n'est venu contredire de ces 4 points. Encore aujourd'hui reconnue comme scientifiquement vraie.

Version moderne de la calculabilité :

5. **Une fonction est calculable s'il existe un programme d'ordinateur qui calcule cette fonction.**
   Conséquence : si une fonction n'est pas calculable en Java, elle ne pourra pas être calculée quel que soit le formalisme de calcul utilisé.

Le langage Java est constitué d'un ensemble de programme.On va se limiter aux programmes qui ont une seule donnée pour calculer les fonctions N -> N et qui imprime un seul résultat entier. L'ensemble de ces programmes Java est récursif (car il existe un compilateur calculable) et énumérable (car chaque programme est une suite finie de caractères).

Une **Fonction de Numérotation** est une fonction qui va associer à chaque entier un programme. La fonction *f* est calculable. P_k dénote le programme *k* dans *P*. ϕ_k dénote la fonction calculée par P_k. Attention : P_k = univers des programmes, ϕ_k = univers des fonctions.

Rappel : Ensemble des fonctions N -> N : non-énumérable & l'Ensemble des programmes Java : énumérable

- ⇨ La plupart des fonctions ne peuvent pas être calculable puisqu'il n'y a pas assez de programme pour les calculer.
- ⇨ L'univers des problèmes à la même taille que l'univers des programmes que l'on a à notre disposition.

Problème de l'Arrêt

On se demande s'il est possible d'avoir une fonction *halt*, qui prend en argument le numéro d'un programme et un entier, et qui va renvoyer *True* si la fonction se termine, et faux dans l'autre cas.

Problème de l'Arrêt → une telle fonction peut-elle exister ?

Soit fonction *halt* qui prend 2 paramètres (numéro programme + entier) et retourne entier.

Cette fonction *halt(n,x)* vaudra :
- 1 si $\phi_n(x) \neq \perp$ (vaut 1 si l'exécution du programme $P_n(x)$ se termine avec résultat)
- 0 sinon (vaut 0 si = ⊥ = exécution programme ne se termine pas)

La fonction *halt* est parfaitement définie (signification claire, pas d'ambiguïté) et totale (retourne toujours un résultat) mais n'est pas calculable. Comme preuve, nous allons supposer que *halt* est calculable (= preuve par l'absurde).

	0	1	2	...	k	...
P_0	halt(0,0)	halt(0,1)	halt(0,2)	...	halt(0,k)	...
P_1	halt(1,0)	halt(1,1)	halt(1,2)	...	halt(1,k)	...
P_2	halt(2,0)	halt(2,1)	halt(2,2)	...	halt(2,k)	...
:	:	:	:	:	:	
P_k	halt(k,1)	halt(k,2)	halt(k,3)	...	halt(k,k)	...
:	:	:	:	:	:	:

1. **On crée une table à 2 dimensions**. Dans la première (VER), nous allons mettre tous les entiers, et dans la seconde (HO) tous les programmes Java numérotés. Une case d'un tableau représente donc le résultat de la fonction du programme numéro *x* avec *y* entiers. Nous allons avoir dans chaque case soit la valeur 0, soit la valeur 1, en fonction de si le programme se termine (1) ou non (0)
2. **On sélectionne la diagonale du tableau** *halt(0,0) halt(1,1) halt(k,k)…* On va créer une fonction de cette diagonale, appelée *diag(n)* qui vaut *halt(n,n)*
3. **On va modifier *diag(n)* en la renommant *diag_mod(n)*** qui vaudra :
   - 1 si *halt(n,n)* = 0
   - ⊥ si *halt(n,n)* = 1

   On décide de définir une nouvelle fonction *diag_mod(n)* basée sur la diagonale du tableau, et qui inverse les résultats de cette diagonale. Cette fonction est calculable (sous l'hypothèse que la fonction *halt* est calculable) puisqu'il est possible d'écrire le code de *diag_mod* en faisant appel à la fonction *halt* ➔ On retourne l'inverse de ce que retourne la fonction *halt*.
4. Si *diag_mod(n)* est calculable, il existe un programme Java qui calcule *diag_mod*, et ce programme est-il est quelque part dans la liste, on suppose qu'il est à la ligne numéro *d*. **Quelle est la valeur de *diag_mod(d)* ?** 2 possibilités : soit c'est 1, soit ⊥ :

   - **Si *diag_mod(n)* = 1** alors *halt(d,d)* = 0, et si cette fonction *halt* vaut 0, c'est que le programme *d* sur la donnée *d* ne se termine pas. Si ce programme ne se termine pas, c'est que *diag_mod(d)* ne se termine pas non plus, ce qui veut dire que *diag_mod(n)* retourne ⊥ ➔ *diag_mod(d)* = 1 et *diag_mod(d)* = ⊥ ABSURDE !
   - **Si *diag_mod(n)* = ⊥** alors *halt(d,d)* = 1, alors *diag_mod* se termine par définition de *halt*, ce qui veut dire que si *diag_mod* se termine, alors il retourne 1 ➔ *diag_mod(d)* = ⊥ et *diag_mod(d)* = 1 ABSURDE !

Quel que soit l'issue de *diag_mod*, on arrive à une contradiction. En fait *diag_mod* n'est pas calculable et *halt* non plus. Il n'existe pas de programme Java calculant la fonction *halt*

Conclusion

- ⇨ Aucun algorithme ne permet de déterminer, étant donné un programme P_n et une donnée *x*, si ce programme $P_n(x)$ se termine ou non
- ⇨ La non-calculabilité de *halt* est l'impossibilité d'avoir un algorithme général qui pourrait décider pour n'importe quelle paire programme-donnée. Ce n'est pas possible d'avoir un programme qui décide quel que soit l'input donné, qui permet de retourner une réponse dans tous les cas.
- ⇨ *halt* est récursivement énumérable mais pas récursif, mais pas calculable

Halt = L'ensemble des paires *n,x* pour lesquelles la fonction *halt(n,x)* retourne 1, donc l'ensemble des paires pour lesquelles le programme se termine. Cet ensemble ne peut pas être récursif car s'il l'était, j'aurais un moyen de calculer *halt*, d'écrire un programme Java.

- ⇨ Non-Récursif
- ⇨ Récursivement énumérable
- ⇨ Non-calculable

K = Ensemble des programmes qui se terminent.

- ⇨ Non-récursif
- ⇨ Récursivement énumérable
- ⇨ Non Calculable

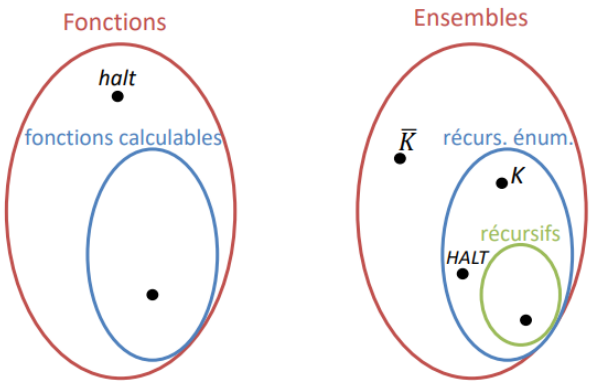
K barre = ensemble des programmes qui ne se terminent pas, qui bouclent en continu.

- ⇨ Non-Récursif
- ⇨ Récursivement Non énumérable
- ⇨ Non Calculable

N = ensemble des programmes. C'est aussi l'union entre K et K barre.

- ⇨ Récursif
- ⇨ Récursivement énumérable
- ⇨ Calculable

Structure des Fonctions Ensembles



Principe de la Diagonalisation

1. **Construction d'une table** : par exemple la liste de tous les grands mathématiciens
2. **Sélection de la diagonale** : On prend la diagonale des caractères de la liste.
3. **Modification de cette diagonale** : On modifie un à un chaque élément de cette diagonale. Par exemple prendre chaque lettre précédente de l'alphabet (*DBOUPS* -> *CANTOR*)
4. **Contradiction** : on remarque que le nouveau mot n'est pas dans la liste de base. Il ne peut pas être dans cette liste : si l'était, quel que soit sa position, il y aurait une contradiction à la position diagonale de la position de *CANTOR*, les lettres ne seraient pas les mêmes. *CANTOR* n'est pas dans cette liste

⇨ Soit on sait que la liste est complète, qu'elle est fixée, connue et contient tous les grands mathématiciens. Dans ce cas-là, la conclusion est que *CANTOR* n'est pas un grand mathématicien puisqu'il n'est pas dans la liste et que celle-ci est complète.

⇨ Soit on sait que *CANTOR* est un grand mathématicien, dans ce cas la liste n'est pas complète.

Rappel :

- **Énumérable** : ensemble qui est fini ou qui peut être mit en bijection avec N
- **Récursif** : va dans tout les cas retourner quelque chose : 1 ou 0
- **Récursivement énumérable** : va soit retourner quelque chose et se terminer, soit ne pas se terminer ou ne rien retourner.
- **Calculable** : ensemble où une fonction caractéristique (programme Java) existe pour le calculer.

Insuffisance des fonctions totales

Il est indispensable qu'un langage de programmation ait des programmes qui ne se terminent jamais.

On pourrait inventer un langage de programmation *Q* qui ne calcule que des fonctions totales, où tous les programmes se terminent toujours. Chaque programme va calculer une fonction que l'on va dénoter ϕ'_k. On pourrait l'appeler *mini-Java*, et on enlève les boucles *while*, les sauts (*goTo*) et tout ce qui est lié à la récursivité, on garde les boucles *for* mais avec une limite fixée. L'interpréteur de ce langage est calculable.

Théorème de Hoare-Allison

Si l'on a un langage (comme *Q*) qui ne calcule que des fonctions totales, alors
- Son interpréteur est calculable (ex : Java)
- La fonction *halt* est calculable

MAIS : *l'interpréteur de ce langage, quoi qu'étant calculable, ne peut pas être programmé dans le langage Q lui-même*. Le langage *Q* ne permet donc pas de calculer la fonction *interpret* bien que celle-ci soit totale et calculable. Le langage *Q* est limitatif.

Preuve :

On va supposer que la fonction *interpret* est calculable

1. **Construction d'une table** dans lequel nous mettons à la verticale les différentes données possibles de mes programmes et à l'horizontale un programme de *Q*. Une cellule va contenir un entier qui est résultat de l'exécution du programme. Vu que tout se termine toujours dans ce langage, on aura toujours un résultat.

	0	…	k	…
Q_0	interpret(0,0)	…	interpret(0,k)	…
:	:	:	:	
Q_k	interpret(k,1)	…	interpret(k,k)	…
:	:	:	:	:

2. **Sélection de la diagonale** du tableau *diag(n) = interpret(n,n)*
3. **Modification de la diagonale** en créant *diag_mod(n) = interpret(n,n)* + 1. On peut affirmer que *diag_mod* est calculable dans *Q*. En effet, sous l'hypothèse que *interpret* est calculable dans ce langage *Q*, on peut écrire *diag_mod(n)* en appelant *interpret* dans le code de *diag_mod* puis en prenant le résultat de *diag_mod* et retourner ça + 1.
4. Si *diag_mod* est calculable dans *Q* (ce que l'on a affirmé au-dessus), il y a donc parmi les programmes de *Q* un programme qui calcule *diag_mod*. Nous arrivons à une contradiction. **Quelle est la valeur de *diag_mod(d)* ?**
   Par définition, *diag_mod(d) = interpret(d,d)* + 1. Mais si nous savons que *diag_mod(d)* est calculé par le programme Q_d, on a *diag_mod(d)* = $\phi'_d(d)$ = *interpret(d,d)* ➔ Contradiction *diag_mod(d) = interpret(d,d)* et *interpret(d,d)* + 1

⇨ *Interpret* n'est pas calculable dans *Q*, l'interpréteur ne peut pas être programmé à l'intérieur de Q

Langage avec fonctions totales et partielles	- Contient des boucles, sauts et tout ce qui est lié à la récursivité - Contient des fonctions totales et partielles - Les programmes ne se terminent pas toujours - Interpréteur programmé dans son langage
Langage avec fonctions totales uniquement	- Ne contient pas de boucles, et tout ce qui est lié à la récursivité - Contient uniquement des fonctions totales - Les programmes se terminent toujours de ce langage - Interpréteur n'est pas programmé dans son langage : imposs

Analyse du théorème de Hoare-Allisson

Si un langage de programmation ne permet de créer que des fonctions totales, alors

- L'interpréteur de ce langage n'est pas programmable dans ce langage
- Ce langage est **restrictif** : il y a des fonctions totales qui sont calculables et intéressantes et qui ne pourront jamais être calculée dans ce langage.

Si on veut qu'un langage de programmation permette la programmation de toutes les fonctions totales calculables, alors ce langage doit également permettre la programmation de fonctions non totales. Le fait d'avoir des programmes qui bouclent est INDISPENSABLE pour les langages de programmation afin de pouvoir calculer le plus de fonctions calculables.

Conséquence de ce Théorème :

⇨ On ne peut pas déterminer si la fonction calculée est totale. Si c'était le cas, on pourrait créer un *mini-java* et on sait que cela est impossible.

Pour qu'un langage de programmation soit le plus utile possible, qui puisse calculer tout ce qui est possible de calculer, il doit permettre de programmer son propre interpréteur.

Un **Interpréteur** est un programme utilisant des principes simples de programmation : lecture de fichiers, structure de données, gestion de tables, opérations élémentaires. C'est un programme particulier qui a un numéro *z*, et quand j'exécute ce programme particulier avec ses 2 paramètres (numéro de programme *n* et donnée *x*), cela va me donner le même résultat que l'exécution du programme *n* sous la donnée *x*

$$\exists z \, \forall n, x : \varphi_z (n,x) = \varphi_n (x)$$

Expression d'un interpréteur

En calculabilité, **interpréteur = fonction universelle**

Extension de Fonctions

Une **Extension** d'une fonction est une fonction totale de la fonction de base, qui va toujours retourner un résultat.

Le passage d'une fonction calculable partielle à une extension totale calculable n'est pas toujours possible ➔ Il existe des fonctions partielle calculable mais pour lequel il n'est pas possible d'avoir une fonction totale calculable, qui soit une extension de la première.

Exemple de fonction partielle qui ne peut pas être étendue en une fonction totale

Nous définissons la fonction *nbStep(n,x)* et cette fonction va donner le nombre d'instructions du programme $P_n(x)$ avant l'arrêt de $P_n(x)$. Est égal à ⊥ si boucle infinie sinon est différent de ⊥. Fonction calculable partielle.

Impossible d'avoir une extension de la fonction *nbStep*(n,x)*. Supposons qu'une telle fonction soit calculable. Dans ce cas, on aurait un moyen de calculer la fonction *halt(n,x)*. Il suffirait d'écrire un programme où :

On stocke dans une variable *k* la valeur retournée par la fonction *nbStep*(n,x)*, ensuite on exécute $P_n(x)$ pendant *k* instructions. Si l'exécution est terminée, ça veut dire que $P_n(x)$ se termine, si ce n'est pas le cas, c'est que *nbStep** nous a donné un résultat pour remplacer le ⊥, ce qui veut dire que $P_n(x)$ et on print 0.

Il n'est donc pas possible d'avoir une extension de cette fonction nbStep, car une telle extension totale conduirait à la calculabilité de la fonction *halt*, que l'on sait non-calculable.

