2 Considérations de base sur le temps

Une fois réalisée la formalisation des algorithmes vue au chapitre précédent, les mathématiciens se sont intéressés à des questions liées au type de problèmes que l'on pouvait ainsi résoudre (calculabilité). Plus tard, lors de l'apparition des premiers ordinateurs, la question de l'efficacité de ces algorithmes s'est posée et le domaine de la complexité algorithmique a vu le jour.

Dans ce chapitre, nous allons voir les prémices de la complexité, qui remontent aux années 1960 : définition des classes de complexité en temps, théorèmes de hiérarchie, relations entre classes, etc. Nous introduirons également les machines non déterministes et le problème « $\mathsf{P} = \mathsf{NP}$? » qui a une grande importance depuis les années 1970.

2.1 Temps déterministe

Le modèle des machines de Turing que nous avons vu au chapitre précédent servira à définir les classes de complexité. Rappelons que le *temps* mis par une machine M sur une entrée x est le nombre d'étapes du calcul de $M(x)$ pour arriver à un état final (définition 1-F). Cela revient en quelque sorte à mesurer le temps d'exécution de l'algorithme pour résoudre le problème sur l'entrée x, mais on veut une mesure indépendante de la puissance de l'ordinateur faisant tourner l'algorithme : c'est pourquoi on ne parle pas en secondes mais en nombre d'étapes de calcul. Grâce à cette notion, nous pouvons définir des classes de complexité en temps.

2.1.1 Classes de complexité en temps

Un peu de recul

Pour la plupart des problèmes, plus la taille de l'entrée est grande, plus il faudra du temps pour trouver la solution. Par exemple, il est évident qu'on mettra plus de temps à trier une liste d'un million d'éléments plutôt qu'une liste de 3 éléments.

Il est alors naturel d'évaluer le temps de calcul d'un algorithme en fonction de la taille de son entrée. Cette observation donne lieu à la définition suivante.

La définition suivante apparaît pour la première fois dans l'article [HS65] d'Hartmanis et Stearns.

2-A Définition (classes de complexité en temps déterministe)

– Pour une fonction $t : \mathbb{N} \to \mathbb{N}$, la classe $\mathsf{DTIME}(t(n))$ est l'ensemble des langages reconnus par une machine de Turing M telle qu'il existe une constante α pour laquelle, sur toute entrée x, $M(x)$ fonctionne en temps $\leqslant \alpha t(|x|)$.

– Si $\mathscr{T}$ est un ensemble de fonctions, alors $\mathsf{DTIME}(\mathscr{T})$ désigne $\cup_{t \in \mathscr{T}} \mathsf{DTIME}(t(n))$.

2-B Remarque Dans le domaine de la complexité (contrairement à la calculabilité), toutes les machines que nous considérerons s'arrêteront sur toute entrée. C'est par exemple implicite dans la définition précédente puisque $M(x)$ est censée fonctionner en temps $\leqslant \alpha t(|x|)$. La question n'est plus de savoir si une machine s'arrête, mais en combien de temps elle le fait.

2-C Remarque Pour résoudre la plupart des problèmes non triviaux, il faut lire la totalité de l'entrée x. Ainsi le temps de calcul mis par la machine est au moins $n = |x|$, et les classes $\mathsf{DTIME}(t(n))$ ont peu d'intérêt pour $t(n) = o(n)$.

Avant de poursuivre, mentionnons quelques propriétés évidentes des classes $\mathsf{DTIME}(t(n))$.

2-D Proposition

1. Si pour tout n, $f(n) \leqslant g(n)$ alors $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DTIME}(g(n))$;
2. pour tout $t(n) \geqslant n$, $\mathsf{DTIME}(t(n))$ est clos par union finie, intersection finie et complémentaire.

Démonstration 1. Si $L \in \mathsf{DTIME}(f(n))$ alors il existe une machine reconnaissant L en temps au plus $\alpha f(n)$ donc a fortiori en temps majoré par $\alpha g(n)$, ce qui implique $L \in \mathsf{DTIME}(g(n))$.

