Fondements de l'informatique
Logique, modèles, et calculs

Chapitre: Bases de l'analyse de complexité d'algorithmes

Cours INF412

de l'Ecole Polytechnique

Olivier Bournez

bournez@lix.polytechnique.fr

Version du 16 juillet 2023

Bases de l'analyse de complexité d'algorithmes

Les discussions précédentes ont fait intervenir l'existence ou non d'algorithmes pour résoudre un problème donné, mais en ignorant un aspect pourtant essentiel en pratique : les ressources nécessaires à son exécution, c'est-à-dire par exemple le temps ou la mémoire nécessaire sur la machine pour l'exécuter.

L'objectif du chapitre suivant est de se focaliser sur l'une des ressources : le temps de calcul. Dans le chapitre qui suit, nous évoquerons d'autres ressources comme l'espace mémoire. On pourrait parler du temps parallèle, c'est-à-dire du temps sur une machine parallèle, etc.

Commençons toutefois par bien comprendre la différence entre le cadre des chapitres qui suivent et les chapitres précédents : dans les chapitres précédents, on parlait de *calculabilité*, c'est-à-dire que l'on se posait la question de l'existence (ou de la non-existence) de solutions algorithmiques à des problèmes donnés. On va maintenant parler de *complexité* : c'est-à-dire que l'on se focalise maintenant uniquement sur des problèmes *décidables*, pour lesquels on connaît un algorithme. La question que l'on se pose maintenant est de comprendre s'il existe un algorithme *efficace*.

Cela nous mène tout d'abord à préciser ce que l'on a envie d'appeler *efficace*, et comment on mesure cette *efficacité*. Toutefois, avant, il faut que notre lecteur ait les idées au clair sur ce que l'on appelle la complexité d'un algorithme, ou la complexité d'un problème, ce qui n'est pas la même chose.

Remarque 1 *Même si nous évoquons les complexités en moyenne dans ce chapitre, nous n'en aurons pas besoin à aucun moment dans les chapitres qui suivent : nous le faisons surtout pour expliquer pourquoi elles sont peu utilisées pour ces types de problèmes.*

1 Complexité d'un algorithme

On considère donc typiquement dans ce chapitre un problème $\mathscr{P}$ pour lequel on connaît un algorithme $\mathscr{A}$: cet algorithme, on sait qu'il est correct, et qu'il termine. Il prend en entrée une donnée d, et produit un résultat en sortie $\mathscr{A}(d)$ en utilisant certaines ressources (on ne parle donc plus que de problèmes décidables).

Exemple 1 *Le problème $\mathscr{P}$ peut par exemple être celui de déterminer si un nombre v est parmi une liste de n nombres.*

Il est clair que l'on peut bien construire un algorithme $\mathscr{A}$ pour résoudre ce problème : par exemple,

— on utilise une variable res que l'on met à 0 ;
— on parcourt la liste, et pour chaque élément :
  — on regarde si cet élément est le nombre v :
    — si c'est le cas, alors on met la variable res à 1
— à l'issue de cette boucle, on retourne res.

Cet algorithme n'est pas le plus efficace que l'on puisse envisager. D'une part, on pourrait s'arrêter dès que l'on a mis res à 1, puisqu'on connaît la réponse. D'autre part, on peut clairement faire tout autrement, et utiliser par exemple une *recherche par dichotomie* (un algorithme récursif), si l'on sait que la liste est triée.

1.1 Premières considérations

On mesure toujours l'efficacité, c'est-à-dire la complexité, d'un algorithme en terme d'une mesure élémentaire μ à valeur entière : cela peut être le nombre d'instructions effectuées, la taille de la mémoire utilisée, ou le nombre de comparaisons effectuées, ou toute autre mesure.

Il faut simplement qu'étant donnée une entrée d, on sache clairement associer à l'algorithme $\mathscr{A}$ sur l'entrée d, la valeur de cette mesure, notée $\mu(\mathscr{A},d)$: par exemple, pour un algorithme de tri qui fonctionne avec des comparaisons, si la mesure élémentaire μ est le nombre de comparaisons effectuées, $\mu(\mathscr{A},d)$ est le nombre de comparaisons effectuées sur l'entrée d (une liste d'entiers) par l'algorithme de tri $\mathscr{A}$ pour produire le résultat $\mathscr{A}(d)$ (cette liste d'entiers triée).

La fonction $\mu(\mathscr{A},d)$ dépend de $\mathscr{A}$, mais aussi de l'entrée d. La qualité d'un algorithme $\mathscr{A}$ n'est donc pas un critère absolu, mais une fonction quantitative $\mu(\mathscr{A},.)$ des données d'entrée vers les entiers.

