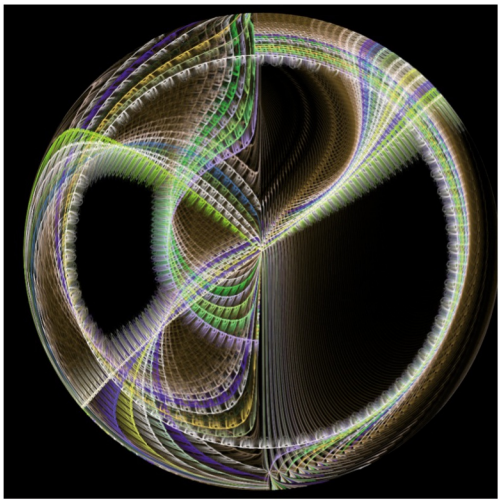
Théorie de la complexité computationnelle

NP-complétude, PCP, hiérarchie polynomiale, circuits et communication



Lucien Sina

Théorie de la complexité computationnelle

NP-complétude, PCP, hiérarchie polynomiale, circuits et communication

Lucien Sina

1.1.2026

Table des matières

Avant-propos

La théorie de la complexité est l'un des domaines les plus fascinants et en même temps les plus exigeants de l'informatique théorique. Elle traite d'une question apparemment simple, mais d'une profonde importance : quel effort est nécessaire pour résoudre un problème ? Tandis que la théorie de la calculabilité étudie quels problèmes sont en principe résolubles par des algorithmes, la théorie de la complexité s'intéresse à l'efficacité de ces solutions. Elle fait ainsi le lien entre les principes fondamentaux du calcul et la faisabilité pratique des procédés algorithmiques.

Cet ouvrage est né du désir de présenter les concepts, méthodes et idées de la théorie de la complexité de manière systématique et en même temps accessible. Il conduit des notions élémentaires, comme les classes de complexité et les réductions, jusqu'à des thèmes modernes tels que la théorie PCP, les systèmes de preuve interactifs, la complexité spatiale, la complexité des circuits et de la communication. À chaque étape, il s'efforce non seulement

de donner des définitions et des théorèmes, mais aussi de dévoiler les intuitions et les liens sous-jacents.

Un objectif central de ce livre est de rendre visible la structure interne de la théorie de la complexité : Comment les différents concepts se construisent-ils les uns sur les autres ? Comment les thèmes de recherche actuels se déduisent-ils de questions classiques ? Et comment peut-on tirer des limites théoriques des enseignements utiles pour la conception pratique d'algorithmes ?

Les chapitres sont conçus pour pouvoir être lus individuellement ou dans leur continuité. Chacun traite d'un domaine thématique clos, des fondements à la NP-complétude puis aux champs de recherche plus récents. Par des exemples, des preuves et des exercices complémentaires, le lecteur est encouragé à travailler activement avec les notions et à en saisir la portée.

Ce livre se veut non seulement une introduction, mais aussi une synthèse dans le contexte d'autres sujets fondamentaux de l'informatique. Mes ouvrages complémentaires — *Algorithmes et structures de données*, *Théorie de la calculabilité*, *Logique : fondements, le problème P vs. NP et perspectives information-théoriques* ainsi que *Programmation orientée objet en Java* — forment, avec le présent ouvrage, une unité cohérente. Ensemble, ils couvrent l'arc allant des assises théoriques à la pensée algorithmique et à la mise en œuvre pratique en programmes.

La théorie de la complexité reste un domaine de recherche ouvert et vivant. Nombre de ses questions

centrales — en particulier le problème P vs. NP — demeurent non résolues à ce jour. Néanmoins, au cours des dernières décennies, il est apparu que les méthodes et les perspectives de la théorie de la complexité dépassent largement le cadre purement théorique. Elles nous aident à comprendre les limites du réalisable, à concevoir des procédés efficaces et, même là où des solutions complètes semblent impossibles, à trouver des compromis optimaux.

J'espère que ce livre offrira aux lecteurs une image claire, cohérente et stimulante de ce champ, qu'il les incitera à poursuivre leur réflexion et qu'il posera le fondement théorique sur lequel reposent de nombreuses avancées pratiques en informatique.

