Polycopié
Calculabilité, Complexité, Modèles de Calcul

Benjamin Hellouin de Menibus, IUT d'Orsay

Version of January 23, 2023

Avant-propos

Ce document est un polycopié pour le cours "Calculabilité, complexité, modèles de calcul" enseigné au MPRI - voie universitaire, Université Paris-Saclay, 2022–2023. Merci à Emma Caizergues, Atte Torri, Léo Kulinski, Matthieu Robeyns, Mohamed Bassiouni, Pablo Arnault et Paul Patault pour leurs suggestions, corrections et notes de cours.

Contents

Quelques notations

Étant donné un ensemble fini A de symboles, appelé **alphabet**, A^* est l'ensemble des **mots finis** ou **chaînes de caractères** utilisant des symboles de A. Par exemple, un entier écrit en binaire est un mot sur l'alphabet $A = \{0, 1\}$. La chaîne vide est notée ε.

Une **fonction partielle** $f : A \to B$ est une fonction qui, sur certaines entrées, n'a pas de sortie ; dans ce cas, on écrit $f(a) = \perp$. Une fonction partielle qui a toujours une sortie est appelée **totale**. Contrairement à la convention habituelle en mathématiques, les fonctions ne sont pas supposées totales en général.

1 Modèles de calcul et calculabilité intuitive

1.1 Modèles de calcul, thèse de Church-Turing

Un **modèle de calcul** est un ensemble de règles formelles qui décrivent des programmes. Chaque programme, après avoir reçu une **entrée**, effectue des opérations (définies par les

règles) et peut renvoyer une **sortie**. Entrée et sortie sont des mots finis sur l'alphabet A. Une function $A^* \to A^*$ calculée par un tel programme est dite **calculable** (dans ce modèle).

Note. *Le choix de A fait partie du modèle (souvent $A = \{0, 1\}$); certains modèles travaillent sur $\mathbb{N}$, ce qui revient au même. On verra plus tard que ces distinctions n'ont pas d'importance, via l'usage des encodages.*

Vous connaissez peut-être des modèles théoriques tels que les machines de Turing, le λ-calcul, les circuits booléens, les fonctions récursives.... Tout langage de programmation définit également, d'une certaine manière, un modèle de calcul[1].

Dans la section Calculabilité de ce cours, nous utiliserons librement divers modèles de calcul, et la raison pour laquelle nous pouvons le faire est que ces modèles calculent les mêmes fonctions :

Thèse de Church-Turing Une fonctions qui est calculable par un modèle de calcul raisonnable est calculable par une machine de Turing. Un modèle qui calcule exactement les mêmes fonctions que les machines de Turing est dit **Turing-complet**; il y en a beaucoup.

C'est un principe général (philosophique, si on veut) ; il s'agit de ce qui est attendu lorsque l'on introduit un modèle nouveau, ce qui ne dispense pas de le prouver ! Il y a des diables dans les détails de ce qui constitue un modèle "raisonnable", mais on utilisera au cours de ce cours de nombreux modèles Turing-complets sans le prouver, et on soulignera explicitement les modèles qui ne le sont pas.

Definition 1. *Une fonction est **calculable** si elle est calculable par un modèle Turing-complet — par exemple, les machines de Turing.*

Puisque la plupart des modèles sont équivalents, pourquoi utiliser différents modèles ? Ils diffèrent en termes de :

- **expressivité**: facilité d'écrire des programmes complexes.
- **minimalisme**: faible nombre d'opérations permises, ce qui rend les preuves plus gérables.
- **applicabilité**: opérations pertinentes pour la machine sur laquelle on programme.

Les langages de programmation sont Turing-universels de manière robuste, donc ces modèles vous permettent d'écrire des programmes dans votre langage préféré ou même en pseudocode sans trop se soucier des détails. Par exemple, voici un programme Python qui constitue une preuve raisonnable que la fonction $n \mapsto 2^n$ (sur $\mathbb{N}$) est calculable :

```
def power2 (int n){
        int result = 1
        for i from 1 to n
                result = 2*result
        output result
}
```

[1] Cela demande de faire abstraction de limitations pratiques — taille de la mémoire vive, taille limite des entrées... — qui sont intégrées au compilateur et à l'interpréteur. Il faut imaginer le comportement de ces programmes sur un ordinateur idéalisé.

1.2 Machines de Turing

Cette section est volontairement compacte et ne sert qu'à fixer les conventions et notations ; vous êtes supposés avoir déjà rencontré une machine de Turing.

Une **machine de Turing** est donnée par un tuple (n, m, δ), où :

- n est le nombre d'états, m le nombre de symboles de ruban ;
- $\delta : \{0, \ldots, n-1\} \times \{\#, 0, \ldots, m-1\} \to \{0, \ldots, n-1\} \times \{\#, 0, \ldots, m-1\} \times \{-1, 0, +1\}$ est une **fonction de transition**.

Une machine de Turing est un programme qui fonctionne comme suit. La machine dispose de:

- un **ruban** infini dont chaque case contient un symbole de $\{\#, 0, \ldots, m-1\}$, où $\#$ est un symbole spécial "vide" ;
- une **tête** dont la position est donnée par un entier $p \in \mathbb{N}$;
- un **état** dans $\{0, \ldots, n-1\}$.

Une **configuration** est un tuple (x, p, e) avec x un mot fini, p une position de la tête, et e un état. Cela signifie que le ruban contient x suivi de symboles $\#$.

- La machine reçoit comme **entrée** un mot fini $x \in \{1, \ldots, m\}^*$. La configuration initiale (au temps 0) est $M^0(x) = (x, 1, 0)$.
- Supposons que la machine soit dans la configuration $M^t(x) = (x, p, e)$ au temps t. Soit $i, d, e' = \delta(x_p, e)$, où x_p est le symbole apparaissant sur le ruban en position p. La configuration au temps $t+1$ est $M^{t+1}(x) = (x', p', e')$, où:
  - x' est égal à x sauf que $x'_p = i$;
  - $p' = p + d$ (déplacement de la tête).
- Si à tout moment $M^t(x) = (x, p, n-1)$ ($n-1$ est le dernier état), la machine **s'arrête** : le calcul est terminé et la **sortie** est x, la partie non vide du ruban. Après cela, on considère que la configuration ne change plus.

Note. *Vous avez probablement déjà rencontré d'autres définitions pour les machines de Turing : autres symboles, plus de rubans... Cela ne change pas la Turing-complétude du modèle, donc ce sont des questions de goût et les fonctions calculées ne changent pas (par la thèse de Church-Turing)*

1.3 Fonctions calculables

Pour une machine de Turing M sur l'entrée x, on écrit:

- $M(x)\downarrow$ si la machine s'arrête après un certain temps. Dans ce cas, on écrit $M(x)\downarrow = y$ ("sur l'entrée x, M s'arrête et renvoie y") où $y \in A^*$ est le mot sur le ruban dans la configuration finale.
- $M(x)\uparrow$ si la machine ne s'arrête jamais.

Les machines de Turing calculent, en général, des fonctions partielles : il n'y a pas de sortie si la machine ne s'arrête jamais.

Definition 2. *Une machine du Turing M* ***calcule*** *une fonction $f : \{0, \dots, m-1\}^* \to \{0, \dots, m-1\}^*$ si on a pour toute entrée x :*

$$f(x) = \bot \Rightarrow M(x)\uparrow$$
$$f(x) \neq \bot \Rightarrow f(x) = M(x)\downarrow .$$

En d'autres termes,

- *si $f(x)$ est défini, la machine sur l'entrée x s'arrête et renvoie la réponse attendue ;*
- *si $f(x)$ n'est pas défini, la machine sur l'entrée x ne s'arrête jamais (elle "boucle").*

Il est important de garder à l'esprit une distinction claire entre fonctions et programmes (= machines de Turing). Un programme calcule une certaine fonction, mais un fonction peut n'être calculée par aucun programme, ou par plusieurs programmes. Une fonction ne peut pas s'arrêter ou boucler, et $f(x)$ a toujourd un sens (même quand c'est $\bot$).

1.4 Quelques opérations sur les fonctions calculables

Notre point de vue "intuitif" sur la calculabilité peut être un peu perturbant si vous êtes habitués à des cadres mathématiques plus solides. J'ai décidé de travailler dans ce cadre car écrire des programmes comme machines de Turing (ou autre modèle minimaliste) est pénible et n'aide pas à construire l'intuition simple qu'une fonction calculable est une fonction pour laquelle on peut écrire un programme.

On voit immédiatement avec cette intuition que la calculabilité est préservée par les opérations habituelles sur les boolées, les chaînes de caractères, les entiers : addition, concaténation, composition, opérateurs logiques...

Faisons une preuve pour, disons, la composition. Supposons que f et g sont calculables, et écrivons le programme:

```
program f\circ g(int n){
        int m = P_g(n)
        output P_f(m)
}
```

où `P_f` et `P_g` sont des programmes qui calculent f et g. Il est clair que ce programme calcule $f \circ g$. Notez tout de même que si $g(n) = \bot$ ou $f(g(n)) = \bot$, notre programme boucle : il semble raisonnable de définir que $f \circ g(n) = \bot$ dans ces conditions, donc c'est la réponse attendue.

Si on voulait une preuve formelle, on se tournerait vers un modèle minimaliste tel que les machines de Turing.

Exercice 1. *Supposons que f et g sont des fonctions calculables totales $\mathbb{N} \to \mathbb{N}$. Parmi les fonctions suivantes, lesquelles sont calculables ? Lesquelles ne semblent pas l'être en général ?*

1. *$n \mapsto 1$ si $f(n) = g(n)$, 0 sinon.*
2. *$n \mapsto$ le plus petit $m > n$ tel que $f(m) = 0$.*
3. *$n \mapsto$ le plus petit m tel que $f(m) = n$ (s'il existe).*

2 Encodages, cardinalité et diagonalisation

2.1 Encodages et dénombrabilité

Pour qu'un objet puisse être manipulé par un programme, il doit être représenté en mémoire — **encodé** — en une suite finie de symboles. C'est le cas pour les entrées et sorties de machines de Turing[2], et pour tous les autres modèles de calcul que nous verrons. Les ensembles dont les membres peuvent être représentés par un nombre fini de symboles (ou de bits) sont dits **dénombrables**.

