École polytechnique — X2011 — INF423 Fondements de l'informatique

Calculabilité et complexité

Sujet proposé par David Monniaux
(corrigé)

Cette PC revient sur les questions de calculabilité et aborde les premières notions de complexité. La NP-complétude fera l'objet de la PC suivante.

1 Pas de langage universel pour les calculs qui terminent toujours

Nos problèmes de décidabilité proviennent de l'usage de systèmes de calcul (p. ex. les machines de Turing, mais aussi bien tout langage de programmation comportant des boucles non bornées) dont les programmes peuvent ne pas terminer. Nous avons déjà vu un système de programmation dont tous les programmes terminent (les fonctions primitives récursives), mais nous avons montré que ce système ne permet pas de représenter certaines fonctions calculables qui terminent toujours (comme la fonction d'Ackermann). Nous allons voir ici que pareille faiblesse est inévitable : aucun système, ou langage, de programmation ne permet de représenter toutes les fonctions calculables qui terminent toujours et exactement celles-ci.

Nous appelons « fonction calculable totale » une fonction calculable qui termine toujours.

Question 1.1. *Supposons qu'il existe un interpréteur I pour ce langage de programmation : la fonction calculable totale $I(x, y)$ retourne le résultat de l'exécution du programme x sur l'entrée y. On dit que ce langage* représente *une fonction calculable s'il existe un x_0 tel que cette fonction est $y \mapsto I(x_0, y)$. Montrez qu'il existe des fonctions calculables qui ne sont pas représentées par ce langage. (Indication : construisez-en une à partir de I.)*

Solution : C'est encore une fois un argument diagonal ! La fonction $f(x) = I(x, x) + 1$ est calculable totale. Supposons qu'elle soient représentable dans ce langage : il existe x_0 tel que $f(x) = I(x_0, x)$ pour tout x. C'est notamment le cas pour $x = x_0$ d'où $I(x_0, x_0) = I(x_0, x_0) + 1$ contradiction. □

En d'autres termes, nous venons de démontrer que si nous définissons un langage de programmation qui ne permet d'écrire que des fonctions qui terminent forcément, alors nous ne pourrons pas écrire l'interpréteur de ce langage avec ce langage (exemple pratique : le langage Coq ne permet d'écrire que des fonctions qui terminent, donc on ne peut écrire d'interpréteur Coq en Coq).

On s'intéresse maintenant au langage des codages de machines de Turing qui terminent forcément :

$$T = \{x \mid \forall y \text{ La machine de codage } x \text{ termine sur } y\}.$$

Question 1.2. *Prouvez que le langage T n'est pas décidable.*

Solution : C'est une application directe du théorème de Rice, par rapport à la propriété non triviale « termine forcément ». □

On pourrait au moins espérer qu'il soit semidécidable (récursivement énumérable) ou de complémentaire semidécidable...

Question 1.3. *Prouvez que le langage complémentaire*

$$\overline{T} = \{x \mid \exists y \text{ La machine de codage } x \text{ ne termine pas sur } y\}$$

n'est pas semidécidable.

Solution : Soit x une machine de Turing. On définit la machine de Turing $g(x)$ ainsi : elle ignore son entrée y puis exécute x sur l'entrée vide. Alors $g(x)$ ne termine pas sur au moins une entrée si et seulement si x ne termine pas sur l'entrée vide. Si $\overline{T}$ est semidécidable, alors l'ensemble des machines de Turing qui ne terminent pas sur l'entrée vide l'est aussi ; comme son complémentaire est clairement semidécidable, alors le problème de l'arrêt est décidable, ce qui est absurde. □

Question 1.4. *Prouvez que le langage T n'est pas semidécidable.*

Solution : Soit x une machine de Turing. On définit la machine de Turing $f(x)$ ainsi :

- Elle prend en entrée y.
- Elle simule x pendant y étapes sur l'entrée vide.
- Si cette simulation a atteint un état final acceptant, elle boucle ensuite indéfiniment.
- Sinon, soit la simulation a atteint un état final refusant, soit elle n'a pas terminé en y étapes, et alors f termine en acceptant.

