RAPPELS DE THÉORIE DE LA COMPLEXITÉ

RÉSUMÉ ET QUESTIONS

Deux références majeures pour la théorie de la complexité sont [Knu] pour les algorithmes fondamentaux et [Pap] pour les notions de bases et la définition des classes de complexité.

La théorie de la complexité mesure le temps et l'espace nécessaires à la résolution d'un problème donné. La question n'a de sens que pour une famille infinie de problèmes similaires qui doivent être résolus par le même algorithme.

Cela sous-entend que l'on se restreint à des problèmes décidables (il existe un algorithme, même très maladroit, qui résout le problème).

1. Première approche : compter les opérations de base

Une première approche pour mesurer l'efficacité d'un algorithme est de compter les opérations élémentaires. La notions d'opération élémentaire dépend du contexte.

1.1. **Tri.**

On se donne une liste d'entiers (c'est l'*input* du problème). L'algorithme doit retourner la même liste triée par ordre croissant (c'est l'*output* du problème). Une *opération élémentaire* dans ce contexte est une comparaison, un accès mémoire, une affectation. On ne comptera ici que les comparaisons.

Soit donc $L = (L_1, L_2, ..., L_n)$ une liste (vecteur) d'entiers. On cherche une liste $M = (M_1, M_1, ..., M_n)$ ordonnée i.e. $M_1 \leq M_2 \leq ... \leq M_n$ telle que $\{L_1, ..., L_n\} = \{M_1, ..., M_n\}$.

1.1.1. *Tri par insertion.*

On suppose d'abord que les k premiers entiers sont déjà dans l'ordre. On a donc $L_1 \leq L_2 \leq \cdots \leq L_k$. On veut insérer L_{k+1} à sa place. On parcours cette liste de gauche à droite jusqu'à trouver le premier entier plus grand que L_{k+1}. Cela demande au plus k comparaisons. Donc le temps $T(n)$ nécessaire pour trier une liste de n entiers avec cette méthode satisfait

$$T(k+1) \leq T(k) + k.$$

On en déduit que

$$T(n) \leq n(n-1)/2.$$

On dit que la complexité en temps de cet algorithme de tri est *quadratique*. En effet, le nombre d'opérations élémentaires est majoré par une fonction quadratique en la taille n de l'input.

La complexité intrinsèque d'un problème est la complexité du meilleur algorithme qui résolve ce problème. Cela signifie que l'on se restreint à une famille ordonnée raisonnable de fonctions.

1.1.2. *Tri fusion.* Ici l'opération de base est la fusion de deux listes triées de taille k. Montrer que cette opération peut être conduite à l'aide de $F(k)$ comparaisons avec $F(k) \leq 2k$. En déduire que l'algorithme récursif ainsi obtenu trie une liste de taille n au prix de $T(n)$ comparaisons où

$$T(2n) \leq 2T(n) + 2n.$$

En déduire que

$$T(2^a) \leq a2^a.$$

1.1.3. *Tri à bulles.* Le principe du *tri à bulles* est de parcourir la liste initiale de gauche à droite et de permuter deux éléments successifs chaque fois qu'ils ne sont pas ordonnés. On réitère jusqu'à ce que la liste soit triée (ce qui se traduit par l'absence de permutations dans la dernière passe).

Montrer que cet algorithme s'arrête. Pour mesurer sa complexité, on majore le nombre de transpositions nécessaires pour trier une liste de n éléments. Montrer que ce nombre est borné par une constante fois n^2. Indication : regarder comment se déplace le plus grand élément de la liste.