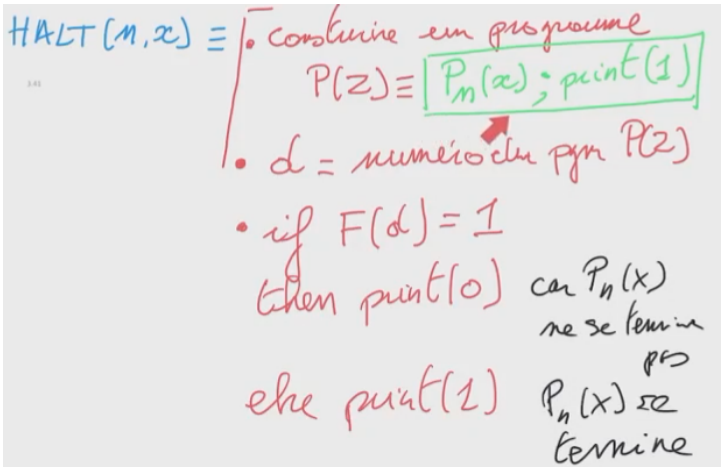
Réduction à Halt

Lorsque l'on veut déterminer si une fonction est calculable ou non, on doit faire un raisonnement sur mesure. Une des méthodes possible est la **méthode de réduction à *halt* :**

1. **On suppose que la fonction est calculable.**
2. **On montre que sous l'hypothèse de la calculabilité de la fonction *f*, la fonction *halt* est calculable.**
3. **Comme on sait que *halt* n'est pas calculable, on peut conclure que *f* ne peut pas être calculable non plus.**

Premier Exemple : *f(n)* = 1 si $\phi_n(x) = \perp$ pour tout X, 0 sinon

1) Supposons *f(x)* calculable. Il existe donc un programme *f* qui calcule la fonction *f(x)*
2) On montre que la fonction *halt(n,x)* est calculable. Pour cela, on construit un programme *halt(n,x)* dont le code est le suivant :



On voit que *halt* est calculable dans ce cas-ci.

3) Étant donné que *halt* est non-calculable, ma fonction *f(x)* est non-calculable.

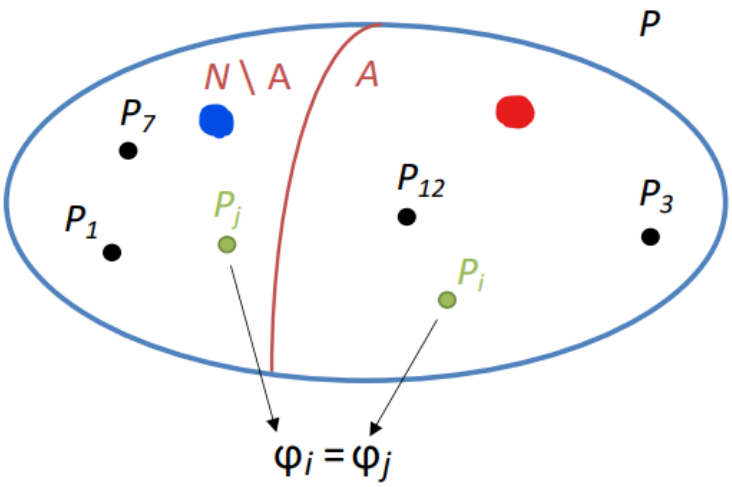
⇨ Toute fonction totale calculable f est l'extension d'une fonction partielle calculable g.

Théorème de Rice

Nous avons ci-dessous l'ensemble des programmes et si l'on divise cet ensemble en 2 parties (programmes de gauche et de droite)

Si *A* est récursif (si l'on peut décider si un programme est rouge ou bleu) et que c'est non-vide et que ≠ *N*, on est sûr qu'il existe un programme rouge et bleu tel que ces 2 programmes calculent la même fonction. (Forme positive)

Si je découpe mon ensemble de programme en 2 parties (A et A barre), et qu'aucun des programmes de A et A barre calculent la même fonction, alors mon ensemble A ne peut pas être décidé par un programme, cet ensemble n'est pas récursif, ou A = ∅ ou A = *N*. (Forme négative).



Analyse du Théorème

Si une propriété de programmes (ex : colorier programmes rouge & bleu) est

1) vérifiée par certains programmes mais pas tous (cas particuliers vide ou égal à N),

2) est décidable (on peut décider si un programme est rouge ou bleu),

➔ Alors il y a un programme rouge et bleu qui calculent la même fonction.

Si une propriété de la fonction calculée par un programme (programmes rouge calculent fonction avec certaine propriété, que programme bleus n'ont pas), donc il n'y a pas de programme bleu qui calculent la même fonction qu'un programme rouge. Et bien cette propriété ne peut pas être décidée par un algorithme.

Exemples

$A_1 = \{i \mid \phi_i$ est totale$\}$: ensemble programmes calculant fonction totale, P_i s'arrête toujours.

Pour appliquer le théorème de Rice, on doit montrer que A_1 est différent de $\emptyset$. On construit un programme $P_k(x)$ qui print 1. Ce programme ne boucle jamais et donc on sait que $k \in A_1$, donc A_1 est non-vide.

On doit montrer que $A_1 \neq N$. On construit un programme $P_d(x)$ qui boucle tout le temps. Il ne finit donc jamais, et $\notin$ à A_1. $A_1 \notin$ à N.

On doit montrer maintenant que quel que soit le programme $\in$ à A_1 et quel que soit le programme $\in$ au complément de A_1, la fonction calculée par le programme numéro i est différente de la fonction calculée par le programme numéro j. Si le programme numéro $i \in$ à A_1, la fonction ϕ_i est totale, alors que la fonction ϕ_j est non-totale et n'appartient pas à A_1. Une fonction totale (ϕ_i) ne peut pas être égale à une fonction qui n'est pas totale (ϕ_j)

⇨ **Grâce au théorème de Rice, on peut affirmer que l'ensemble A_1 n'est pas récursif.**

Dès que l'on souhaite voir ce que fait un programme, sous l'angle de la fonction qu'il calcule, c'est impossible de faire de manière automatique. On ne peut pas se faire aider par un programme. Ceci est une limitation très grande au niveau de la calculabilité.

Il est impossible de faire ces choses-là automatiquement :

➔ Voir si 2 programmes calculent la même fonction
➔ Corriger automatiquement un programme informatique lors d'un examen

Bad News : propriétés et problèmes pour l'informaticien ne peuvent pas être automatisés.

Good News : être humain est nécessaire pour corriger programmes informatiques, les évaluer.

Démonstration de la forme Négative du Théorème de Rice

Supposons que A est $\neq \emptyset$ et que A $\neq N$
Supposons $\forall$ i $\in$ A, $\forall$ j $\in$ A barre, ϕ_i est $\neq \phi_j$
Alors A est non-récursif.
Pour démontrer cela, nous allons utiliser la technique de la réduction à Halt.

Supposons que A est récursif. Sous cette hypothèse, nous allons montrer que *halt* est calculable, et comme nous savons que *halt* ne l'est pas, on pourra déduire que *A* est non-récursif. L'ensemble de programmes est découpé en 2 parties : A et A barre. Nous allons écrire un programme $P_k(x)$ qui boucle tout le temps. Nous faisons l'hypothèse que $k \in$ A barre.
Nous savons que $A \neq \emptyset$, il y a bien un programme $\in$ A. Nous prenons un programme quelconque *m* qui $\in$ A. Nous pouvons affirmer que $\phi_k \neq \phi_m$.

Nous allons démontrer que A récursif, nous allons construire un programme qui va calculer la fonction *halt(n,x)*, avec comme code la création de 2 programmes : $P(z) = P_n(x)$; $P_m(z)$. On va assigner à la variable *d* le numéro du programme que l'on vient de construire. À chaque exécution de mon programme, nous allons obtenir un programme *P(z)* qui va être différent. On dépose dans *d* le numéro du programme qu'on vient de construire. On teste ensuite si $d \in$ A ou non. Que va calculer *P(z)* ? Il y a 2 possibilités :

- Si $P_n(x)$ se termine, l'exécution du programme va se réduire à l'exécution du programme $P_m(z)$. $\phi_d = \phi_m$ (alors dans ce cas $d \in$ A).
- Si $P_n(x)$ ne se termine pas, alors $\phi_d = \phi_k$ (alors dans ce cas $d \notin$ A).

Étant donné qu'entre les éléments de A et A barre, il n'y a jamais 2 programmes qui calculent la même fonction, tester si $\phi_d = \phi_m$ revient à tester si $d \in$ A. Tandis que $\phi_d = \phi_k$ c'est comme si on testait si $d \notin$ A. J'ai donc un moyen de déterminer si $P_n(x)$ se termine ou non

- Si d $\in$ A, on print 1.
- Si d $\notin$ A, on print 0.

On peut conclure par la réduction à *halt* que *A* n'est pas un ensemble récursif

Théorème de la Paramétrisation

Nous avons vu pour l'instant que des théorèmes dits « négatifs », ils donnent des limites à ce qui est calculable. Existe-t-il des théorèmes positifs ?

Une fonction f : N -> N peut être vue comme un transformateur de programme : le Programme numéro A va donner le programme numéro B : $P_A \rightarrow P_B$

Théorème S-M-N
On peut spécialiser un programme qui a 2 paramètres, en un programme qui n'en a plus qu'un, en fixant le second restant à une valeur particulière.

Il existe un transformateur de programme S^1_1, qui va recevoir 2 données :

- Un programme P_k composé de 2 arguments
- La valeur v_2 pour laquelle nous voulons spécialiser le $2^{ème}$ argument de P_k

Et fournir comme résultat :

- Un Nouveau programme P_{x1} composé d'1 argument, qui calcule la même fonction que $P_k(x_1,v_2)$

Il est possible de généraliser le théorème avec un programme S^m_n, qui va recevoir m+n arguments et spécialiser m arguments finalement. S^m_n est calculable en toute généralité, indépendamment des programmes à transformer.

Preuve du Théorème S-M-N

On va prendre S^1_1. Nous devons montrer qu'il existe un transformateur de programmes S^1_1 qui reçoit comme paramètre :

- *Un Programme P_k qui a 2 paramètres x_1 x_2*
- *Une valeur v_2*

Comment spécialiser le programme P_k ?

Une des technique possible : insérer l'instruction $x_2 = v_2$ au début du programme P_k

Version Affaiblie de S-1-1 :

Il s'agit d'avoir un transformateur spécifique, un programme qui lui est donné

On donne un programme *k* et on affirme qu'il existe un transformateur de programme qui va donner une fonction totale calculable S : N -> N tel que $\phi_k(x_1,x_2) = \phi_{S(x2)}(x_1)$

Cette fonction S est spécifique à la spécialisation de P_k

Théorème du Point-Fixe

Résultat positif de la calculabilité : il dit quelque chose qui est possible

Étant donné une fonction *f* totale calculable, quel que soit la fonction *f*, on sait qu'il existe un programme *k* tel que la fonction calculée par le programme numéro *k* est la même que la fonction calculée par le programme transformé par cette fonction *f* : $\phi_k^{(n)} = \phi_{f(k)}^{(n)}$

Quel que soit le transformateur, il va y avoir les 2 programmes P_k et P_j tels que

- P_j est la transformation de P_k
- P_k et P_j calculent la même fonction

Application du Théorème du Point Fixe

$K = \{n \mid \phi_n(n) \neq \perp\}$ n'est pas récursif
Preuve : Soient 2 programmes :

- $\phi_n(x) = \perp$ pour tout *x* (programme *n* qui boucle tout le temps)
- $\phi_m(x) = x$ pour tout *x* (fonction identité)

Définissons une fonction f(x) = n si $x \in K$ et f(x) = *m* si $x \notin K$
f est une fonction totale et calculable (si K est récursif)
Le Théorème du Point-Fixe me dit qu'il existe un programme *k* tel que la fonction calculée par le programme numéro k est la même que la fonction calculée par le programme numéro *f(k)*.
2 possibilités :

- Soit $k \in K$ alors f(k) = n (par définition de *f*). Par le point-fixe, on sait que $\phi_k^{(k)} = \phi_n^{(k)}$. Le programme se termine vu qu'il appartient à K, donc résultat différent de $\perp$. Or on sait dans la définition de *f* que $\phi_n(x) = \perp$, ce qui donne $\neq\perp = \perp$ ABSURDE
- Soit $k \notin K$ alors f(k) = m (par définition de *f*). Par le point fixe, on sait que $\phi_k^{(k)} = \phi_n^{(k)}$. Le programme ne se termine pas vu qu'il $\notin K$, donc résultat vaut $\perp$. Or on sait que dans la définition de *f* que $\phi_m(x) = x$, ce qui donne $\perp = \neq\perp$ ABSURDE

Nos 2 possibilités sont absurdes, notre hypothèse initiale comme quoi *K* est récursif est donc fausse. K n'est donc pas récursif.

Théorème de Rice via le Théorème du Point Fixe

Si A ≠ Ø et que A ≠ *N* ∀ i ∈ A, ∀ j ∈ A barre : ϕ_i est ≠ ϕ_j

Alors A est non-récursif.

Démonstration :

Nous savons que A ≠ Ø, il y a bien un élément dans A qui est *n*. Nous savons que A A ≠ *N*, il y a au moins un élément qui appartient au complément de A, qu'on va nommer *m*. Définissons f(x) = m si *x* ∈ A, et f(x) = *n* si *x* ∈ A barre. La fonction *f* est totale, il y a toujours un résultat, et elle est calculable, sous l'hypothèse que *A* est récursif.

Par le Point-Fixe, je sais qu'il existe *k* tel que $\phi_k^{(n)} = \phi_{f(k)}^{(n)}$. Est-ce que *K* ∈ A ?

2 possibilités :

- Soit *k* ∈ A alors f(k) = m et $\phi_k = \phi_m$. Ce qui signifie que *k* ∈ A barre car *m* ∈ A barre et ∀ i ∈ A, ∀ j ∈ A barre : ϕ_i est ≠ ϕ_j. Contradiction : voir soulignés.
- Soit *k* ∉ A barre alors f(k) = n et $\phi_k = \phi_n$. Ce qui signifie que *k* ∈ *A* car *n* ∈ A et ∀ i ∈ A, ∀ j ∈ A barre : ϕ_i est ≠ ϕ_j. Contradiction : voir soulignés.

Les 2 contradictions nous amènent à dire que A est non-récursif.

Analyse du Théorème du Point-Fixe

Le Théorème du Point-Fixe est un théorème central en calculabilité car :

- ⇨ Il implique le théorème de Rice
- ⇨ Il implique la non-récursivité de *K*
- ⇨ Il implique la non-calculabilité de *HALT*

Démonstration du Théorème du Point-Fixe

Voir vidéo du cours sur le sujet, trop la flemme de retaper tout ça.

Autres Problèmes non-calculables

On sait du Théorème de Rice que dès que l'on veut analyser ce que fait un programme, cette analyse ne peut pas être réalisée de manière automatique. Il y a d'autres choses aussi qui ne peuvent pas être réalisés de manières automatiques :

- ⇨ **Problème de Correspondance de Post :** 2 listes de mots, chaque mot étant une chaînes de caractère. Est-il possible de trouver des indices de ces 2 listes de telle sorte que les mots qui utilisent les indices de cette suite d'entier soit identique, que l'on prenne la première ou la seconde liste ?
  Exemple :
  A = {a,b}
  U = {u_1 = b, u_2 = babbb, u_3 = ba}

$V = \{v_1 = bbb, v_2 = ba, v_3 = a\}$
Je souhaite trouver une chaîne de caractères qui sont identiques dans les 2 listes comme par exemple : 2 1 1 3 telles que les mots
$u_2\, u_1\, u_1\, u_3$ = babbb b b ba
$v_2\, v_1\, v_1\, v_3$ = ba bbb bbb a
Il n'existe pas d'algorithme qui, recevant *U* et *V* comme données, permettre de décider de l'existence d'une telle suite. ➔ Problème Non Calculable (Indécidable)

⇨ **Équations Diophantiennes**
Une **Équation Diophantienne** est un polynôme à coefficient entier qui peut avoir plusieurs variables ($3x^5 + 4x^4 + 1x^2 + 9x = 0$).
Il n'existe pas de polynôme permettant de trouver les solutions entières de l'équation diophentienne ➔ Problème Non Calculable

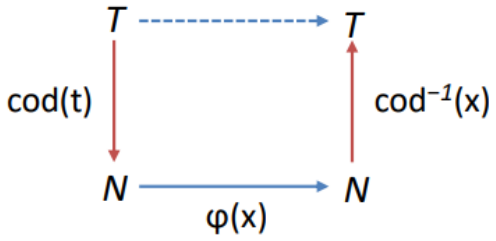
Codage et Représentation

Jusqu'à présent, nous nous sommes limités aux entiers. Pourtant, les algorithmes utilisent d'autres types de données.