2. Si $L_1, L_2 \in \mathsf{DTIME}(t(n))$, soit M_1 et M_2 deux machines reconnaissant L_1 et L_2 en temps $\leqslant \alpha_1 t(n)$ et $\leqslant \alpha_2 t(n)$ respectivement. Alors une machine pour $L_1 \cup L_2$ (resp. $L_1 \cap L_2$) est la machine $M_\cup(x)$ (resp. $M_\cap(x)$) qui exécute $M_1(x)$ puis $M_2(x)$ et accepte ssi l'une des deux accepte (resp. les deux acceptent). Puisque cette machine doit revenir au début du ruban de lecture entre les calculs de M_1 et de M_2, elle fonctionne en temps $\leqslant \alpha_1 t(n) + \alpha_2 t(n) + O(n)$. Puisque $t(n) \geqslant n$, on a donc $L_1 \cup L_2 \in \mathsf{DTIME}(t(n))$ (resp. $L_1 \cap L_2 \in \mathsf{DTIME}(t(n))$).

   Une machine pour ${}^c L_1$ est la machine $M_c(x)$ qui exécute $M_1(x)$ et accepte ssi $M_1(x)$ rejette : elle fonctionne en temps $\leqslant \alpha_1 t(n)$ donc ${}^c L_1 \in \mathsf{DTIME}(t(n))$. □

Pourquoi définir la classe DTIME à une constante α près ? La raison vient du résultat suivant qui montre que le fonctionnement d'une machine peut être accéléré par n'importe quel facteur constant. En d'autres termes, le temps de calcul d'une machine n'est une mesure pertinente qu'à une constante près.

Un peu de recul

Il n'y a pas de magie dans ce résultat : pour accélérer une machine, il suffit d'agrandir l'alphabet pour faire plus de calculs en une étape. C'est grosso-modo ce qu'on fait en remplaçant les micro-processeurs 32 bits par des micro-processeurs 64 bits.

Comme on l'a vu, on ne peut pas descendre en dessous de n pour le temps d'exécution d'une machine raisonnable, ce qui explique la forme $(1+\epsilon)n + \epsilon t(n)$ utilisée dans le théorème suivant. Si $n = o(t(n))$ alors cela signifie réellement une accélération par n'importe quelle constante. Ce résultat est dû encore à Hartmanis et Stearns [HS65].

2-E Théorème (accélération linéaire)

Pour toute constante $\epsilon > 0$, si un langage L est reconnu par une machine M fonctionnant en temps $\leqslant t(n)$, alors il existe une machine M' reconnaissant L et fonctionnant en temps $\leqslant (1+\epsilon)n + \epsilon t(n)$.

Idée de la démonstration L'astuce consiste à agrandir l'alphabet pour pouvoir réaliser en une seule fois plusieurs étapes de calcul de la machine de départ. Pour réaliser c étapes d'un coup, il faut connaître le voisinage de rayon c autour de la cellule en cours : chaque nouvelle cellule contiendra des $(2c+1)$-uples d'anciennes cellules (ce qui revient à passer à l'alphabet de travail Γ^{2c+1}). En réalité nous n'allons pas simuler c anciennes étapes en une seule nouvelle, mais en 6 nouvelles : les informations nécessaires sont en effet contenues dans les voisines de gauche et de droite qu'il nous faut donc visiter (voir la figure 2.1).

Démonstration La machine M' contient un ruban de travail de plus que M. On pose $c = \lceil 6/\epsilon \rceil$: si l'alphabet de travail de M est Γ, alors celui de M' est $\Gamma \cup \Gamma^c$ (où Γ^c, produit cartésien c fois de Γ avec lui-même, est l'ensemble des c-uples d'éléments de Γ). Sur les rubans de travail de M', chaque cellule contient un bloc de c symboles consécutifs de M. Le ruban d'entrée de M' contient bien sûr l'entrée x sur n cases sur l'alphabet $\Sigma \subset \Gamma$: la première phase de M' consiste à recopier x sur le ruban de travail supplémentaire sur l'alphabet Γ^c : le mot x prend maintenant $\lceil n/c \rceil$ cases. Puisqu'il faut revenir au début du ruban de travail supplémentaire, cette phase requiert $n + \lceil n/c \rceil \leqslant (1+\epsilon)n$ étapes de calcul. Ce ruban de travail supplémentaire simulera le ruban d'entrée de M.

Chaque groupe de c transitions de M est simulé par 5 ou 6 transitions de M' : d'abord, indépendamment du mouvement de M, quatre transitions pour lire les cases voisines (un déplacement à droite suivi de deux à gauche puis un à droite), puis un ou éventuellement deux dernières transitions pour simuler l'écriture et le déplacement de M (voir la figure 2.1).