Lucien Sina

Chapitre 1

Introduction

1.1 Qu'est-ce que la théorie de la complexité ?

La théorie de la complexité étudie quelles ressources minimales sont nécessaires pour résoudre des problèmes algorithmiques. Contrairement à la conception d'algorithmes concrets, qui fournit des bornes supérieures sur le besoin en ressources, la théorie de la complexité tente de prouver des bornes inférieures — c'est-à-dire des assertions sur la quantité de temps, d'espace ou d'autres ressources rares que *tout* algorithme doit nécessairement utiliser pour résoudre correctement un problème donné. De telles assertions permettent de distinguer ce qui est pratiquement réalisable de ce qui est fondamentalement impossible et d'éviter ainsi des recherches infructueuses vers des solutions inaccessibles.

Un problème algorithmique est généralement formulé de façon générale en précisant les entrées, les sorties attendues et les critères de qualité. Si la version générale d'un problème ne permet pas d'obtenir des algorithmes efficaces, on suit classiquement deux stratégies : soit on restreint la classe d'entrées ou les exigences (spécialisation), soit on accepte des solutions approximatives ou heuristiques (affaiblissement des exigences). Les deux voies peuvent conduire à des résultats praticables et souvent très utiles, car elles mettent au jour le cœur de la difficulté d'un problème.

À la différence de la simple conception d'algorithmes, qui fournit des procédures concrètes, la théorie de la complexité propose un formalisme pour classer les problèmes : des classes telles que P (temps polynomial), NP (vérifiabilité non déterministe en temps polynomial) ou des classes d'espace regroupent les problèmes selon leur coût asymptotique en ressources. Les réductions entre problèmes permettent de comparer leur difficulté relative : si l'on peut réduire efficacement un problème A à un problème B, alors A n'est pas plus difficile que B au regard des ressources considérées.

La question la plus célèbre et toujours non résolue de la discipline est le problème P vs. NP : toute propriété dont la validité se vérifie en temps polynomial est-elle aussi décidable en temps polynomial ? De nombreux problèmes décisionnels combinatoires importants (par ex. SAT, Clique, cycle hamiltonien) sont NP-complets, c'est-à-dire qu'ils appartiennent à la tranche la plus

difficile de NP. Si l'on trouvait pour l'un de ces problèmes un algorithme en temps polynomial, cela entraînerait une borne polynomiale pour tous les problèmes de NP ; inversement, un résultat d'impossibilité exclurait toute classe entière d'algorithmes efficaces.

Outre le temps et l'espace, d'autres ressources prennent une place centrale dans les considérations pratiques et théoriques : capacité d'entrée/sortie, consommation d'énergie, bande passante de communication, taille et profondeur de circuits, accès quantiques, etc. Selon l'application et le modèle de calcul, les classes pertinentes et les techniques d'analyse évoluent. Les méthodes de preuve sont elles aussi variées : techniques de diagonalisation, simulations entre modèles, outils information-théoriques, preuves par réduction, méthodes adversariales et — dans des développements plus récents — des approches issues de l'algèbre et des statistiques.

La théorie de la complexité et la conception d'algorithmes se complètent : les algorithmes montrent ce qui est atteignable ; les bornes inférieures indiquent ce qui ne l'est pas. Ces deux perspectives conjointes mènent à une compréhension plus profonde des problèmes algorithmiques — que ce soit par la découverte de procédés efficaces, la mise en lumière d'obstacles ou l'identification des causes structurelles de la difficulté.

1.2 Problèmes algorithmiques

Définition 1. *Un* ***problème algorithmique*** *est un ensemble d'entrées admissibles (instances), représentées comme des mots finis sur un alphabet fini Σ, accompagné d'une affectation qui associe à chaque entrée admissible $x \in \Sigma^*$ un ensemble non vide de sorties correctes $S(x)$ (réponses, certificats, valeurs). Les sorties sont elles aussi des mots finis sur un alphabet fini (éventuellement différent). On appelle une entrée concrète une* instance *du problème ; le problème lui-même est l'ensemble de toutes les instances avec la spécification des sorties admissibles.*

Par cette formalisation, nous adaptons les problèmes aux capacités de traitement des ordinateurs : entrées et sorties sont codées comme des mots finis en bits ou en caractères, de sorte que la question de la calculabilité et de l'utilisation des ressources peut être posée de manière précise.

