Université de Lille
FST- Département Informatique

Algorithmique et ComplexiTé

2018–2019
M1 Informatique
S.Tison

Quelques rappels de base
à propos de la complexité et de la correction d'algorithmes

---

1 Complexité d'un algorithme

S'intéresser à la complexité (dynamique) d'un algorithme, c'est chercher à évaluer les ressources utilisées par l'algorithme. Plus précisément, on cherche à exprimer le coût de l'algorithme - la quantité de ressources utilisées - en fonction de la taille des données: les ressources considérées peuvent être le temps, la mémoire et le matériel (par exemple le nombre de processeurs, le nombre de canaux de communications ...); nous nous intéresserons la plupart du temps à la complexité temporelle. Avant de chercher à estimer la complexité, il faut donc préciser:

- La *taille des données*: il faut d'abord définir ce qu'on entend par *taille des données*: c'est "normalement" **le nombre de bits de leur représentation en binaire**; dans certains cas, on étudie la complexité en fonction d'une autre notion de taille, comme le nombre d'éléments pour une liste, la dimension pour une matrice, ...

- La notion de coût: on définit pour une donnée d, le coût de l'algorithme A pour cette donnée d: $cout_A(d)$. Pour évaluer ce coût indépendamment de la machine (physique) sur lequel l'algorithme sera exécuté, on utilise un modèle de machines séquentielles "classiques", en général la RAM ("Random Access Machine"). (Bien sûr, si on évalue la complexité d'un algorithme parallèle, le modèle choisi sera différent, par exemple PRAM.) Dans ce modèle, chaque instruction "simple" (arithmétique de base, accès, ...) a un coût de 1; on supposera donc en général qu'une instruction élémentaire a un *coût uniforme*, i.e. indépendant de la taille de ses opérandes, par opposition au *coût logarithmique* où on tient compte de la taille des données.

  On simplifie en général encore, se bornant à "compter" certaines instructions: par exemple, les comparaisons pour un tri par comparaisons, les accès à la mémoire externe pour un tri externe, les opérations arithmétiques de base pour des produits de matrices ...

- Quelle(s) complexité(s) on cherche à calculer. En effet, pour deux données de même taille, l'exécution d'un algorithme peut bien sûr utiliser des quantités de ressources différentes. On définit donc trois mesures de complexité:

  – *La complexité dans le meilleur des cas:* elle est définie par

  $$Inf_A(n) = inf\{cout_A(d)/d \text{ de taille } n\}$$

  – *La complexité dans le pire des cas:* elle est donnée par

  $$Sup_A(n) = sup\{cout_A(d)/d \text{ de taille } n\}$$

  – *La complexité en moyenne:* pour la définir, il faut connaître la distribution des données et donc disposer d'une loi de probabilité modélisant cette distribution; celle-ci ne fait souvent qu'approximer la distribution réelle des données, car il est très difficile de proposer un modèle des données "réelles": on suppose par exemple souvent que toutes les données de même taille sont équiprobables, ce qui est peu réaliste. La complexité en moyenne est alors donnée par

  $$Moy_A(n) = \sum_{d \text{ de taille } n} p(d) * cout(d)$$

  où $p(d)$ désigne la probabilité d'avoir la donnée d parmi toutes les données de taille n.

Quand on parle de la complexité d'un algorithme sans préciser laquelle, c'est souvent de la complexité temporelle dans le pire des cas qu'on parle. La complexité dans le meilleur des cas n'est pas très utilisée; la complexité en moyenne est d'une certaine façon celle qui révèle le mieux le comportement "réel" de l'algorithme à condition de disposer d'un modèle satisfaisant de la distribution des données, mais bien sûr elle ne garantit rien sur le pire des cas ... et elle est souvent difficile à calculer, même de façon approximative!

On ne calcule pas en général la complexité exacte mais on se contente de calculer son ordre de grandeur asymptotique voire de borner celui-ci. Par exemple, si on fait $n^2 + 2n - 5$ opérations, on retiendra juste que l'ordre de grandeur est n^2. On utilisera donc les notations classiques sur les ordres de grandeur:

Ordres de grandeur, Comportements asymptotiques: Quelques rappels

Soient f et g deux fonctions de N dans$R$:

. $f \in O(g)$ Ssi_{def} $\exists c \in R^{+*}, \exists A$,tels que $\forall n > A, f(n) \leq c * g(n)$
  On dit que $f$est dominée asymptotiquement par g. On notera souvent $f = O(g)$

. $f \in \Theta(g)$ Ssi_{def} $\exists c, C \in R^{+*}, \exists A$,tels que $\forall n > A, C * g(n) \leq f(n) \leq c * g(n)$
  On dit alors que f et g sont de même ordre de grandeur asymptotique. On notera souvent $f = \Theta(g)$.