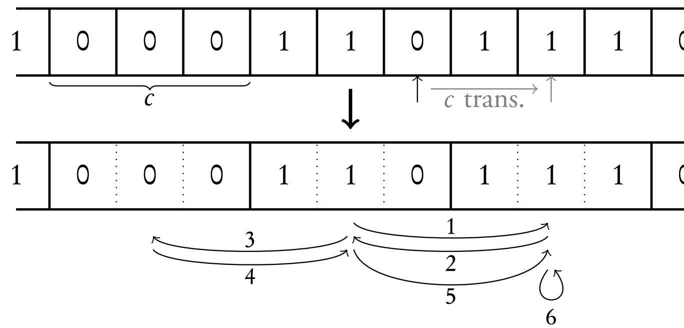


Figure 2.1 – Simulation de c transitions en 6 étapes.

Pour simplifier l'explication, nous traiterons chaque ruban de travail séparément : il faudrait en réalité ajouter des états pour retenir le contenu des cases voisines sur tous les rubans.

Pour chaque état q de M, la machine M' contient $c|\Gamma^{2c}| + 2c|\Gamma^c| + 2c$ états pour retenir le contenu des cases voisines et les déplacements à effectuer :

- pour $1 \leqslant i \leqslant c$, un état $q'_{i,D}$ pour débuter la séquence de 5 ou 6 étapes : $q'_{i,D}$ permet de retenir la position i de la tête dans le bloc de c cases et de prévoir un mouvement D ;
- puis M' se déplace à droite en passant dans l'état $q'_{i,G}$ pour prévoir un mouvement G ;
- elle se déplace ensuite à gauche et retient le contenu $u \in \Gamma^c$ de la case qu'elle vient de lire (celle de droite) en passant dans l'état $q'_{i,u,G}$ (le prochain mouvement sera G) ;
- elle se déplace encore à gauche en passant dans l'état $q'_{i,u,D}$ pour prévoir un mouvement D ;

– elle se déplace à droite et retient le contenu $v \in \Gamma^c$ de la case qu'elle vient de lire (celle de gauche) en passant dans l'état $q'_{i,u,v}$.

Après ces 4 déplacements, M' est revenue à la case de départ. Grâce à son état, elle connaît maintenant le contenu de ses deux voisines et, grâce à la lecture de la case courante, celui de la case en cours. En d'autres termes, du point de vue de M, elle connaît le contenu d'au moins c cases vers la gauche et vers la droite : elle peut donc simuler c transitions de M.

Pour cela, il suffit de définir la fonction de transition δ' de M' de sorte que, sur l'état $q'_{i,u,v}$ et en lisant le uple $w \in \Gamma^c$, elle effectue l'équivalent de c transitions de M à partir de la i-ème case du bloc courant. Cela implique de modifier la case en cours en fonction de ce qu'écrit M, puis si nécessaire d'aller modifier la voisine de gauche ou de droite (l'une seulement des deux pouvant être modifiée en c transitions de M) et soit revenir sur la case de départ, soit rester sur la voisine modifiée. Pour effectuer cela, la cinquième transition de M' modifie la case de départ et se déplace du bon côté (G, S ou D) ; si nécessaire, la sixième transition modifie la nouvelle case pointée par la tête et se déplace éventuellement (G ou S si l'on est sur la case de droite, D ou S si on est sur la case de gauche). On passe également dans un nouvel état selon la nouvelle position de la tête de M et son état après c étapes.

Ainsi, $c \geqslant 6/\epsilon$ étapes de M sont simulées par $\leqslant 6$ étapes de M'. En comptant la phase initiale de recopie de l'entrée x sur un ruban de travail supplémentaire, le temps de calcul de $M'(x)$ est majoré par $(1+\epsilon)n + \epsilon t(n)$. □

Un peu de recul

Il est crucial de garder à l'esprit que la complexité d'un problème est une **mesure asymptotique**, un temps d'exécution lorsque la taille de l'entrée tend vers l'infini. Ainsi, pour montrer qu'un langage A est dans $\mathsf{DTIME}(n^2)$, il suffit par exemple de donner un algorithme pour A qui fonctionne en temps $3n^2$ *pour n suffisamment grand*. Peu importe son comportement si l'entrée est petite : s'il fonctionne en temps $3n^2$ pour n suffisamment grand, alors il existe une constante c tel qu'il fonctionne en temps $\leqslant cn^2$ pour tout n. Puisque la plupart du temps, la valeur de la constante nous importe peu, nous dirons simplement que l'algorithme fonctionne en temps $O(n^2)$.