On distingue plusieurs types usuels de problèmes selon la nature de la sortie demandée :

- **Problèmes de décision :** la sortie est l'une des deux réponses (par ex. `Oui`/`Non`). On formule souvent les problèmes de décision comme des langages $L \subseteq \Sigma^*$: une instance x appartient à L si et seulement si la bonne réponse est `Oui`.
- **Problèmes de recherche :** il suffit, pour une instance x, de trouver un $y \in S(x)$ quelconque (par ex. une preuve ou un certificat).
- **Problèmes d'optimisation :** à chaque instance

est associée une famille de solutions admissibles munie d'une fonction d'évaluation ; on cherche une solution de qualité maximale (ou minimale) (par ex. le plus court cycle hamiltonien).

— **Problèmes d'évaluation :** au lieu d'une solution explicite, seul importe la valeur associée (par ex. la longueur du plus court chemin entre deux nœuds).

Souvent, on peut transformer des problèmes les uns en les autres : un problème d'optimisation peut être converti en problème de décision en paramétrant un seuil (par ex. «Existe-t-il un cycle de longueur $\leq k$?»). Réciproquement, un problème de décision peut être vu comme un cas particulier d'un problème de recherche ou d'évaluation.

Pour les problèmes solvables exactement, toutes les sorties correctes sont considérées comme équivalentes ; pour des problèmes plus difficiles, on introduit souvent une métrique de qualité et l'on accepte des approximations. Une formulation courante de la qualité d'approximation est un facteur $\alpha \geq 1$: pour un problème de minimisation, on dit qu'une solution est une α-approximation si sa valeur est au plus α fois la valeur optimale. De telles relaxations de l'exigence d'optimalité sont pertinentes tant sur le plan pratique que théorique.

Un problème algorithmique est dit *résolu* s'il existe un mécanisme effectif et bien défini (algorithme, machine de Turing, etc.) qui fournit, pour chaque instance admissible, une sortie correcte de $S(x)$. En théorie de la complexité, on s'intéresse en outre au coût de ce procédé — par exemple en temps, en espace, en nombre d'échanges

de communication, en consommation d'énergie ou en profondeur de circuits.

Enfin, la représentation des entrées est importante : différentes codages d'une même entrée mathématique peuvent donner lieu à des problèmes algorithmiques différents. En pratique, on considère deux codages comme équivalents s'ils se transforment efficacement l'un en l'autre (par ex. en temps polynomial). Cette idée d'encodages efficaces permet de comparer la difficulté des problèmes indépendamment de détails de représentation arbitraires.

Exemples : Exemples typiques illustrant les distinctions ci-dessus :

- **SAT** (décision) : existe-t-il une affectation satisfaisante des variables pour une formule booléenne donnée ?
- **Plus court chemin** (évaluation/optimisation) : déterminer la longueur ou le chemin de distance minimale entre deux nœuds dans un graphe pondéré.
- **TSP** (optimisation / pertinent pour l'approximation) : trouver le plus court cycle hamiltonien dans un graphe complet pondéré.

La formulation précise des problèmes algorithmiques et de leurs codages constitue la base de toutes les définitions ultérieures des mesures et classes de complexité, qui seront abordées dans le chapitre suivant.

1.3 Algorithme

Qu'est-ce qu'un algorithme ?

Définition 2. *Un* ***algorithme*** *est une règle bien définie et univoque qui, pour chaque entrée admissible du problème algorithmique considéré, décrit une suite finie et ordonnée d'opérations de calcul qui — en cas de terminaison — produit une sortie correcte.*

Pour la théorie de la complexité, en plus du concept d'« algorithme », différents modèles de calcul et modes d'exécution sont pertinents. Nous donnons ici des définitions succinctes des types usuels :

Définition 3. *Un algorithme est dit* ***déterministe*** *si, pour chaque configuration (constituée de l'entrée, du contenu de travail et de l'état), l'opération suivante est déterminée de façon unique. Les algorithmes déterministes correspondent au déroulement intuitif d'un programme où chaque décision est fixée par les informations disponibles jusqu'alors.*

Définition 4. *Un algorithme est dit* ***non déterministe*** *(au sens théorique) s'il admet, en certains points, plusieurs pas possibles et qu'une entrée est* acceptée *dès qu'au moins une branche d'exécution mène à une configuration finale acceptante. On modélise formellement les algorithmes non déterministes par exemple par des machines de Turing non déterministes, qui « devinent » simultanément toutes les possibilités de choix (ou,*

de manière équivalente, par l'existence d'un certificat approprié).