Definition 3. *Un ensemble est dit **dénombrable** s'il existe un alphabet A tel qu'on ait une des deux conditions équivalentes:*

- *il existe un **encodage** de X dans A^*, c'est-à-dire une injection totale $\varphi_X : X \to A^*$;*
- *il existe un **décodage** de A^* dans X, c'est-à-dire une surjection partielle $\varphi_X^{-1} : A^* \to X$.*

Comme la notation le suggère, encoder puis décoder renvoie l'objet initial. Les conditions d'injectivité / surjectivité signifient que tout objet de X peut être représenté par un mot de A^*, ou peut-être plusieurs. Un mot de A^* peut ne rien encoder, mais il est interdit pour deux objets de X d'avoir le même encodage.

Tout ensemble fini X est dénombrable; si X est infini, on peut même trouver des bijections. $X = \mathbb{N}$ peut être encodé sur $\{0,1\}^*$ par l'encodage binaire, ou sur $\{0,\dots,9\}$ par l'encodage décimal, ou de multiples autres façons.

Une définition équivalente important est la suivante :

Definition 4. *Un ensemble X est **dénombrable** s'il existe une **enumération** de X, i.e. une surjection $\mathbb{N} \to X$, ou (de manière équivalente) un encodage de X dans $\mathbb{N}$, i.e. une injection $X \to \mathbb{N}$.*

Si X est infini, l'énumération nous fournit une liste $x_1, x_2, x_3 \dots$ qui contient tous les éléments de X.

La dénombrabilité vue comme "la capacité à représenter des objets par un nombre fini de symboles" devrait donner une bonne intuition. Entraînons-nous avec une preuve formelle.

Theorem 1. *Soit D et D' deux ensembles dénombrables. Alors $D \times D'$ est dénombrable*

Proof. Intuitivement, on peut encoder un élément $(d, d') \in D \times D'$ en concaténant un encodage de d et un encodage de d'. Pour s'assurer qu'il n'y a pas d'ambiguité, on ajoute un nouveau symbole **,** pour marquer la limite entre les deux éléments.

Formellement, soit $\varphi_D : D \to S^*$ un encodage pour D et $\varphi_{D'} : D' \to S'^*$ pour D'. On définit $\varphi_{D\times D'}$ comme l'encodage qui, sur l'entrée (d, d'), renvoie la chaîne $\varphi_D(d), \varphi_{D'}(d')$. Vérifions que $\varphi_{D\times D'}$ est une injection : prenons deux paires $(a, a') \neq (b, b')$. Si $a \neq b$, alors les préfixes apparaissant avant la virgule dans $\varphi_{D\times D'}(a, a')$ et $\varphi_{D\times D'}(b, b')$ sont différents, donc $\varphi_{D\times D'}(a, a') \neq \varphi_{D\times D'}(b, b')$. De même si $a' \neq b'$. □

[2]Rien n'empêche de réfléchir au comportement des machines de Turing sur des entrées infinies, mais cela n'a pas vraiment de sens dans un cadre standard où le calcul doit s'arrêter après un nombre fini d'étapes.

2.2 Encodages et calculabilité

Nous avons défini plus tôt les fonctions calculables sur les mots (chaînes de caractères). Nous allons maintenant étendre cette définition aux fonctions $D \to D'$, où D et D' sont des ensembles dénombrables, en utilisant des encodages.

Definition 5. *Soient φ_D et φ'_D des encodages de D et D', respectivement. Une fonctions $f : D \to D'$ est calculable si la fonction $\varphi_D(x) \mapsto \varphi'_D(f(x))$ est calculable (au sens de la définition précédente, puisque cette fonction travaille sur des mots).*

Cette définition devrait être intuitive : si vous voulez calculer une fonction $\mathbb{N} \to \mathbb{N}$, commencez par encoder l'entrée en binaire, exécutez le programme, et décoder la réponse pour avoir le résultat.

Cela signifie que, hors des chaînes de caractères, le choix d'encodage peut influer sur le fait que la fonctions est calculable ou non. En pratique, ce n'est pas un problème et la calculabilité est invariante par tout choix raisonnable d'encodage[3]. Par exemple, les fonctions calculables sur les entiers sont les mêmes en encodage binaire ou décimal.

Note. *Dans la définition précédente, pour calculer la fonction $\varphi_D(x) \mapsto \varphi'_D(f(x))$, remarquez que tous les mots de A^* ne sont pas des entrées valides (ils n'encodent pas nécessairement un élément de D — par exemple, l'encodage binaire interdit généralement de commencer par 0). Le comportement d'un programme qui calcule cette fonction n'est pas défini pour ces entrées : il peut faire ce qu'il souhaite.*

Exercice 2. *Parmi les fonctions suivantes, pour lesquelles cela a-t-il du sens de demander si elles sont calculables ?*

$\mathbb{Z} \to \mathbb{Z}$ *:* $n \mapsto -n$*;*

$\mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *:* $(a, b) \mapsto (a + b, a - b)$*;*

$\mathbb{N} \to \mathbb{Q}$ *:* $n \mapsto 1/n$*;*

$\mathbb{N} \to \mathbb{R}$ *:* $x \to \sin(x)$.

2.3 Fonctions incalculables: la technique de diagonalisation

Nous faisons un petit détour pour comprendre comment l'hypothèse de dénombrabilité sur les entrées et sortie des programmes limite l'univers des fonctions calculables.

Theorem 2. *L'ensemble des fonctions calculables est dénombrable*

Proof. Pour toute fonction calculable, il existe un programme qui la calcule. Ce programme est une chaîne de caractères finie. Nous pouvons donc encoder une fonction calculable par le plus court programme qui la calcule. Comme un programme ne peut calculer qu'une seule fonction, c'est un injection. □

Cette preuve est valide dans tout modèle. Par la définition alternative, on peut énumérer les programmes : $P_0, P_1 \ldots$.

Exercice 3. *Trouver un encodage des machines de Turing.*

[3]Pour une preuve formelle, il faudrait montrer que la fonction de changement d'encodage est calculable.

Pour montrer l'existence de fonctions incalculables, on montre qu'il y a davantage de fonctions possibles que de programmes. La preuve suivante est écrite pour les fonctions $\mathbb{N} \to \mathbb{N}$ mais marcherait sur n'importe quels ensembles dénombrables infinis.

Theorem 3. *L'ensemble des fonctions totales $\mathbb{N} \to \mathbb{N}$ est indénombrable. Il s'ensuit que l'ensemble des fonctions partielles $\mathbb{N} \to \mathbb{N}$ est également indénombrables (i y en a encore plus).*

La preuve suivant utilise une technique très importante appelée **diagonalisation**.

Proof. Supposons, par contradiction, que l'ensemble des fonctions totales $\mathbb{N} \to \mathbb{N}$ soit dénombrable. Enumérons-les : $f_0, f_1 \ldots$ (c'est une surjection, donc toutes les fonctions totales devraient être énumérées). Définissons :

$$F : \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & f_n(n) + 1 \end{array}$$

F est totale, et j'affirme que F n'apparait pas dans la liste. Pour tout k, on a par définition $F(k) \neq f_k(k)$, et par conséquent $F \neq f_k$. C'est une contradiction. □

Exercice 4. *Avec la même méthode de preuve, montrez que l'ensemble des nombres réels est indénombrable.*

Le prochain théorème est également vrai pour n'importe quels ensembles dénombrables infinis.

Theorem 4. *Il existe une fonction incalculable $\{0,1\}^* \to \{0,1\}^*$.*

Proof. L'ensemble des fonctions $\{0,1\}^* \to \{0,1\}^*$ est indénombrable, et l'ensemble des fonctions calculables $\{0,1\}^* \to \{0,1\}^*$ est dénombrable. Il doit donc exister une fonction incalculable, et même une quantité indénombrable. □

La fonction incalculable produite par la méthode de diagonalisation n'est pas très explicite. On trouvera des exemples plus concrets plus bas.

Ce résultat offre une perspective nouvelle sur les objets mathématiques classiques. Tous les entiers peuvent être décrits par une quantité d'information finie, et c'est également vrai pour les fonctions calculables (représentées par un programme), mais aucun langage ne nous permet de décrire toutes les fonctions $\mathbb{N} \to \mathbb{N}$ en une quantité d'information finie.

Vous avec pourtant rencontré de nombreuses fonctions ayant une description finie, comme $f : n \mapsto 2n$. En fait, toutes les fonctions explicitement définies en mathématiques ont une description finie (une définition), donc on ne rencontre qu'une petite partie dénombrable des fonctions $\mathbb{N} \to \mathbb{N}$. Il s'agit souvent de fonctions calculables, mais pas toujours.

2.4 Objets calculables et représentations.

Jusqu'à présent, nous avons défini la calculabilité uniquement pour les fonctions. On peut également parler de calculabilité pour d'autres objets, tels que des ensembles ou des langages, en définissant une **représentation** de l'objet sous la forme d'une fonction qui contient la même information et dont les entrées et sorties sont calculables.

Note. *Essayez de bien clarifier la différence entre:*

- *un* ***encodage*** *: représente des objets comme mots finis pour qu'ils puissent être manipulés comme entrées–sorties de programmes ;*
- *une* ***représentation*** *: représente des objets comme des fonctions pour qu'on puisse parler de leur (absence de) calculabilité.*

Exercice 5. *Parmi les objets suivants, essayez de donner du sens à la question de savoir s'ils sont calculables. Autrement dit, pouvez-vous les représenter sous la forme d'une fonction dont l'entrée et la sortie sont dénombrables ?*

1. *le nombre entier* 7*;*
2. *la valeur de vérité (*0 *ou* 1*) de la conjecture de Riemann;*
3. *le problème de, étant donné un graphe, trouver un chemin Hamiltonien;*
4. *le langage* $\mathcal{L} \subset \{0,1\}^*$ *des palindromes;*
5. *le sous-ensemble* $P \subset \mathbb{N}$ *des nombres premiers;*
6. *Le nombre réel* π*.*

1. Parler de représentation pour un objet fini n'a presque pas de sens ; n'importe quelle représentation devrait montrer que tous les entiers sont calculables. Par exemple, l'entier n peut être représenté par la fonction $\{\varepsilon\} \to \mathbb{N} : \varepsilon \mapsto n$.
2. Remarquez bien qu'il ne s'agit pas d'une fonction : c'est un unique booléen, même s'il dépend du monde dans lequel on vit. Comme un entier, un booléen peut être représenté par la fonction qui ignore son entrée et renvoie toujours vrai, ou toujours faux. On ne sait pas quelle fonction est la bonne, mais dans tous les cas il existe un programme qui calcule le bon booléen.

