Algorithmes et Complexité

25 septembre 2015

Table des matières

1 Classes de complexité

1.1 Rappel sur les notions de comportements asymptotiques

On note $\mathcal{S}$ l'ensemble des suites à valeur dans $\mathbb{R}^+$. En algorithmique, nous nous intéressons spécifiquement au comportement au voisinage de l'infini de suite de $\mathcal{S}$. Un exemple classique est la suite de terme général t_n, où t_n représente le temps d'exécution d'un algorithme A dans le pire des cas sur les instances de tailles n.
Le comportement au voisinage de l'infini de suite est donc une (bonne ?) mesure permettant de comparer l'efficacité des algorithmes.

Définition 1.1 (Petit o) *Étant données deux suites*[1] f *et* g *de* $\mathcal{S}$*, on dira que* f *est* négligeable *devant* g *au voisinage de l'infini si et seulement si pour tout réel* $\varepsilon > 0$*, il existe un entier positif* n_0 *tel que pour tout* $n > n_0$*, on a* $f_n < \varepsilon g_n$*.*
On notera alors $f = o(g)$*.*
En d'autres termes :
$f = o(g) \Leftrightarrow \forall \varepsilon > 0, \exists n_0 > 0, \forall n > n_0, f_n < \varepsilon g_n$

Proposition 1.2 *Etant données deux suites* f *et* g *de* $\mathcal{S}$ *:*

1. Les mêmes définitions s'appliquent aux fonctions

$$si \lim_{n\to\infty} \frac{f_n}{g_n} = 0 \ alors \ f = o(g).$$

Définition 1.3 (Grand O, Grand Omega) *Étant données deux suites f et g de $\mathcal{S}$, on dira que f est* dominée *par g au voisinage de l'infini (ou encore que g est une* borne asymptotique supérieure *de f) si et seulement s'il existe deux constantes strictement positives c et n_0 telles que pour tout $n > n_0$ on a $f_n < cg_n$.*
On notera alors $f = O(g)$ ou bien $g = \Omega(f)$.
En d'autres termes :
$f = O(g) \Leftrightarrow \exists c > 0, \exists n_0, \forall n > n_0, f_n < cg_n$
$f = \Omega(g) \Leftrightarrow \exists c > 0, \exists n_0, \forall n > n_0, f_n > cg_n$

Proposition 1.4 *Étant données deux suites f et g de $\mathcal{S}$:*

$$si \lim_{n\to\infty} \frac{f_n}{g_n} = c < \infty \ alors \ f = O(g).$$

Définition 1.5 (Grand Theta) *On dira que f et g sont* semblables *au voisinage de l'infini (ou encore que g est une* borne asymptotique approchée *de f) si et seulement si $f = O(g)$ et $f = \Omega(g)$.*
On notera alors $f = \Theta(g)$.
En d'autres termes :
$f = \Theta(g) \Leftrightarrow \exists c_m > 0, \exists c_M > 0, \exists n_0, \forall n > n_0, c_m g_n < f_n < c_M g_n$

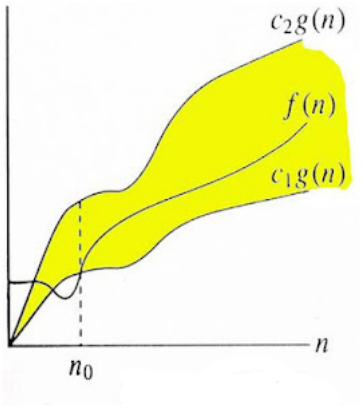

Figure 1 – Notion Theta. (source internet http ://gamemiaoqi.blogspot.fr/2012/03/training-artists-to-do-cs-stuff.html)

Proposition 1.6 *Etant données deux suites f et g de $\mathcal{S}$:*

$$si \lim_{n\to\infty} \frac{f_n}{g_n} = c \ avec \ 0 < c < \infty \ alors \ f = \Theta(g).$$

La réciproque des propositions 1.2,1.4,1.6 est fausse, mais ces propositions sont souvent utiles.
En général, étant donnée une suite ou une fonction f, on essayera de la comparer suivant une échelle simple (par exemple du type $\log(n)^a n^b \exp(cn)$). Les comportements rencontrés classiquement sont $\Theta(\ln(n))$ comme les algorithmes par dichotomie, $\Theta(n)$ les algorithmes linéaires, $\Theta(n\ln(n))$ comme certains tris par comparaison,...