En conséquence, dans nombre de raisonnements nous nous intéresserons seulement aux entrées suffisamment grandes.

2.1.2 Théorème de hiérarchie

Nous allons maintenant voir l'un des résultats fondateurs de la complexité : le théorème de hiérarchie en temps déterministe. Il s'agit de montrer que l'on peut résoudre strictement plus de problèmes si l'on dispose de plus de temps, ce qui n'est pas très surprenant. Pour formaliser cela, nous devons introduire le concept de *fonction constructible en temps*.

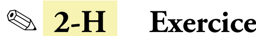
2-F Définition (fonction constructible en temps)

Une fonction $t : \mathbb{N} \to \mathbb{N}$ est *constructible en temps* s'il existe une constante α et une machine de Turing M qui, sur l'entrée 1^n (l'entier n en unaire) renvoie $1^{t(n)}$ (l'entier $t(n)$ en unaire) en temps $\leqslant \alpha t(n)$.

En quelque sorte, les fonctions constructibles en temps sont des fonctions $t : \mathbb{N} \to \mathbb{N}$ « raisonnables » qui ont de bonnes propriétés. En réalité, la plupart des fonctions que nous utilisons sont constructibles en temps. En voici quelques exemples.

2-G Exemple Les fonctions suivantes $t : \mathbb{N} \to \mathbb{N}$ sont constructibles en temps :

- $t(n) = c$ pour une constante $c \in \mathbb{N}$;
- $t(n) = n$;
- $t(n) = 2^{n^c}$ pour une constante $c \in \mathbb{N}$;
- de plus, si t_1 et t_2 sont constructibles en temps, alors il en est de même de $t_1 + t_2$ et $t_1 t_2$: ainsi, tout polynôme $t \in \mathbb{N}[n]$ est constructible en temps.



2-H Exercice

Montrer que les fonctions ci-dessus sont constructibles en temps.

2-I Exercice

Montrer qu'une fonction constructible en temps $f(n)$ telle que $f(n) = o(n)$ est ultimement constante.

Indication : une machine fonctionnant en temps $o(n)$ se comporte de la même façon sur les entrées 1^n et 1^{n+1} pour n suffisamment grand.

Le rôle des fonctions constructibles en temps s'illustre par exemple dans le théorème de hiérarchie suivant : une machine disposant de plus de temps peut décider plus de choses. C'est à nouveau un résultat de Hartmanis et Stearns [HS65] mais sous la forme qui suit il est dû à Hennie et Stearns [HS66] grâce à leur machine universelle avec ralentissement logarithmique vue au théorème 1-S.

2-J Théorème (hiérarchie en temps déterministe)

Soit $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$ des fonctions telles que $f(n) \neq 0$ (pour tout $n \in \mathbb{N}$), g est constructible en temps et $f \log f = o(g)$. Alors $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n))$.

Idée de la démonstration Il s'agit de construire un langage $L \in \mathsf{DTIME}(g(n))$ tel que $L \notin \mathsf{DTIME}(f(n))$. Pour cela, nous allons faire en sorte que toute machine fonctionnant

en temps $f(n)$ et censée reconnaître L se trompe sur au moins une entrée. Dans la lignée des paradoxes logiques célèbres, on utilisera l'auto-référence par une question de la forme « est-ce qu'une machine M sur son propre code rejette en temps $\leqslant f(n)$? », sur laquelle toute machine fonctionnant en temps $f(n)$ doit se tromper. Modulo quelques détails techniques, le langage L sera l'ensemble des machines M pour lesquelles la réponse à la question précédente est positive.

Pour décider L, il faudra simuler une machine fonctionnant en temps $f(n)$ par une machine universelle, ce qui lui prend $O(f(n)\log f(n))$ étapes (théorème 1-S), d'où la condition du théorème.