Pour ces deux exemples, et plus généralement pour des objets individuels ayant une représentation finie, la notion de calculabilité est presque dégénérée (tout est calculable). Pour les autres cas, je veux souligner que, contrairement aux encodages, la notion de calculabilité peut dépendre du choix de représentation.

En pratique, on se fixe une représentation standard qui définit la notion de calculabilié pour chaque type d'objet. Si une autre représentation semble intéressant, elle induira une autre notion de calculabilité avec un nom différent. Voici quelques représentations stadard :

Definition 6. *Dans ce qui suit,* X *est un ensemble dénombrable.*

- *Un problème (étant donné* $x \in X$*,* x *satisfait-il la propriété* $\mathcal{P}$ *?) est représenté par sa fonction booléenne* $X \to \{0,1\}$*.*

- *Un ensemble $S \subset X$ est représenté par sa fonction indicatrice $1_S : X \to \{0,1\}$, définie par $1_S(x) = 1 \Leftrightarrow x \in S$.*
- *Un langage $\mathcal{L} \in \{0,1\}^*$ est un type d'ensemble, donc la définition précédente s'applique.*
- *Un nombre réel est représenté par une fonction d'approximation $a_r : \mathbb{N} \to \mathbb{Q}$ telle que $|r - a_r(n)| \leqslant 2^{-n}$.*

La définition pour les nombres réels est sans doute la moins intuitive. Ce sujet sera développé dans une section ultérieure.

Noms Il est habituel de dire qu'une fonction est calculable, qu'un problème est décidable, qu'un langage ou ensemble est récursif. Ces adjectifs sont synonymes et ne sont que du bagage historique.

Voyons à présent une autre représentation naturelle pour les ensembles qui nous amènera à une autre définition de calculabilité :

Definition 7. *Un ensemble $S \subset X$ est **calculablement énumérable**[4], abrévié c.e., s'il existe une surjection calculable $f : \mathbb{N} \to S$.*

On appelait déjà une telle fonction **énumération** dans la Définition 4. Attention : contrairement à la Définition 4, on ne peut pas utiliser une des définitions alternatives dans cette définition.

Theorem 5. *Si S est calculable, alors S est calculablement énumérable.*

Proof. Faisons la preuve quand $S \subset \mathbb{N}$. Puisque S est calculable, par définition, la fonction 1_S est calculée par un certain programme `P_S`. Considérons le programme suivant :

```
program enum(int n){
        int count = 0;
        for i from 0 to infinity:
                if P_S(i):
                        count += 1;
                if count = n:
                        output i;
}
```

Sur l'entrée $n \in \mathbb{N}$, le programme répond le n-ième élément de S, dans l'ordre croissant. Il devrait être clair que c'est une surjection. Remarquez que le programme peut boucler s'il y a moins de n éléments dans S ; ce n'est pas un souci. □

Un ensemble calculablement énumérable n'est, en revanche, pas nécessairement calculable. Nous prouverons cela plus tard, quand nous aurons les outils nécessaires.

[4]Souvent appelé **récursivement énumérable** — cf. la remarque précédente.

3 Des programmes dans des programmes, réductions, oracles

3.1 Programmes en entrée – sortie

On a mentionné plus haut que les programmes forment un ensemble calculable. Autrement dit, ils peuvent être encodés sous forme d'une chaîne de caractères pour être manipulés, en entrée ou en sortie, par un autre programme.

Le plus simple est d'imaginer que les programmes sont encodés sous forme de leur code source. Ce n'est pas une idée si étrange : un compilateur, par exemple, prend en entrée un programme et renvoie en sortie un autre (dans un langage différent).

Definition 8. *Un **programme universel** U est un programme qui, reçevant en entrée un programme P et un mot x, se comporte comme suit :*

- *si $P(x)\uparrow$, alors $U(P,x)\uparrow$;*
- *si $P(x)\downarrow$, alors $U(P,x)\downarrow = P(x)\downarrow$.*

U simule le comportement de tout programme qui lui est donné. En informatique réelle, on dit que U exécute le programme, et on trouve de telles fonctions (souvent appelées **exec**) dans de nombreux langages.

3.2 Le problème de l'arrêt

Le problème de l'arrêt consiste à décider, étant donné un programme et une chaîne de caractères, si le programme s'arrête sur cette entrée. Formellement :

$$\begin{array}{rccl} & \mathcal{P}\times A^* & \to & \{0,1\} \\ \text{halt} : & (P,x) & \mapsto & \begin{cases} 1 \text{ si } P(x)\downarrow \\ 0 \text{ si } P(x)\uparrow \end{cases} \end{array}$$

Theorem 6. *Le problème de l'arrêt est indécidable (= incalculable).*

Proof. Pour raisonner par contradiction, supposons que le problème de l'arrêt peut être calculé par un programme `stop`. Considérons maintenant le programme suivant :

```
program f(program P){
        if stop(P, P) = 0
                output 0
        else
                loop
}
```

Quel est la valeur renvoyée par $\texttt{stop}(\texttt{f},\texttt{f})$? Si $\texttt{stop}(\texttt{f},\texttt{f}) = 0$, alors par définition $\texttt{f}(\texttt{f})\downarrow = 0$; si $\texttt{stop}(\texttt{f},\texttt{f}) = 1$, alors par définition $\texttt{f}(\texttt{f})\uparrow$. Mais dans les deux cas, c'est en contradiction avec la définition de `stop`. □

On dirait un résultat amusant avec une preuve magique, mais l'impossibilité de décider le problème de l'arrêt a des conséquences profondes pour des problèmes réels de vérification de programmes. On verra dans un moment que le problème de l'arrêt n'est pas unique, et que prévoir le comportement d'un programme est en général indécidable.

3.3 Réductions Turing

Nous allons développer un outil pour montrer que de nouvelles fonctions sont incalculables en comparant la difficulté de calculer différentes fonctions. Intuitivement, f est "plus facile à calculer" que g si

Si je peux obtenir la valeur de $g(x)$ pour tout x, je peux écrire un programme qui calcule f.

Cela signifie, en particulier, que si j'avais un programme qui calculait g, je pourrais en tirer un programme qui calcule f.

Prenons un exemple-jouet[5]. Imaginons qu'on travaille avec un modèle étrange sur les entiers où les opérations autorisées sont l'addition, la soustraction, et la division par deux. Quelle opération est la plus facil, la multiplication ou la mise au carré ?

- Mettre au carré est plus facile que multiplier, puisque $\texttt{square}(x) = \texttt{mult}(x, x)$.
- Multiplier est plus facile que mettre au carré, puisque $\texttt{mult}(x, y) = \frac{\texttt{square}(x+y)-\texttt{square}(x)-\texttt{square}(y)}{2}$.

Ce sont des déclarations purement comparatives : on n'a aucune raison de penser que ces opérations sont calculables dans ce modèle étrange, mais nous sommes pourtant capables de comparer leur difficulté. Comment avons-nous fait cela ? Nous avons écrit une espèce de programme, appelé **réduction**, qui calcule une fonction en utilisant une opération additionnelle en plus de celles autorisées par le modèle. On dit que la fonction supplémentaite est **donnée en oracle**.

Faisons à présent une définition formelle.

Definition 9 (Réduction Turing)**.** *Soit f et g deux fonctions $A^* \to A^*$. On dit que f est **se réduit** à g (pour la réduction Turing), et on écrit $f \leqslant_T g$, si f peut être calculé étant donné g en oracle. Autrement dit, f est calculable dans le modèle auquel on ajoute une opération qui calcule g (une "boîte noire").*

La notation $f \leqslant_T g$ peut être comprise comme "f est moins difficile que g".

On définit $\geqslant_T, \equiv_T, <_T \ldots$ de la même manière.

Quand on écrit des programmes en pseudocode, recevoir g en oracle nous permet d'écrire des opérations comme $y = g(x)$, que g soit calculable ou non. Dans les machines de Turing, cette notion peut être formalisée par l'ajout d'un ruban et d'un état supplémentaire dits **d'oracle**; quand le calcul entre dans l'état d'oracle, la machine remplace "magiquement" le contenu x du ruban d'oracle par $g(x)$.

La motivation pour définir la réduction de Turing est le théorème suivant :

Theorem 7. *Si $f \leqslant_T g$ avec g calculable, alors f est calculable.*

Si $f \leqslant_T g$ avec f incalculable, alors g est incalculable.

[5]Merci à `https://fr.wikipedia.org/wiki/Réduction_(complexité)`

Proof. $f \leqslant_T g$ signifie qu'il existe un programme P recevant g en oracle qui calcule f. Comme g est calculable, il existe un programme P_g qui calcule g. Chaque fois que l'opération d'oracle qui calcule g est utilisée dans f, on la remplace par un appel au programme P_g. On a construit un programme sans oracle qui calcule f. □

Il est important de comprendre que des fonctions incalculables peuvent être comparées, et que certains seront "plus incalculables" que d'autres. En réalité, un thème de recherche à part entière est dédié à l'étude de la structure mathématique de $\leqslant_T$ et $\equiv_T$ et leurs classes d'équivalence (**Turing degrees**).

La réduction Turing n'est qu'un type de réduction qui est appropriée à l'étude de la calculabilité et de l'incalculabilité, mais pas (par exemple) à P̧ et N̨, pour la raison suivante :

Exercice 6. *Prouvez que deux fonctions calculables sont toujours Turing-équivalentes ($\equiv_T$).*

C'est pourquoi, dans d'autres contextes, nous introduirons d'autres types de réductions.

3.4 Théorème de Rice

Nous allons mettre en pratique les réductions que nous avons apprises.

Theorem 8 (Théorème de Rice)**.** *Soit T une propriété non triviale des fonctions calculables (qui n'est pas toujours vraie ou fausse).*

Alors halt $\leqslant_T T$. *En particulier, T est incalculable.*

Cela peut paraître inattendu que l'arrêt, bien qu'incalculable, soit la propriété "la plus facile" des fonctions calculables.

Ce théorème, malgré sa simplicité apparente, cache des difficultés. Puisque les entrées de T sont des fonctions calculables, il est naturel de les encoder par un programme qui les calcule. Cependant, T n'est pas une propriété des programmes, c'est une propriété des fonctions. Pour les exemples suivants, essayez de comprendre s'il s'agit d'une propriété de la fonction ou du programme, et dites si le problème vous semble décidable ou non.