Le codage permet de définir une bijection entre le nouveau type voulu et les entiers.

Propriété d'un codage cod : $T \rightarrow N$

1. la fonction cod est bijective
2. la fonction cod est calculable
3. la fonction cod $^{-1}$ est calculable



Nombres Calculables

Il y a beaucoup de trop de nombres réels par rapport aux programmes ➔ Impossible de représenter tout les nombres réels par un programme. Un **Nombre Réel** est défini comme la limite convergente de nombre rationnels.

Pour tout $x \in R$, il existe une fonction totale s : N → Q telle que la limite entre le nombre réel et la suite est 0. La fonction tend vers *x*

Un nombre réel est **Calculable** s'il existe une fonction totale calculable s : N → Q telle que je peux m'approcher aussi près que je veux du nombre. Pour n, La différence entre x et s(n) me donne une valeur approchée à 2^{-n} près de ce nombre réel. Cette fonction totale calculable me permet d'obtenir des nombres réels rationnels qui se rapprochent de ce nombre rationnel. Mais la fonction ne peut pas donner le réel directement, c'est une séquence infinie de digit donc impossible de la donner en input.

Exemple de Nombre Réels Calculables :

- Nombres Réels Algébriques
- Nombre Réels Non-algébriques π et e

⇨ Ensemble des nombres réels calculables est énumérable puisqu'en fait, nous avons autant de programmes que de nombres réels calculables, donc cet ensemble est forcément énumérables.

⇨ Comme les nombres réels ne sont pas énumérables, certains ne sont pas calculables.

- Le réel

$$x = \sum_{0 \le n \le \infty} \chi_K(n)\, 3^{-n}$$

n'est pas calculable

Chapitre 4 : Modèles

Il y a 2 familles de Modèles de Calculabilité :

- **Modèles basés sur le calcul**
  Modèles qui ont pour objectif de modéliser, formaliser le concept de fonctions calculables, d'algorithmes par un ordinateur. Certains modèles sont volontairement limitatifs (ne peuvent pas calculer tout ce qui est calculable).
- **Modèles basés sur les langages (ensemble de chaînes de caractères)**
  Modèles qui ont pour objectif d'être capable, via une description, de pouvoir proposer, définir un ensemble de mots, de chaînes de caractères. Une grammaire formelle a pour objectif de définir un ensemble de chaînes de caractères (ex : Langage Java).

Dans le cadre de ce cours, nous allons parler uniquement des modèles basés sur le calcul. Dans ces modèles basés sur le calcul, on retrouve aussi ici une distinction dans 2 sous-familles :

a) Modèles déterministes
Celui que nous avons habitude de rencontrer dans les cours de programmation. Il n'y a qu'une seule exécution possible pour un input donné

b) Modèles non-déterministes
Pour un input particulier, il peut exister plusieurs exécutions possibles, donc maybe plusieurs résultats. Il faut donner sens à ces différents résultats.

Langage de Programmation

Les Langages de Programmation sont un modèle possible de la Calculabilité. Ce sont les modèles les plus utilisés de la calculabilité. Ce modèle est en fait le modèle que nous avons utilisé pour démontrer les résultats fondamentaux de la calculabilité (Chapitre 3).

Un **programme** représenté par un langage de programmation décrit comment calculer un résultat à partir d'une donnée.

Lorsque l'on veut définir un langage de programmation comme étant un modèle de calculabilité, on doit définir :

- *La Syntaxe du langage*
- *La sémantique du langage*
- *La convention de représentation d'une fonction par un programme*

Existe-t-il des langages de programmations plus puissant que d'autres ?

= Tout les langages de programmations sont aussi puissant les uns que les autres. Si une fonction est calculable dans un langage de programmation, cette fonction l'est également dans un autre langage. Les langages sont équivalents d'un point de vue calculabilité.

Ils ont tous la caractéristique d'être **complet** : dès qu'une fonction est calculable, elle peut être calculée dans ces langages de programmation. Mais en pratique, certains langages de programmations sont + adaptés à certaines classes de problèmes.

Langage Bloop

Version limitée du langage Java, un sous-ensemble qu'on va appeler *Bloop* (*bounded bloop* = boucle bornée). Le langage *Bloop* est un programme Java tel que :

- Pas de boucle while
- Dans le corps d'une boucle for, pas de modification de la variable compteur
- Pas de méthodes récursives ni mutuellement récursives

Propriétés du Langage Bloop :

- Tous les programmes Bloop se terminent
- Bloop ne calcule que des fonctions totales
- Bloop ne calcule pas toutes les fonctions totales → Théorème de Hoare-Allison, L'interpréteur de Bloop est une fonction totale non programmable en Bloop.
- Le langage Bloop n'est pas un modèle complet de la calculabilité

Langage de Programmation non-déterministe

Nous allons définir une extension du langage Java non-déterministe :

On ajoute une fonction *choose(n)* qui renvoie un entier entre 0 et n. La particularité de cette fonction est qu'elle est **non-déterministe** : On doit considérer les différentes exécutions possibles, en fonction des résultats. Il peut y avoir des résultats différents entre les différentes exécutions, et il est possible d'avoir des exécutions finies ou infinies.

Différentes approches possibles

1. Programme ND calcule relation plutôt qu'une fonction (considérer plusieurs résultats).
2. Voir programme ND comme un moyen de décider si un élément ∈ à un ensemble

⇨ En calculabilité, on ne considère les programmes non-déterministes que pour les problèmes de décision dont la réponse unique est oui ou non. On considère toutes les exécutions possibles pour donner un seul résultat. Donc second choix

Un ensemble est **récursif de manière non-déterministe** s'il existe un programme Java ND tel que s'il reçoit une donnée un nombre naturel *x*,

- Si $x \in A$, alors il existe une exécution fournissant (tôt ou tard) comme résultat 1
- Si $x \notin A$, alors toutes les exécutions possibles fournissent (tôt ou tard) comme résultat 0

Un ensemble est **récursivement énumérable de manière non-déterministe** s'il existe un programme Java ND tel que s'il reçoit une donnée un nombre naturel *x*,

- Si $x \in A$, alors il existe une exécution fournissant (tôt ou tard) comme résultat 1
- Si $x \notin A$, alors il ne peut pas y avoir de résultat fournissant 1 (soit résultat $\neq 1$ soit ne se termine pas)

Propriétés :

⇨ **Un ensemble ND-Récursif est récursif et vice-versa**
⇨ **Un ensemble ND-Récursivement énumérable est récursivement énumérable et vice-versa**

D'un point de vue de la calculabilité, avoir un langage de programmation non-déterministe ne change rien quant à la définition de ce qui est possible de calculer. La différence théorique est que d'un point de vue complexité, dans un programme non-déterministe, la branche qui donne le bon résultat peut-être d'une longueur *n*. Alors qu'un programme déterministe nécessite une complexité exponentielle.

Automates Finis

Modélisation élémentaire du concept de *calcul* :

- Nombre fini d'états
- Lecture d'une donnée : une chaîne de caractères (réponse : OUI ou NON)
- Chaque symbole de la donnée est lu une et une seule fois
- Transitions entre états en fonction du symbole lu
- État final = état avoir lu tous les symboles de la donnée
- Pas de possibilité de mémorisation (pas de variable, de mémoire)

L'objectif d'un **Automate Fini** est de décider si un mot donné appartient ou non à un langage (ensemble de mots). La réponse sera oui ou non (pas de bouclage). Il est composé de :

➔ Ensemble de symboles Σ
➔ Ensemble d'états S
➔ État initial S_0
➔ Ensemble des états acceptants
➔ Fonction de transition

Exemple :

- Σ = { 0, 1}
- S = { pair1, impair1}
- pair1 : état initial
- impair1: état acceptant
- fonction de transition :

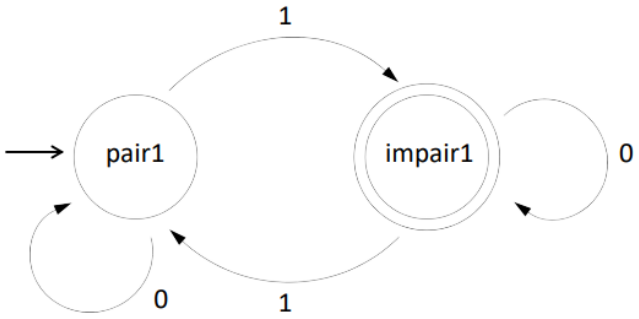
	0	1
pair1	pair1	impair1
impair1	impair1	pair1

Un automate Fini est un modèle de calcul :

- ⇨ Au départ, état initial
- ⇨ Puis on parcourt les symboles du mot d'entrée, un à un
- ⇨ À chaque symbole lu, on change d'état (fonction de transition) en fonction de l'état courant et du symbole lu
- ⇨ État final est l'état après avoir parcouru tous les symboles du mot d'entrée

On va simplifier la description d'un automate, si la fonction de transition n'est pas totale (si pour certains états, on a pas déterminé en fonction de l'input quel était l'état suivant), on va mettre un état fail, qui va permettre de continuer le calcul.

L'Automate peut être représenté par un **graphe** dans lequel les états sont les nœuds, on trouve à gauche une flèche qui indique l'état initial. Les états acceptants ont un double cercle, ceux qui ne le sont pas ont un simple cercle, les transitions sont des arcs entre les états.



Quels sont les états successifs pour l'input 10101 ?

1 : pair1 → impair1

0 : impair1 → impair1

1 : impair1 → pair1

0 : pair1 → pair1

1 : pair1 → impair1

Automate fini = détermine un langage qui est l'ensemble des chaînes de caractères qui contient un nombre impair de 1.

Automate Fini et Ensemble Récursifs

Étant donné un automate fini,

- Un mot *m* est **accepté** si après exécution, l'état final est acceptant
- Un mot *m* **n'est pas accepté** si après exécution, l'état final n'est pas acceptant

L'Ensemble Récursif est l'ensemble des mots *m* qui sont acceptés par l'automate.

Dans un automate fini, l'exécution se termine toujours. Impossible d'avoir un programme qui boucle.

Propriétés des Automates Finis

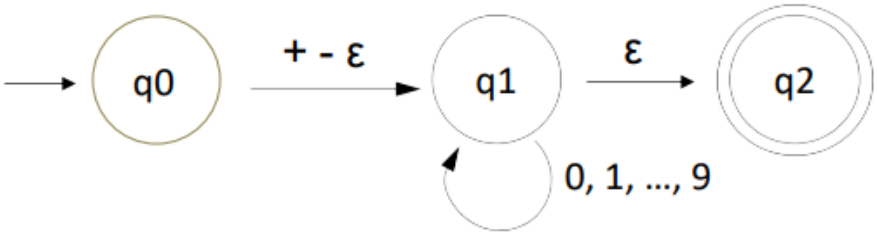
- ⇨ **Les Automates finis définissent des ensembles récursifs** (de mots)
- ⇨ **Certains ensembles récursifs ne peuvent être reconnus par un automate fini** (Conséquence du théorème de Hoare-Allison)
  Exemple : L = $a^n b^n$: Les automates n'ont pas de mémoires, il faudrait mémoriser le nombre d'occurrences de A avant de s'attaquer à la seconde partie du string.
- ⇨ **La fonction interpréteur des automates finis est calculable.** On peut la programmer en Java
- ⇨ **L'interpréteur des automates finis ne peut pas être représenté par un automate fini** (Hoare Allison)
- ⇨ **Le modèle des automates finis n'est pas un modèle complet de la calculabilité.** On ne peut pas calculer tous les ensembles récursifs.

Automates Finis : Extensions

Automates Finis non Déterministes (NDFA)

Même puissance que les automates finis déterministes, mais permettent une expression plus courte et plus simple pour certains problèmes.

On peut avoir plusieurs transitions possibles, c'est donc une **Relation de Transition**. Pour une chaîne de caractère donnée, il y aura plusieurs exécutions possibles.



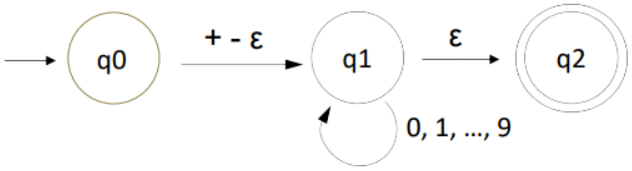
Comment savoir si un mot est accepté ou pas ?

Étant donné un Automate Fini non Déterministes :

- Un mot *m* est **accepté** s'il existe une exécution qui se termine dans un état acceptant.
- Un mot *m* **n'est pas accepté** s'il n'existe pas d'exécution qui se termine dans un état acceptant pour tout exécution.

Automates Finis avec Transition Vides (ε-NDFA)

Pas plus de puissances que les automates finis, mais représentation plus courte et est plus simple à comprendre.

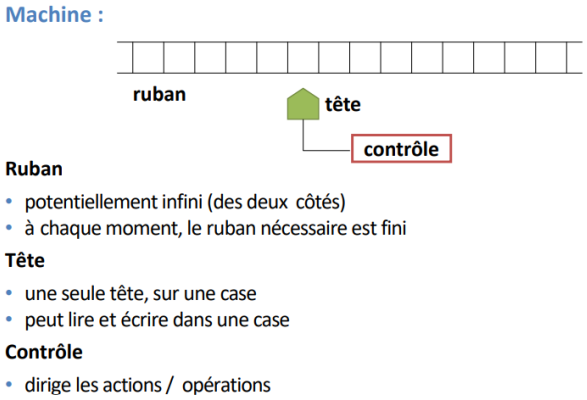


Applications des Automates Finis

- ⇨ Analyse Lexicale dans un compilateur : découpage d'un programme en symboles
- ⇨ Recherche dans un éditeur de texte
- ⇨ Interface Utilisateurs : interface pour distributeur de billets, ne pas avoir de mémoire permet d'avoir un comportement qui ne dépend pas de ce qui a été fait précédemment.

Machines de Turing

Modèle le plus simple, le plus élémentaire, mais le plus puissant possible.



Dans chaque case peut se trouver un caractère. La tête de lecture est capable de lire le caractère, et d'écrire un nouveau caractère dans une case. Il y a enfin le mécanisme de contrôle qui va diriger les opérations.

Une **Machine de Turing** comporte un nombre finis d'états. L'exécution va passer d'un état à un autre. Le premier état est celui qu'on appelle l'état initial, le dernier l'état d'arrêt. Quand elle s'arrête, le contenu du ruban est le résultat de l'exécution.

On peut voir le **Contrôle** de la machine de Turing comme un programme, un ensemble d'instructions dont la forme d'instruction est la suivante :

< q , c > → < new_q , Mouv , new_c>

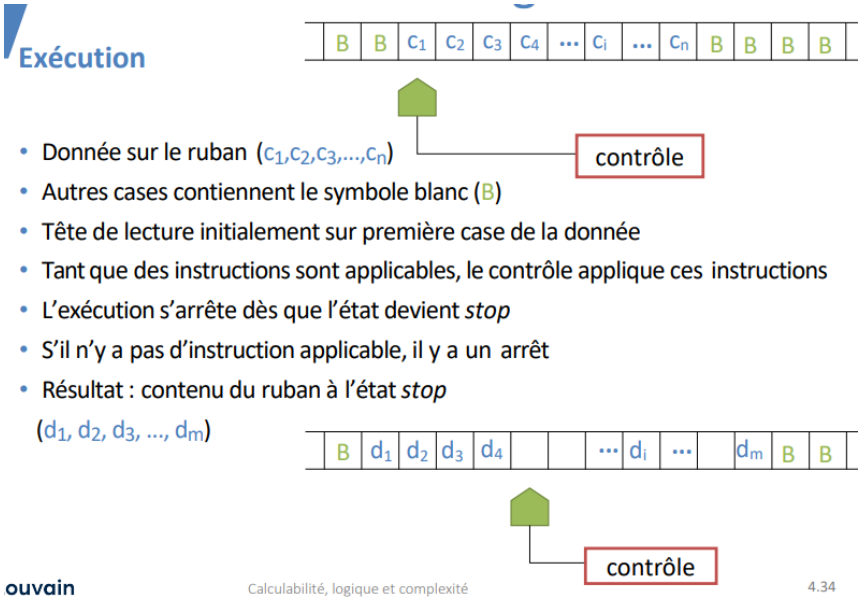
- q : état courant
- c : symbole sous la tête de lecture
- new_c : symbole à écrire sous la tête de lecture
- Mouv : mouvement (G ou D) de la tête de lecture à effectuer (aller à gauche ou aller à droite d'une case)
- new_q : état suivant (après exécution de cette instruction)

Une machine de Turing (MT) est composée de

- Σ : ensemble fini de symboles d'entrée
- Γ : ensemble fini de symboles du ruban
  - $\Sigma \subset \Gamma$
  - $B \in \Gamma, B \notin \sum$ (symbole blanc)
- S : ensemble fini d'états
- $s_0 \in S$: état initial
- stop $\in S$: état d'arrêt
- $\delta: S \times \Gamma \rightarrow S \times \{G,D\} \times \Gamma$: fonction de transition (fini)

Il est indispensable d'avoir un symbole supplémentaire, le symbole blanc, qui va être le contenu implicite de toutes les cases du ruban, sauf celles de l'input.