. $f \in \Omega(g)$ Ssi_{def} $\exists c \in R^{+*}, \exists A$,tels que $\forall n > A, f(n) \geq c * g(n)$
  On dit que f domine asymptotiquement g. On notera souvent $f = \Omega(g)$

. $f \in o(g)$ Ssi_{def} $\forall \epsilon \in R^{+*}, \exists A$,tels que $\forall n > A, f(n) \leq \epsilon * g(n)$
  On dit que $f$est négligeable asymptotiquement devant g. On notera souvent $f = o(g)$

On dira alors par exemple que la complexité d'un algorithme est "en $\Theta(n \ log \ n)$" ou en $O(n^2)$ ou ...

Remarque: s'intéresser à l'ordre de grandeur plutôt qu'à la complexité exacte se justifie si les données manipulées sont de grande taille. Il ne faut pourtant pas négliger trop vite les constantes.
Q?: à partir de quelle taille de données, un algorithme de complexité exacte $20 * n * \log n$ sera-t-il plus intéressant qu'un algorithme en n^2 ?.

Vocabulaire

Un algorithme est dit

. en temps **constant** si sa complexité dans le pire des cas est bornée par une constante.

. **linéaire** (resp. linéairement borné) si sa complexité (dans le pire des cas) est en $\Theta(n)$ (resp. $O(n)$).

. **quadratique** (resp. au plus quadratique) si sa complexité (dans le pire des cas) est en $\Theta(n^2)$ (resp. en $O(n^2)$).

. **polynomial** ou polynomialement borné, si sa complexité (dans le pire des cas) est en $O(n^p)$ pour un certain p.

. (au plus) **exponentiel** ou exponentiellement borné si elle est en $O(2^{n^p})$, pour un certain $p > 0$.

Abus de langage On dit souvent simplement qu'un algorithme est linéaire si il est linéairement borné, i.e. si sa complexité (dans le pire des cas) est en $O(n)$; similairement on dira souvent qu'il est quadratique si sa complexité dans le pire des cas est en $O(n^2)$.
On dira qu'il est super-polynomial si il n'est pas polynomialement borné.
"Exponentiel" étant considéré abusivement comme synonyme d'impraticable, on dit parfois qu'un algorithme est exponentiel si sa complexité est **au moins** exponentielle, i.e. en $\Omega(2^n)$, voire en $\Omega(k^n)$, avec $k > 1$.

Algorithmes Praticables

Par "convention", un algorithme est dit **praticable** si il est polynomial; cette convention appelle quelques remarques en sa défaveur ou en sa faveur:

- Si un algorithme est exponentiel dans le pire des cas, il peut être polynomial en moyenne; si les pires cas sont exceptionnels, il peut être praticable ... en pratique (cf le simplexe). Des solveurs de plus en plus sophistiqués (Sat, programmation linéaire en entiers, ...) permettent de résoudre sur des données de taille raisonnable des problèmes a priori exponentiels .

- L'exécution d'un algorithme polynomial dont la complexité moyenne est de l'ordre de grandeur de n^5 microsecondes prendrait 30 ans si $n = 1000$.

+ En pratique, il s'avère que la plupart des algorithmes polynomiaux ont un comportement asymptotique équivalent à un polynôme de faible degré, 2 ou 3.

+ La classe des algorithmes polynomiaux a de bonnes propriétés de clôture: par exemple, une "séquence" de deux algorithmes polynomiaux est polynomiale.

+ Elle est indépendante du modèle séquentiel "classique" choisi: un algorithme polynômial pour le modèle des machines de Turing, le sera pour une RAM et vice-versa .

+ Si un algorithme est polynomial, quand la taille de la donnée double, le temps d'exécution est au plus multiplié par une certaine constante (indépendante de la taille de la donnée).

Le tableau ci-dessous montre le temps d'exécution de cinq algorithmes selon leur comportement et la taille des données; on suppose que la durée d'une instruction élémentaire est de l'ordre de la μs.