Proposition 1.7

- *La relation R définie par $fRg \Leftrightarrow f = \Theta(g)$ est une relation d'équivalence.*
- *La relation R_1 (resp. R_2) définie par $fR_1g \Leftrightarrow f = \Omega(g)$ (resp. $fR_2g \Leftrightarrow f = O(g)$) est un préordre (i.e. une relation binaire, réflexive et transitive) sur $\mathcal{S}$.*
- *$f = o(g)$ implique $f = O(g)$.*

1.2 Un mot sur les modèles de calcul

L'objectif de ce cours n'est pas de définir des modèles de calculs. Néanmoins, une façon rigoureuse de définir les notions de complexité consiste à les définir sur des modèles élémentaires de calcul, comme les machines de Turing. Nous rappelons, ici, très brièvement les définitions de machine de Turing [2] et les notions de complexité en temps et en espace que l'on peut définir dessus. En fait, il peut montrer que les classes de complexité telles que P ou NP sont indépendante des modèles de calcul sous-jacent (en d'autres termes, la plupart des modèles de calculs sont polynomialement équivalents)
Une machine de Turing se compose d'une tête de lecture/écriture qui travaille sur une bande constituée d'une infinité de cases dans laquelle sont écrits des symboles. La tête de lecture/écriture peut effectuer des opérations de modification (écriture) sur la bande et se déplacer localement, pour cela elle dispose d'un "programme" qui lui indique suivant son état et ce qu'elle lit comment elle doit écrire sur la bande puis se déplacer. Ce programme, ce sont les *transitions* de la machine. Plus formellement, nous pouvons écrire la définition suivante :

Définition 1.8 (Machine de Turing) *Nous disposons de 7 symboles spéciaux que sont*
- *Les 2 états particuliers : $\triangle$ l'état d'acceptation, $\triangledown$ l'état de rejet.*
- *Les 3 déplacements de la tête de lecture $\rightarrow$ vers la droite, $\leftarrow$ vers la gauche, et $\downarrow$ reste sur place.*
- *Deux symboles particuliers, $\blacktriangleright$ appelé* symbole initial *et $\square$ appelé* blanc.

Une machine de Turing M est un quadruplet (Q, Σ, T, q_0) où
- *$Q = \{q_0, \ldots, q_n\}$ est un ensemble fini d'*états *(sans les 3 états particuliers).*
- *$\Sigma = S \cup \{\square, \blacktriangleright\}$ est l'*alphabet*. C'est à dire un ensemble fini de symboles S disjoint de Q.*
- *$T \cup \{(q, \blacktriangleright, p, \blacktriangleright, \rightarrow)\}$ est un ensemble fini de transitions de la forme (p, a, q, b, x) où $p \in Q$ et $q \in Q \cup \{\triangle, \triangledown\}$ sont des états, a et b sont des symboles et x est un élément de $\{\rightarrow, \leftarrow \downarrow\}$. Une transition (p, a, q, b, x) est aussi notée $(p, a) \rightarrow (q, b, x)$.*
- *$q_0 \in Q$ est l'état initial dans lequel se trouve machine au début d'un calcul.*

Définition 1.9 *Soit une machine de Turing $M = (Q, \Sigma, T, q_0)$, si pour toute transition (p, a, q, b, x) et (p, a, q', b', x') dans T, on a $q = q', b = b', x = x'$, la machine de Turing est dite* déterministe.

2. référence O.Carton "Langages formels, Calculabilité et Complexité", Papadimitriou "Computational complexity"

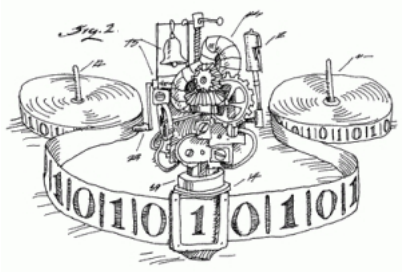

Figure 2 – Une machine de Turing. (source internet https ://project.inria.fr/turing2012/francais-le-programme-des-animations/francais-la-machine-de-turing-ou-lordinateur-de-papier/)

1.3 Comment marche une machine de Turing déterministe

Soit M une machine de Turing et un mot initial w dans $\blacktriangleright (\Sigma)^*$ écrit sur la bande (et suivi d'une infinité de symboles blancs). Alors les transitions de M vont modifier la case de la bande au niveau de la tête de lecture, puis la position et l'état de la tête de lecture. La première transition va faire passer de la situation $(q, \blacktriangleright)$ à la situation $(p, \blacktriangleright, \rightarrow)$, c'est à dire l'état de la tête de lecture passe à p et elle avance d'une case sans modifier la bande. Chacune de ces transformations est appelé un *calcul*.
Si en partant de w après effectué un nombre fini de calcul, la machine atteint l'un des 2 états spéciaux $\triangle, \triangledown$ alors elle s'arrête et on note $M(w) = x$ où x est l'état atteint sinon on note $M(w) = \infty$