Exécution d'une Machine de Turing



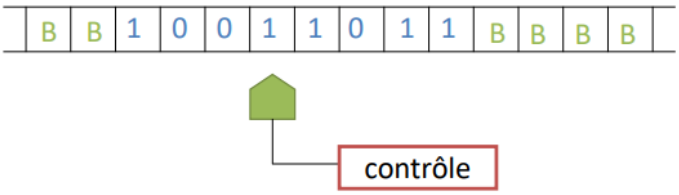
Exemple :

Machine de Turing calculant la fonction

$$f(x) = x + 1$$

Représentation des entiers

- sous forme binaire
- Σ = { 0,1 }
- Γ= { 0,1, B} pas d'autres symboles (à part B)



1. Positionner la tête de lecture sur le bit de poids le plus faible
2. Réaliser l'addition et les reports nécessaires
3. Report final et fin

Positionner tête de lecture à droite

état	symb.	état	mouv.	symb.
début	0	début	D	0
début	1	début	D	1
début	B	report	G	B

Addition

état	symb.	état	mouv.	symb.
report	0	stop	G	1
report	1	report	G	0
report	B	stop	G	1

La première étape consiste à positionner la tête de lecture à droite. Dès que l'on voit un symbole 0 ou 1, on va à Droite (*D* dans la colonne mouvement) jusqu'au moment où on arrive au symbole blanc (annoté B). Dans ce cas-là, on va à gauche (G dans la colonne mouvement). Dès que l'on est sur l'état report, c'est que l'on est sur le bit de poids le plus faible.
On passe maintenant à la partie addition. Si l'on a le symbole 0 sur la tête de lecture, on remplace le symbole 0 par 1, et on fait un mouvement vers la gauche. L'état devient stop, le résultat est mis sur le ruban. Si l'on a un symbole 1, on met un symbole 0 et on fait un report de la somme (on fait +1 sur le bit se trouvant à sa gauche). Si l'on rencontre un symbole blanc, c'est que c'est le premier blanc à gauche de mon nombre binaire, et on le remplace par un 1.

Exemple D'Exécution Machine de Turing

état	gauche	tête	droite
début		1	0011011
début	1	0	011011
début	10	0	11011
...	...	...	...
début	1001101	1	
début	10011011		
report	1001101	1	
report	100110	1	0
report	10011	0	00
stop	1001	1	100

Le début est marqué par la ligne verte.

On décale le ruban à chaque fois vers la gauche, jusqu'au moment où l'on tombe sur le premier symbole blanc. À ce moment-là, on va dans le sens inverse et on glisse vers la droite. On fait le report, le 1 est remplacé par un 0. On effectue un second report, et puis on arrive à la fin.
On tombe ensuite sur un zéro, on décale encore une fois à droite et on stop.

Il est possible de faire boucler à l'infini une machine de Turing ➔ bonne nouvelle, pour être complet, il est indispensable d'avoir des programmes qui bouclent.

Fonctions Calculables en Machine de Turing

Une fonction *f* est **Calculable en Machine de Turing (T-Calculable)** si et seulement s'il existe une machine de Turing qui, recevant comme donnée un nombre *x* :

- Fourni tôt ou tard comme résultat **si f(x) est défini**
- Ne se termine pas **si f(x) = ⊥**

Un ensemble A est **Récursif en Machine de Turing (T-Récursif)** s'il existe une machine de Turing qui, recevant comme donnée un nombre *x* fournit tôt ou tard comme résultat :

- 1 si $x \in A$
- 0 si $x \notin A$

Un ensemble A est **Récursivement Énumérable en Machine de Turing (T-Récursivement Énumérable)** s'il existe une machine de Turing qui, recevant comme donnée un nombre *x* fournit tôt ou tard comme résultat :

- 1 si $x \in A$
- Fournit un résultat différent de 1, s'arrête avec un résultat différent de *stop* ou ne se termine pas si $x \notin A$

Thèse de Church-Turing

1. Toute fonction T-calculable est calculable
2. Toute fonction calculable est T-calculable
3. Tout ensemble T-récursif est récursif
4. Tout ensemble récursif est T-récursif
5. Tout ensemble T-récursivement énumérable est récursivement énumérable
6. Tout ensemble récursivement énumérable est T-récursivement énumérable

Extensions Machines de Turing

Possibilité de modifier le modèle de base des machines de Turing

Le nouveau modèle est-il plus puissant ? ➔ Calculer plus de fonctions

Le nouveau modèle est-il plus efficace ? ➔ Moins d'étapes pour obtenir le résultat

Autres conventions :

- Possibilité de se déplacer de plusieurs cases à la fois
- Plusieurs états stop

⇨ Même puissance et speedup linéaire

Symboles et états :

- Possibilité de réduire les symboles

⇨ Même puissance et efficacité

Si l'on limite le nombre d'états (avec symboles fixés), alors seulement nombre fini de machines de Turing différentes ➔ Modèle moins puissant

Autres Rubans :

- Ruban Limité d'un côté (unidirectionnel). Arrêt si déplacement à gauche de la dernière case.
- Ruban multicases

⇨ Même Puissance et slowdown linéaire

Plusieurs Rubans :

Chaque ruban ayant sa propre tête de lecture.

⇨ Même puissance et speedup linéaire

Machine de Turing Non Déterministe :

Même principe que les automates non-déterministes. Plusieurs exécutions possibles pour une même donnée. Relation de transition finie. Uniquement utilisé pour décider si un élément appartient à un ensemble.

Ensemble Récursif et Récursivement Énumérables pour Machine de Turing

Un ensemble A est **Récursif pour une Machine de Turing (NDT-Récursif)** s'il existe une Machine de Turing non-déterministe telle qu'elle reçoit comme donnée n'importe quel naturel *x* :

- Si $x \in A$, alors il existe **UNE exécution** fournissant tôt ou tard comme résultat 1
- Si $x \notin A$, alors **toutes les exécutions** possibles fournissent tôt ou tard comme résultat 0

Un ensemble A est **Récursivement Énumérable pour une Machine de Turing (NDT-Récursivement Énumérable)** s'il existe une Machine de Turing non-déterministe telle qu'elle reçoit comme donnée n'importe quel naturel *x* :

- 1$x \in A$, alors il existe **UNE exécution** fournissant tôt ou tard comme résultat 1
- Si $x \notin A$, alors un résultat différent de 1, s'arrête avec un état différent de stop ou ne se termine pas.

Puissance et Efficacité

Un langage de programmation non-déterministe peut être simulé par un langage déterministe. À chaque Machine de Turing ND, je peux construire une machine de Turing qui a le même effet.

Speed-up exponentiel. En pratique, impossible de l'exploiter car aucun ordinateur capable de l'exécuter.

Machine de Turing avec Oracle :

Se base sur un ensemble A, où l'on va ajouter 3 états :

- **Oracle$_{ask}$** : demander à l'oracle si l'entier représenté à droite de la tête de lecture appartient à l'ensemble A
- **Oracle$_{yes}$** : l'entier appartient à A
- **Oracle$_{no}$** : l'entier n'appartient pas à A

L'état suivant n'est pas spécifié dans l'instruction. L'état sera en fonction de la réponse de l'oracle Oracle$_{yes}$ ou Oracle$_{no}$.

Même Puissance ?

- ⇨ **Si A récursif, même puissance que machine de Turing**
- ⇨ **Si A n'est pas récursif, modèle + puissant que machine de Turing**

Mais impossibilité d'exécuter un tel programme ! Quelle utilité alors ?

➔ Permet d'établir une hiérarchie parmi les problèmes indécidables (ensembles non-récursifs)

Si K est récursif, quels seraient les problèmes ?

Machine de Turing Universelle :

Comme un interpreteur de Machine de Turing.

L'interpreteur lui-même peut être représenté par une machine de Turing.

Chapitre 5 : Logique

L'objectif de la logique est :

- Représentation de connaissances
- Raisonnement automatisé sur la représentation des connaissances
- Il existe de multiples formes de logique : logique des propositions, logique des prédicats, logique modale, logique temporelle,…

Une Logique est composée de 3 parties :

1. La **Syntaxe** : détermine la forme des formules acceptées dans la logique, les phrases qui sont reconnues dans cette logique
2. La **Sémantique** : définit le sens des formules de la logique
3. Le **Raisonnement** : propose des méthodes permettant de manipuler les formules afin d'obtenir des informations pertinentes, ou de répondre à différentes questions

Logique la plus simple et la plus ancienne. Le principe de la base est la **Proposition** qui est une affirmation qui est soit vraie soit fausse. *Exemple de Proposition :*

- Le soleil brille
- Le cours de calculabilité se donne le jeudi après-midi

Partie 1 : La Syntaxe

En logique des propositions, chacune des propositions est représentée par un symbole : appelé **Variable Propositionnelle** (A,B). Et ces variables propositionnelles vont être reliées par des connecteurs logiques tels que la négation, le ET ou le OU

La Syntaxe de la Logique Propositionnelle

Constante : True et False

Variables propositionnelles : représentées par des chaînes de caractères

Connecteurs logiques : ¬ (Négation), Conjonction (∧), Disjonction (V), implication (⇒) et équivalence (⇔)

Parenthèses

Syntaxe : Formule Propositionnelle

Si l'on veut construire une formule, une variable propositionnelle est une formule propositionnelle. Si *p* et *q* sont des formules propositionnelles, il est possible de former de nouvelles formules propositionnelles ¬*p*, *p*∧*q*, *pVq*, *p =>q* et *p* ⇔*q*

Syntaxe : Conventions

Cette syntaxe particulière a quelques inconvénients comme le fait qu'il y a beaucoup de parenthèses.

- Les parenthèses externes sont supprimées
- Ordre de précédence entre les opérateurs : **¬, ∧, V, =>, ⇔**
- ∧ et V sont associatifs à gauche
- => et ⇔ sont associatifs à droite

$A \wedge B$	$(A \wedge B)$
$\neg A \wedge B$	$((\neg A) \wedge B)$
$A \wedge \neg B$	$(A \wedge (\neg B))$
$A \wedge B \vee C$	$((A \wedge B) \vee C)$
$A \Rightarrow B \vee C$	$(A \Rightarrow (B \vee C))$
$(A \Rightarrow B) \vee C$	$((A \Rightarrow B) \vee C)$
$A \vee B \vee C$	$((A \vee B) \vee C)$
$A \Rightarrow B \Rightarrow C$	$(A \Rightarrow (B \Rightarrow C))$

Partie 2 : La Sémantique

La **Sémantique** d'une formule vise à déterminer le sens de cette formule. Le sens d'une formule dépend d'un contexte où certaines propositions sont considérées comme vraies et d'autres comme fausses. Ce contexte va s'appeler l'Interprétation.

Une **Interprétation** définit un contexte possible pour une formule propositionnelle. C'est une assignation d'une valeur de vérité (*true/false*) à chacune des variables propositionnelles de la formule.

Si une formule a *n* variables différentes, il y a donc 2^n interprétations possibles.

Une fois que l'on a une interprétation, il est possible d'analyser une formule pour déterminer quelle est sa valeur de vérité dans cette interprétation.

- La valeur de vérité des constantes True et False sont respectivement true et false
- La valeur de vérité d'une variable propositionnelle A est la valeur de cette variable dans l'interprétation I

La valeur de vérité d'une formule propositionnelle composée dépend de la valeur de vérité des formules *p* et *q* dans *I*, selon la table suivante :

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

(Implication => n'est fausse que lorsque que *p* est vrai et *q* est faux)

Exemple :

- Soit l'interprétation avec l'assignation (A=*true*, B=*false*, C=*false*)
- Valeur de vérité de la formule $(A \vee B) \wedge (B \Rightarrow \neg C)$ dans cette interpretation ?
- $(true \vee false) \wedge (false \Rightarrow \neg false)$ qui se réduit à $true \wedge (false \Rightarrow true)$ qui se réduit à $true \wedge true$ qui se réduit à *true*

Une interprétation dans laquelle une formule est vraie est un **Modèle** de cette formule. Le sens d'une formule propositionnelle, c''est-à-dire sa sémantique, est défini par les différents modèles de cette formule.

La signification d'une formule est déterminée par l'ensemble de ses modèles.

Deux formules (avec même variables propositionnelles) qui ont la même signification sont **Équivalentes** si elles ont le même modèle. Lorsque 2 formules sont équivalentes, on peut remplacer une formule par une autre, au sein de n'importe quelle formule plus large.

$\neg (\neg p)$	est équivalent à	p	(double négation)
$p \wedge q$	est équivalent à	$q \wedge p$	(commutativité de $\wedge$)
$p \vee q$	est équivalent à	$q \vee p$	(commutativité de $\vee$)
$p \Leftrightarrow q$	est équivalent à	$(p \Rightarrow q) \wedge (p \Rightarrow q)$	($\Leftrightarrow$ élimination)
$p \Rightarrow q$	est équivalent à	$\neg p \vee q$	($\Rightarrow$ élimination)
$p \Rightarrow q$	est équivalent à	$\neg q \Rightarrow \neg p$	(contraposition)
$\neg (p \vee q)$	est équivalent à	$\neg p \wedge \neg q$	(de Morgan)
$\neg (p \wedge q)$	est équivalent à	$\neg p \vee \neg q$	(de Morgan)
$p \vee (q \wedge r)$	est équivalent à	$(p \vee q) \wedge (p \vee r)$	(distributivité)
$p \wedge (q \vee r)$	est équivalent à	$(p \wedge q) \vee (p \wedge r)$	(distributivité)

p	q	$\neg p$	$\neg p \vee q$	$p \rightarrow q$	$(\neg p \vee q) \leftrightarrow (p \rightarrow q)$
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

L'objectif d'une formule logique, est de pouvoir déduire de l'information, de pouvoir effectuer un raisonnement sur cette formule.

Le problème de satisfiabilité se pose la question suivante : *La formule F a-t-elle un modèle ?* Habituellement, rechercher un modèle d'une formule revient à essayer de trouver une solution de la formule *F*, une solution qui rend cette formule vraie.

Étant donné une formule *F*, quelles sont les formules qui sont des conséquences de *F* ? Étant donné *F*, on souhaite déduire d'autres formules qui sont vraies qui sont vraies partout où *F* est vraie.

Une formule propositionnelle est **Satisfaisable** si elle possède au moins un modèle. Si elle ne possède aucun modèle la formule est non-satisfaisable. On appelle dans ce cas-là la formule une **Contradiction**.

Ensemble SAT = Ensemble des Formules Propositionnelles Satisfaisables.