1.1.4. *Tri par tournoi.* Une autre façon de trier consiste à sélectionner le plus grand élément par éliminations successives, à la manière d'un tournoi. On groupe donc les éléments par paires et on désigne le plus grand de chaque paire. Les vainqueurs de ce premier tour sont alors confrontés par paires et ainsi de suite, jusqu'à la finale. Cet algorithme détermine seulement le plus grand des éléments. Pour le transformer en algorithme de tri, on imagine une structure hierarchique en forme d'arbre binaire. Au sommet de l'arbre, le chef, qui a deux sous-chefs, chacun ayant deux sous-sous-chefs etc. On suppose qu'au début de l'algorithme toutes les places de l'organigramme sont vacantes sauf celles du plus bas niveau hiérarchique, où l'on place la liste à trier. Ces places du niveau inférieur sont les feuilles de l'arbre binaire. Dans chaque paire de feuilles on choisit la plus grande valeur et on la promeut au niveau supérieur. Les entrées du deuxième niveau sont ensuite comparées par paires et la meilleure est promue au troisième niveau. Et ainsi de suite. Mais on prend bien garde de pourvoir en cascade chaque place laissée vacante par une promotion en promouvant ensuite le meilleur des deux subordonnés du dernier promu, et ainsi de suite récursivement. Ce processus sélectif par paliers se poursuit jusqu'au niveau suprême et détermine la plus grande valeur de la liste (le premier secrétaire du Parti). On retire alors cette valeur de l'arbre (mort du premier secrétaire) et on organise son remplacement récursif en promouvant le meilleur des deux secrétaires adjoints puis en pourvoyant la place de secrétaire adjoint ainsi libérée et ainsi de suite jusqu'à la base.

Montrer que cet algorithme trie une liste de taille n avec moins de $An \log n$ comparaisons pour un réel A qu'on ne cherchera pas à déterminer.

1.2. **Opérations sur les entiers.** On considère le problème de l'addition de deux entiers positifs a et b. L'input est (a, b). L'output est $a+b$. La taille d'un entier peut être définie comme le nombre de décimales. Donc la taille de l'input est

$$\lceil \log_{10}(a+1) \rceil + \lceil \log_{10}(b+1) \rceil.$$

On calcule $a + b$ avec la méthode élémentaire apprise à l'école. Le nombre d'opérations élémentaires est proportionnel à la taille de l'input. On dit que l'algorithme est linéaire.

L'algorithme appris à l'école pour multiplier deux entiers a une complexité en temps quadratique en la taille de l'input. Même chose pour l'algorithme de division euclidienne. Il existe cependant des algorithmes *quasi-linéaires* pour la multiplication des entiers et pour la division euclidienne [Gat]. La complexité en temps est majorée par $AS(\log S)^B$ où S est la taille de l'input et A, et B sont deux constantes.

Un autre algorithme important est l'algorithme d'Euclide pour calculer le pgcd de deux entiers. Il existe une variante quasi-linéaire de cet algorithme. La méthode standard est quadratique.

2. Deuxième approche : modèles de calcul

Les définitions du paragraphe précédent souffrent d'un certain manque de rigueur. Si l'on se restreint à un cadre assez étroit (les algorithmes de tri, les algorithmes de transformée de Fourier, etc.), le point de vue adopté jusqu'alors est justifié : il suffit de compter les opérations les plus coûteuses, celles qui sont le pain quotidien des algorithmes concernés. Mais si l'on prétend définir des classes plus générales de problèmes, il est nécessaire de disposer d'un modèle uniforme de calcul.

On de donne d'abord un **cadre général** pour définir les problèmes. On distingue premièrement deux types de problèmes. Les problèmes dont la réponse ne peut être que OUI ou NON sont appelés problèmes de *décision* (par exemple, *ce nombre est il premier?* ou *ce graphe est il connexe ?* Les autres problèmes dont la réponse est plus longue (par exemple, *quel est le produit de ces deux nombres?* ou *quel est le cardinal de ce groupe?*) sont appelés problèmes fonctionnels.

Un problème décisionnel revient à décider si un mot m appartient à un langage $L \subset \{0,1\}^*$. Ici l'input est m est l'output est oui ou non. Il y a donc autant de problèmes décisionnels que de langages. Un problème fonctionnel revient à évaluer une fonction $f : \{0,1\}^* \to \{0,1\}^*$. Ici l'input est m est l'output est $f(m)$.

Exemples de problèmes : PRIME, SQUARE, MULTIPLICATION, ...

On a besoin aussi de définir un **modèle de calcul**. Le plus connu est celui des machines de Turing. Une machine de Turing est un automate déterministe muni d'une bande mémoire infinie à droite sur laquelle l'automate peut lire ou écrire à l'aide d'un unique curseur. Le temps est discret. Le comportement de la machine est déterminé par une fonction de transition qui, connaissant l'état de la machine et le caractère lu sur la bande à l'endroit où se trouve le curseur, détermine l'état suivant, le caractère à écrire à la place du caractère courant, et le mouvement du curseur.