1.2 Complexité d'un algorithme au pire cas

En pratique, pour pouvoir appréhender cette fonction, on cherche souvent à évaluer cette complexité pour les entrées d'une certaine *taille* : il y a souvent une fonction $taille$ qui associe à chaque donnée d'entrée d, un entier $taille(d)$, qui correspond à un paramètre naturel. Par exemple, cette fonction peut être celle qui compte le nombre d'éléments dans la liste pour un algorithme de tri, la taille d'une matrice pour le calcul du déterminant, la somme des longueurs des listes pour un algorithme de concaténation.

Pour passer d'une fonction des données vers les entiers, à une fonction des entiers (les tailles) vers les entiers, on considère alors la complexité *au pire cas* : la complexité $\mu(\mathscr{A},n)$ de l'algorithme $\mathscr{A}$ sur les entrées de taille n est définie par

$$\mu(\mathscr{A},n) = \max_{d \text{ entrée avec } taille(d)=n} \mu(\mathscr{A},d).$$

Autrement dit, la complexité $\mu(\mathscr{A}, n)$ est la complexité la pire sur les données de taille n.

Par défaut, lorsqu'on parle de *complexité d'algorithme* en informatique, il s'agit de complexité au pire cas, comme ci-dessus.

Si l'on ne sait pas plus sur les données, on ne peut guère faire plus que d'avoir cette vision pessimiste des choses : cela revient à évaluer la complexité dans le pire des cas (le meilleur des cas n'a pas souvent un sens profond, et dans ce contexte le pessimisme est de loin plus significatif).

1.3 Complexité moyenne d'un algorithme

Pour pouvoir en dire plus, il faut en savoir plus sur les données. Par exemple, qu'elles sont distribuées selon une certaine loi de probabilité.

Dans ce cas, on peut alors parler de complexité en moyenne : la complexité moyenne $\mu(\mathscr{A}, n)$ de l'algorithme $\mathscr{A}$ sur les entrées de taille n est définie par

$$\mu(\mathscr{A}, n) = \mathbb{E}[\mu(\mathscr{A}, d) | d \text{ entrée avec } taille(d) = n],$$

où $\mathbb{E}$ désigne l'espérance (la moyenne).

Si l'on préfère,

$$\mu(\mathscr{A}, n) = \sum_{d \text{ entrée avec } taille(d)=n} \pi(d)\mu(\mathscr{A}, d),$$

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

En pratique, le pire cas est rarement atteint et l'analyse en moyenne peut sembler plus séduisante.

Mais, d'une part, il est important de comprendre que l'on ne peut pas parler de moyenne sans loi de probabilité (sans distribution) sur les entrées. Cela implique que l'on connaisse d'autre part la distribution des données en entrée, ce qui est très souvent délicat à estimer en pratique. Comment anticiper par exemple les listes qui seront données à un algorithme de tri par exemple ?

On fait parfois l'hypothèse que les données sont équiprobables (lorsque cela a un sens, comme lorsqu'on trie n nombres entre 1 et n et où l'on peut supposer que les permutations en entrée sont équiprobables), mais cela est parfois arbitraire, et pas totalement justifiable.

Et enfin, comme nous allons le voir sur quelques exemples, les calculs de complexité en moyenne sont plus délicats à mettre en œuvre.

2 Complexité d'un problème

On peut aussi parler de la *complexité d'un problème* : cela permet de discuter de l'optimalité ou non d'un algorithme pour résoudre un problème donné.

On fixe un problème $\mathscr{P}$: par exemple celui de trier une liste d'entiers. Soit $Alg(\mathscr{P})$ la classe de tous les algorithmes qui résolvent $\mathscr{P}$: un algorithme $\mathscr{A}$ de $Alg(\mathscr{P})$ est

un algorithme qui répond à la spécification du problème $\mathscr{P}$: pour chaque donnée d, il produit la réponse correcte $\mathscr{A}(d)$.

La complexité du problème $\mathscr{P}$ est définie comme l'infimum [1] de la complexité des algorithmes de $Alg(\mathscr{P})$. Par conséquent, un algorithme $\mathscr{A}$ est *optimal* si sa complexité est égale à la complexité optimale d'un algorithme de $Alg(\mathscr{P})$.

Autrement dit, si aucun algorithme $\mathscr{B}$ de $Alg(\mathscr{P})$ ne possède une complexité plus faible. On écrit $\mu(\mathscr{P}, n)$ pour la complexité d'un algorithme optimal [2] sur les entrées de taille n. Autrement dit, on ne fait plus seulement varier les entrées de taille n, mais aussi l'algorithme. On considère le meilleur algorithme qui résout le problème. Le meilleur étant celui avec la meilleure complexité au sens de la définition précédente, et donc au pire cas. C'est donc la complexité du meilleur algorithme au pire cas.