Démonstration Nous aimerions définir $L = \{\langle M\rangle \mid M(\langle M\rangle) \text{ rejette en temps } \leqslant f(n)\}$ où M désigne une machine de Turing, mais nous ne savons pas décider ce langage en temps $g(n)$ (à cause de la constante α_M dépendant de M dans le théorème 1-S). En gardant la même idée, il va falloir être légèrement plus subtil. On va d'abord ajouter un mot x quelconque à l'entrée du problème (donc travailler sur l'entrée $(\langle M\rangle, x)$ à la place de $\langle M\rangle$) pour pouvoir agrandir arbitrairement la taille de l'entrée, puis définir le langage en termes de fonctionnement d'une machine.

Soit V la machine suivante sur l'entrée $(\langle M\rangle, x)$:

- exécuter $U(\langle M\rangle, (\langle M\rangle, x))$ pendant $g(n)$ étapes, où U est la machine universelle du théorème 1-S et $n = |(\langle M\rangle, x)|$ (c'est-à-dire qu'on simule $M(\langle M\rangle, x)$ tant que la simulation prend moins de $g(n)$ étapes) ;
- si U n'a pas terminé son calcul, alors rejeter ;
- sinon, accepter ssi U rejette.

On note L le langage reconnu par V.

On remarquera que V calcule d'abord $g(n)$ pour savoir quand arrêter l'exécution de U : puisque g est constructible en temps, cela prend $O(g(n))$ étapes. Puis l'exécution de U prend à nouveau $g(n)$ étapes, donc en tout V fonctionne en temps $O(g(n))$: ainsi, $L \in \mathsf{DTIME}(g(n))$.

Montrons maintenant que $L \notin \mathsf{DTIME}(f(n))$. Soit M un machine fonctionnant en temps $\alpha f(n)$. Ainsi U simule M en temps $\alpha_M \alpha f(n)\log f(n)$ (où α_M est une constante dépendant seulement de M). Pour n assez grand, par hypothèse

$$g(n) \geqslant \alpha_M \alpha f(n)\log f(n),$$

donc pour x assez grand, la simulation de M par U se termine avant $g(n)$ étapes, donc $V(\langle M\rangle, x)$ accepte ssi $M(\langle M\rangle, x)$ rejette. Ainsi, M « se trompe » sur l'entrée $(\langle M\rangle, x)$ et ne reconnaît donc pas L. □

2-K Remarques

– Nous verrons au théorème 6-O une astuce permettant de réduire un peu l'écart

entre les fonctions f et g dans le théorème précédent (c'est-à-dire obtenir mieux que $f \log f = o(g)$). On pourrait également essayer de construire une machine universelle plus efficace que celle du théorème 1-S, mais il s'agit d'une question ouverte comme on l'a vu (remarque 1-T).

– Si l'on se restreint aux machines de Turing ayant *toutes* exactement k rubans de travail pour $k \geqslant 2$ fixé, alors Fürer [Für82] montre un théorème de hiérarchie optimal, c'est-à-dire sans le facteur $\log(f(n))$.

Le théorème de hiérarchie est très intuitif puisqu'il nous apprend qu'en disposant de plus de temps de calcul, on peut résoudre plus de problèmes. Cependant, l'hypothèse de constructibilité en temps de la fonction g est cruciale : en effet, le résultat suivant montre ce que l'on peut faire sans supposer g constructible. Il est dû à Trakhtenbrot [Tra64] en 1964 (rédigé en russe) et a été redécouvert par Borodin [Bor72] en 1972 pour le monde occidental sous le nom de « Gap theorem ».

2-L Théorème (de la lacune)

Il existe une fonction $f : \mathbb{N} \to \mathbb{N}$ telle que $f(n) \geqslant n$ et $\mathsf{DTIME}(f(n)) = \mathsf{DTIME}(2^{f(n)})$.

Idée de la démonstration L'idée est de définir la fonction f de sorte qu'aucune machine de Turing ne s'arrête en un temps compris entre $f(n)$ et $2^{f(n)}$, ce qui implique bien sûr qu'il n'y a aucun langage dans $\mathsf{DTIME}(2^{f(n)}) \setminus \mathsf{DTIME}(f(n))$.

Démonstration Si M est une machine, on note Σ_M son alphabet d'entrée. La propriété suivante sera vérifiée : pour tout $n \in \mathbb{N}$, aucune machine M telle que $|\langle M \rangle| \leqslant n$ et $|\Sigma_M| \leqslant n$ ne s'arrête sur une entrée de taille n en un temps compris entre $f(n)$ et $n2^{f(n)}$.