Plus précisément une machine de Turing est la donnée d'un quadruplet $M = (S, \Sigma, \delta, s)$ où S est l'ensemble fini des états de la machine, $s \in S$ est l'état initial, Σ est l'alphabet utilisé par M et contient au moins les deux symboles $\triangleright$ (symbole initial) et $\square$ (blanc) (souvent $\Sigma = \{0, 1, \triangleright, \square\}$), et δ est la fonction de transition qui va de $S \times \Sigma$ dans $S \cup \{\text{STOP, OUI, NON}\} \times \Sigma \times \{\leftarrow, \rightarrow, \bullet\}$. Cette fonction détermine l'état suivant (qui peut être l'un des trois états finaux OUI, NON ou STOP) le caractère à écrire et le déplacement du curseur (vers la gauche, vers la droite ou pas de déplacement). Initialement, le curseur est sur la première case de la bande mémoire et cette case contient le symbole initial $\triangleright$ suivi des entrées (ou données du problème). Le reste de la bande est vierge (il ne contient que des blancs $\square$). La machine n'est pas autorisée à effacer le symbole $\triangleright$ et ce symbole l'oblige à se déplacer vers la droite, de sorte qu'elle ne sort jamais de la bande

mémoire. Ces contraintes s'expriment par la condition que pour tout état p il existe un état q tel que $\delta(p, \triangleright) = (q, \triangleright, \rightarrow)$.

Les machines de Turing sont bien adaptées à l'étude des problèmes de décision. Si $L \subset \{0,1\}^*$ est un langage, on dit que la machine de Turing T décide L si pour toute entrée $x \in \{0,1\}^*$ la machine s'arrête dans l'état OUI si $x \in L$ et dans l'état NON si $x \notin L$. Si un langage est décidé par une machine de Turing, on dit qu'il est **décidable**. Il existe des langages non-décidables (ne serait-ce que pour des raisons simples de cardinalité) mais la théorie de la complexité se restreint d'ordinaire aux problèmes décidables. Comme certains problèmes très naturels sont indécidables (comme le problème de l'appartenance dans les groupes de présentation finie) il est prudent de ne pas méconnaître ces notions.

Les logiciens introduisent une notion plus faible que la décidabilité. On dit qu'un langage L est **récursivement énumérable** s'il existe une machine de Turing T qui s'arrête pour toute entrée $x \in L$ et qui ne s'arrête pas si $x \notin L$. Cette notion est plus faible que la décidabilité car dans le cas où $x \notin L$ on n'est jamais fixé. Un exemple naturel et fameux est celui des équations diophantiennes (équations algébriques dont les inconnues sont des nombre entiers). L'existence d'une solution à une équation diophantienne est un problème de décision évidemment récursivement énumérable (il suffit d'essayer toutes les solutions) mais un résultat fameux de Matiyasevich prouve que ce problème n'est pas décidable, résolvant ainsi le dixième problème de Hilbert.

Un langage est décidable si et seulement si lui et son complémentaire sont récursivement énumérables.

Les machines de Turing sont utiles aussi pour l'étude des problèmes fonctionnels. Si une machine de Turing M initialisée avec l'entrée x s'arrête dans l'état STOP, on appelle sortie et on note $M(x)$ la chaîne présente alors sur la bande mémoire de M à droite du $\triangleright$ initial. On dit qu'une fonction $f : \{0,1\}^* \rightarrow \{0,1\}^*$ est **récursive** si elle est calculée par une machine de Turing. Le temps de calcul est le nombre d'étapes avant l'arrêt de la machine.

Si $T : \mathbb{N} \rightarrow \mathbb{N}$ est une fonction on note **TIME**(T) l'ensemble des langages reconnus par une machine de Turing en temps $\leq T(n)$ où n est la taille des entrées. Cette définition ne présente pas un très grand intérêt car la moindre altération dans la définition des machines de Turing en modifierait le sens. En revanche, la classe **PTIME** est définie comme l'union de **TIME**$(n \mapsto n^d)$ pour $d \geq 1$ et cette définition résiste à toutes les variations raisonnables dans les définition des machines de Turing.

REFERENCES

[Gat] J. Von Zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press.
[Knu] Donald E. Knuth. *The Art of Computer Programming*. Four volumes. Addison-Wesley,
[Pap] C.H. Papadimitriou. *Computational complexity*. Addison Wesley, Reading, 1967.