Définition 5. *Un algorithme est* ***randomisé*** *(ou* ***probabiliste****) si son exécution dispose en outre d'un accès à des bits aléatoires et si certaines décisions au cours de l'exécution dépendent des résultats de ces bits aléatoires. On classe les algorithmes randomisés selon leur comportement d'erreur (par ex. Monte-Carlo avec un taux d'erreur maximal* $< \frac{1}{2}$ *ou Las-Vegas, qui fournissent toujours des résultats corrects mais avec un temps d'exécution aléatoire).*

Point important : non déterminisme et randomisation sont des concepts différents. Le non déterminisme est un modèle théorique exigeant l'existence d'une branche de choix « correcte » (existence d'une solution), tandis que la randomisation introduit de véritables décisions aléatoires avec des erreurs quantifiables ou des espérances de temps/qualité. Il n'est pas correct de considérer le non déterminisme de manière générale comme un cas particulier de la randomisation — la relation exacte entre les classes de complexité correspondantes (par ex. P, NP, RP, BPP, ZPP) fait l'objet de questions centrales et est en partie ouverte.

Exemples :

— Un algorithme **déterministe** : le tri fusion (Mergesort) — pour chaque entrée le déroulement est unique et il renvoie une suite triée en temps $O(n \log n)$.

— Une procédure **non déterministe** pour SAT : « deviner » une affectation des variables et vérifier déterministement que la formule est satisfaite. Ceci montre formellement que SAT appartient à NP.
— Un algorithme **randomisé** : Quicksort randomisé (temps attendu $O(n \log n)$) ; le test de primalité de Miller–Rabin en tant que test Monte-Carlo (faible taux d'erreur, exécution très rapide).

En théorie de la complexité, ces concepts formalisent des modèles de calcul distincts et conduisent à différentes classes, qui seront introduites et comparées de façon systématique par la suite.

1.4 Temps d'exécution d'un algorithme

Le temps d'exécution d'un algorithme est une mesure centrale de son efficacité. Une définition naïve — le temps de calcul absolu mesuré en secondes sur une machine concrète — est inadaptée aux comparaisons théoriques, car elle dépend de nombreux facteurs externes (matériel, langage de programmation, détails d'implémentation, système d'exploitation, optimisations du compilateur, etc.). L'objectif d'une analyse théoriquement propre est de choisir une mesure du temps qui ne dépende que de l'algorithme lui-même et de son entrée, et qui compare de manière compatible différents modèles de calcul raisonnables.

Cela implique deux étapes fondamentales :

1. le choix d'un modèle de calcul abstrait qui fixe les opérations élémentaires et leur coût ;
2. la formulation d'une mesure asymptotique du temps qui décrit la dépendance des ressources nécessaires à la longueur de l'entrée.

1.4.1 Opérations élémentaires et mesures de coût

On mesure habituellement le temps par le nombre d'opérations élémentaires, les opérations élémentaires étant celles considérées comme « constantes » sur un modèle de calcul réaliste : opérations arithmétiques de base sur des mots de longueur fixe, lecture/écriture d'une cellule mémoire, transitions d'état, déplacement d'une tête de lecture, etc. Deux mesures de coût répandues sont :

Modèle à coût unitaire : Chaque opération élémentaire coûte 1. Ce modèle est simple à manipuler, mais il peut donner des estimations irréalistes pour des opérations sur des nombres très grands (par ex. l'addition d'entiers de taille arbitraire serait comptée comme constante).

Mesure de coût logarithmique (coût en bits) : Le coût d'une opération dépend proportionnellement de la longueur en bits des opérandes ; le modèle compte en quelque sorte les opérations au bit près. Cette mesure reflète mieux la difficulté réelle des opérations sur de grands nombres.

Pour de nombreuses affirmations théoriques (en particulier pour des classes de complexité telles que P et NP), il est souhaitable de choisir une mesure de coût robuste vis-à-vis du choix d'un modèle « raisonnable » concret. La robustesse signifie ici que deux modèles sensés diffèrent seulement par un facteur polynomial dans leurs temps d'exécution — l'équivalence à un facteur polynomial est le critère d'équivalence usuel en théorie de la complexité.

1.4.2 Machine de Turing : modèle de calcul formel

Le modèle formel qui se rapproche le plus de l'intuition d'algorithme et qui s'est imposé comme standard en théorie de la complexité est la machine de Turing. Nous donnons ici la définition usuelle (déterministe) sous une forme condensée.