Cette machine vérifie $f(x) \in T$ si et seulement si x ne termine pas sur l'entrée vide.

Supposons maintenant que T est semidécidable. On en déduit donc que l'ensemble des machines de Turing qui ne terminent pas sur l'entrée vide est semidécidable. Or, le complémentaire de cet ensemble est clairement semidécidable. On obtient donc une propriété non-triviale décidable, ce qui contredit le théorème de Rice. □

2 Complexité

2.1 Étoiles

Étant donné un langage $A \subseteq \Sigma^*$, on note A^* le langage formé des mots de la forme $\omega_1 \cdots \omega_n$ où pour tout i, $\omega_i \in A$, y compris le mot vide.

Question 2.1. *Soit A un langage dans NP. Montrez que A^* est dans NP.*

Solution : Pour le mot vide, le certificat est le certificat que le mot vide $\varepsilon \in A$.

Pour des mots non vides, on peut se restreindre sans perte de généralité au cas où ω se décompose en $\omega_1 \cdots \omega_n$ où chacun des ω_i est non vide et dans A ; notamment $n \leq |\omega|$. Le certificat consiste alors en cette décomposition (on donne les indices des lettres où l'on coupe) et en certificats que les $\omega_i \in A$. □

Question 2.2. *Soit A un langage dans P. Montrez que A^* est dans P.*

Solution : L'approche brutale « étant donné ω on teste tous les découpages de ω en $\omega_1 \cdots \omega_n$ pour $n \leq |\omega|$ » donnerait une complexité exponentielle.

On procède par *programmation dynamique* : on construit une matrice M de bits indexée par (i, j) où $0 \leq i < j \leq |\omega|$ où $M_{i,j}$ dit si oui ou non la portion $\omega_{i,j}$ de ω située entre les indices i inclus et j exclu est dans A^*. On calcule $M_{i,j}$ en fonction des $M_{i',j'}$ avec $0 \leq i' < j' \leq j$ et $j' - i' < j - i$, ainsi : on commence par tester si $\omega_{i,j} \in A$; sinon c'est qu'il faut tester un découpage en $\omega_{i,k} \cdot \omega_{k,j}$ avec $i < k < j$, et on les teste tous.

L'astuce est de ne jamais recalculer un bit déjà calculé (autrement dit on garde non seulement la matrice M, mais aussi une matrice auxiliaire qui dit quels bits ont déjà été calculés). Il y a un nombre quadratique de bits à calculer, et pour chacun il faut tester un nombre linéaire de découpages en deux morceaux, d'où un nombre au plus cubique de tests d'appartenance à A. □

2.2 Primalité

Soit PRIMES l'ensemble des entiers naturels premiers, écrits en binaire.

Question 2.3. *Montrez que PRIMES est dans co-NP (c'est-à-dire que son complémentaire est dans NP).*

Solution : Un certificat de non-primalité de n, de taille linéaire en celle de n, consiste juste à donner (a, b) avec $n = ab$ et $1 < a, b < n$. □

On admet le résultat suivant (théorème de Lehmer) [1] : un nombre n est premier si et seulement s'il existe $1 < a < n$ tel que $a^{n-1} \equiv 1 \pmod{n}$ et pour tout facteur premier q de $n-1$, $a^{(n-1)/q} \not\equiv 1 \pmod{n}$ (autrement dit, il existe un élément d'ordre $n-1$ dans $(\mathbb{Z}/n\mathbb{Z})^*$).

Question 2.4. *Montrez que PRIMES est dans NP.*

Solution : Par le théorème de Lehmer, un certificat de primalité pour p consiste en a (de taille linéaire en p), en une décomposition de $p-1$ en facteurs $p_1 \cdots p_k$ et en certificats de primalité des p_k. En effet, avec ces informations, on peut vérifier en temps polynomial les conditions ci-dessus. Les certificats de primalité forment donc un arbre, dont les feuilles correspondent à $p = 2$.