- ⇨ L'Ensemble SAT est un ensemble Infini
- ⇨ Une formule qui possède *n* variables peut avoir 2^n interprétations (nombre grand mais fini). Un algorithme donne la valeur *true* ou *false*.
- ⇨ Pour déterminer si une formule est satisfaisable, il suffit d'énumérer chacune des interprétations, et pour chacune des interprétations tester si la formule est vraie ou fausse.
  Si pour une des interprétations la formule est vraie, la réponse est OUI
  Si pour toutes les interprétations la formule est fausse, la réponse est NON
- ⇨ L'ensemble SAT est donc récursif

Exemple :

$$(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee C)$$

- non satisfaisable
- Pour chacune de ses 8 interpretations possibles, cette formule s'évalue à *false*

3 variables (A,B et C) ➔ 2^3 = 8 interprétations possibles

Une **Tautologie** est une formule vraie dans toutes les interprétations. Quelque soit les valeurs de vérité que j'assigne aux variables, la formule est toujours vraie (ex : A V ¬A)

p est une tautologie si et seulement si ¬*p* est non-satisfaisable.

Conséquence Logique

- ⇨ Une formule *q* est une **Conséquence Logique** d'une formule *p* si la formule p => q est une tautologie.
- ⇨ Tous les modèles de *p* sont aussi des modèles de *q*
- ⇨ La formule *q* est vraie dans tous les modèles de *p*
- ⇨ ***p* ⊨ *q***

Propriété

- $p \models q$ ssi $(p \wedge \neg q)$ est non satisfaisable

Preuve

- $p \models q$ ssi $p \Rightarrow q$ est une tautologie
- ssi $p \Rightarrow q$ est vrai dans toutes les interpretations
- ssi $\neg (p \Rightarrow q)$ est faux dans toutes les interpretations
- ssi $\neg (p \Rightarrow q)$ est non satisfaisable
- ssi $\neg (\neg p \vee q)$ est non satisfaisable
- ssi $(\neg \neg p \wedge \neg q)$ est non satisfaisable
- ssi $(p \wedge \neg q)$ est non satisfaisable

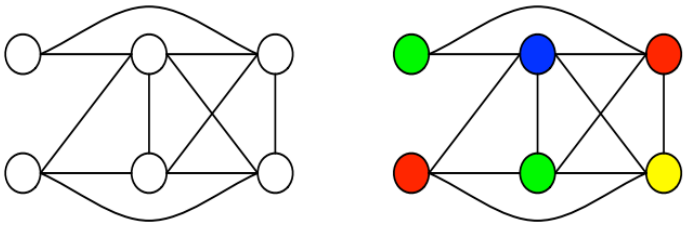
- ⇨ **Démonstration du lien fondamental entre les conséquences logiques et les formules satisfaisables**

Attention

- ➔ Si *q* n'est pas une conséquence logique de *p*, on n'a pas nécessairement que ¬*q* est une conséquence logique de *p*
- ➔ Il est donc possible d'avoir *p* !⊨ *q* et *p* !⊨ ¬*q*. C'est-à-dire qu'aucune des formules *p* => *q* et *p* => ¬*q* n'est une tautologie.

Modélisation

Problème de Coloration d'un Graphe : Comment colorier les nœuds d'un graphe de telle sorte que deux nœuds adjacents ont toujours des couleurs différentes ?



Un graphe G = (V,E) où V = ensemble des nœuds et E = ensemble des arcs, C = ensemble de couleurs. Définir les variables propositionnelles suivantes :

- Couleur[v,c] avec v ∈ V et c ∈ C
- Variable couleur[v,c] est vraie si le nœud *v* est de couleur *c*. Fausse sinon.
- Il y a #V x #C variables propositionnelles distinctes

- ⇨ Pour chaque valeur v,c nous avons une variable propositionnelles distincte

Comment représenter le graphe sous forme d'une conjonction de formules :

- Pour chaque arc(v,w) de mon graphe, la formule **¬(couleur[v,c] ∧ couleur[w,c])** empêche 2 nœuds reliés par un arc d'avoir la même couleur.
- Pour chaque nœud, la disjonction couleur[v,c] assure que chaque nœud a au moins une couleur
- Pour chaque nœud, la conjonction **¬(couleur[v,c_1] ∧ couleur[v,c_2])** pour chaque paire de couleurs (c_1,c_2) assure que chaque nœud n'a jamais plus d'une seule couleur.

Si j'ai un modèle pour cette conjonction de formules, ce modèle va représenter une solution pour la coloration du graphe. Si la formule est non-satisfaisable, le problème n'a pas de solutions.

Raisonnements

2 raisonnements pour la logique des Propositions :

a) Model Checking

Lorsque l'on veut raisonner au sujet des formules propositionnelles, il y a en fait 2 problèmes distincts que l'on peut résoudre :

- ***p* !⊨ *q* ?** Formule *q* est-elle une conséquence logique de *p* ? Est-ce que *q* est vrai partout où *p* est vrai ?
- ***p* ∈ SAT ?** Existe-t-il au moins un modèle pour cette formule *p* ? *p* est-elle satisfaisable ?

Technique de Raisonnements

- Solveurs SAT permettant de décider si une formule appartient à cet ensemble.
- Technique d'Inférence, qui est une méthode de déduction, qui à partir de règles d'inférences, permet de dire si une formule est une conséquence logique d'une autre formule.

1) Solveurs SAT élémentaire

Nous avons une donnée qui est la formule *p* avec *n* variables propositionnelles. Et l'algorithme élémentaire consiste à énumérer les interprétations *I* parmis chacune des 2^n interprétations possibles de *p*. Si la valeur de vérité dans cette interprétation *I* est vraie, alors return <true, *I*>. Sinon return <*false, ∅*>.

Forme Normale Conjonctive (CNF) qui est une suite de conjonctions de disjonctions.

$$(\neg A \vee \neg C) \wedge (\neg A \vee D) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee D) \wedge (A \vee B \vee C \vee \neg D)$$

Litéral = variable propositionnelle A ou sa négation ¬A
Clause = disjonction de littéraux
CNF = Conjonction de clauses

⇨ Toute formule propositionnelle peut se représenter sous la forme d'une CNF

$$(A \vee B) \Leftrightarrow (\neg C \wedge D)$$

$$(\neg(A \vee B) \vee (\neg C \wedge D)) \wedge (\neg(\neg C \wedge D) \vee (A \vee B))$$

$$((\neg A \wedge \neg B) \vee (\neg C \wedge D)) \wedge (C \vee \neg D \vee A \vee B)$$

$$(\neg A \vee \neg C) \wedge (\neg A \vee D) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee D) \wedge (A \vee B \vee C \vee \neg D)$$

b) *Résolution*

Transformation de formules propositionnelles en appliquant à chaque étape une **règle d'inférence**/ de transformation de manière successive. Exemple de règle d'inférence :

Equivalence

$$\frac{p \Leftrightarrow q \quad q \Leftrightarrow r}{p \Leftrightarrow r}$$

Modus ponens

$$\frac{p \Rightarrow q \quad p}{q}$$

$$\frac{\textit{prémices}}{\textit{conclusion}}$$

Si j'ai un ensemble de formules propositionnelles qui représentent la conjonction de ces formules, si les prémices appartiennent à *f*, alors on transforme *F* en *F'* en lui ajoutant la conclusion. Les règles d'inférences m'assurent que *F* et *F'* sont équivalents.

La résolution est une règle particulière mais très intéressante car à elle seule, elle permet de prouver une conséquence logique.

Règles de Résolution

Si j'ai 2 clauses c_1 et c_2 qui sont les prémices de notre application, la première clause contient un littéral positif $L_1 = V$ (variable) et la seconde un littéral négatif $L_2 = \neg V$ (négation de cette même variable). On va avoir une clause conclusion contenant tous les littéraux de C_1 et C_2, sauf les littéraux L_1 et L_2.

Exemple :

$$\frac{A \vee B \quad \neg A}{B} \qquad \frac{A \vee B \vee \neg C \quad \neg D \vee \neg B \vee F}{A \vee \neg C \vee \neg D \vee F} \qquad \frac{A \vee B \quad \neg A \vee C}{B \vee C} \qquad \frac{A \quad \neg A}{\textit{false}}$$

1er exemple : je prends la première variable A et la seconde variable ¬A (ce qui est en rouge), on les enlève et on obtient B.

- ⇨ On part d'une formule CNF sous la forme d'un ensemble de clauses
- ⇨ À chaque étape, on applique la règle de résolutions sur 2 clauses pour lesquelles la règle est applicable et dont la conclusion n'est pas une clause existante
- ⇨ Ajouter la clause conclusion à la formule CNF
  - Si la clause *false* est ajoutée à une étape, alors **formule non-satisfaisable**
  - Si la règle de résolution ne peut plus s'appliquer, alors **formule satisfaisable**

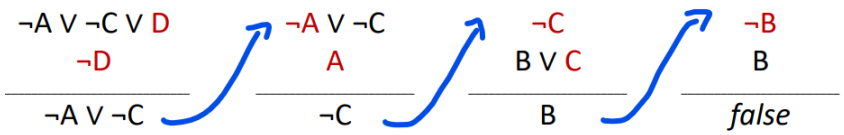
La résolution nous indique si une formule CNF appartient ou n'appartient pas à SAT

Si elle appartient à SAT, la résolution ne donne pas de modèle comme résultat

Résolution utilisée comme technique pour prouver $p \models q$

- Transformer $(p \wedge \neg q)$ en CNF
- Si résolution déduit *false* ,
  Alors $(p \wedge \neg q) \notin$ SAT , donc $p \models q$
- Si résolution se termine sans déduire *false* ,
  Alors $(p \wedge \neg q) \in$ SAT , donc $p \not\models q$

- Soit la formule p sous forme CNF (ensemble de clauses)
  {A, ¬B, ¬A ∨ ¬C ∨ D, ¬E ∨ C, B ∨ C}
- Question $p \models$ D ?
- Appliquer la résolution à $p \cup \{\neg D\}$



- Donc $p \models$ D

Chapitre 6 : Complexité Algorithmique

Notion de Complexité

On doit exiger qu'un programme soit correct par rapport à sa spécification.

On peut exiger qu'un programme correct soit efficace.

L'Efficacité est la mesure des ressources nécessaires pour que ce programme produise les résultats attendus.

La **Complexité** est le prix à payer pour résoudre le problème.

Différence entre

- Théoriquement calculable (*calculabilité*)
- Pratiquement calculable (*complexité*)

Un problème calculable peut être pratiquement infaisable

Ressources nécessaires à l'exécution d'un programme

- CPU (Processeur) : temps que le programme met à s'exécuter, nombre d'opérations à effectuer.
- Mémoire : espace nécessaire pour exécuter le programme (RAM, Disque)
- Nombre d'entrées/sorties, temps de développement, etc…µ

On mesure l'efficacité d'un programme pour un type de ressources :

1) **Complexité TEMPORELLE** : temps d'exécution
2) **Complexité SPATIALE** : espace mémoire

En pratique, complexité temporelle plus souvent plus importante que complexité spatiale. Complexité spatiale toujours inférieure à la complexité temporelle.

Algorithme VS Complexité

Quand on parle d'une complexité, on parle souvent de la complexité d'un algorithme (= temps nécessaire pour exécuter cet algorithme). Mais nous nous intéressons à l'univers des problèmes. *Comment étudier la complexité d'un problème ?*

Complexité d'un problème = complexité de l'algorithme le plus efficace résolvant le problème.

La complexité dépend de la taille des données, de la représentation des données, du modèle de calcul utilisé,…

Pour mesurer la complexité d'un algorithme, on utilise 3 types de mesures

- ***Pire Cas***
- ***Cas Moyen***
- ***Meilleur Cas***

Calcul de la Complexité

Complexité d'un Algorithme	- Jeux d'essais : - Analyse d'Algorithmes
Complexité d'un Problème	- Complexité d'un algorithme résolvant le problème - Technique de Réduction

a) Jeux d'Essais :

- Mesure du temps d'exécution : de programmes données, sur des machines données
- Permettent une comparaison : de programmes différents sur la même machine, d'un programme sur des machine différentes

⇨ Un programme peut avoir des complexités différentes sur des données différentes, mais de même taille (tableau trié/tableau pas trié)

⇨ On fait l'hypothèse que les jeux d'essais sont représentatifs de l'éventail des données réelles possibles.

b) Analyse d'Algorithmes :

- Exprimer la complexité d'un programme comme une fonction de la taille des données
- Trouver une fonction *T(n)* exprimant la complexité du programme pour une donnée de taille *n*
- *T(n)* = temps d'exécution du programme
- *T(n)* obtenu par analyse du texte du programme

c) Technique de Réduction

- Déduire la complexité d'un problème si l'on dispose d'un algorithme résolvant un autre problème. *Exemple :*

Problème 1: trouver le maximum d'une suite de nombres

Problème 2: trier en ordre décroissant une suite de nombres

Si l'on dispose d'un algorithme résolvant le problème 2,
alors on a aussi un algorithme résolvant le problème 1

(pas nécessairement le meilleur)

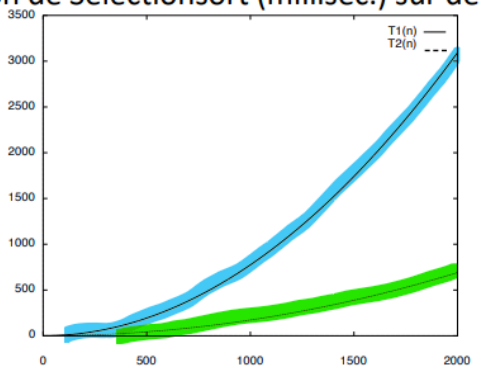
Techniques de réduction déjà beaucoup utilisées pour déterminer la (non) calculabilité de problèmes

Notation Grand O

Prenons l'exemple d'un programme de tri type *BubbleSort*. Si l'on mesure le temps d'exécution de ce programme sur des données de différentes tailles sur 2 ordinateurs différents, on voit que l'ordinateur sera plus rapide que le petit.

taille n tableau	petit ordinateur	ordinateur
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	172.9
2000	3114.9	690.5

Temps d'exécution de Selectionsort (millisec.) sur deux ordinateurs différents



Si l'on essaie de superposer à ces courbes une fonction quadratique, on va obtenir 2 fonctions quadratique différentes, mais avec des coefficients différents.

$$T_1(n) = 0.000772\ n^2 + 0.00305\ n + 0.001$$

$$T_2(n) = 0.0001724\ n^2 + 0.0004\ n + 0.1$$

$$T(n) = a\ n^2 + b\ n + c$$

Peu importe l'ordinateur utilisé, le temps d'exécution sera une fonction quadratique de la taille (n) du tableau à trier

Pour déterminer la fonction de complexité, on peut la calculer :

- ⇨ Dans le PIRE des cas
- ⇨ Dans le MEILLEUR des cas
- ⇨ Faire un calcul MOYEN

En pratique, la complexité en moyenne est très difficile à calculer car on ne dispose pas de la distribution statistique des données.
On se concentre très souvent en réalité de la complexité dans le pire des cas. La complexité dans le meilleur des cas est fort peu utile : savoir que j'ai un temps constant dans le meilleur des cas ne me donne pas de borne sur le temps que je risque d'attendre pour obtenir le résultat final.
On ne considère la complexité dans le pire des cas car c'est la plus simple à calculer et car elle nous donne la borne supérieure sur le temps d'exécution.

Classe de Complexité

Analyse d'un programme ➔ Forme générale de la courbe (pas intéressé par les coefficients)

La notation grand O va nous permettre de caractériser un ensemble d'équations/de courbes pour lesquels on ne connaît pas les coefficients, mais on aura simplement toutes les courbes de type quadratique.

Une **Classe de Complexité** caractérise une famille de courbes.

Exemple : $T(n) = an^2 + bn + c$ → $T(n)$ est $O(n^2)$

Les paramètres a,b et c dépendent de l'ordinateur utilisé. Or nous ne voulons avoir une complexité indépendante de la machine utilisée et des progrès technologique.

Objectifs

La **notation grand O** a donc pour objectif d'être une mesure indépendante des caractéristiques techniques de l'environnement technologique (vitesse du CPU, nombre d'instructions générées par le compilateur, nature des instructions).