Définition 6 (Machine de Turing déterministe)**.** *Une machine de Turing (déterministe) est un tuple*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}),$$

où Q est un ensemble fini d'états, Σ l'alphabet d'entrée (sans le symbole blanc $\sqcup$), Γ l'alphabet de travail avec $\Sigma \subseteq \Gamma$, $q_0 \in Q$ l'état initial et $q_{acc}, q_{rej} \in Q$ les états finaux acceptant et rejetant. La fonction de transition

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

détermine à chaque pas l'état suivant, le symbole à écrire et le déplacement de la tête (gauche, droite, arrêt).

Une configuration d'une machine de Turing décrit l'état courant de la machine, le contenu de la bande et la position de la tête. Un pas correspond à l'application de δ. Le *temps d'exécution* de M sur une entrée x est le nombre de pas jusqu'à la première atteinte d'un état final (si M s'arrête). Le temps au pire cas d'une machine pour des entrées de longueur n s'écrit

$$T_M(n) = \max\{\ \text{étapes}(M, x) \mid |x| = n\ \}.$$

En pratique, on travaille souvent avec des machines de Turing à plusieurs rubans (plusieurs bandes de travail), car elles modélisent plus naturellement de nombreux algorithmes ; les différentes variantes sont cependant équivalentes à un facteur polynomial (les machines multi-rubans simulent les machines à ruban unique avec un surcoût quadratique ou, plus généralement, polynomial, et réciproquement).

1.4.3 Robustesse polynomiale et thèse étendue de Church–Turing

La version polynomiale de la thèse de Church–Turing (souvent appelée *polynomial Church–Turing thesis* ou *extended Church–Turing thesis*) affirme essentiellement ce qui suit :

Énoncé (simulation polynomiale) : *Tout modèle de calcul « raisonnable » R (par ex. machine à accès aléatoire, registre machine, modèles matériels réalistes sans opérations magiques illimitées) peut être simulé par*

une machine de Turing de sorte qu'une exécution de t opérations élémentaires sur R (sous une mesure de coût appropriée, par ex. la mesure logarithmique/en bits) puisse être effectuée en au plus $p(t)$ pas sur une machine de Turing (à plusieurs rubans), où p est un polynôme dépendant de la méthode de simulation choisie.

Cette robustesse polynomiale justifie de formuler les classes de complexité (par ex. P, NP) et les énoncés asymptotiques de temps sur le modèle de Turing — les affirmations deviennent alors indépendantes du modèle à un facteur polynomial près et sont ainsi stables pour la plupart des questions théoriques.

Nous retenons

Soit R un modèle de calcul réaliste et soit le temps d'exécution d'un algorithme sur R sous la mesure de coût logarithmique donné par $t(n)$ pour des entrées de longueur n. Alors il existe une machine de Turing (à plusieurs rubans) M et un polynôme p tels que

$$T_M(n) \leq p(t(n))$$

pour tout n suffisamment grand. Ainsi, la complexité temporelle est, au sens polynomial, indépendante du modèle — un pilier central de l'analyse théorique de la complexité.

Remarques

— La forme précise de p dépend des opérations autorisées sur R ; les modèles qui permettent des opérations « injustes » (par ex. multiplication directe d'entiers de taille arbitraire en temps constant ou des oracles puissants) ne sont pas couverts par cet énoncé.
— Pour des détails pratiques (par ex. le coût des opérations sur de grands entiers), la mesure de coût logarithmique est plus réaliste que le modèle à coût unitaire.
— L'équivalence polynomiale motive l'étude des classes de complexité sous la relation d'équivalence « égalité à un facteur polynomial près », en particulier pour des questions telles que P vs. NP.

1.5 Complexité des problèmes algorithmiques

Soit $t_A(x)$ le temps de calcul d'un algorithme A pour une entrée x mesuré dans la mesure de coût unitaire sur un modèle de référence (par ex. une machine de Turing). Pour rendre comparables les temps d'exécution de différents algorithmes résolvant le même problème algorithmique, quelques précautions sont nécessaires ; une règle de comparaison naïve point par point

A est au moins aussi rapide que B si $t_A(x) \leq t_B(x)\ \forall x$

présente plusieurs difficultés pratiques et théoriques :

1. Les temps absolus de calcul dépendent du modèle de référence choisi (machine de Turing à ruban unique vs. machine à plusieurs rubans, modèle RAM, etc.).
2. Exiger que la comparaison porte sur toutes les entrées x est souvent trop contraignant et peu adapté aux énoncés asymptotiques.
3. Pour de petites entrées, un algorithme « plus simple » peut être plus rapide, tandis que des algorithmes spécialisés montrent leurs avantages asymptotiques seulement pour des entrées suffisamment grandes.