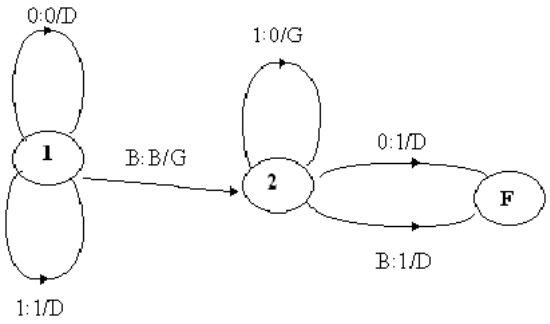

Figure 3 – Représentation graphique d'une machine de Turing. (source internet http ://www.grappa.univ-lille3.fr/polys/intro-info/informatique003.html)

Définition 1.10 *Une machine de Turing déterministe* accepte *(resp.* rejette*) un mot w dans $\blacktriangleright (\Sigma)^*$ si $M(w) = \triangle$ (resp. $M(w) = \triangledown$). Le* langage accepté *est l'ensemble des mots acceptés.*
Si la machine de Turing est non déterministe, un mot w est accepté *s'il existe un chemin de calcul parmi tout les chemins de calcul qui arrive à un état d'acceptation.*

Définition 1.11 *Le* langage accepté *L_M par une machine de Turing M est* décidé *si les mots n'appartenant pas à L_M sont refusés par la machine*[3].

3. C'est une notion plus forte, car la machine pourrait ne pas s'arrêter sur ces mots

Bien qu'élémentaires les machines de Turing sont en fait très puissantes et permettent de simuler "tous les algorithmes" (plus exactement, c'est une façon abstraite de définir ce qu'est un algorithme).

1.4 Complexité d'une machine de Turing : TIME, NTIME et SPACE

Soit M une machine de Turing déterministe, on définit naturellement la complexité en temps pour une entrée x acceptée comme le nombre t de calculs fait par M pour calculer x (i.e. atteindre l'arrêt) quand la bande est initialisée à $\blacktriangleright\, x$. On note alors $TIME_M(x) = t$. On définit aussi la complexité en espace pour une entrée x comme le nombre maximum s de symboles sur la bande lors des calculs faits par M. On note $SPACE_M(x) = s$.

Si la machine n'est pas déterministe, on définit la complexité en temps pour une entrée x acceptée comme le nombre minimum t de calculs fait par M pour atteindre un état d'acceptation (il peut y avoir plusieurs chemins de calcul qui mènent à l'état d'acceptation et d'autres chemins menant à des états de rejet) quand la bande est initialisée à $\blacktriangleright\, x$. On note alors $NTIME_M(x) = t$.

Définition 1.12 *Les complexités (dans le pire des cas) en temps TIME et en espace SPACE d'une machine M déterministes sont définies comme suit :*

$$TIME_M(n) = max_{x \text{ tel que } M(x)=\triangle, |x|=n} TIME_M(x)$$

et

$$SPACE_M(n) = max_{x \text{ tel que } M(x)=\triangle, |x|=n} SPACE_M(x).$$

Notons que la notion de complexité en espace SPACE n'est pas adaptée aux machines qui prennent un espace de calcul sous-linéaire car elle contient implicitement dans sa définition la taille de l'entrée (mot initial). Dans le cas des algorithmes peu couteux en espace, on peut adapter la définition des machines de Turing en définissant deux rubans un contenant le mot initial et un sur lequel on fait les calculs. La complexité en espace étant alors la taille sur le ruban de calcul.
Si cette vision est fondamentale pour avoir une définition rigoureuse de la notion de complexité. Nous voyons qu'elle est assez inadaptée pour traiter de manière pratique la complexité d'algorithmes. La suite de ce chapitre donne les bases d'une théorie de la complexité plus appropriée pour l'analyse d'algorithmes.