On travaille dans le pire des cas car grand O nous donne une borne supérieure pour toute donnée de taille *n* + c'est une mesure de la complexité pour une donnée de taille *n*.

Terme Dominant

La quantité de la fonction obtenue en ne regardant uniquement que le terme du plus au degré tend vers 100% lorsque l'on augmente la taille de *n*. On ne garde que cette partie-là.

$T(n) = a\,n^2 + b\,n + c$

$T(n) = 0.0001724\,n^2 + 0.00040\,n + 0.100$

n	*T(n)*	*a n²*	*n² vs % du total*
125	2.8	2.7	94.7
250	11.0	10.8	98.2
500	43.4	43.1	99.3
1000	172.9	172.4	99.7
2000	690.5	689.6	99.9

Seul le *terme dominant* est utilisé dans la notation "grand O"

- ⇨ *On ne garde que le terme au plus degré + on ne prend pas en compte le coefficient de ce terme au plus haut degré.* Exemple : $T(n) = 2x^3 + 1x \rightarrow T(n) = O(n^3)$
- ⇨ *On se concentre sur les problèmes de grande taille et on ignore ceux de petite taille*

Définition Plus Précise

Si l'on souhaite prouver que T(n) est en $O(n^2)$, on doit prouver que $T(n) \leq xn^2$, où *x* est une constante de notre choix (1ère liberté). On peut aussi de poser cette condition sur un certain intervalle : vrai lorsque $n > n_0$, où n_0 est une valeur de notre choix (2ème liberté).

T(n) est O(g(n)) ssi

$$\exists\, c, n_0\ \forall n \geq n_0 : |T(n)| \leq c\,|g(n)|$$

- ⇨ **On ne se préoccupe pas du coefficient**
- ⇨ **On s'intéresse pour des valeurs de *n* plus grand qu'une première valeur n_0**

Propriétés

- **Grand O est une borne supérieure** : si mon temps d'exécution est en $3n^2$, il est aussi en $O(n^5)$. Quand un algorithme est $O(n^3)$, dans le pire des cas, il est inférieur ou égal à n^3.
- **Les facteurs sont totalement inutiles** : on n'écrit jamais $O(2n^2)$ ➔ $O(n^2)$
- **Les Termes d'ordre inférieurs sont négligeables** : $n^5 + n^3 = O(n^5)$
- **Bases logarithmes inutile** : $\log_2 n = O(\log_{10} n) = O(\log n)$

Transitivité

Si $T(n) = O(f(n))$

et $f(n) = O(g(n))$

Alors $T(n) = O(g(n))$

$4n^2 = O(n^2)$ et $n^2 = O(n^3)$

Donc $4n^2 = O(n^3)$

Limites

Si $\lim_{n \to \infty} \frac{g(n)}{f(n)} = a$ *(avec* $a \neq \infty$*)*

Alors $g(n) = O(f(n))$

$g(n)$ ne croît pas plus vite que $f(n)$

(Attention, l'inverse n'est pas nécessairement vrai)

Application

$n^2 = O(2^n)$

$n^{1000} = O(2^n)$

$\log_2 n = O(\sqrt{n})$

$(\log_{10} n)^{100} = O(n)$

$2^n = O(3^n)$

Sommation

Si

- $T_1(n) = O(f_1(n))$
- $T_2(n) = O(f_2(n))$
- $f_2(n) = O(f_1(n))$

Alors $T_1(n) + T_2(n) = O(f_1(n))$

Problèmes avec notation Grand O

La notation grand O n'est pas très précise :

1) Pour une complexité en $O(n^3)$, rien ne me dit que la complexité est également en $O(n^2)$, qui est une estimation + favorable du temps d'exécution. Dans le pire des cas, c'est borné par $O(n^2)$
2) Existence d'autres notations : borne inférieure $\Omega(n)$ et exacte $\Theta(n)$ de complexité

Hiérarchie des Complexités

Nom	notation
Constante	$O(1)$
Logarithmique	$O(\log n)$
Linéaire	$O(n)$
n log n	$O(n \log n)$
Quadratique	$O(n^2)$
Cubique	$O(n^3)$
Exponentielle	$O(2^n)$
Exponentielle	$O(10^n)$

En vert : Complexité Polynomiales et en bleu : Complexité Exponentielles. Les algorithmes de complexité exponentielle sont intrinsèquement complexes.

Complexité : temps d'exécution

- Hypothèses:
  - ordinateur 100.000 Mips
  - traitement de 1 élément = 1.000 instructions machine

	10	50	100	500	1000	10000
n	.0000001 sec.	.0000005 sec.	.000001 sec.	.000005 sec.	.00001 sec.	.0001 sec.
n^2	.000001 sec.	.000025 sec.	.0001 sec.	.0025 sec.	0.01 sec.	1 sec.
n^3	.00001 sec.	.00125 sec.	0.01 sec	1.25 sec.	10 sec.	2.8 heures
n^5	.001 sec.	3.13 sec.	1.67 min.	3.6 jours	115.7 jours	31.7 siècles
2^n	.0000102 sec.	130 jours	$4x10^{12}$ siècles	...	...	...
3^n	.00059 sec.	$2x10^5$ siècles	...	...	...	...

vain

Algorithmes Constants	- Imprimer un élément d'un tableau - Ajouter un élément à une queue - Rechercher un élément dans une table
Algorithmes Logarithmiques	- Recherche dichotomique dans un tableau - Recherche dans un arbre binaire de recherche
Algorithmes Linéaires	- Recherche d'un élément dans un tableau non-trié
Algorithmes n log n	- Tri par tas, quicksort
Algorithmes Quadratiques	- Tri par échange - BubbleSort - Tri par insertion - Addition de 2 matrices *n x n*
Algorithmes Cubiques	- Multiplication de 2 matrices *n x n*
Algorithmes Exponentiels	- Voyageur de commerce - Coloration de graphes

Problèmes Intrinsèquement Complexes

Un **Problème Intrinsèquement Complexes** est un problème qui est théoriquement calculable, mais en pratique infaisable car il nécessite des ressources incompatibles avec les réalités physiques.

- S'il existe un algorithme de complexité polynomiale, alors le problème pratiquement faisable.
  Existe algo complexité polynomiale → pratiquement faisable
- S'il n'existe pas d'algorithme de complexité polynomiale, alors le problème est pratiquement infaisable.
  N'existe pas d'algo complexité polynomiale → pratiquement infaisable

Un problème est intrinsèquement complexe si et seulement s'il n'existe pas d'algorithme de complexité polynomiale résolvant le problème. Un problème intrinsèquement complexe est pratiquement infaisable. Beaucoup de problèmes intéressants sont intrinsèquement complexes.

Pourquoi est-ce gênant d'avoir un problème qui est intrinsèquement complexe ?

Car si par exemple on a un algorithme de complexité exponentielle, quel que soit la complexité polynomiale que l'on envisage, ma complexité exponentielle sera de toute façon plus élevée que n'importe quel polynôme.

Regardons de plus près ce qui se passe si les ordinateurs deviennent de plus en plus puissants.

Supposons N_i la taille de la plus grande donnée qui peut être calculée en 1h de temps calcul. Que peut-on résoudre comme problème en 1h de temps calcul avec N_i ?

Si la complexité est linéaire, on peut résoudre des problèmes extrêmement grands. Avec des ordinateurs 100x et 1000x plus puissants, on peut résoudre le même problème avec une augmentation de la donnée avec un facteur multiplicatif.

Complexité	Ordinateur d'aujourd'hui	Ordinateur 100x	Ordinateur 1000x
n	$N_1 = 3.6x10^{11}$	$100\ N_1$	$1000\ N_1$
n^2	$N_2 = 60000$	$10\ N_2$	$31.6\ N_2$
n^3	$N_3 = 7110$	$4.64\ N_3$	$10\ N_3$
n^5	$N_4 = 205$	$2.5\ N_4$	$3.98\ N_4$
2^n	$N_5 = 38$	$N_5 + 6$	$N_5 + 10$
3^n	$N_6 = 24$	$N_6 + 4$	$N_6 + 6$

Pour les complexités exponentielles, la taille du problème que l'on peut résoudre est très petite, et on voit que pour des ordinateurs 100x et 1000x plus grands, on sait résoudre le même problème avant + quelques unités. Le facteur ajouté est ici additif.

⇨ Peu importe les évolutions technologiques, un problème intrinsèquement complexe ne peut et ne pourra être résolu que pour de petits exemples

Influence du Modèle de Calcul

Le modèle de calcul influence-t-il le fait qu'un algorithme soit efficace ou non-efficace ?

Si un algorithme, exprimé dans un modèle de calcul particulier, est de complexité polynomiale, alors il sera également de complexité polynomiale dans un autre modèle de calcul (complexité spatiale et temporelle).

Exemple : Si pas d'algorithme polynomial en Java, alors pas d'algorithme polynomial en Machine de Turing.

- ⇨ **La classe de problèmes intrinsèquement complexes est donc indépendante du modèle de calcul.**
- ⇨ **Il est inutile de se focaliser sur un modèle de calcul en particulier pour déterminer qu'un problème est intrinsèquement complexe.**
- ⇨ **Si dans 20 ans on invente un nouveau modèle de calcul, un problème intrinsèquement complexe aujourd'hui le sera aussi dans 20 ans.**

Influence de la représentation des données

La manière dont on représente les données dans l'algorithme, ne va rien changer du caractère intrinsèquement complexe de mon problème.

Remarques :

- Un problème est intrinsèquement complexe car parmi toutes les données de taille n, il existe au moins une pour laquelle le temps d'exécution est exponentiel, ou pire.
- Pour certains problèmes intrinsèquement complexes, il existe des algorithmes polynomiaux pour presque toutes les données de taille n.
- Les cas où une donnée de taille n donne lieu à une complexité exponentielle peuvent parfois correspondre à des cas particuliers que l'on rencontre peu en pratique.
- Il existe des algorithmes (efficaces) permettant de calculer des approximations de la solution exacte.

Chapitre 7 : Classes de Complexité

L'objectif de base d'une **Réduction** est, à partir d'un algorithme d'un autre problème *P'*, de déduire un autre algorithme pour un autre problème *P*.

Pour nous faciliter la vie, nous allons nous limiter aux **Problèmes de Décision**, qui sont des problèmes qui nous permettent de décider si l'input appartient ou n'appartient pas à un ensemble donné. L'objectif d'une réduction revient donc à déduire un algorithme de décision (oui/non) pour décider un ensemble *E* à partir d'un algorithme de décision pour un autre ensemble *E'*

Cette technique de réduction va être utilisée pour 2 objectifs très différents :

- **Objectif de Calculabilité : déduire de cette réduction des propriétés sur la calculabilité ou non-calculabilité d'un problème.**
- **Obtenir des informations sur la complexité d'un problème.**

En fonction de l'objectif que l'on poursuit, on va voir différentes méthodes de réductions

- ⇨ Réduction Algorithmique : déduire des propriétés sur la calculabilité
- ⇨ Réduction Fonctionnelle : déduire des informations sur la complexité
- ⇨ Réduction Polynomiale : déduire des informations sur la complexité

Réduction Algorithmique

Un ensemble A est **Algorithmiquement Réductible** à un ensemble *B* si en supposant B récursif, alors A est récursif. On dénote cela A ≤ B, A est plus facile à résoudre que B d'un point de vue algorithmique. ➔ Relation Réflexive (= un ensemble est réductible à lui-même A ≤ A) et Transitive (= si A ≤ B et B ≤ C alors A ≤ C).

Ensemble Complet

Un **Ensemble Complet** d'une classe d'ensembles est l'ensemble le plus difficile à décider parmi tout les ensembles de la classe. Le problème s'appelle le Problème Complet par rapport à la classe A (A-Complet). Ce problème complet doit respecter 2 conditions :

1. **E ∈ A** (doit appartenir à la classe A)
2. **∀ B ∈ B : B ≤ E** (Ce problème doit être le plus difficile que tous les autres de la classe)

En trouvant cet algorithme et en construisant un algorithme pour ce point rouge, on va pouvoir en déduire des algorithmes pour tous les autres éléments de la classe.

Cas Particulier :

Si on a un ensemble qui se trouve à l'extérieur de A mais qu'il a la propriété d'être plus difficile que tout les éléments, on dit que ce problème E est **A-difficile** par rapport à une relation de réduction. Une seule condition :

1. **∀ B ∈ B : B ≤ E** (Ce problème doit être le plus difficile que tous les autres de la classe)

⇨ On n'exige pas que le problème soit dans la classe de problème, il peut l'être comme il ne peut pas l'être.

Propriétés

- Si A $\leq_a$ B et B récursif, alors A récursif
- Si A $\leq_a$ B et A non récursif, alors B non récursif
- A $\leq_a \bar{A}$
- A $\leq_a$ B $\Leftrightarrow$ $\bar{A} \leq_a \bar{B}$
- Si A récursif, alors pour tout B, A $\leq_a$ B
- Si A $\leq_a$ B et B récursivement énumérable, alors A *pas nécessairement* récursivement énumérable

Exemples

- DIAG $\leq_a$ HALT
- HALT $\leq_a$ DIAG
- *HALT* est r.e.-complet par rapport à $\leq_a$ (r.e. = classes des ensembles récursivement énumérables)
- *DIAG* est r.e.-complet par rapport à $\leq_a$

Réduction Fonctionnelle

Un ensemble A est **fonctionnellement Réductible** à un ensemble B ($A \leq_f B$) si et seulement s'il existe une fonction totale calculable *f* telle que **$a \in A \Leftrightarrow f(a) \in B$** (décider l'appartenance à A revient à tester l'appartenance de f(a) à B).

Attention : implication dans les 2 sens : si f(a) appartient à B, on peut déduire que a appartient à A, et si f(a) n'appartient pas à B, on peut déduire que a n'appartient pas à A.

Pour décider si un élément appartient à A, il suffit de calculer f(a) et tester si $f(a) \in B$.

Si l'on sait que B a une complexité en $O(n^2)$, peut-on deviner la complexité de A ?

On sait que le test $f(a) \in B$ sera de l'ordre de $O(n^2)$. On n'a cependant pas d'informations sur le temps nécessaire pour calculer f(a).
Donc la complexité de *a* = complexité du calcul de *f* + la complexité de *B*

Propriétés

- Si $A \leq_f B$ et B récursif, alors A récursif
- Si $A \leq_f B$ et A non récursif, alors B non récursif
- Si $A \leq_f B$ et B récursivement énumérable, alors A récursivement énumérable
- $A \leq_f B \Leftrightarrow \bar{A} \leq_f \bar{B}$
- Si A récursif, alors pour tout B, $A \leq_f B$
- Pas nécessairement $A \leq_f \bar{A}$
- Si $A \leq_f B$, alors $A \leq_a B$
- $A \leq_a B$ n'implique *pas nécessairement* $A \leq_f B$

Exemples

- $DIAG \leq_f HALT$
- $HALT \leq_f DIAG$
- *DIAG* est r.e.-complet par rapport à $\leq_f$

Différence entre Réduction Algorithmique et Fonctionnelle

$\leq_a$ = Réduction Algorithmique

$\leq_f$ = Réduction Fonctionnelle

➔ **Réduction Algorithmique ($\leq_a$) :**
Point de vue calculabilité.
Pour décider si $a \in A$, on peut utiliser quand on veut et autant de fois que l'on veut, le fait que B soit récursif. *Est-ce que quelque chose est récursif ou non ? calculable ?*

➔ **Réduction Fonctionnelle ($\leq_f$) :**
Point de vue complexité.
Pour décider si $a \in A$, le schéma d'algorithme est le suivant :

```
input a
a' := f(a)
si a' ∈ B alors output OUI (a ∈ A)
        sinon output NON (a ∉ A)
```

On calcule f(a) sur la donnée, puis on teste l'appartenance de a à B pour l'élément calculé. La réponse est OUOI si $a \in A$, sinon c'est NON.

Si on connaît la complexité de la fonction *f* et du problème de décision de l'ensemble *B*, alors on peut déduire des information sur la complexité de A.