Pour surmonter ces problèmes, on utilise en théorie de la complexité les concepts standards suivants.

1.5.1 Temps d'exécution au pire cas

Pour une machine A on définit la fonction de temps (au pire cas)

$$t_A(n) := \sup\{\, t_A(x) \mid |x| \leq n \,\},$$

ou bien, de manière souvent équivalente,

$$t_A(n) = \max\{\, t_A(x) \mid |x| = n \,\},$$

lorsque le maximum existe. Pour la plupart des modèles raisonnables, $t_A(n)$ est croissante en n. La comparaison d'algorithmes se fait alors en termes asymptotiques : on

dit que A est *asymptotiquement au moins aussi rapide* que B si

$$t_A(n) = O\big(t_B(n)\big).$$

1.5.2 Cas moyen et distributions

Si des instances « extrêmes » dominent le temps au pire cas, il peut être pertinent de considérer une mesure de probabilité ν_n sur les entrées de longueur n. Le temps moyen de A sous la loi ν s'écrit alors

$$t_A^{\nu}(n) := \sum_{|x|=n} \nu_n(x)\, t_A(x),$$

où ν_n est pour chaque n une distribution de probabilité sur l'ensemble $\{x \mid |x| = n\}$. Les analyses en cas moyen n'ont de sens que dans la mesure où l'hypothèse sur la distribution des entrées est plausible.

Mesures de complexité asymptotiques

Les notations asymptotiques usuelles $O(\cdot)$, $\Omega(\cdot)$ et $\Theta(\cdot)$ condensent l'ordre de croissance des fonctions de temps et rendent les comparaisons robustes vis-à-vis du choix du modèle à un facteur polynomial près (cf. la formulation polynomiale de la thèse de Church–Turing).

Définition 7. *Un algorithme A résout un problème en temps $O(f(n))$ si $t_A(n) = O(f(n))$. On dit que la* ***complexité algorithmique*** *(temporelle) du problème est $f(n)$ si*

1. *il existe un algorithme A tel que $t_A(n) = O(f(n))$ (borne supérieure), et*
2. *toute machine résolvant le problème nécessite au moins un temps $\Omega(f(n))$ (borne inférieure).*

Si une borne supérieure et une borne inférieure sont données par la même fonction asymptotique $f(n)$, on écrit la complexité sous la forme $\Theta(f(n))$, ce qui fixe l'ordre de croissance temporel du problème.

1.5.3 Indépendance par rapport au modèle

La formulation polynomiale de la thèse de Church–Turing implique que, pour des modèles de calcul « raisonnables » R_1, R_2, il existe un polynôme p tel qu'une exécution de R_1 nécessitant t opérations élémentaires peut être simulée en au plus $p(t)$ opérations sur R_2. De ce fait, les énoncés sur la complexité temporelle sont, au sens polynomial, indépendants du modèle — c'est l'une des justifications fondamentales de l'emploi des machines de Turing comme modèle de référence en théorie de la complexité.

Remarques

— Pour des analyses plus fines (par ex. le coût des opérations sur de grands entiers), la mesure de coût choisie (coût unitaire vs coût en bits) est déterminante.

— Outre le temps, on considère bien entendu d'autres ressources (mémoire, communication, bits aléatoires, profondeur de circuits), et pour chaque ressource on définit des fonctions analogues au pire cas et au cas moyen.
— Les bornes inférieures sont souvent difficiles à démontrer et constituent le cœur de nombreux résultats de la théorie de la complexité.

Chapitre 2

Classes de complexité

2.1 La classe de complexité P

Les temps polynomiaux occupent une place particulière en théorie de la complexité. Sous la version polynomiale de la thèse de Church–Turing, divers modèles de calcul « raisonnables » sont polynômialement équivalents, de sorte que des fonctions de temps qui ne diffèrent que par des facteurs polynomiaux sont, pour la plupart des questions théoriques, considérées comme équivalentes. C'est pourquoi les temps polynomiaux servent de formalisation robuste et indépendante du modèle du concept d'*efficace calculable*.

Une autre raison pratique de l'importance des temps polynomiaux est que de nombreux algorithmes doivent lire l'ensemble de l'entrée ; cela induit fréquemment une borne inférieure linéaire $\Omega(n)$ pour le temps d'exécution,