Taille \Comportement	$\log n$	n	$n \log n$	n^2	2^n
10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$1000\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$1/100s$	10^{14} siècles
1000	$10\mu s$	$1000\mu s$	$1/100s$	$1s$	astronomique
10000	$13\mu s$	$1/100s$	$1/7s$	$1,7mn$	astronomique
100000	$17\mu s$	$1/10s$	$2s$	$2,8h$	astronomique

2 La correction des programmes: rappel des définitions de base de la logique de Hoare

"Program testing can be used to show the presence of bugs, but never to show their absence !" Edsger W. Dijkstra

"One does not need to give a formal proof of an obviously correct program; but one needs a thorough understanding of formal proof methods to know when correctness is obvious." John C. Reynolds

Prouver qu'un programme ou un algorithme est correct, c'est prouver qu'il correspond à la spécification donnée. La *l*ogique de Hoare-Floyd a été introduite à la fin des années 60 pour formaliser la relation entre langage de programmation (impératif) et langage de spécification. Elle est basée sur la notion de *triplet de Hoare*[1], *de la forme* $\{P\}$ `A` $\{Q\}$*; P est la précondition, Q la postcondition, A étant une expression du langage de programmation.*

P et Q sont des assertions, i.e. des formules du langage de spécification décrivant les valeurs des variables, l'état de la mémoire ou du système. En toute rigueur, le langage de spécification devrait être un langage mathématique comme la logique des prédicats, mais dans un objectif de documentation plus que de preuve de programmes, le langage de spécification peut être le langage naturel (en évitant les ambiguités!).

[1]Charles Anthony Richard Hoare est lauréat du prestigieux Prix Turing (1980). Il a eu de nombreuses contributions remarquables,comme la logique de Hoare mais aussi le Quicksort ou la définition de CSP, un modèle pour la programmation concurrente.

L'interprétation de $\{P\}$ `A` $\{Q\}$ *est donc: si le système vérifie* $\{P\}$*, après l'exécution de* `A`*, on peut assurer qu'il vérifie* $\{Q\}$*; en fait, c'est un peu plus compliqué que cela; en effet, il se peut que l'exécution de* `A` *ne termine pas! On a donc deux interprétations possibles:*

si $\{P\}$ *est vérifiée, après l'exécution de* `A`*, si celle-ci termine, on peut assurer* $\{Q\}$*;*

si $\{P\}$ *est vérifiée, la terminaison de* `A` *est assurée et après son exécution, on peut assurer* $\{Q\}$*;*

Dans le premier cas, on parle de correction partielle, dans le deuxième de correction totale. La preuve de correction d'un programme selon la logique de Hoare s'appuie sur les règles de la logique de Hoare qui permettent de prouver le comportement d'instructions composées à partir du comportement de ses composants; bien sûr, cela suppose que la sémantique des primitives du langage utilisé soit bien définie.

Les règles d'inférence de Hoare

Prouver un programme revient donc à prouver:

$\{$*condition sur les données*$\}$ `Programme` $\{$*condition exprimant les résultats attendus*$\}$

La preuve est construite à partir des axiomes -qui définissent la sémantique des primitives du langage-et des règles d'inférences de Hoare. Voici les principales:

La séquence: $$\frac{\{P\}\ \texttt{A}\ \{Intermede\} \quad \{Intermede\}\ \texttt{B}\ \{Q\}}{\{P\}\ \texttt{A;B}\ \{\texttt{Q}\}}$$

La conditionnelle: $$\frac{\{P \wedge cond\}\ \texttt{A}\ \{Q\} \quad \{P \wedge \neg cond\}\ \texttt{B}\ \{Q\}}{\{P\}\ \texttt{si cond alors A sinon B}\ \ \{Q\}}$$

Le "tant que":

si on trouve une assertion i, qu'on appelle invariant*, telle que soient vérifiées les trois conditions:*

$P \Rightarrow i$*: la précondition assure l'invariant*

$\{i \wedge cond\}$ `A` $\{i\}$ *: l'invariant reste ... invariant à chaque pas de boucle*

$\{i \wedge \neg cond\} \Rightarrow \{Q\}$*: l'invariant et la condition de sortie assurent la postcondition*

alors: $\{P\}$ `tant que cond faire A` $\{Q\}$

Attention: pour avoir une preuve de correction totale, il faut aussi faire une preuve d'arrêt de la boucle!

L'affectation:

$P[x \leftarrow exp]$ `x:=exp` P*, en supposant que exp ne contient pas de fonctions à effet de bord.*

Par exemple, $x + y < 5$ `x:=x+y` $x < 5$

Le renforcement de la précondition $$\frac{P \Rightarrow Q \quad \{Q\}\ \texttt{A}\ \{R\}}{\{P\}\ \texttt{A}\ \{R\}}$$

Et dualement, l'affaiblissement de la postcondition $$\frac{\{P\}\ \texttt{A}\ \{Q\} \quad Q \Rightarrow R}{\{P\}\ \texttt{A}\ \{R\}}$$

Soit: Qui peut le plus, peut le moins!