1.5 Problèmes abstraits, problèmes concrets, problèmes de décision

On va maintenant définir de manière théorique la notion de problèmes. [4]

Définition 1.13 *Un* problème abstrait *est une relation binaire sur le produit cartésien $I \times S$, où I est l'ensemble des instances du problèmes et S l'ensemble des solutions.*

A noter qu'une instance pour avoir plusieurs solutions (d'où l'usage de relation binaire plutôt qu'une fonction).
Nous définissons une sous-classe importante des problèmes abstraits que sont les problèmes de décisions :

4. référence : Cormen, Leiserson, Rivest, "Introduction à l'algorithmique", chapitre 36

Définition 1.14 *Un* problème de décision *est une fonction de l'ensemble des instances I dans l'espace $S = \{0, 1\}$.*

En d'autres termes, un problème de décision, est un problème où l'on attend une réponse de type oui ou non. Par exemple, $I = \mathbb{N}^*$, soit $n \in I$, n est-il premier ? est un problème P de décision, on a $P(2) = P(3) = 1$ car 2 et 3 sont premiers et $P(4) = 0$,...
Nous verrons que se réduire aux problèmes de décision ne constitue pas en soit une grande limitation. Il nous reste néanmoins une question importante qui est celle du codage des instances.

Définition 1.15 *Un* codage *d'un problème est une fonction de l'ensemble des instances I dans les mots binaires $\{0, 1\}^*$.*

En effet, pour qu'une machine (machine de Turing ou ordinateur) puisse résoudre un problème, il faut que les instances soient codée par un mot et ce codage va influer grandement sur la complexité du problème comme le montre l'exemple classique suivant. Reprenons le problème de décision "est-premier ?", considérons trois codages pour l'entier n, le premier est l'écriture de n en base 2, le second est le mot constitué de n 1, le troisième est le premier est l'écriture de n en base 2 précédé un bit valant 1 ou 0 suivant que n est premier ou non. Dans un cas la taille de l'instance codée est en $\Theta(\ln(n))$ alors que dans les deux autres la taille de l'entrée est en $\Theta(n)$ et comme la complexité (TIME) est décrite en fonction de la taille de l'entrée, il en résulte que cela risque avoir un impact important. Pour le dernier codage, il existe un algorithme en $O(1)$ car il suffit de lire le premier bit ! Ceci nous amène à considérer la notion de problèmes concrets.

Définition 1.16 *Un* problème concret *Q un couple (P, e) formé d'un problème de décision P et d'un codage e des instances.*

Sur lequel on peut donner une définition de complexité d'un algorithme :

Définition 1.17 *Un algorithme (une machine de Turing) résout un problème de décision (P, e) en temps $f(n)$ si le langage $e(P^{-1}(1))$ (c'est à dire l'encodage des instances qui renvoient vrai) est décidé pour toutes les instances de taille n (où la taille est le nombre de lettres du codage) en au plus $f(n)$ calculs.*

La section suivante va permettre d'introduire une définition de classes de complexité.

1.6 Quelques classes de complexité "déterministe"

Nous pouvons définir une première classe de complexité :

Définition 1.18 *La classe $DTIME(f(n))$ est la classe des problèmes concrets qui sont résolubles en temps $O(f(n))$.*
La classe $DSPACE(f(n))$ est la classe des problèmes concrets qui sont résolubles en espace en $O(f(n))$.

Définition 1.19 *La classe $PTIME$ (ou P) est la classe des problèmes concrets qui sont résolubles en temps polynomial.*
C'est à dire $PTIME = \bigcup_{k>0} DTIME(n^k)$
La classe $PSPACE$ est la classe des problèmes concrets qui sont résolubles en espace polynomial.
C'est à dire $PSPACE = \bigcup_{k>0} DSPACE(n^k)$

Définition 1.20 *La classe $EXPTIME$ (ou EXP) est la classe des problèmes concrets qui sont résolubles en temps poly-exponentiel.*
C'est à dire $EXPTIME = \bigcup_{k>0} DTIME(\exp(n^k))$
La classe $EXPSPACE$ est la classe des problèmes concrets qui sont résolubles en espace poly-exponentiel.
C'est à dire $EXPSPACE = \bigcup_{k>0} DSPACE(\exp(n^k))$

Chacune de ces définitions admet un analogue dans le cas de machines de Turing non déterministes. À la classe P correspond la classe NP des problèmes concrets qui sont résolubles en temps polynomial par une machine de Turing non déterministe. De même, nous avons la classe $NEXP$.

1.7 Autre manière de définir la complexité "non-déterministe"

Il existe une autre façon, équivalente de définir la classe des problèmes NP. Cette approche repose sur la notion de certificat.