L'intérêt de cette définition est de permettre d'affirmer qu'un algorithme est optimal : C'est-à-dire que tout autre algorithme correct est nécessairement moins efficace par définition.

3 Exemple : Calcul du maximum

Nous allons illustrer la discussion précédente par un exemple : le problème du calcul du maximum. Ce problème est le suivant : on se donne en entrée une liste d'entiers naturels $e_1, e_2, \cdots, e_n$, avec $n \geq 1$, et on cherche à déterminer en sortie $M = \max_{1 \leq i \leq n} e_i$, c'est-à-dire le plus grand de ces entiers.

3.1 Complexité d'un premier algorithme

En considérant que l'entrée est rangée dans un tableau, la fonction Java suivante résout le problème.

```
static int max(int T[]) {
    int l = T.length-1;
    int M =  T[l];
    l = l-1;
    while (l ≥ 0) {
     if (M < T[l])  M = T[l];
     l = l-1;
    }
    return M;
  }
```

Si notre mesure élémentaire μ correspond au nombre de comparaisons, nous en faisons 2 par itération de la boucle, qui est exécutée $n-1$ fois, plus 1 dernière du type $l \geq 0$ lorsque l vaut 0. Nous avons donc $\mu(\mathscr{A}, n) = 2n-1$ pour cet algorithme $\mathscr{A}$, où n est la taille de l'entrée, c'est-à-dire le nombre d'entiers dans la liste $e_1, e_2, \cdots, e_n$. Ce nombre est indépendant de la donnée d, et donc $\mu(\mathscr{A}, n) = 2n-1$.

1. S'il existe.
2. En supposant qu'il existe. Il peut ne pas exister.

Par contre, si notre mesure élémentaire μ correspond au nombre d'affectations, nous en faisons 3 avant la boucle **while**. Chaque itération de la boucle effectue soit 1 ou 2 affectations suivant le résultat du test M<T[l]. On a donc pour une entrée d de taille n, $n+2 \leq \mu(\mathscr{A}, d) \leq 2n+1$: la valeur minimum est atteinte pour une liste ayant son maximum dans son dernier élément, et la valeur maximum pour une liste sans répétition triée dans l'ordre décroissant. Cette fois $\mu(\mathscr{A}, d)$ dépend de l'entrée d. La complexité au pire cas est donnée par $\mu(\mathscr{A}, n) = 2n$.

3.2 Complexité d'un second algorithme

Si l'entrée est rangée dans une liste, définie par exemple par

```
class List {
  int val ;      // La valeur
  List next ;    // La suite

  List (int  val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

alors la fonction suivante résout le problème.