Modèle de Calcul

Théorie "classique" de la complexité:

- utilisation du modèle des machines de Turing
- permet une définition précise de complexité temporelle (nombre de transitions)
- permet une définition précise de complexité spatiale (nombre de cases utilisées)

Mais

- modèle peu intuitif pour un informaticien
- complexité d'une machine de Turing est précise mais éloignée d'une complexité "en pratique"
- on s'intéresse au ordre de grandeur (grand *O*)
- on s'intéresse à la frontière : problèmes intrinsèquement complexes
- modèle des machines de Turing simple à utiliser dans certaines preuves

Possibilité d'utiliser le modèle des **programmes Java**

- déterministes
- non déterministes

En pratique, utiliser le modèle le plus adéquat à la situation à traiter

- Machine de Turing pour les preuves complexes
- Langage Java pour les définitions et les intuitions

Quel que soit le modèle utilisé, il y a un rapport polynomial entre une machine de Turing et une complexité d'un programme Java qui exécute la même chose. **Comme on s'intéresse aux problèmes pratiquement faisables (= ceux qui ont un algo polynomial), Il existera un algo polynomial en Java et en Machine de Turing et vice-versa.**

Tous les modèles de complexité ont entre eux des complexités spatiales et temporelles reliées de façon polynomiale. **Si un problème est pratiquement faisable dans un modèle (non déterministe), alors il est pratiquement faisable dans tout les modèles (non déterministes).**

Classes de Complexité

a) Classes basées sur un modèle déterministe

- **DTIME(f)**
  Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité temporelle *O(f)*.

- **DSPACE(f)**
  Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité spatiale *O(f)*.

Remarque : dès que l'on parle de complexité temporelle, les programmes se terminent toujours. Un programme qui boucle n'a pas de complexité. Donc dès que l'on parle de complexité, on parle d'ensemble récursifs. On aura toujours une réponse (*oui/non*)

Remarque : Nous parlons de complexité ici par rapport aux programmes Java.

DTIME(n^2) en programme Java n'est pas la même chose que DTIME(n^2) en Machine de Turing. Si l'on a un programme Quadratique en Java, rien ne dit qu'il le sera aussi en Machine de Turing.
Seule chose dont on est sûr est que si c'est Polynomial en Java, ça le sera aussi en Machine de Turing (peut-être avec un exposant différent)

b) Classes basées sur un modèles non-déterministe

- **NTIME(f)**
  Famille des ensembles ND-récursifs pouvant être décidés par un programme Java non-déterministes de complexité temporelle *O(f)*.
  On va considérer la complexité de la branche d'exécution la plus longue.

- **NSPACE(f)**
  Famille des ensembles ND-récursifs pouvant être décidés par un programme Java non-déterministe de complexité spatiale *O(f)*.

Non-déterministes : Plusieurs exécutions possibles. Complexité = les programmes se terminent toujours. Donc ici toutes les exécutions se terminent tous (toutes les branches).

Classe P

- Union de tous les DTIMES polynomiaux
- P = U DTIME(n^i)
- Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité temporelle polynomiale
- Indépendante du modèle de calcul

Classe NP

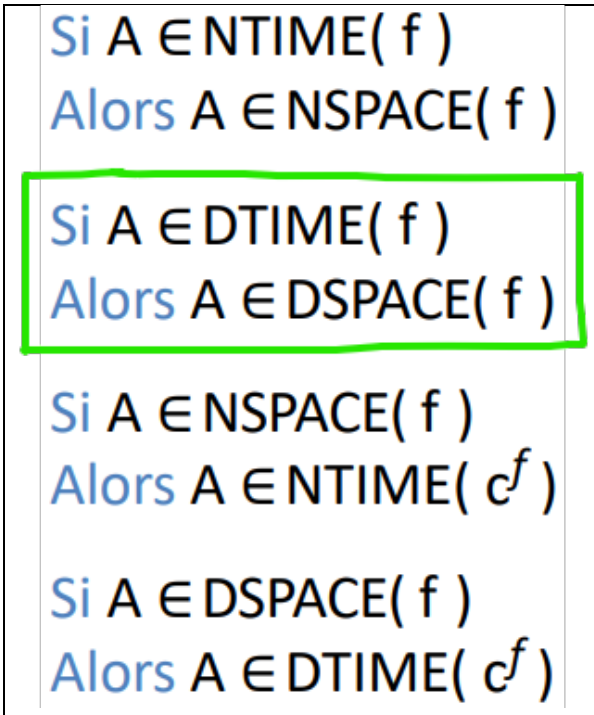
- Union de tous les NTIMES polynomiaux
- NP = U NTIME(n^i)
- Famille des ensembles ND-récursifs pouvant être décidés par un programme Java non-déterministe de complexité temporelle polynomiale

Relations entre Classes de Complexité

Si $A \in NTIME(f)$ Alors $A \in DTIME(c^f)$

Si $A \in NSPACE(f)$ Alors $A \in DSPACE(f^2)$

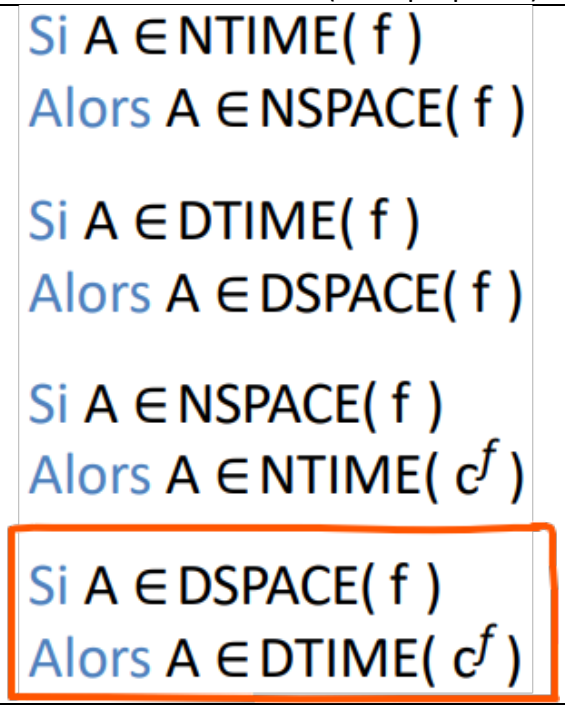
Existe-t-il des relations entre les classes de complexité du même modèle ?

<table>
<tr>
<td> 
</td>
<td>La seconde propriété est importante. Si par exemple on a un problème ayant une complexité DTIME(n^3), on est assuré que sa complexité spatiale déterministe est également bornée par n^3. Il ne peut pas utiliser plus que n^3 cellules mémoires.
Pourquoi ?

À chaque instruction exécutée, dans le pire des cas on va utiliser 1 cellule mémoire en +. Il n'est pas possible de consommer plus de cases mémoires que d'instructions exécutées puisqu'à chaque instruction on ne peut consommer qu'une case en plus.

C'est pour cela que la complexité temporelle est une borne de la complexité spatiale.

⇨ Même résultat pour une complexité non-déterministe (la 1er propriété)</td>
</tr>
<tr>
<td>Si l'on sait que la complexité spatiale est linéaire, qu'est-ce qu'on peut en déduire sur sa complexité temporelle ?

La Complexité Temporelle est de l'ordre de par exemple 2^n, est bornée par 2^n. D'un point de vue pratique, ce résultat ne sert à rien car avoir une complexité exponentielle n'est pas un bon résultat.</td>
<td> 
</td>
</tr>
</table>

Hiérarchie de Complexités

Pour toute fonction totale calculable *f*, il existe un ensemble A récursif tel que A ∉ DTIME(f).

Peu importe la complexité que l'on envisage, même si l'on considère une complexité gigantesque, il est toujours possible de trouver un problème plus complexe que l'a complexité que l'on a donné (sa complexité ne sera pas bornée par *f*, complexité encore pire que *f*).

NP-Complétude

Question Fondamentale de la Théorie de la Complexité :

S'il existe un algorithme non-déterministe polynomial, existe-t-il alors un algorithme déterministe polynomial résolvant ce même problème ?

Si un ensemble récursif A appartenant à la classe NP (=pour lesquels il existe un algorithme non-déterministe polynomial),

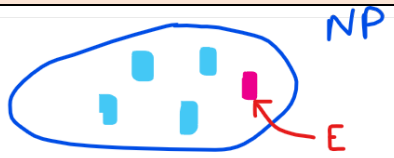
Est-ce que ce problème-là appartient aussi à la classe P ?

⇨ Si c'est le cas, alors **P = NP ?**

On sait que $P \subseteq NP$

Comment démontrer

- $NP \subseteq P$
- Ou non ($NP \subseteq P$)



Prendre un élément E "représentatif" de *NP* (le plus difficile) et essayer de démontrer On va montrer uniquement pour E

- $E \in P$ ⟶ Alors P = NP
- Ou $E \notin P$ ⟶ Alors P ≠ NP

Choisir un élément E qui soit *NP*-complet par rapport à une relation de réduction ≤

- $E \in NP$
- $\forall B \in NP : B \leq E$

Quelles relation de réduction choisir ? ≤a ≤f

1) **Réduction Algorithmique**
   Ne donne aucune informations sur la complexité. Or l'objectif est de, si je suis capable de démontrer que E est dans P, alors P = NP. Tous les autres problèmes de NP ont aussi un algorithme polynomial déterministe.

2) **Réduction Fonctionnelle**
   Permet de donner une information sur la complexité temporelle. Il existe une fonction totale calculable telle que pour décider si une fonction a appartient à A, il suffit de calculer f(1) et de tester si f(1) appartient à B.

Réduction Polynomiale

Un ensemble A est **Polynomialement Réductible** à un ensemble B ($A \leq_p B$) s'il existe une fonction totale calculable *f* de complexité temporelle polynomiale telle que $a \in A \Leftrightarrow f(a) \in B$.

⇨ $\leq_p$ est une relation réflexive et transitive
⇨ Si $A \leq_p B$ et $B \in P$ alors $A \in P$
⇨ Si $A \leq_p B$ et $B \in NP$ alors $A \in NP$

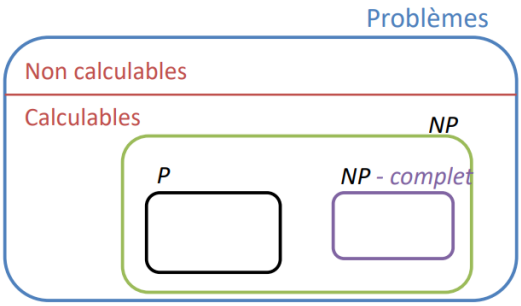
NP-Complétude

Un problème E est **NP-Complet** (par rapport à $\leq_p$) si

1. **E ∈ NP**
2. **∀ B ∈ NP : B $\leq_p$ E**

Un problème E est **NP-Difficile** (par rapport à $\leq_p$) si

1. **∀ B ∈ NP : B $\leq_p$ E**



SAT = Ensemble des formules propositionnelles qui sont satisfaisables.

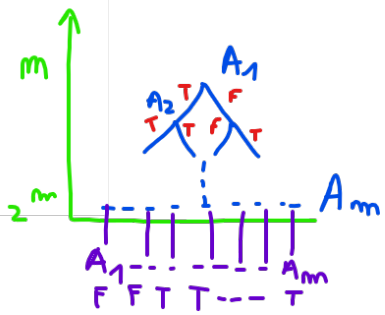
La logique Propositionnelle est composée de :

- **Connecteurs Logiques** : ¬, ∧, V, =>, ⇔, ()
- **Variables A_i** représentées par l'entier *i* (décimal/binaire)
- **Une Formule** W(A_1,…,A_m) est **Satisfaisable** s'il existe des valeurs logiques (true/false) pour les variables A_1,…,A_m tel que W(A_1,…,A_m) est vrai. La longueur d'une formule W est représenté par *n* (si *n* = nombre d'occurrences des variables, alors la taille(W) = O(n log n)). Le nombre d'occurrences est d'office plus grand que le nombre de variables.

Théorème de Cook

Le théorème de Cook dit que SAT est NP-Complet. Pour cela, il faut prouver 2 choses :

1. **SAT ∈ NP**
   Il faut montrer qu'il existe un algorithme qui permet de décider si une formule est satisfaisable par un algorithme non-déterministe de complexité non-polynomiale.
   - Générer (de façon non déterministe) une séquence de m valeurs logiques
     Complexité: O(m) avec m ≤ n
   - Substituer les occurrences des variables A_i par leur valeur
     Complexité : *O*(n log n)
   - Evaluer l'expression (par technique de réduction)
     Complexité polynomiale



⇨ Exécution non-déterministe car plusieurs branches
⇨ Si un *true* formule satisfaisable, si toutes les branches *false* formule non-satisfaisable
⇨ Profondeur totale polynomiale, cela appartient bien à NP

2. **$\forall B \in NP : B \leq_p SAT$**
   Quel que soit le problème qui est dans NP, ce problème peut se réduire de manière polynomiale à mon ensemble SAT.
   Il existe une machine de Turing non-déterministe M qui reconnaît en un temps polynomial p(n), si élément *x* appartient à B (*p* = polynôme, *n* = longueur donnée *x*). Si l'on donne une donnée x, la machine de Turing va construire un arbre dont la profondeur de cet arbre est *p(n)*.
   On doit démontrer qu'il existe une transformation totale calculable de complexité polynomiale. Cette transformation transforme *x* en une formule appelée W_x
   $x \in B \Leftrightarrow W_x \in SAT$
   On peut tester si W_x est satisfaisable pour déterminer si *x* appartient à B. à partir de la donnée *x* et d'une machine de Turing, on va construire une immense formule propositionnelle dont l'objectif est de simuler le fonctionnement de la machine de Turing sur *x*. W_x sera satisfaisable si M s'arrête dans un état satisfaisant.

Quelques Problèmes NP-Complets

Comment prouver la N-Complétude ?

2 techniques :

1. Réaliser une preuve semblable à celle pour SAT (difficile)
2. Utiliser la propriété : **Si $E \leq_p B$ avec E NP-Complet et $B \in NP$ alors B est NP-Complet.**
   Les problèmes considérés sont des problèmes de décision : la réponse est oui ou non, l'élément appartient (pas) à l'ensemble.

(Voir exemples d'applications dans les slides 37-41)

P = NP ?

Similitudes avec Church-Turing. Il y a eu beaucoup d'efforts pour trouver des algorithmes polynomiaux pour les problèmes NP-Complets, mais échec. Il y a de fortes présomptions que les problèmes NP-Complets soit effectivement intrinsèquement complexes P ≠ NP, mais pas encore de preuves à l'heure actuelle.

Que signifie qu'un problème est NP Complet ?

Le problème n'a pas de solution algorithmique polynomiale, mais il s'agit de la complexité dans le pire des cas. Possible que la complexité soit polynomiale pour la plupart des données.

Comment résoudre un problème NP Complet ?

- ➔ Il ne faut pas abandonner. Il faut modifier le problème en un problème plus simple, le particulariser pour certaines instances.
- ➔ Utiliser un algorithme dans le pire des cas si la plupart des instances à résoudre sont de complexité polynomiale
- ➔ Utiliser la technique d'exploration, de recherche. Plutôt que de rechercher la meilleure solution, chercher une solution approximative. Algorithme incomplet.

Transition de Phase

Lorsque l'on aborde un problème NP-Complet, pour certaines instances le problème est très facile et pour d'autres très difficile. Est-il possible de caractériser les instances facile des difficiles ?

- ⇨ Possible de le faire à l'aide d'un **Paramètre de Contrôle**. En fonction de la valeur du paramètre, on aura 2 types d'instances :
- ⇨ Instances où presque tout est solution
- ⇨ Instances où solution quasi inexistante
- ⇨ Transition : située entre ces 2 régions d'instances

La valeur du paramètre va déterminer si l'on est dans la première ou seconde région, et quand on est entre les deux, on est dans ce qu'on appelle dans une **Transition de Phase**

Problème SAT : pouvoir déterminer si une formule est satisfaisable. Cette formule est composée de clause, reliée par des haies. Chaque clause est une composition de littéraux.