Posons arbitrairement $f(0) = 0$ et définissons $f(n)$ pour un $n > 0$ donné. Pour cela on considère la suite $(u_i)_{i \geqslant 1}$ définie par $u_1 = n$ et $u_{i+1} = n2^{u_i} + 1$: ainsi, les intervalles $[u_i, n2^{u_i}]$ forment une partition de $[n, +\infty[$. Soit M une machine telle que $|\langle M \rangle| \leqslant n$ et $|\Sigma_M| \leqslant n$ et soit $x \in \Sigma_M$ de taille n. Si $M(x)$ ne s'arrête pas, alors M ne décide pas de langage. Sinon, $M(x)$ s'arrête en un temps $t_{M,x}$: alors soit $t_{M,x} < n$, soit $t_{M,x} \in [u_i, 2^{u_i}]$ pour un certain i.

Il y a $\leqslant 2^{n+1} - 1$ machines M telles que $|\langle M \rangle| \leqslant n$ et il y a $\leqslant n^n$ mots de taille n dans Σ_M (car $|\Sigma_M| \leqslant n$), donc il y a $< n^n 2^{n+1}$ valeurs $t_{M,x}$ distinctes. On en déduit que l'un des intervalles $[u_i, n2^{u_i}]$, pour $1 \leqslant i \leqslant n^n 2^{n+1}$, ne contient aucun $t_{M,x}$. Si $[u_k, n2^{u_k}]$ est un tel intervalle, on définit $f(n) = u_k$ (on remarquera que $f(n) \geqslant n$ puisque $u_1 = n$). Ainsi, si $|\langle M \rangle| \leqslant n$ et $|\Sigma_M| \leqslant n$, alors pour tout mot $x \in \Sigma_M$ de taille n, $M(x)$ ne s'arrête pas en un temps compris entre $f(n)$ et $n2^{f(n)}$.

Maintenant que f est définie, soit $L \in \mathsf{DTIME}(2^{f(n)})$. Alors une machine M reconnaît L en temps $\leqslant \alpha 2^{f(n)}$ pour une certaine constante α. Pour n suffisamment grand, on a $|\langle M \rangle| \leqslant n$, $|\Sigma_M| \leqslant n$ et $\alpha 2^{f(n)} \leqslant n2^{f(n)}$. Donc, pour $|x|$ suffisamment grand, $M(x)$ ne s'arrête pas en un temps compris entre $f(|x|)$ et $|x|2^{f(|x|)}$: puisqu'elle s'arrête en un temps $< |x|2^{f(|x|)}$, on en déduit que $M(x)$ doit s'arrêter en un temps $< f(|x|)$. D'où $L \in \mathsf{DTIME}(f(n))$. □

✎ **2-M Exercice**

Montrer qu'on peut prendre f strictement croissante dans le théorème ci-dessus.
Indication : modifier la valeur de départ de la suite (u_i) dans la démonstration.

2-N Remarque Le nom de ce théorème contre-intuitif vient du « trou » qu'il y a entre $f(n)$ et $2^{f(n)}$, intervalle dans lequel on ne trouve aucun langage. Rappelons que f n'est pas constructible en temps, sinon on contredirait le théorème de hiérarchie.

En réalité, n'importe quelle fonction $s : \mathbb{N} \to \mathbb{N}$ conviendrait à la place de $n \mapsto 2^n$ (c'est-à-dire $\mathsf{DTIME}(f(n)) = \mathsf{DTIME}(s(f(n)))$ pour une fonction f construite comme ci-dessus), c'est-à-dire qu'on peut rendre le « trou » aussi grand que l'on veut. Ce théorème est habituellement donné pour une fonction s calculable quelconque (notion que nous n'avons pas définie) ; dans ce cas, f est elle-même calculable.

2.1.3 Temps polynomial et temps exponentiel

Nous pouvons maintenant définir deux classes de complexité qui joueront un rôle important par la suite. L'idée de considérer comme notion d'algorithmes « efficaces » les algorithmes fonctionnant en temps polynomial, remonte aux articles de Cobham [Cob65] et d'Edmonds [Edm65] en 1965.