Définition 1.21 *Un* algorithme de validation [5] *est un algorithme de décision qui prend 2 arguments, x, y, où x est l'entrée standard et y est un* certificat. *L'algorithme A valide la chaine x s'il existe un certificat y tel que $A(x, y) = 1$. Le langage* validé *par un algorithme de validation est*

$$L = \{x \in \{0,1\}^*, \exists y \in \{0,1\}^*, A(x, y) = 1\}$$

Définition 1.22 (Classe NP) *La classe NP est la classe des problèmes de décision (I, e) tel que le langage L des codages acceptés est validé par un algorithme polynomial. En d'autres termes, il existe un algorithme polynomial A tel que :*

$$L = \{x \in \{0,1\}^*, \exists y \in \{0,1\}^*, \exists c > 0, |y| = O(|x|^c) \text{ et } A(x, y) = 1\}$$

Définition 1.23 *Une fonction f de $\{0,1\}^*$ dans $\{0,1\}^*$ est* calculable en temps polynomial *s'il existe un algorithme* [6] *dont le temps de calcul est polynomial et qui prend en entrée le mot w et restitue le mot $f(w)$.*

Définition 1.24 *Deux codages e_1 et e_2 d'un même ensemble I sont polynomialement équivalent s'il existe deux fonctions calculables f_{12} et f_{21} en temps polynomial permettant de passer de l'un à l'autre des codages ($\forall i \in I, f_{12}(e_1(i)) = e_2(i)$ et $\forall i \in I, f_{21}(e_2(i)) = e_1(i)$)*

En particulier, l'on voit que si un problème a deux codages polynomialement équivalent alors il appartient à la classe PTIME pour les deux codages ou pour aucun des deux.

1.8 NP-complétude et réductibilité

Définition 1.25 *Un langage L_1 est réductible en temps polynomial à un problème L_2, s'il existe une fonction f calculable en temps polynomial telle que $w \in L_1 \Leftrightarrow f(w) \in L_2$. On note alors $L_1 \leq_P L_2$.*

5. Il existe une définition utilisant des machines de Turing, mais nous quittons progressivement ce niveau de formalisme
6. Une machine de Turing

Notons qu'un problème concret Q sur les instances I et avec le codage e, le langage accepté L est $e(Q^{-1}(1))$, c'est à dire l'encodage des instances vraies.

Lemme 1.26 *Soit L_1, L_2 deux langages acceptés par deux problèmes concrets Q_1, Q_2 tels que $L_1 \leq_P L_2$, alors $Q_2 \in P$ implique $Q_1 \in P$.*

Définition 1.27 *Un problème concret Q est NP-complet si et seulement si :*

- $Q \in NP$
- $\forall Q' \in NP, L' \leq_P L$ *où L et L' sont les langages acceptés de Q et Q'.*

Un problème concret qui vérifie 2) est dit *NP-difficile.*

Proposition 1.28 *Si un problème concret NP-Complet était résoluble en temps polynomial alors on aurait $P = NP$.*

1.9 La classe NP-Complet est non-vide

Un circuit booléen est composé de portes ET, OU, et NON et de cables.

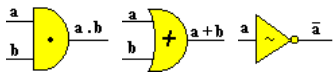

Figure 4 – Les 3 portes logiques ET, OU, NON

Nous considérons ici que les circuits ont une unique sortie, en voici un exemple :

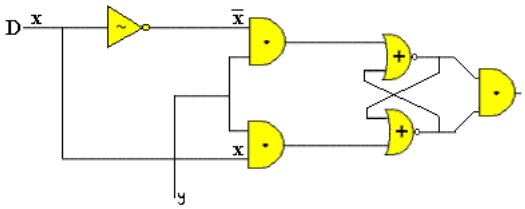

Figure 5 – Un circuit logique

Le problème de la satisfaisabilité d'un circuit, est de savoir s'il existe des valeurs d'entrée (ici x et y) telles que la sortie donne 1.

Théorème 1.29 *Le problème de la satisfaisabilité d'un circuit est NP-complet (quelque soit le codage "raisonnable" des instances). On le note CIRCUIT-SAT.*

1.10 Preuve de NP-Complet et Réduction

Lemme 1.30 *Si un problème concret Q est dans NP et qu'il existe un problème NP-Complet Q' tel que $Q' \leq_P Q$, alors Q est aussi NP-Complet.*

Cela donne la démarche a adopter pour montrer qu'un problème X est NP-Complet :

1. Montrer que le problème X est dans NP.
2. Choisir un problème Q, NP-complet approprié.
3. Construire une réduction polynomiale f de Q vers X. En d'autres termes, $Q \leq_P X$.