Montrons par induction que pour cet arbre a au plus $4 \log_2 p - 4$ nœuds internes (les nœuds qui ne sont pas des feuilles, c'est-à-dire ceux correspondant à $p' > 2$). Le cas de base $p = 2$ est trivial ; passons à $p > 2$. Considérons un sous-arbre dont la racine est nœud étiqueté par p et dont les fils sont étiquetées par $p_1, \ldots, p_k$. $k = 0$ est absurde car cela veut dire $p - 1 = 1$. Le cas $k = 1$ veut dire que $p - 1$ est premier, ce qui ne peut se produire que pour $p = 3$, pour lequel la propriété est vraie par calcul. Considérons maintenant $k \geq 2$. Il a par hypothèse de récurrence au plus pour taille $1 + \sum_{i=1}^{k} (4 \log_2 p_k - 4)$ soit $(1 - 4k) + \log_2 p \leq 4 \log_2 p - 4$. □

Note : on sait depuis seulement 2002 que PRIMES est en fait dans P, mais la preuve est trop compliquée pour être donnée en PC...

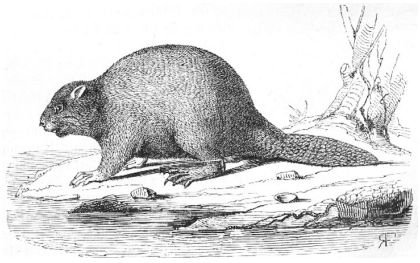
En pratique, on n'utilise de toute façon pas le test déterministe en temps polynomial, mais un test probabiliste beaucoup plus rapide, dont on peut rendre aussi petite que l'on veut la probabilité de pouvoir se tromper en déclarant qu'un nombre est premier alors qu'il ne l'est pas.

La suite en BONUS pour vous occuper si vous avez du temps libre...

---

3 Le castor affairé

La fonction du « castor affairé » (*busy beaver*) a été introduite en 1962 par Tibor Radó [3]. Considérons les machines de Turing déterministes à n états $(1 \ldots n)$, parmi lesquels on distingue un état initial, plus un état final (noté 0), opérant sur un seul ruban infini dans les deux directions, sur l'alphabet $\{0, 1\}$ (le 0 servant de « blanc »). Chaque machine est donc déterminée par une fonction qui à chaque état $1 \leq k \leq n$, et chaque valeur lue $\{0, 1\}$, associe une indication de direction dans $\{-1, +1\}$ et une valeur à écrire dans $\{0, 1\}$.



Le castor, par Alcide Raillet, 1895

1. Le petit théorème de Fermat dit que si p est premier, alors pour tout a, $a^{p-1} \equiv 1 \pmod{p}$. La réciproque n'est pas vraie sans une hypothèse supplémentaire, qu'apporte le théorème de Lehmer, à la base d'algorithmes de preuves de primalité.

Question 3.1. *Donnez le nombre $N(n)$ de machines de Turing à n états du type ci-dessus. (On distinguera les machines identiques à renommage d'états près.)*

Solution : À chaque couple (état, valeur lue), dans $\{1, \ldots, n\} \times \{0, 1\}$ (ensemble à $2n$ éléments), on associe un triplet (nouvel état, direction, valeur écrite) dans $\{0, \ldots, n\} \times \{-1, +1\} \times \{0, 1\}$, ensemble à $4(n+1)$ éléments. On a donc

$$N(n) = [4(n+1)]^{2n} \tag{1}$$

□

Parmi ces machines, on nomme « castors à n états » les machines qui, sur une entrée entièrement vide (ruban constitué uniquement de 0), terminent dans l'état final. Le score d'un castor est le nombre de 1 écrits sur le ruban.

Question 3.2. *Montrez que l'ensemble des castors à n états est non vide.*

Solution : Prendre une machine triviale qui saute immédiatement à l'état final. □

L'ensemble des castors à n états étant non vide et fini, il admet donc un score maximal, noté $\Sigma(n)$. On appelle Σ *fonction du castor affairé.*

Question 3.3. *Montrez qu'il n'y a pas de fonction calculable f telle que $f(n) \geq \Sigma(n)$ pour tout n.*

Solution : Une autre démonstration est proposée par Radó [3].