- A-t-on $f(\varepsilon) = \bot$? (entrée vide)
- Le programme contient-il une boucle while ?
- Peut-on trouver une entrée x telle que $f(x) \neq 2$?
- Le programme boucle-t-il sur toutes les entrées ?

Une autre formulation est que le théorème s'applique aux propriétés **sémantiques** des programmes — leur sens, i.e. la fonction calculée — et non à leurs propriétés **syntaxiques** — ce qui est écrit dans le programme.

Proof. Soit $\infty : x \mapsto \bot$, et supposons que ∞ ne satisfait pas T (l'autre cas est symétrique). Puisque T est non triviale, il existe une fonction f qui satisfait T.

Maintenant, on va définir une fonction ψ qui transforme un programme en un autre. Étant donné un programme P et une chaîne x, $\psi(P, x)$ est le programme suivant :

```
program (input y){
        P(x);
        output f(y);
}
```

- ψ est calculable : il suffit d'écrire le code ci-dessus, de remplacer la lettre x par la valeur donnée et P par un appel au programme correspondant.
- si $P(x)\downarrow$, alors $\psi(P,x)$ calcule f et satisfait T. Si $P(x)\uparrow$, $\psi(P,x)\uparrow$, donc $\psi(P,x)$ calcule ∞ et ne satisfait pas T.

Nous sommes prêts à écrire la réduction qui prouve que halt $\leqslant_T T$.

```
program stop(program P, string x) oracle T{
        if T(ψ(P,x)): output 1
        else: output 0
}
```

□

3.5 Quelques exercices de réduction

Exercice 7. *Les problèmes suivants sont-ils calculables ou incalculables ?*

$$=_1 : \begin{array}{rcl} \mathcal{M}\times\mathcal{M}\times\{0,1\}^* & \to & \{0,1\} \\ (f,g,x) & \mapsto & \begin{cases} 1 \text{ if } f(x)\downarrow = g(x)\downarrow \\ 0 \text{ if } f(x)\downarrow \neq g(x)\downarrow \end{cases} \end{array}$$

$$=_2 : \begin{array}{rcl} \mathcal{M}\times\mathcal{M}\times\{0,1\}^* & \to & \{0,1\} \\ (f,g,x) & \mapsto & \begin{cases} 1 \text{ if } f(x)\downarrow = g(x)\downarrow \\ 0 \text{ otherwise.} \end{cases} \end{array}$$

Exercice 8. *Les fonctions suivantes sont-elles calculables ou incalculables ?*

1. *Arrêt sur l'entrée vide : $P \mapsto 1$ si $P(\varepsilon)\downarrow$, 0 si $P(\varepsilon)\uparrow$.*
2. *Arrêt pour P: $x \mapsto 1$ si $P(x)\downarrow$, 0 si $P(x)\uparrow$.*
3. *Arrêt avant k étapes : $(P,n,k) \mapsto 1$ si $P(x)$ s'arrête avant k étapes de calcul, 0 sinon.*
4. *Détection de code mort : soit P un programme en `<langage préféré>` et f le nom d'un sous-programme. f est-elle appelée durant l'exécution de P ?*

3.6 Calculabilité par-delà l'arrêt

A-t-on des fonctions strictement plus difficiles que le problème de l'arrêt, ou toutes les fonctions incalculables sont-elles équivalentes ?

Definition 10. *Soit $g : A^* \to A^*$ une fonction arbitraire. Appelons* halt^g *le problème de l'arrêt des programmes qui reçoivent g en oracle. Autrement dit, pour tout programme P qui reçoit g en oracle et pour toute entrée x,* $\text{halt}^g(P,x) = 1$ *si $P(x)\downarrow$, et 0 si $P(x)\uparrow$.*

Theorem 9 (Saut Turing)**.** *Pour toute fonction g,* $\text{halt}^g >_T g$.

Cela signifie que $\text{halt}^g \geqslant_T g$ et $\text{halt}^g \not\leqslant_T g$.

Proof. On définit un programme P qui reçoit en entrée deux chaînes de caractères x, y et construit un programme :

```
program () oracle g{
        if g(x) = y: output 1
        else: loop
}
```

La réduction $\text{halt}^g \geqslant_T g$ est :

```
program (string x) oracle halt^g{
        for any string y:
                if halt^g(prog, x, y): output y
}
```

La preuve de $\text{halt}^g \not\leqslant_T g$ (inégalité stricte) est la même que celle du problème de l'arrêt, en fournissant un oracle g à tous les programmes dans la preuve. □

4 Énumérations calculables

4.1 Généralités

On a vu plus tôt que les ensembles, langages et problèmes de décision (des objets que l'on peut représenter par des fonctions à images dans $\{0, 1\}$) ont plusieurs représentations qui mènent à différentes notions de calculabilité :

- par la fonction indicatrice 1_X (si cette fonction est calculable, l'ensemble est **calculable**)
- par une énumération, i.e. une surjection $enum_X : \mathbb{N} \to X$ (si une telle énumération est calculable, l'ensemble est **calculablement énumérable**)

Il y a d'autres représentations mais elles sont le plus souvent équivalentes à ces deux-là. Remarquez que la deuxième définition utilise la notion d'énumération introduite dans la définition d'un ensemble dénombrable (Definition 4), donc on pourrait appeler cette notion "calculablement dénombrable". Attention : on ne peut pas utiliser la fonction inverse ici (injection $X \to \mathbb{N}$).

Theorem 10. *Soit X un ensemble, 1_X sa fonction indicatrice, $enum_X$ une énumération de X. Alors $enum_X \leqslant_T 1_X$. Il s'ensuit qu'un ensemble calculable est calculablement énumérable.*

Avant d'aller plus loin, prouvons une importante définition équivalente :

Theorem 11. *Un ensemble X est calculablement énumérable si, et seulement si, la fonction indicatrice partielle $1_X^\perp$ est calculable, où $1_X^\perp : x \mapsto 1$ si $x \in X$, $\perp$ si $x \notin X$.*

Ce théorème nous donne une intuition précise sur la différence entre calculable et calculablement énumérable. Avant d'en faire la preuve, notez que X est calculable si et seulement si X^c (le complément de X) l'est. En effet, il suffit d'échanger 0 et 1 dans un programme qui calcule 1_X. Cependant, ce théorème montre qu'être calculablement énumérable n'est **pas symétrique** par l'échange de 0 et de 1.

Si X^c est c.e., on dit que X est **co-calculablement énumérable**.

Proof. ($\Leftarrow$) Étant donné une énumération calculable $enum_X$, le programme suivant calcule $1_X^\perp$:

```
program 1 (input x){
        for all n from 0 to ∞:
                if enum_X(n) = x:
                        output 1
}
```

Il renvoie 1 si $x \in X$ (parce que $enum_X$ est une surjection), et boucle sinon.

($\Rightarrow$) Supposez que $1_X^\perp$ est calculable. Le programme suivant calcule une énumération de X :

```
program enum_X (input n){
    S = []
    for all t from 0 to ∞:
        for all strings x of length ⩽ t:
            simulate 1_X^⊥(x) during t steps of computations
            if this simulation stops:
                add x to S
                if S contains n elements: output S[n]
}
```

Si $x \in X$, alors $1_X^\perp(x)$ s'arrêtera après un certain temps t. Quand n est assez grand, $enum_X(n)$ finira pas atteindre la t-ième étape de la boucle et répondra x. Donc $enum_X$ est une énumération (= surjection sur X). □

Exercice 9. *Montrez que l'ensemble $S = \{(P, x) \in A^* \times A^* : P(x)\downarrow\}$ est calculablement énumérable mais pas calculable.*

Theorem 12. *Si X est c.e. et co-c.e., alors X est calculable.*

Proof. On dispose de deux énumérations calculables : ϕ qui énumère les éléments de X, et ϕ^c qui énumère les éléments de X^c. Cela signifie que chaque élément de $\{0,1\}^*$ est énuméré par exactement une de ces deux fonctions. Par conséquent, le programme suivant :

```
program f(input x){
    for all i from 0 to ∞:
                if x = φ(i): output 1
                if x = ψ(i): output 0
}
```

termine toujours et renvoie 1_X. □

Ce monde de la calculabilité asymétrique, où 0 et 1 jouent des rôles différents, ne peut exister que pour des ensembles, des langages, des problèmes de décision, et plus généralement les fonctions à image dans $\{0, 1\}$

4.2 Réductions fortes (many-one)

Si on prend une fonction $f : X \to \{0, 1\}$ (problème de décision, fonction indicatrice d'un ensemble...), et $\bar{f}$ la fonction opposée obtenue en inversant 0 et 1, il est clair que $f \equiv_T \bar{f}$. Cependant, il est possible que f soit c.e. et pas $\bar{f}$ (dés que f n'est pas calculable). Ceci nous apprend que la réduction Turing est trop grossière pour travailler sur des énumérations calculables.

Definition 11 (Réduction fortes (many-one))**.** *Soient f et g deux fonctions $A^* \to A^*$. On dit que f se réduit à g au sens de la réduction forte, et on écrit $f \leqslant_m g$, si f est calculable par un programme qui reçoit g en oracle et qui n'appelle g qu'une seule fois à la fin du programme, sans pouvoir changer le résultat de l'appel.*

Une définition équivalente (peut-être plus formelle) est qu'il existe une fonction calculable p telle que $f(x) = g \circ p(x)$ pour toute entrée x.

Il devrait être clair, par définition, que les réductions fortes sont plus fortes que les réductions Turing au sens où $f \leqslant_m g \Rightarrow f \leqslant_T g$. En particulier, comme pour la réduction Turing, si on a $f \leqslant_m g$ et que g est calculable, alors f est calculable. Ce qui est nouveau, en revanche, est le résultat suivant :

Theorem 13. *Si g is c.e. et $f \leqslant_m g$, alors f est c.e.; la même chose est vraie pour co-c.e.*

Cela signifie que les réductions fortes sont adaptés pour travailler sur des notions asymétriques de calculabilité.

5 Fonctions récursives et récursives primitives

Les fonctions récursives sont un modèle de calcul dit fonctionnel, au lieu d'une définition "mécanique" où les fonctions calculables sont des combinaisons d'opérations de calcul élémentaires sur une entrée. Ici, les fonctions calculables sont définies en partant de fonctions de base supposées calculables et en définissant des opérations sur les fonctions qui maintiennent la calculabilité. Pour ce modèle, les fonctions sont $\mathbb{N}^k \to \mathbb{N}$.