```
static int max(List a) {
  int M = a.val;
  for (a = a.next; a ≠ null;  a = a.next) {
    if (a.val > M)
      M = a.val;}
  return M;
}
```

Si notre mesure élémentaire μ correspond au nombre de comparaisons entre entiers (nous ne comptons pas les comparaisons entre variables de type référence sur le type List) nous en faisons 1 par itération de la boucle, qui est exécutée $n-1$ fois, soit au total $n-1$.

La complexité $\mu(\mathscr{A}, n)$ de cet algorithme $\mathscr{A}$ sur les entrés de taille n est donc $n-1$.

3.3 Complexité du problème

On peut se poser la question de savoir s'il est possible de faire moins de $n-1$ telles comparaisons : la réponse est non, à condition de se limiter aux algorithmes qui fonctionnent seulement par comparaisons [3]. En effet, cet algorithme est optimal en nombre de comparaisons.

3. Si les entrées sont des entiers et que l'on s'autorise de l'arithmétique, il peut peut-être être possible de limiter les comparaions. On ne discutera pas ce type d'algorithmes ici.

En effet, considérons la classe $\mathscr{C}$ des algorithmes qui résolvent le problème de la recherche du maximum de n éléments en utilisant comme critère de décision les comparaisons entre éléments, avec les hypothèses plus haut.

Commençons par énoncer la propriété suivante : tout algorithme $\mathscr{A}$ de $\mathscr{C}$ est tel que tout élément autre que le maximum est comparé au moins une fois avec un élément qui lui est plus grand.

En effet, soit i_0 le rang du maximum M retourné par l'algorithme sur une liste $L = e_1.e_2.\cdots.e_n : e_{i_0} = M = \max_{1\leq i\leq n} e_i$. Raisonnons par l'absurde : soit $j_0 \neq i_0$ tel que e_{j_0} ne soit pas comparé avec un élément plus grand que lui. L'élément e_{j_0} n'a donc pas été comparé avec e_{i_0} le maximum.

Considérerons la liste $L' = e_1.e_2.\cdots.e_{j_0-1}.M+1.e_{j_0+1}.\cdots.e_n$ obtenue à partir de L en remplaçant l'élément d'indice j_0 par M+1.

L'algorithme $\mathscr{A}$ effectuera exactement les mêmes comparaisons sur L et L', sans comparer $L'[j_0]$ avec $L'[i_0]$ et donc retournera $L'[i_0]$, ce qui est incorrect. D'où une contradiction, qui prouve la propriété.

Il découle de la propriété qu'il n'est pas possible de déterminer le maximum de n-éléments en moins de $n-1$ comparaisons entre entiers. Autrement dit, la complexité du problème $\mathscr{P}$ du calcul du maximum sur les entrées de taille n est $\mu(\mathscr{P}, n) = n-1$.

L'algorithme précédent fonctionnant en $n-1$ telles comparaisons, il est optimal pour cette mesure de complexité.

3.4 Complexité de l'algorithme en moyenne

Si notre mesure élémentaire μ correspond au nombre d'affectations à l'intérieur de la boucle **for**, on voit que ce nombre dépend de la donnée.

On peut s'intéresser à sa complexité en moyenne : il faut faire une hypothèse sur la distribution des entrées. Supposons que les listes en entrée dont on cherche à calculer le maximum sont des permutations de $\{1,2,\cdots,n\}$, et que les $n!$ permutations sont équiprobables.

On peut montrer [Sedgewick and Flajolet, 1996, Froidevaux et al., 1993] que la complexité moyenne sur les entrées de taille n pour cette mesure élémentaire μ est alors donnée par H_n, le $n$ième nombre harmonique : $H_n = 1+\frac{1}{2}+\frac{1}{3}+\cdots\frac{1}{n}$. H_n est de l'ordre de $\log n$ lorsque n tend vers l'infini.

Cependant, le calcul est technique, et serait laborieux dans le cadre seul de ce cours.

Simplifions, en nous intéressons à un problème encore plus simple : plutôt que de rechercher le maximum dans la liste $e_1, e_2, \cdots, e_n$, avec $n \geq 1$, donnons nous une liste d'entiers de $\{1,2,\ldots,k\}$ et un entier $1 \leq v \leq k$, et cherchons à déterminer s'il existe un indice $1 \leq i \leq n$ avec $e_i = v$.

L'algorithme suivant résout le problème.

```
static boolean trouve(int[] T, int v) {
    for (int i = 0; i < T.length; i++)
        if (T[i] == v)
            return true;
```