L'idée est de montrer que cette fonction pourrait servir à borner le nombre de pas des machines de Turing qui terminent, et qu'ainsi on pourrait décider de la terminaison : on simule la machine jusqu'à la borne, et si elle n'a pas encore terminé, c'est qu'elle ne terminera jamais.

Soit x le codage d'une machine de Turing à 1 bande. À partir de cette machine, on construit une machine qui simule la machine x, mais en traitant les cases du ruban t par paquets de 3 cases consécutives :

- t_{3i} simule la case numéro i du ruban de la machine simulée.
- $t_{3i+1} = 1$ veut dire que la tête de la machine simulée est sur la case i (il n'y a donc qu'une seule case $t_{3i+1} = 1$ à 1, les autres sont à 0).
- Les cases $t_{3i+2} = 1$ pour $0 \leq k \leq i$ sont à 1, les autres à 0, pour dire que la machine simulée a exécuté exactement 1 pas.

Il est possible de construire une telle machine, de codage $s(x)$, avec un nombre d'états linéaires en celui de x, telle que s soit calculable. Remarquons que la machine $s(x)$ termine sur l'entrée vide si et seulement si la machine x termine sur l'entrée vide.

Notons $|s(x)|$ le nombre d'états de $s(x)$. On peut construire une machine $s'(x)$ qui simule $s(x)$, en vérifiant à chaque fois que l'on écrit un 1 dans une case t_{3i+2} que le nombre de ces cases ne dépasse pas $f(|s(x)|)$. Si ce nombre le dépasse, on sait que la machine $s(x)$ ne s'arrêtera jamais.

On a donc une méthode qui permet de déterminer si la machine de Turing x s'arrête, ce qui contredit le théorème de l'arrêt. □

Notons que ceci montre que Σ n'est pas calculable.

Question 3.4. *Montrez que l'ensemble des codages des machines de Turing définissant des castors est non récursif.*

Solution : Si c'était le cas, on pourrait énumérer toutes les machines de taille n, ne garder que les castors, les exécuter, et calculer $\Sigma(n)$. □

Question 3.5. *Montrez qu'il n'y a pas de fonction calculable f et d'indice N telle que $f(n) \geq \Sigma(n)$ pour tout $n \geq N$.*

Solution : Supposons qu'une telle fonction existe. Soit la fonction définie ainsi : pour $0 \leq k < N$, $f'(k) = \Sigma(k)$, et pour $k \geq N$, $f'(k) = f(k)$. f' vérifie les hypothèses de la question précédente, d'où la contradiction.

Remarque : pour passer du codage de la machine de Turing calculant f au codage de la machine de Turing calculant f', nous aurions besoin, avec cette construction, de calculer $\Sigma(k)$ pour $0 \leq k < N$. Cependant, nous n'avons pas besoin de faire de calcul algorithmiquement. Il nous suffit de savoir que $\Sigma(k)$ est un certain entier ; autrement dit notre construction de f' est non effective. □

En informatique théorique, h est *élémentaire* s'il existe une tour d'exponentielles telle que $h(n) \leq 2^{\cdot^{\cdot^{2^n}}}$ à partir d'un certain n.

Question 3.6. *Montrez que Σ est non élémentaire.*

Solution : Les tours d'exponentielles sont calculables, on conclut par la question précédente. □

Références

[1] Dexter C. Kozen. *Theory of computation.* Springer, 2006. ISBN 1846282977.

[2] Michael Machtey and Paul Young. *An introduction to the general theory of algorithms.* Computer science library, Theory of computation series. North-Holland, New York, 1978. ISBN 044400226X.

[3] Tibor Radó. On non-computable functions. *Bell System Technical Journal*, 41(3) :877–884, May 1962.

[4] Hartley Rogers, Jr. *Theory of recursive functions and effective computability.* MIT Press, 1987. ISBN 0-262-68052-1.

[5] Carl A. Smith. *A recursive introduction to the theory of computation.* Graduate texts in computer science. Springer-Verlag, 1994. ISBN 0387943323.