5.1 Fonctions récursives primitives

Definition 12. *Les fonctions primitives récursives sont les fonctions de base suivantes :*

Constante 0*;*

Successeur $n \mapsto n + 1$*;*

Projections $\pi_i^k : x_1, \ldots, x_k \to x_i$*,*

qui peuvent être combinées via les opérations suivantes :

Composition *si $f, g_1, \ldots, g_k$ sont récursive primitive, alors $h(x) = f(g_1(x), \ldots, g_k(x))$ le sont .*

Récursion primitive *si f et g sont récursives primitives, alors la fonction définie récursivement par :*

$$h(0, x) = f(x)$$
$$h(n+1, x) = g(h(n, x), n, x)$$

est récursive primitive.

Exercice 10. *Montrez que les fonctions suivantes sont récursives primitives :*

1. *addition add* : $a, b \mapsto a + b$
2. *multiplication mult* : $a, b \mapsto a \times b$

Remarquez que les fonctions récursives primitives s'arrêtent toujours. En effet, la récursion primitive est "contrôlée" dans le sens où le premier argument indique, dés le début, le nombre d'itérations dans la récursion. C'est équivalent à une boucle `for` (en supposant que l'index n'est pas modifié dans la boucle).

Il s'avère que la récursion contrôlée de cette manière est insuffisante pour être Turing-complet, même si on ne considère que les fonctions totales.

Theorem 14. *Il existe une fonction totale calculable qui n'est pas récursive primitive.*

Proof. Commencez par vous convaincre que l'ensemble des fonctions récursives primitives est dénombrable. Soit $f_0, f_1, \ldots$ une énumération calculable de cet ensemble.

Il n'est pas difficile d'écrire un programme universel U pour les fonctions récursives primitives : étant donnée une fonction récursive primitive f et un entier n, $U(f, n)\downarrow = f(n)$.

Par diagonalisation, la fonction calculée par le programme suivant :

```
program diago(int n){
        output U(f_n,n)+1
}
```

ne peut pas être récursive primitive. □

Cet argument est en réalité plus général et s'applique à tout modèle de calcul dont les programmes s'arrêtent toujours. Mieux, tout modèle de calcul où le problème de l'arrêt est décidable ne peut pas être Turing-complet.

Ce constat a des conséquences profondes. L'indécidabilité du problème de l'arrêt n'est pas un aspect secondaire étrange de nos modèles de calcul mais un aspect essentiel. Si vous aviez un mécanisme qui vous empêchait d'écrire des boucles infinies, vous pouvez être sûrs qu'il vous empêcherait également d'écrire des programmes qui s'arrêtent mais où le mécanisme est trop grossier pour le comprendre.

5.2 La fonction d'Ackermann

Definition 13. *La **fonction d'Ackermann*** $\mathrm{Ack} : \mathbb{N}^2 \to \mathbb{N}$ *est définie par :*

- $\mathrm{Ack}(0, p) = p + 1$
- $\mathrm{Ack}(n + 1, 0) = \mathrm{Ack}(n, 1)$
- $\mathrm{Ack}(n + 1, p + 1) = \mathrm{Ack}(n, \mathrm{Ack}(n + 1, p))$

Exercice 11. *Montrez que la fonction d'Ackermann est bien définie (la définition ne boucle pas).*

Theorem 15. *La fonction d'Ackermann n'est pas récursive primitive.*

Proof. On peut facilement prouver par induction les faits suivants : $\mathrm{Ack}(2, x) = 2x + 3$, Ack est croissante en ses deux entrées, $\mathrm{Ack}(x, y + 1) \leqslant \mathrm{Ack}(x + 1, y)$.

On va montrer que pour toute fonction récursive primitive f il existe un t tel que :

$$f(x_1, \ldots, x_n) \leqslant \mathrm{Ack}(t, \max(x_i)).$$

Cela implique qu'il n'existe aucune telle fonction f qui calcule la fonction d'Ackermann.

Fonctions de base La propriété est vraie pour $t = 0$ pour les trois fonctions de base.

Composition Supposons que la propriété est vraie pour un certain t pour les fonctions f et $g_1, \ldots, g_n$. Alors :

$$h(x) = f(g_1(x), \ldots, g_k(x)) \leqslant \mathrm{Ack}(t, \mathrm{Ack}(t, x)) \leqslant \mathrm{Ack}(t + 1, x + 1) \leqslant \mathrm{Ack}(t + 2, x)$$

Récursion primitive Supposons que la propriété est vraie pour un certain t pour les fonctions f et g. Montrons qu'elle est vraie pour $t + 4$ pour la fonction h.

D'abord, montrons par induction sur n que pour tout n, $h(n, x) \leqslant \mathrm{Ack}(t + 1, n + x)$:

- $h(0, x) = f(x) \leqslant \mathrm{Ack}(t, x)$.
- Supposons que c'est vrai pour un certain n. Alors $h(n + 1, x) = g(h(n, x), n, x) \leqslant \mathrm{Ack}(t, \mathrm{Ack}(t + 1, x + n)) = \mathrm{Ack}(t + 1, x + n + 1)$.

Pour conclure, en fixant $m = \max(x, n)$, on a $\mathrm{Ack}(t + 1, n + x) \leqslant \mathrm{Ack}(t + 1, 2m) \leqslant \mathrm{Ack}(t + 1, \mathrm{Ack}(2, m)) \leqslant \mathrm{Ack}(t + 3, m + 1) \leqslant \mathrm{Ack}(t + 4, m)$. □

5.3 Fonction récursives

Une fonction récursive est une fonction obtenus par les opérations précédentes ainsi que :

Minimisation Si f est une fonction récursive, alors

$$\mu(f)(x) = \min\{n \in \mathbb{N} : f(n, x) = 0\} \quad \text{ou} \quad \perp \quad \text{si min n'existe pas}$$

est également récursive.

Theorem 16. *L'ensemble des fonctions récursives est l'ensemble des fonctions calculables.*

C'est un exemple de plus de la thèse de Church-Turing.

6 Classes de complexité

6.1 Panorama

La théorie de la complexité est l'étude des fonctions calculables avec des ressources limitées. Elle est motivée par l'étude de la performance des algorithmes dans un sens large. En fonction du contexte, cette étude peut prendre en compte différents types de ressources : temps ou nombre d'étapes avant qu'un programme termine, nombre de variables ou quantité de mémoire, nombre de composants (portes logiques, états...), quantité de communication (algorithmes sur des réseaux), nombre d'appels d'une certaine opération (e.g. comparaisons pour les algorithmes de tri).

Dans la partie "Calculabilité" de ce cours, on a essayé autant que possible d'oublier le modèle de calcul sous-jacent, parce que tous les modèles amenaient le même ensemble de fonctions calculables (thèse de Church-Turing). Dans la partie "Complexité", la notion de ressource est un aspect du modèle et la plupart des classes ne sont pas robustes à un changement de modèle (si la définition a même du sens).

Il y a à la fois un travail de réflexion important pour comprendre quels modèles et ressources représentent le mieux les limitations de la performance des programmes réels, et un travail théorique pour comprendre la structure et les relations entre les différents types de classes. Par exemple, c'est toujours super de voir deux classes définies par des modèles différents se révéler égales. En tout cas, rappelez-vous que toutes les fonctions qu'on considère à présent sont calculables.

6.2 Classes limitées en temps ou en espace

Definition 14 (Notations o et O)**.** *On note $f(n) = o(g(n))$ (et on dit "f est un petit o de g") quand $\frac{f(n)}{g(n)} \to 0$.*

On note $f(n) = O(g(n))$ (et on dit "f est un grand O de g") s'il existe un constante $C > 0$ telle que $f(n) \leqslant Cg(n)$.

On va travailler dans le modèle classique des machines de Turing.

Definition 15 (Time complexity)**.** *Soit $T : \mathbb{N} \to \mathbb{N}$.*

*Une fonction f est **calculable en temps** $T(n)$ s'il existe une machine de Turing qui, sur toute entrée w de taille n, s'arrête après $T(n)$ étapes de calcul au plus et renvoie $f(w)$. Ceci est noté $f \in TIME(T(n))$.*

- $\mathrm{P} = \mathrm{PTIME} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(n^k)$
- $\mathrm{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathrm{TIME}(2^{n^k})$

Pour des algorithmes concrets, il est également courant de parler de complexité linéaire $O(n)$, complexité quadratique $O(n^2)$, complexité quasi-linéaire $O(n \log n)$...

For real algorithms, it is also usual to talk about linear complexity $O(n)$, quadratic complexity $O(n^2)$, quasi-linear complexity $O(n \log n)$...

This is an example of a larger phenomenon: sublinear time complexity classes are very sensitive to the details of the model. The situation is better for larger classes such

as P, and there is a stronger version of the Church-Turing thesis: Vous vous demandez peut-être pourquoi on ne définit pas une classe LOGTIME = TIME($\log n$). Un temps logarithmique ne laisse pas à une machine de Turing le temps de lire son entrée, ce qui semble un peu absurde. Il est possible de donner un sens à cette notion dans un modèle différent : les machines RAM (Random Access Memory), qui peuvent accéder à n'importe quelle case mémoire en une étape de calcul. Dans ce modèle, la recherche dichotomique termine en temps logarithmique comme on s'y attend.

Il s'agit d'un exemple d'un phénomène plus large : la complexité en temps sous-linéaire est très sensible aux détails du modèle. La situation est meilleure pour des classes plus grandes telles que P, et on a une version plus forte de la thèse de Church-Turing :

Thèse de Church-Turing pour la complexité en temps Tous les modèles de calcul Turing-complets possédant une notion raisonnable de temps peuvent se simuler entre eux en temps polynomial. En particulier, ils définissent les mêmes classes P et EXPTIME.

Un temps polynomial peut être excessif si vous vous intéressez à des calculs réels, mais c'est tout à fait suffisant pour les théoriciens qui s'intéressent à P $\stackrel{?}{=}$ NP, par exemple.