2-O Définition (temps polynomial et exponentiel)

La classe P est l'ensemble des langages reconnus en temps polynomial, c'est-à-dire

$$\mathsf{P} = \mathsf{DTIME}(n^{O(1)}) = \cup_{k\in\mathbb{N}} \mathsf{DTIME}(n^k).$$

La classe EXP est l'ensemble des langages reconnus en temps exponentiel, c'est-à-dire

$$\mathsf{EXP} = \mathsf{DTIME}(2^{n^{O(1)}}) = \cup_{k\in\mathbb{N}} \mathsf{DTIME}(2^{n^k}).$$

Enfin, une dernière classe que l'on étudiera moins est notée E, il s'agit de l'ensemble des langages reconnus en temps exponentiel avec exposant linéaire, c'est-à-dire :

$$\mathsf{E} = \mathsf{DTIME}(2^{O(n)}) = \cup_{k\in\mathbb{N}} \mathsf{DTIME}(2^{kn}).$$

On a bien sûr $\mathsf{P} \subseteq \mathsf{E} \subseteq \mathsf{EXP}$. Nous pouvons d'ores et déjà appliquer le théorème de hiérarchie 2-J pour séparer ces classes.

2-P Corollaire

$$\mathsf{P} \subsetneq \mathsf{EXP}$$

Démonstration Il suffit de remarquer que $\mathsf{P} \subseteq \mathsf{DTIME}(2^n)$ puis, par le théorème 2-J, que $\mathsf{DTIME}(2^n) \subsetneq \mathsf{DTIME}(2^{n^2})$ et enfin que $\mathsf{DTIME}(2^{n^2}) \subseteq \mathsf{EXP}$. □

2-Q Exercice

> Séparer de même E et P.

2-R Exemple Voici quelques exemples de problèmes dans P et dans EXP.

1. `MULTIPLICATION D'ENTIERS` :
   - *entrée* : deux entiers a et b donnés en binaire ; un entier k ;
   - *question* : le k-ème bit du produit ab vaut-il 1 ?

   Ce problème est dans P : il suffit d'effectuer la multiplication ab grâce à l'algorithme de l'école primaire et de regarder le k-ème bit du résultat. Cela prend un temps polynomial.
2. `MULTIPLICATION DE MATRICES` :
   - *entrée* : deux matrices A et B de taille $m \times m$ à coefficients entiers donnés en binaire ; un couple (i, j) avec $1 \leqslant i, j \leqslant m$; enfin, un entier k ;
   - *question* : le k-ème bit du coefficient (i, j) du produit AB vaut-il 1 ?

   Ce problème est dans P : le coefficient (i, j) de AB n'est rien d'autre que la somme des m produits $a_{i,k}b_{k,j}$, ce qui se calcule en temps polynomial.
3. `ACCESSIBILITÉ` :
   - *entrée* : un graphe orienté G et deux sommets s et t ;
   - *question* : existe-t-il un chemin dans G de s à t ?

   Ce problème est dans P : il suffit de faire un parcours du graphe en partant de s et de voir si l'on atteint t (parcours en largeur ou en profondeur, peu importe).
4. `COUPLAGE PARFAIT` :
   - *entrée* : un graphe non orienté G ;
   - *question* : G admet-il un couplage parfait ?

   Un couplage parfait est un ensemble d'arêtes tel que chaque sommet est relié à exactement une arête de cet ensemble. Ce problème est dans P mais l'algorithme polynomial n'est pas trivial et est dû à Edmonds [Edm65].

5. `PRIMALITÉ` :
   - *entrée* : un entier N donné en binaire ;
   - *question* : N est-il premier ?

   Ce problème est dans P mais la démonstration n'est pas évidente (l'algorithme naïf consistant à essayer tous les diviseurs potentiels jusqu'à $\sqrt{N}$ est exponentiel puisque la taille de l'entrée est $n = \log N$) : il a fallut attendre Agrawal, Kayal et Saxena [AKS04] pour l'obtenir.
6. `RANGEMENT` :
   - *entrée* : des entiers $p_1, \dots, p_n \in \mathbb{N}$ en binaire (les poids de n objets) et deux entiers m, p en binaire (le nombre de boîtes de rangement et le poids maximal que chacune peut contenir) ;
   - *question* : peut-on ranger les n objets dans les m boîtes en respectant le poids maximal p de chaque boîte ?