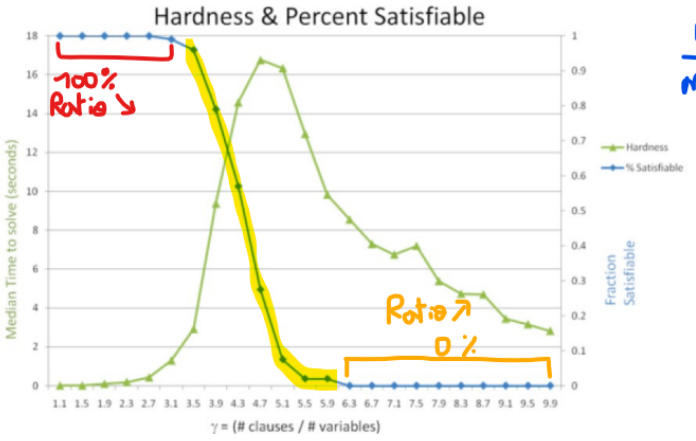
Nous allons mesurer SAT par un ratio de 2 valeurs :

1. Le nombre de clauses *n*
2. Le nombre de variables qui apparaissent dans tout les littéraux et clauses

Transition de phase : SAT $C_1 \wedge C_2 \wedge \dots \wedge C_m$, $L_1 \vee L_2 \vee L_m$

Paramètre de SAT : ratio entre nombre de clauses et nombre de variables

- Ratio faible pour SAT : instances (presque) toujours satisfaisables
- Ratio élevé pour SAT : instances (presque) toujours non satisfaisables



nb clauses / nb variable

https://arxiv.org/pdf/1304.0145

➔ Si le ratio est faible, c'est qu'on a très peu de clauses par rapport aux nombre de variables. Cela veut dire qu'on a beaucoup de variables *L*, et vu que celles-ci sont entourées de V (ou), on aura très probablement une qui sera vrai → Simple à résoudre.

➔ Si le ratio est élevé, beaucoup de clauses (entourées de ET ∧) ce qui veut dire que c'est beaucoup plus dur à résoudre.

On peut voir sur le graphe que lorsque le ratio est faible, on est à 100% de satisfiabilité. Et si le ratio est élevé on est à 0% de satisfiabilité. Transition rapide très rapide entre des formules toujours satisfaisable et jamais satisfaisable.

On aperçoit aussi que c'est dans la transition de phase (Transition entre les 2 ratios, de faible à élevé) que la complexité est la plus élevée. La Transition de Phase permet d'analyser les instances et permet d'avoir une indication sur les familles d'instances qui demandent le plus de temps d'exécution.

Le **Pic de Difficulté** est la petite variation du paramètre, lors du passage d'instances faciles à très difficiles. Ce pic est indépendant de l'algorithme choisi. Si l'on est dans ce pic de difficulté, ça ne sert à rien d'imaginer un algorithme.

Autres Classes de Complexité

Classes Co-NP : Ensemble ND-Récursifs dont le complément appartient à NP

Classe EXPTIME : Ensemble récursifs qui peuvent être décidés par un algorithme de complexité temporelle $O(2^P)$, où P est un polynôme.

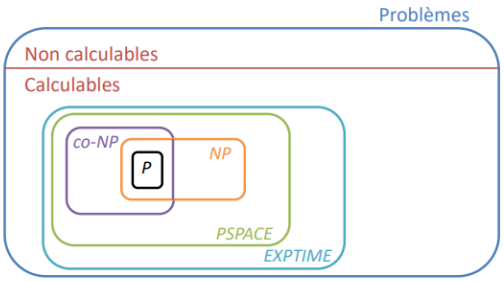
Classe LOGSPAGE : Ensemble récursifs qui peuvent être décidés par un algorithme de complexité spatiale O(log)

Classe PSPACE : Ensemble récursifs qui peuvent être décidés par un algorithme de complexité spatiale O(p), où P est un polynôme.

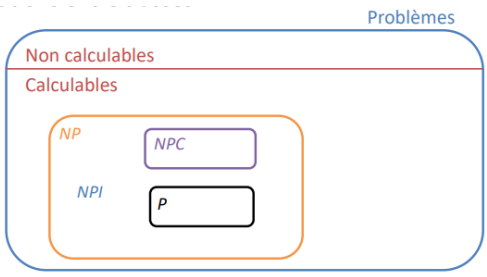
Classe NSPACE : Ensemble ND-récursifs qui peuvent être décidés par un algorithme de complexité spatiale O(p), où P est un polynôme.

Propriété : **PSPACE = NPSPACE**

Théorème qui dit que pour passer d'un algorithme déterministe à un non-déterministe, il y a une perte quadratique d'espaces, et comme ce sont des polynômes, cela revient à la même chose.



NP = PSPACE ? problème ouvert
On sait que *P* ≠ EXPTIME
Si *P=NP*, alors EXPTIME=NEXPTIME



NPC : problèmes NP-complets
NPI = NP \ NPC \ P
On peut montrer que NPI est non vide (supposant *P* ≠ *NP*)

NPC = NP-Complet

NPI = NP-Incomplet : pas très connue, très rare. ex : isomorphisme de graphes

P = problème pratiquement faisable. Si pas dans P, problème pratiquement infaisable.

Attention, il en existe des milliers d'autres classes de problèmes, non citées dans le cours.

Chapitre 8 : Analyses et Perspectives

Thèse de Church Turing (forme originale)

- **Première partie** : Toute fonction calculable par une machine de Turing est effectivement calculable
- **Seconde partie** : Toute fonction effectivement calculable est calculable par une machine de Turing

Première partie peut être facilement prouvée. La seconde ne peut pas être prouvée car la notion de « effectivement calculable » n'est pas formellement définie. Seconde partie reconnue comme vraie car nombreuses évidences qui supportent la seconde partie :

⇨ **Évidences Heuristiques**

Définition des machines de Turing s'approche de la notion intuitive de procédé effectif (d'un point de vue humain)

Toutes fonctions particulières s'avérant être calculables ont été montrées être calculables par une machine de Turing.

Difficile d'imaginer une fonction effectivement calculable qui ne soit pas calculable par une machine de Turing.

⇨ **Équivalence des Formalismes**

Différentes définitions des mêmes formalismes de calculabilité ont conduit à la définition de la même classe de fonctions calculables.

Différentes personnes de différents domaines scientifiques, de différentes époques ont inventé des modèles de calculs totalement différents (logique mathématique, lambda calcul, fonctions récursives,..). Toutes ces définitions recouvrent la même classe de fonction calculable.

Tout les mécanismes qui permettraient de construire une machine en utilisant la mécanique newtonienne ne peuvent calculer que des fonctions programmables.

Parmi tous ces formalismes différents, qu'est-ce qu'un bon formalisme pour la calculabilité ? ➔ Un formalisme qui vérifie les fondements de la thèse

Soit *D*, un nouveau formalisme de calculabilité. Nous avons un ensemble de caractéristiques que doit satisfaire ce nouveau formalisme :

a) **SD : Soundness des Définitions**
   Toute fonction D-Calculable (qui peut être représentée dans mon langage de programmation) est calculable. Première partie Church-Turing
b) **CD : Complétude des Définitions**
   Toute fonction calculable est D-Calculable (Si une fonction est calculable dans un certain langage, elle doit aussi l'être dans le langage *D*). Seconde partie Church-Turing.
c) **SA : Soundness Algorithmique**
   L'interpréteur de *D* est calculable.
d) **CA : Complétude Algorithmique**
   S'il existe un autre formalisme qui peut être exécuté, il faut que l'on puisse transformer n'importe quel programme Java en un programme de mon nouveau langage, de telle sorte que le programme initial et le programme transformé soit équivalents.

e) U : Description Universelle
L'interpréteur de *D* est D-Calculable. (Hoare-Allisson) Le compilateur Java doit être représenté par un programme Java par exemple.

f) S : Propriété S-M-N affaiblie
Il existe un transformateur de programmes (Théorème S-M-N).
Soit un programme *d* à 2 valeurs
Soit une valeur *x*
Fournit comme résultat un programme *d'* à 1 argument tel que d'(y) calcule la même fonction que d(x,y)

⇨ **SD,CD,SA,CA,U & S**

Liens entre toutes ces caractéristiques

- **SA => SD**
  Il existe un interpréteur de D qui est calculable. Vu que l'interpréteur est D est calculable, forcément toute fonction calculable en D peut être calculée grâce à l'interpéteur
- **CA => CD**
  On peut transformer n'importe quel programme calculable dans un programme du langage *D*. Donc toute fonction qui est calculable l'est aussi dans le langage *D*
- **SD et U => SA**
  Toute fonction calculable dans *D* l'est aussi en général. L'interpréteur de *D* est D-Calculable. Les 2 propriétés assemblées font que toute fonction calculable (interpréteur + fonctions calculables) sont D-Calculable.
- **CD et S => CA**
  Toute fonction calculable est D-Calculable, et il existe un transformateur de programme (S-M-N). Cela implique que s'il existe un autre formalisme qui peut être exécuté, il faut que l'on puisse transformer n'importe quel programme Java en un programme de mon nouveau langage.
- **SA et CD => U**
  L'interpréteur de D est calculable et toute fonction calculable est D-Calculable, ce qui implique que l'interpréteur de *D* est D-Calculable
- **CA et SD => S**
- **S et U => S-M-N**
- **SA et CA ⇔ SD et CD et U et S**
- **SA et CD et S ⇔ CA et SD et U**

⇨ **Un bon formalisme de calculabilité doit posséder toutes ces caractéristiques.**

⇨ **Il existe aussi des formalismes incomplets, pas bon. Exemple : Fonctions primitives récursives, langage BLOOP (uniquement fonctions totales).**

Comment démontrer la calculabilité d'une fonction, d'un problème ?

1. Essayer de trouver un algorithme qui résout le problème
2. Si 1 ne marche pas, se demander si suspicion possible de non-calculabilité du problème
3. Essayer de trouver la non-calculabilité
4. Si problème non-calculable, essayer de définir un problème approché qui est calculable

Techniques de Preuve

- ⇨ Le + commun : Théorème de Rice
- ⇨ Démonstration directe de la non-calculabilité : diagonalisation + preuve par l'absurde
- ⇨ Méthode de réduction : réduire solution d'un problème A à celle d'un autre problème A', puis à partir de A' construire algorithme pour A.

Si A' calculable, alors A calculable.

Si A non-calculable, A' non-calculable

Comment montrer qu'un problème est intrinsèquement complexe ?

- ➔ Le fait d'avoir un algorithme exponentiel non-polynomial (complexité dans le pire des cas) ne rend pas nécessairement le problème exponentiel ! Cela ne veut pas dire que pour chaque donnée, l'algorithme aura un comportement exponentiel. Une complexité exponentielle est une borne supérieure.
- ➔ Si suspicion que le problème est exponentiel, regarder si un problème NP-Complet connu peut être réduit polynomialement à ce problème.
- ➔ Existe-t-il un algorithme non-déterministe polynomial pour ce problème ? Si oui, le problème est dans NP. Sinon ce sera encore + difficile.

Comment absorber un problème intrinsèquement complexe ?

- ➔ Utiliser un algorithme exponentiel si pour nos données, le comportement est polynomial. Pour certaines données, cet algorithme sera polynomial.
- ➔ Changer le problème en un problème plus simple (polynomial) dont la pertinence est proche de celle du problème initial.
- ➔ Utiliser une technique d'exploration, mais en se limitant à un nombre polynomial.
- ➔ Si problème d'approximation, calculer solution approximative.

Au-delà de la calculabilité

Une fonction calculable possède un programme qui calcule cette fonction. Si une fonction est non-calculable, inutile d'espérer la calculer. Si elle est calculable, on peut essayer de trouver un algorithme efficace pour la calculer.

Une fonction calculable en théorie peut être non-calculable en pratique (nécessite trop de ressources).

Une fonction est **H-Calculable** si un être humain particulier est capable de calculer cette fonction.

H-Calculable = T-Calculable ?

- **Thèse de Church-Turing (Procédés Publics) :**

Si une fonction est H-Calculable et que l'être humain calculant cette fonction est capable de décrire (à l'aide du langage) à un autre être humain sa méthode de calcul, de telle sorte que l'autre humain soit capable de résoudre la fonction, alors la fonction est T-Calculable (calculable par une Machine de Turing).

- **Thèse de Church-Turing (Version Réductionniste) :**

Si une fonction est H-Calculable, alors elle est T-Calculable. Comportement des éléments constitutifs d'un être vivant peut être simulé par machine de Turing.

- **Thèse de Church-Turing (Version Anti-Réductionniste) :**

H-Calculable est différent de T-Calculable car même si le cerveau est capable de calculer certaines opérations, certains aspects seront toujours hors de portée des ordinateurs.

Pas de réponses scientifique, différentes positions possibles :

Croyance	⇨ H-Calculable = T-Calculable ⇨ Réductionniste
Athée	⇨ H-Calculable ≠ T-Calculable ⇨ Anti-Réductionniste
Agnostique	⇨ Pas intéressé
Iconoclaste	⇨ Question mal posée, pas de sens

Calculabilité et Intelligence Artificielle

2 approches possibles pour l'Intelligence Artificielle :

a) **IA Forte** : Simulation du cerveau humain à l'aide d'un ordinateur. En copiant le fonctionnement, on espère obtenir des résultats similaires.
b) **IA Faible** : Programmation de méthodes et techniques de raisonnement en exploitant les caractéristiques propres des ordinateurs. S'apparente à version réductionniste.

Cerveau Humain : 10^{18} instructions par secondes.

Ordinateur le plus Puissant : 10^{18} instructions par seconde (uniquement hardware).

L'ordinateur se rapproche de + en + du cerveau humain.

Les machines peuvent-elles penser ?

Test de Turing (1950)

Principes

- 3 acteurs : un ordinateur (A), un être humain (B), un interrogateur (humain)
- communication exclusivement via message écrit
- l'interrogateur ne sait qui parmi (A) et (B) est l'ordinateur, ni qui est l'être humain
- l'interrogateur peut poser des questions à (A) et (B) qui doivent répondre
- objectif de l'interrogateur : trouver qui est l'ordinateur et qui est l'être humain
- objectif de l'ordinateur : essayer de faire perdre l'interrogateur
- objectif de l'être humain: essayer de faire gagner l'interrogateur

Si l'interrogateur se trompe en moyenne une fois sur deux,
alors *les machines peuvent penser…*

Les machines pensent-elles ?

Principe de raisonnement (Turing 1950):

- Hypothèse: les machines pensent
- Si on ne peut réfuter cette hypothèse, c'est que celle-ci est vraie
- Envisager toutes les objections possibles et ensuite réfuter chacune d'elle

Exemples de champs philosophiques ayant été bouleversés par les progrès scientifiques

- Conception de la vie et de l'homme
  - Théorie Darwinienne
- Conception du temps et de l'espace
  - Théorie de la relativité
- Conception du déterminisme et des objets physiques
  - Mécanique quantique
- Conception du raisonnement et de ses limitations
  - Calculabilité et logique mathématique

De nouveaux modèles de calcul, basés sur de nouvelles technologies peuvent-ils modifier les frontières de la calculabilité et de la complexité ?

- ⇨ Frontière de la non-calculabilité est indépendant des progrès technologiques
- ⇨ Les problèmes NP-Complets ne pourraient-ils pas être résolus en temps polynomial grâce à de nouvelles technologies ? Avec des ordinateurs quantiques, est ce que cela ne changerait pas la donne ?

Ordinateurs Quantiques

Un ordinateur Quantique ne donne pas un mais plusieurs résultats superposés les uns sur les autres, chacun ayant leurs probabilité d'être visible. Un ordinateur quantique ne calcule pas de réponses exacte, mais plutôt des réponses avec probabilité.

Un problème A ∈ **BQP (Bounded-Error Quantum Computing)** si et seulement si un algorithme quantique polynomial qui donne la bonne réponse au moins 2 fois sur 3. Pas de réponse exacte mais précision aussi précise que voulu en un temps polynomial.

- ⇨ Relation Supposée en BPQ et NP