Definition 16 (Complexité en espace)**.** *Une fonction f est **calculable en espace** $S(n)$ s'il existe une machine de Turing qui, sur toute entrée w de taille n, s'arrête et renvoie $f(w)$ sans jamais écrire en dehors des $S(n)$ premières cases du ruban au cours du calcul, excepté pour écrire le résultat. Ceci est noté $f \in SPACE(S(n))$.*

La phrase "excepté pour écrire le résultat" peut être formalisée en utilisant un ruban spécial dit "de résultat" uniquement utilisable en écriture. L'intuition est que la complexité en espace cherche à mesurer la mémoire nécessaire pour effectuer un calcul, et que faire "payer" un programme pour la taille de sa sortie ne correspond pas à cette intuition.

- $L = \text{LOGSPACE} = \text{SPACE}(\log n)$
- $\text{PSPACE} = \bigcup_{k\in\mathbb{N}} \text{SPACE}(n^k)$
- $\text{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \text{TIME}(2^{n^k})$

Exercice 12.
1. *Montrez que $TIME(T(n)) \subset SPACE(T(n))$ (si une fonction est calculable en temps $T(n)$, alors elle est calculable en espace $T(n)$).*
2. *Quelle est la complexité en temps et en espace de la fonction $n \mapsto n + 1$? (il y a un piège)*
3. *Quelle est la complexité en temps et en espace de la fonction $f \vee g$ (ou logique) en fonction des complexités de f et g ?*
4. *Plus difficile : montrez que $PSPACE \subset EXPTIME$. Indice : montrez que $SPACE(S(n)) \subset TIME(C^{S(n)})$ pour une certaine constante C.*

Le prochain résultat est donné sans preuve, mais il est bien plus technique que son apprence semblerait le suggérer. La page Wikipédia sur le sujet est bien faite.

Theorem 17 (Théorème de hiérarchie)**.** *Soient t_1 et t_2 deux fonctions telles que $t_1(n) = o(t_2(n))$. Sous certaines hypothèses techniques, on a $TIME(t_1(n)) \subsetneq TIME(t_2(n))$ (l'inclusion est évidente, mais pas le fait qu'elles ne sont pas égales). Un résultat similaire existe pour l'espace.*

Ce résultat est faux sans les hypothèses techniques et il y a des cas hautement artificiels où deux fonctions différentes décrivent les mêmes classes de complexité.

6.3 Modèles nondéterministes de calcul

Le nondéterminisme est la capacité pour un modèle d'avoir plusieurs comportements dans une configuration donnée. Ce n'est pas une notion très naturelle quand on écrit des programmes; cependant on verra que ces classes peuvent être interprétées comme des fonctions pour lesquelles il est facile de vérifier si une valeur donnée est le bon résultat. Cette classe est également d'un interêt théorique considérable, comme vous le savez sans doute.

CommenCons par la définition directe, où on étend le modèle des machines de Turing pour autoriser du nondéterminisme dans le calcul. La fonction de transition produit maintenant un ensemble de comportements possibles (état, symbole sur le ruban, mouvement de la tête). En partant d'une configuration donnée on a à présent plusieurs calculs possibles, et on doit définir ce qui constitue le résultat du calcul dans ce contexte.

Definition 17. *Une fonction $f : A^* \to \{0,1\}$ est **calculable en temps nondéterministe** $T(n)$ (noté $f \in NTIME(T(n))$) s'il existe une machine de Turing nondéterministe M telle que :*

- *M s'arrête **toujours** après au plus $T(n)$ étapes de calcul ;*
- *Si $f(x) = 0$, alors la machine répond **toujours** 0 sur l'entrée x.*
- *Si $f(x) = 1$, alors la machine répond **parfois** 1 sur l'entrée x (elle peut répondre 0 le reste du temps).*

Notez d'abord que cette définition est **asymétrique** : 0 et 1 ne jouent pas le même rôle, comme pour les énumérations calculables. On peut donc parler de coNTIME, coNSPACE, etc.

Pour une vision plus intuitive du nondéterminisme, on peut écrire des programmes en pseudocode comme auparavant, mais en ayant accès (en plus des opérations habituelles) à une fonction `guess()` qui peut renvoyer 0 ou 1 de manière nondéterministe. Ces programmes nondéterministes sont exactement aussi puissants que les machines de Turing nondéterministes.

Exercice 13. *L'ensemble PRIME des nombres premiers est dans coNP.*

Il existe une définition alternativement des classes nondéterministes qui utilise des machins de Turing standard; cela rend certaines preuves plus faciles, comme la comparaison entre classes déterministes et nondéterministes. Le nondéterminisme est remplacé par une entrée spéciale appelée **certificat** qui indique à la machine quels choix elle doit faire au lieu de deviner.

Theorem 18.

Pourquoi limitons-nous les certificats à une longueur $T(n)$? On a rencontré une idée similaire plus tôt : pusique le programme s'arrête en temps $T(n)$, il n'aurait pas le temps de lire un certificat plus long de toute faCon.

Ici, l'interprétation alternative est qu'un problème est dans NP si un certficat qui garantit que la réponse est "oui" peut être **vérifié** en temps polynômial.

Definition 18 (Classes de complexité nondéterministes)**.** – *$NTIME(T(n))$: fonctions calculable en temps nondéterministe $T(n)$.*

– *$NSPACE(S(n))$: fonctions calculable en espace nondéterministe $S(n)$.*

On définit de la même manière les classes NL = NLOGSPACE, NP = NPTIME, NPSPACE, NEXPTIME...

Mentionnons que PSPACE = NPSPACE (théorème de Savitch), donc les classes d'espace nondéterministes ne sont étudiées que pour les fonctions polynômiales.

Faisons une preuve formelle pour se convaincre que la deuxième définition peut être plus pratique.

Theorem 19.

$$NP \subset PSPACE$$

Proof. Si $f \in$ NP, cela signifie qu'il existe un programme P tel que $f(x) = 0$ si et seulement si $P(x, y) = 0$ pour tous les certificats y de taille $T(|x|)$, pour un certain polynôme T. Considérons le programme suivant :

```
program (input x){
        for all words y of length T(|x|):
                if P(x,y) = 1:
                        output 1
        output 0
}
```

On vérifie facilement que c'est un programme déterministe qui calcule f. La boucle utilise un espace $T(|x|)$ pour écrire successivement les différents certificats, et le programme P utilise un temps polynômial, donc il utilise également un espace polynômial. Par conséquent, le programme entier utilise un espace polynômial. □

Theorem 20 (Panorama)**.**

$$L = LOGSPACE \subset NL \subset P = PTIME \subset NP \subset PSPACE \subset EXPTIME\ldots$$

On a $P \subsetneq PSPACE$ (les classes ne sont pas égales). Cependant, on ne sait pas si $P = NP$, si $NP = PSPACE$, ou si NP est quelque part entre les deux.

Le prochain résultat est donné sans explication; prenez-le comme une anecdote. Pour une classe de complexité C, notez C^f la même classe où toutes les machines de Turing dans la définition reCoivent la fonction f en oracle.

Theorem 21. *Il existe deux fonctions A et B telles que*

$$P^A = NP^A \qquad \text{and} \qquad P^B \neq NP^B$$

Ce résultat infirme imméiatement de nombreuses "preuves" proposées pour P $\stackrel{?}{=}$ NP : si votre preuve continue à fonctionner après avoir ajouté des oracles, elle doit être fausse.

6.4 Complexité randomisée

Un programme randomisé est un programme ayant accès à une opération `random()`, qui répond 0 ou 1 avec probabilité 1/2. Les programmes utilisant l'aléa sont courants, et il semble naturel de développer un cadre permettant d'étudier leur complexité.

Pour formaliser cette notion, la situation est similaire à celle du nondéterminisme:

- on peut définir un nouveau modèle de machine de Turing probabilistes qui ont accès à une opération additionnelle `random`, ou
- on trouve une définition alternative qui n'utilise que les machines de Turing standard.

Les classes de complexité randomisées offrent bien plus de variété que le nondéterminisme ; BPP n'est qu'un exemple parmi d'autres.

Definition 19. *Une fonction f est **calculable en temps** $T(n)$ **avec probabilité bornée** (noté $f \in BPTIME(T(n))$) s'il existe une machine de Turing probabiliste qui, sur toute entrée x de taille n,*

- *s'arrête toujours en temps $T(n)$*
- *$M(x)\downarrow = f(x)$ avec probabilité au moins $\frac{2}{3}$.*

Tout comme dans la section précédente, voici une définition qui utilise les certificats.

Theorem 22. *Une fonction f est calculable en temps $T(n)$ avec probabilité bornée si et seulement s'il existe une machine de Turing telle que, pour toute entrée x de taille n,*

- *$M(x, y)$ s'arrête en temps $T(n)$ pour tout certificat y ;*
- *$M(x, y)\downarrow = f(x)$ pour au moins $\frac{2}{3}$ des certificats ypossibles.*

Comme précédemment, les certificats y ont une longueur $T(n)$. Si le certificat est choisi aléatoirement, la machine s'arrêtera toujours dans le temps attendu et renverra la réponse correcte au moins 2/3 du temps.

Exercice 14.
1. *Que se passe-t-il si on remplace $\frac{2}{3}$ par une autre probabilité, telle que $\frac{1}{2}$ ou $\frac{3}{4}$?*
2. *Montrez que $P \subset BPP$ et que $BPP \subset PSPACE$.*

La question ouverte $\mathrm{P} \stackrel{?}{=} \mathrm{BPP}$ est connue sous le nom de conjecture de la dérandomisation et est une des grosses questions ouvertes en complexité ; elle est liée à des questions sur l'existence de générateurs pseudoaléatoires déterministes forts.

Nous n'avons fait qu'un petit détour par la complexité randomisée, et il existe de nombreuses autres classes qui utilisent l'aléa :

- La réponse est toujours correcte mais le temps n'est borné qu'en moyenne (ZPP).
- Classes asymétriques : la réponse peut être fausse d'un coté mais pas de l'autre (RP).
- Classes n'ayant qu'un nombre limité d'appels à la fonction `random`, etc.

La théorie de la complexité possède de nombreux sous-domaines avec des outils et approches bien spécifiques. En arrivant dans un nouveau domaine, vous verrez souvent que vos outils habituels ne fonctionnent plus, mais le plus important est de comprendre le sens et la raison derrière les définitions.

7 Réductions et complétude en complexité

7.1 Réductions, le retour

Nous retrouvons les réductions, qui servent à comparer la complexité calculatoire des fonctions.