   Ce problème est dans EXP car il suffit de tester toutes les possibilités pour ranger les objets (c'est-à-dire pour chaque i, décider dans laquelle des m boîtes mettre l'objet i) et de voir si l'une d'entre elles respecte les contraintes. Il y a $m^n = 2^{n \log m}$ possibilités, donc ce calcul prend un temps exponentiel en la taille de l'entrée. En réalité, bien que ce problème ne soit probablement pas dans P, on peut quand même faire mieux que EXP car il appartient à la classe NP que nous verrons par la suite (il est NP-complet). Il est appelé `BIN PACKING` en anglais.
7. `ARRÊT EXP` :
   - *entrée* : le code d'une machine M et un entier k en binaire ;
   - *question* : est-ce que $M(\epsilon)$ s'arrête en $\leqslant k$ étapes ?

   Ce problème est dans EXP car on peut simuler $M(\epsilon)$ pendant k étapes : puisque la taille de l'entrée k est $\log k$, cela prend un temps exponentiel. Nous n'avons pas encore vu cette notion, mais mentionnons au passage que ce problème est EXP-complet.
8. Un problème de EXP plus « naturel » est, sur l'entrée C, de calculer la valeur du circuit booléen D dont le code est décrit par le circuit booléen C donné en entrée (nous définirons les circuits au chapitre 5). Puisque D peut être de taille exponentielle par rapport à C, ce calcul prend un temps exponentiel. Comme le précédent, ce problème est EXP-complet (exercice B-C).

2-S Exercice

Justifier plus rigoureusement que ces problèmes sont chacun dans la classe annoncée. Pour le problème du couplage parfait, voir [Edm65] ou un livre d'algorithmique comme [Cor+10]. Pour le problème de la primalité, voir [AKS04].

Un peu de recul

La classe P est généralement présentée comme celle des problèmes résolubles « efficacement ». Bien qu'en pratique on sache résoudre rapidement certains problèmes hors de P, et par ailleurs qu'il soit difficile d'admettre qu'un algorithme fonctionnant en temps $1\,000\,000 n^{1\,000}$ soit efficace, il n'en reste pas moins que cette présentation est très souvent pertinente : la plupart des problèmes « naturels » dans P ont un algorithme en n^k pour un petit exposant k et pour la plupart des problèmes « naturels » hors de P, on ne connaît pas d'algorithme efficace.

On verra plus tard que les algorithmes probabilistes polynomiaux ajoutent encore à la confusion. Toujours est-il que la classe P est « robuste » (bonnes propriétés, notamment de clôture) et donne un bon cadre d'étude en complexité.

Notre première propriété de P sera la clôture de P par changements finis. Si A et B sont deux langages, rappelons que leur différence symétrique $A \Delta B$ est l'ensemble des mots contenus dans exactement un seul des deux ensembles, c'est-à-dire $A \Delta B = (A \setminus B) \cup (B \setminus A)$.

2-T Lemme

Soit L et L' deux langages sur un alphabet Σ. Si $L \in \mathsf{P}$ et $L \Delta L'$ est fini (c'est-à-dire que L et L' diffèrent seulement en un nombre *fini* de mots), alors $L' \in \mathsf{P}$.

Idée de la démonstration La machine pour L' va contenir dans son code la liste (finie) des mots de $L \Delta L'$. Pour décider si $x \in L'$, on décide d'abord si $x \in L$ puis on corrige éventuellement la réponse en testant si x appartient à la liste des mots de $L \Delta L'$.

Démonstration Soit $\{x_1, \dots, x_m\}$ l'ensemble des mots de $L \Delta L'$. Soit M une machine polynomiale pour le langage L. Voici une machine M' pour L' sur l'entrée x :

- exécuter $M(x)$;
- si $x \in \{x_1, \dots, x_m\}$, alors `accepter` ssi $M(x)$ rejette ;
- sinon `accepter` ssi $M(x)$ accepte.

L'exécution de $M(x)$ prend un temps polynomial. Pour tester si $x \in \{x_1, \dots, x_m\}$, on compare x à chacun des mots x_i, lesquels sont codés « en dur » dans le code de la machine M'. Chaque comparaison prend un temps linéaire ; puisque le nombre de comparaisons est fixé une fois pour toute, le test $x \in \{x_1, \dots, x_m\}$ prend lui aussi un temps linéaire. Au total, M' fonctionne en temps polynomial et reconnaît L', donc $L' \in \mathsf{P}$. □