```
        return false;
    }
```

Sa complexité au pire cas en nombre d'instructions élémentaires est linéaire en n, puisque la boucle est effectuée n fois dans le pire cas.

Observons que les listes en entrée sont des fonctions de $\{1,2,\ldots,n\}$ dans $\{1,2,\ldots,k\}$, que l'on appellera *tableau*. Supposons que chacun des tableaux est choisi de façon équiprobable.

Remarquons qu'il y a k^n tableaux. Parmi ceux-ci, $(k-1)^n$ ne contiennent pas l'élément v et dans ce cas, l'algorithme procède à exactement n itérations. Dans le cas contraire, l'entier est dans le tableau et sa première occurrence est alors i avec une probabilité de

$$\frac{(k-1)^{i-1}}{k^i}$$

et il faut alors procéder à i itérations. Au total, nous avons une complexité moyenne de

$$C = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i$$

Or

$$\forall x, \sum_{i=1}^{n} i x^{i-1} = \frac{1 + x^n(nx - n - 1)}{(1-x)^2}$$

(il suffit pour établir ce résultat de dériver $\sum_{i=1}^{n} x^i = \frac{1-x^{n+1}}{1-x}$) et donc

$$C = n\frac{(k-1)^n}{k^n} + k\left(1 - \frac{(k-1)^n}{k^n}(1 + \frac{n}{k})\right) = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right)$$

4 Asymptotiques

4.1 Complexités asymptotiques

On le voit sur l'exemple précédent, réaliser une étude précise et complète de complexité est souvent fastidieux, et parfois difficile. Aussi, on s'intéresse en informatique plutôt à l'ordre de grandeur (l'asymptotique) des complexités quand la taille n des entrées devient très grande.

4.2 Notations de Landau

Comme il en est l'habitude en informatique, on travaille souvent à un ordre de grandeur près, via les notations $\mathcal{O}(.)$. Rappelons les définitions suivantes :

Définition 1 (Notation $\mathcal{O}(.)$) *Soient f et g deux fonctions $f, g : \mathbb{N} \to \mathbb{R}^{>0}$. On note $f(n) = \mathcal{O}\left(g(n)\right)$ lorsqu'il existe des entiers c et n_0 tel que pour tout $n \geq n_0$,*

$$f(n) \leq cg(n).$$

Intuitivement, cela signifie que f est inférieur à g à une constante multiplicative près, pour les instances (données) de tailles suffisamment grandes.

De même on définit :

Définition 2 (Notations o, Ω, Θ) *Soient f et g deux fonctions $f, g : \mathbb{N} \to \mathbb{R}^{>0}$.*

- *On note $f(n) = o(g(n))$ lorsque pour tout réel positif c, il existe un entier n_0 tels que pour tout $n \geq n_0$,*
$$f(n) \leq cg(n).$$
- *On note $f(n) = \Omega(g(n))$ lorsqu'il existe des entiers c et n_0 tels que pour tout $n \geq n_0$,*
$$cg(n) \leq f(n).$$
(on a dans ce cas $g = \mathcal{O}(f)$)
- *On note $f(n) = \Theta(g(n))$ lorsque $f(n) = \mathcal{O}(g(n))$ et $f(n) = \Omega(g(n))$.*

Exercice 1 *(corrigé page 243) Soient f et g deux fonctions telles que*
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
existe et vaut un nombre $c > 0$. Prouver que $f(n) = \Theta(g(n))$.

Exercice 2 *Prouver :*

- *Si $f = \mathcal{O}(g)$ et $g = \mathcal{O}(h)$ alors $f = \mathcal{O}(h)$.*
- *Si $f = \Omega(g)$ et $g = \Omega(h)$ alors $f = \Omega(h)$.*
- *Si $f = \Theta(g)$ et $g = \Theta(h)$ alors $f = \Theta(h)$.*

Exercice 3 *(corrigé page 243) Donner des exemples d'algorithmes respectivement de complexité :*

- *linéaire*
- *$O(n \log n)$*
- *cubique*
- *non-polynomial*

Exercice 4 *(corrigé page 243) Supposons que l'on a des algorithmes avec les temps listés plus bas (en supposant que ce sont des temps exacts). De combien sont ralentis ces algorithmes lorque l'on* (*a*) *double la taille de l'entrée,* (*b*) *augmente la taille de l'entrée de* 1.

1. n^2
2. n^3
3. $100n^2$
4. $n\log n$
5. 2^n

5 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture du polycopié du cours INF421 (ancienne version, version 2010), ou de l'ouvrage [Kleinberg and Tardos, 2005], et en particulier de ses premiers chapitres.

Bibliographie Le texte de ce chapitre est repris du texte que nous avions écrit dans le polycopié INF421. Il est inspiré de [Kleinberg and Tardos, 2005]. L'analyse du problème du calcul du maximum, et de ses variations est basée sur leur analyse dans le livre [Froidevaux et al., 1993].

Index

Ω, *voir* notations de Landau, 10
Θ, *voir* notations de Landau, 10
$\mathscr{A}(d)$, 3
$\mu(\mathscr{A}, d)$, 4, *voir* mesure élémentaire
$\mu(\mathscr{A}, n)$, 5, *voir* complexité d'un algorithme au pire cas, *voir* complexité d'un algorithme en moyenne
o, *voir* notations de Landau, 10

algorithme, 3
    efficace, 3

calculabilité, 3
complexité, 3
    asymptotique d'un algorithme, 9
    d'un algorithme, 5
        au pire cas, 4
        en moyenne, 5
    d'un problème, 5

décidable, 3
dichotomie, 4

efficace, *voir* algorithme

mémoire, 3
mesure élémentaire, 4
    notation, voir $\mu(\mathscr{A}, d)$

notations
    de Landau, 9

optimal, 6

problème, 3

recherche par dichotomie, 4
ressources, 3

temps de calcul, 3

Bibliographie

[Froidevaux et al., 1993] Froidevaux, C., Gaudel, M., and Soria, M. (1993). *Types de donnees et algorithmes*. Ediscience International.

[Kleinberg and Tardos, 2005] Kleinberg, J. and Tardos, E. (2005). *Algorithm design*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Sedgewick and Flajolet, 1996] Sedgewick, R. and Flajolet, P. (1996). *Introduction à l'analyse d'algorithmes*. International Thomson Publishing, FRANCE.