Nous avons déjà rencontré les réductions pour la calculabilité dans les sections 3.3 et 4.2 (pour la calculabilité générale et asymétrique, respectivement). Ces réductions ne sont pas adaptées aux classes de complexité: souvenez-vous que deux fonctions calculables sont toujours Turing-équivalentes. Pour comparer deux fonctions d'une même classe C, nous allons chercher des réductions appropriées.

Definition 20. *Soit C une classe de complexité et $\leqslant$ une réduction.*
Une fonction f est C-difficile pour $\leqslant$ si $g \leqslant f$ pour toute $g \in C$.
f est C-complète si, de plus, $f \in C$.

Les fonctions C-complètes sont "les fonctions les plus difficiles de C".
On verra dans un moment comment choisir $\leqslant$ pour que ces notions se révèlent utiles.

Exercice 15. *1. Si C est l'ensemble des fonctions calculables, quelles sont les fonctions C-complètes pour $\leqslant$? Les fonctions C-difficiles ?*

2. Si f est C-complète et $f \leqslant g$, que cela signifie-t-il pour g ? et si $g \leqslant f$?

*3. Une réduction $\leqslant_1$ est **plus fine** qu'une réduction $\leqslant_2$ si $f \leqslant_2 g \Rightarrow f \leqslant_1 g$. Quelle relation a-t-on entre les ensemble de fonctions C-complètes pour $\leqslant_1$ et $\leqslant_2$?*

Échauffons-nous avec un peu de complétude en calculabilité :

Theorem 23. *Le problème de l'arrêt est c.e.-complet pour $\leqslant_m$.*

Proof. Pour toute fonction c.e. $f : A^* \to \{0,1\}$, il existe un programme $prog_f$ qui, sur l'entrée x, s'arrête si et seulement si $f(x) = 1$ (Theorem 11). Utilisez ce programme pour écrire une réduction au problème de l'arrêt. □

Remarquez qu'une fonction calculable ne peut pas être c.e.-complète.

7.2 Réductions limitées en temps ou en espace

Comme une réduction n'est qu'un programme qui a accès à un oracle, on peut définir le temps et la mémoire utilisées par le programme en considérant qu'appeler l'oracle prend 1 étape de calcul et n'utilise aucune mémoire supplémentaire.

Definition 21. *Si f et g sont deux fonctions, g se réduit à f en temps $T(n)$ s'il existe un programme recevant g en oracle qui, sur toute entrée x de taille n, s'arrête et répond $f(x)$ en temps $T(n)$. De même pour l'espace.*

On définit de la même manière les réductions fortes (many-one).

Comment choisir une réduction adaptée à l'étude d'une certaine classe ? Il y a deux critères principaux :

1. La réduction doit être moins puissante (utiliser moins de ressources) que les fonctions de la classe.
2. La réduction doit être forte (many-one) si on travaille dans une classe asymétrique.

Premier point. Si la réduction a accés à plus de ressources que les fonctions de la classe, alors on peut avoir $f \leqslant g$ et $g \in C$ mais $f \notin C$, ce qui va à l'encontre de l'intuition de "plus difficile". Si la réduction a accès à exactement les mêmes ressources, alors la classe entière sera C-complète: par exemple, toutes les fonctions de P sont équivalentes pour la réduction Turing en temps polynômial[6]. Cette réduction serait cependant un choix raisonnable pour étudier PSPACE.

Second point. Ce point est la motivation derrière les réductions fortes. Comme les réductions Turing permettent d'échanger librement une réponse 0 ou 1 en restant équivalent, elles ne permettent pas de distinguer NP et co-NP, par exemple. Si ce point n'est pas clair, retournez lire la Section 11.

Quelques exemples

- La P-complétude est habituellement définie pour des réductions LOGSPACE[7].
- La NP-complétude est définie pour des réductions fortes en temps polynômial, également appelées **réductions de Karp**.
- La PSPACE-complétude et l'EXPTIME-complétude sont définies par des réductions de Turing en temps polynômial.

En dehors de ces cas classiques, la plupart des papiers soulignerons quelle réduction est considérée.

7.3 Problèmes complets naturels

Pour le moment, il n'est pas clair qu'il existe des problèmes complets. Nous allons montrer une technique générale pour produire un premier problème complet dans chaque classe: prédire le comportement d'une machine de Turing correspondant à la définition de la classe.

Voici le premier problème de prédiction P-complet :

$$pred_P : \begin{array}{rcl} \mathcal{M} \times \{0,1\}^* \times \mathbb{N}[X] & \to & \{0,1\} \\ (M,x,p) & \to & \begin{cases} 1 \text{ si } M(x)\downarrow = 1 \\ \quad \text{après au plus } p(|x|) \text{ étapes de calcul} \\ 0 \text{ sinon.} \end{cases} \end{array}$$

Les problèmes d'arrêt ou de prédiction sont équivalents mais ces derniers rendent les preuves un peu plus claires.

Theorem 24.

[6]Indice : La réduction peut faire le calcul sans utiliser l'oracle.

[7]Il existe d'autres possibilités, comme les réductions NC. La question de savoir si elles amènent au même ensemble de fonctions P-complètes est un problème ouvert.

Proof. Pour faire simple, prenons une fonction $f : A^* \to \{0,1\} \in \mathrm{P}$. Par définition de P, il y a un programme M et un polynôme T tel que $M(x)\downarrow = f(x)$ en temps $T(n)$ pour toute entrée x, où n est la taille de x.

Pour calculer f en ayant accès à un oracle $pred_P$, il suffit de calculer $pred_P(M, x, T)$. Rappelez-vous que l'appel à l'oracle (en particulier, la longueur de x) n'est pas compté dans la complexité en espace. □

Si vous retournez lire vos notes sur la NP-complétude, vous pouvez vérifier que le théorème de Cook – qui prouve que SAT est NP-complet – est en réalité une réduction au problème de prédiction dans les machines de Turing nondéterministes :

$$\begin{array}{rcl} \mathcal{M} \times \{0,1\}^* \times \mathbb{N}[X] & \to & \{0,1\} \\ (M, x, p) & \to & \begin{cases} 1 \text{ s'il existe un certificat } y \text{ tel que } M(x,y)\downarrow = 1 \\ \quad \text{en au plus } p(|x|) \text{ étapes de calcul} \\ 0 \text{ sinon.} \end{cases} \end{array}$$

8 Calculabilité sur les nombres réels

Cette section a deux objectifs : voir comment on peut étendre nos définitions de calculabilité sur des espaces indénombrables, et comprendre comment la calculabilité est la motivation derrière des propriétés mathématiques classiques.

8.1 Nombres réels individuels

Nous avons vu en Section 2.1 que notre définition de calculabilité nécessitait de représenter les objets concernés comme des fonctions sont les entrées et sorties sont dénombrables (possédant une description finie).

Note. *L'ensemble de telles fonctions n'est pas dénombrable; on dit qu'il a la puissance du continu. La raison pour laquelle les nombres réels peuvent être représentés comme des fonctions dont les entrées–sorties sont dénombrables est qu'ils ont également la puissance du continu.*

De fait, les nombres réels ont deux représentations principales. On peut représenter $x \in \mathbb{R}$:

1. par une représentation binaire (ou décimale) :

$$bin_x : \begin{array}{rcl} \mathbb{N} & \to & \{0,1\} \\ n & \mapsto & n - \text{ième bit de } x \end{array}$$

2. par une approximation rationnelle :

$$approx_x : \begin{array}{rcl} \mathbb{N} & \to & \mathbb{Q} \\ n & \mapsto & q \text{ tel que } |q - x| \leqslant \frac{1}{2^n} \end{array}$$

Dans le second cas, le choix de la borne d'approximation $\frac{1}{2^n}$ n'est pas significatif. Cela pourrait être, par exemple, $\frac{1}{n}$.

Remarquez que tout nombre réel peut être approché par une séquence de nombres rationnels, et que tout nombre rationnel est calculable. Ce qui est important ici est qu'un seul programme calcule la séquence entière de nombres rationnels. Une telle séquence est parfois appelée **uniformément calculable** pour insister sur le fait qu'un unique programme calcule la séquence entière.

Theorem 25. *Ces deux représentations sont équivalentes. Autrement dit,*

$$\forall x \in \mathbb{R}, bin_x \equiv_T approx_x.$$

Proof. On sépare les cas où x est rationnel ou irrationnel.

- $x \in \mathbb{Q}$. Dans ce cas, bin_x and $approx_x$ sont calculables. la représentation de x est ultimement périodique, et on peut donc écrire un programme qui considère tous les cas et répond le bit correspondant. Par exemple, si $x = 0,11010010010010\ldots$, bin_x serait:

  $$1, 2 \mapsto 1 \quad \text{et sinon,} \quad n \mapsto 1 \text{ if } n \equiv 1 \bmod 3, 0 \text{ sinon,}$$

  et $approx_x$ n'est que la fonction constante qui répond x.

- $x \notin \mathbb{Q}$.

  1. $bin_x \geqslant_T approx_x$: en ayant accès à un oracle bin_x, on peut calculer une approximation rationnelle de x à $\frac{1}{2^n}$ près en ne gardant que les n premiers bits de sa représentation binaire. Formellement,

     ```
     program approx_x (input n) oracle bin_x{
             out = 0
             for i from 1 to n:
                     out = 2*out + bin_x(i)
             output out/2^n
     }
     ```

  2. $bin_x \leqslant_T approx_x$: il s'agit du cas subtil. Remarquez d'abord que le long de la ligne réelle, le n-ième bit change à intervalle 2^{-n} (plus précisément, les limites des intervalles sont de la forme $k2^{-n}$ avec $k \in \mathbb{Z}$).

     puisque $x \notin \mathbb{Q}$, x n'est pas la limite d'un intervalle, donc pour k assez grand, $x - 2^{-k}$ et $x + 2^{-k}$ appartiennent au même 2^{-n}-intervalle. Cela signifie qu'on connait le k-ième bit de x si on en a une approximation à 2^{-n} près.

     ```
     program bin_x (input k) oracle approx_x{
             for n from k to infinity:
                     a = approx(x, n)
             if a − 2^−k and a + 2^−k have the same n-th bit:
                 output this bit
     }
     ```

     Si $x \in \mathbb{Q}$, ce programme boucle : quand x est sur une limite d'intervalle, aucune approximation ne sera suffisante. □

8.2 Fonctions à valeurs réelles

Dans cette section, on va étendre petit à petit notre cadre pour définir la calculabilité de fonctions $\mathbb{R} \to \mathbb{R}$. $\mathbb{R}$ n'est pas dénombrable, et ces fonctions ne peuvent pas en général être représentées comme des fonctions dont les entrées–sorties sont dénombrables[8], et on a donc besoin de nouvelles définitions.

Premièrement, réflechissons aux stratégies utilisées par les programms réels pour faire du calcul sur les réels :

- Approche **symbolique** : on fixe un certain nombre de symboles (π, $\log(n)$, $\cos(n)\dots$) et on ne manipule que les nombres réels exprimables de cette manière – un sous-ensemble dénombrable.
- Approche par **approximation** : on travaille sur des nombres rationnels (e.g. nombres de la forme $\frac{q}{2^n}$ à précision fixée) et les entrées et sorties des fonctions sont des approximations.

Cette deuxième approche va amener à des définitions plus robustes (la première n'est "que" de la réecriture de chaînes de caractères). Tout en travaillant sur les définitions, assurons-nous qu'elles correspondent à un sens pratique de capacité à calculer le résultat, et on va retrouver des notions mathématiques bien connues.

Definition 22. *Une fonction $f : \mathbb{Q} \to \mathbb{R}$ est calculable s'il existe un programme M qui, étant donné une entrée x, fournit des approximations de $f(x)$ à toute précision voulue :*

$$\forall x \in \mathbb{Q}, \forall n \in \mathbb{N}, \quad |M(x,n)\downarrow - f(x)| \leqslant \frac{1}{2^n}$$

Exercice 16. *Si $f : \mathbb{Q} \to \mathbb{R}$ est calculable, montrez que $f(q)$ est calculable pour tout $q \in \mathbb{Q}$.*

Pour travailler avec des fonctions $\mathbb{R} \to \mathbb{R}$, l'entrée comme la sortie devront être fournies comme des approximations. Cependant, considérez la fonction $1_{\mathbb{Q}}$ ($1_{\mathbb{Q}}(x) = 1$ si $x \in \mathbb{Q}$ et 0 sinon). Quel est le sens que pourrait avoir le fait de calculer cette fonction ?

Si on souhaite approcher $f(x)$ mais qu'on ne dispose que d'une approximation de x (disons, on connait un $x_n \in \mathbb{Q}$ tel que $|x - x_n| \leqslant \frac{1}{2^n}$), on doit s'assurer que $f(x_n)$ est une bonne approximation de $f(x)$ – sinon, aucun calcul ne donnera un résultat raisonnable.

Cette hypothèse revient à supposer que f est **continue** dans un sens assez fort :

Pour calculer une approximation de $f(x)$ avec précision $\frac{1}{2^n}$,
on demande une approximation de x avec précision $\frac{1}{2^{g(n)}}$ où g est une fonction calculable.

g est appelé le **module de continuité** en analyse. Finalement, on voit qu'il y a deux sources d'erreurs à gérer :

- notre entrée n'est qu'une approximation de l'entrée voulue.
- notre programme ne peut répondre qu'une approximation de la valeur demandée, même sur une entrée exacte (rationnelle).

[8] C'est dû au fait que l'ensemble de ces fonctions est plus grand que le continu

On parvient enfin à la définition suivante qui, bien qu'un peu complexe, correspond à la notion intuitive de calculer une fonction réelle :

Definition 23. *Une fonction $f : \mathbb{R} \to \mathbb{R}$ est calculable s'il existe deux programmes C (calcul) et P (précision) tels que :*

$$\forall x \in \mathbb{R}, \forall n \in \mathbb{N}, |x_n - x| \leqslant \frac{1}{2^{P(n)\downarrow}} \Rightarrow |C(x_n, n)\downarrow - f(x)| \leqslant \frac{1}{2^n}$$

Theorem 26. *Si f est calculable, alors f est continue.*

Comment utiliser cette définition pour calculer $f(x)$ à précision $\frac{1}{2^n}$ en pratique ?

1. Calculer $P(n+1)$.
2. Fournir un rationnel x_n qui approche x avec précision $\frac{1}{2^{P(n+1)}}$.
3. Répondre $C(x_n, n+1)$.

Exercice 17.
1. *Montrez que si $f : \mathbb{R} \to \mathbb{R}$ est calculable et que x est calculable, vous pouvez écrire un programme qui, sur l'entrée n, répond une approximation de $f(x)$ à précision $\frac{1}{2^n}$.*
2. *Corollaire de ce qui précède : montrez que si x est calculable, alors $f(x)$ est calculable.*

Nous avons utilisé pour ces définitions des représentations des réels par approximations rationnelles. On aurait pu demander au programme de répondre les n premiers bits de x à la place. Ca ne fonctionnerait pas du tout. Essayez de prouver que $x \mapsto 3x$ est calculable en décimal et vous rencontrerez des problèmes.

Quelques exemples de fonctions réelles incalculables :

– N'importe quelle fonction discontinue ;

– La fonction constante $x \mapsto r$ où r est un nombre réel incalculable.

9 Dynamique symbolique ; calculabilité dans d'autres domaines

9.1 Espaces de pavage et problème du domino

Jusqu'à présent, tous les problèmes indécidables ou complts concernaient des programmes. Ici Turing FM, les calculabilistes parlent aux calculabilistes.

Vous connaissez probablement déjà plusieurs problèmes naturels NP-complets ; la plupart des classes de complexité ont des problèmes naturels qui apparaissent dans divers domaines mathématiques et informatiques. Allons en rencontrer quelques-uns dans le domaine des **pavages**.

Definition 24.

Le problème fondamental du domain est le **problème du domino** : étant donné un espace de pavage (A et $\mathcal{F}$), existe-t-il un pavage admissible ?

Ce problème est indécidable. Nous allons ici traiter un problème plus simple pour rendre les preuves plus accessibles.

Definition 25. *Le **problème du domino avec graine** :*

Entrée *Un alphabet, un ensemble de motifs interdits, un motif de départ appelé **graine** et un entier n (en unaire),*

Sortie *La réponse à l'une des questions suivantes (suivant la variante considérée) :*

- *Seed(∞, ∞) : la graine peut-elle être prolongée par un pavage admissible de $\mathbb{Z}^2$?*
- *Seed(n, n) : la graine peut-elle être prolongée par un pavage admissible du carré $[0, n]^2$?*
- *Seed(n, ∞): la graine peut-elle être prolongée par un pavage admissible de la bande $[0, n] \times \mathbb{Z}$?*

Ces problèmes semblent plutôt faciles à première approche, mais des difficultés combinatoires apparaissent dés que le nombre de symbole augmente, comme on le verra. Déjà, quels programmes peut-on écrire ?

- *Seed(n, n)* : ce n'est pas difficile d'écrire un programme de force brute en espace polynomial qui teste toutes les possibilités. En devinant les n^2 couleurs du pavage et en vérifiant que c'est correct en temps polynômial, on peut voir que le problem est en fait dans NP.
- *Seed(n, ∞)* : pour simplifier, supposons que les motifs interdits sont des contraintes d'adjacence (deux symboles). Essayez de paver le rectangle $n \times |\mathcal{A}|^n$. Si vous ne trouver aucune manière admissible de le faire, la réponse est non. Si vous y parvenez, alors une ligne est répétée (principe pigeon-trou) et on peut paver la bande infinie en répétant la partie périodique, donc la réponse est oui.

  En devinant successivement les lignes du pavages on peut faire de cette idée un programme NPSPACE = PSPACE.
- *Seed(∞, ∞)*: on peut essayer de paver chaque carré $[0, n]^2$ par force brute comme précédemment. Si, pour un certain n, un carré n'a pas de pavage admissible, la réponse est non. Si on trouve un pavage admissible périodique, la réponse est oui. Est-ce certain qu'on trouver toujours un tel pavage ? Pour le moment, on peut dire avec certitude que le problème est co-calculablement énumérable, car le programme s'arrête si la réponse est non.

9.2 Simulation de calcul universel

On peut produire un alphabet, un ensemble de motifs interdits et une graine tels que tout pavage admissible représente le calcul d'une machine de Turing sur une entrée donnée par la graine (la machine peut être nondéterministe si nécessaire) :

(todo pictures)

Ainsi, tout pavage admissible qui étend la graine doit contenir la calcul de la machine de Turing.

Cette simulation amène aux résultats de complétude suivants :

Theorem 27. *1. $Seed(\infty, \infty)$ est co-c.e.-complet.*

2. $Seed(n, n)$ est NP-complet.

3. $Seed(n, \infty)$ est PSPACE-complet.

Proof. □

J'affirme sans preuve que le problème $Seed(1, n)$ est NL-complet ; si Ca vous intéresse, renseignez-vous sur les problèmes d'atteignabilité dans les graphes.

9.3 Interprétation de ces résultats

Dans cet exemple, comme souvent, les problèmes de complétude en complexité et en calculabilité viennent main dans la main suivant qu'on considère les versions finies et infinies des problèmes, et ces résultats proviennent d'une capacité à simuler du calcul universel dans un contexte combinatoire apparamment innocent.

Notez d'abord qu'on travaille dans un contexte de complexité / calculabilité dans le pire des cas. Ces problèmes pourraient être résolubles dans 99,9% des cas par des heuristiques — c'est un gros souci en cryptographie, par exemple. Il est possible de fournir de meilleures garanties, c'est un autre domaine de la complexité.

Néanmoins, ces résultats nous disent qu'on ne trouvera pas de méthode générale pour résoudre un problème (ou au moins pour le résoudre rapidement) : il s'agit de déclarations d'impossibilité ou de bornes inférieures pour la complexité informatique, mais aussi mathématique.

Conceptuellement, ces résultats d'impossibilité devraient avoir la même valeur que des résultats positifs, mais on peut trouver Ca un peu aride[9]. Un point de vue plus positif est qu'une méthode générale "tue" le problème — il n'y a plus rien à faire — tandis qu'un résultat de complétude fournit une bonne justification pour travailler sur des variantes plus simples, des approximations, des hypothèses supplémentaires, etc., et donne indirectement plus de valeurs aux résultats partiels.

Si vous êtes nostalgiques des Hic Svnt Dracones des anciennes cartes, maintenant qu'il ne reste rien à explorer, les résultats d'indécidabilité et de complétude sont vos dragons — et ils resteront là, prouvablement, pour toujours.

[9] pour candidater à des bourses...