Calculabilité et Complexité (11X008) - Printemps 2026

5. Notations asymptotiques & classes de complexité

Enseignant: Arnaud Casteigts

Assistants: M. De Francesco, M. Marseloo
Moniteurs: E. Bussod, N. Beghdadi

Dans ce cours, nous présentons les principales notations asymptotiques ($O, \Omega, \Theta, ...$) et les illustrons en analysant la complexité de plusieurs algorithmes. Nous introduisons également les classes de complexité générique pour le temps et l'espace : TIME et SPACE.

5.1 Notations asymptotiques

Nous rencontrons souvent les notations O, Ω, Θ (prononcer "grand o", "grand omega" et "grand theta") et parfois aussi les notations o et ω ("petit o", "petit omega"). Ces notations ne sont pas exclusives à l'informatique. Elles permettent de manipuler des expressions mathématiques en ignorant des détails jugés non significatifs, afin de simplifier les calculs.

Elles se définissent généralement en termes de limites. Cependant, on peut aussi les définir intuitivement comme suit. Soit n la quantité que l'on fait varier (typiquement, la taille de l'entrée de notre algorithme). Toutes ces notations ont en commun :

- d'ignorer les termes dominés (contribution marginale quand $n \to \infty$)
- d'ignorer les facteurs constants (facteurs qui ne font pas intervenir n)

Par exemple, imaginons qu'un algorithme effectue $2n^3+3n^2+4$ opérations. Quand $n \to \infty$, la contribution des termes $3n^2$ et 4 deviennent négligeables par rapport à $2n^3$ (formellement, la limite du rapport entre les deux tend vers 0). On peut donc simplifier l'expression en $2n^3$. Puis on ignore le facteur 2, qui est constant, ce qui donne n^3. Autre exemple : $n + \log n$ se simplifie en n (car $\log n$ devient négligeable par rapport à n). Par contre, $n \cdot \log n$ n'est pas simplifiable en n car on n'ignore pas les *facteurs* dominés (attention) !

Une fois ces simplifications faites, on a les équivalences suivantes :

Notation	$o()$	$O()$	$\Theta()$	$\Omega()$	$\omega()$
Signification	$<$	$\leq$	$=$	$\geq$	$>$

Ainsi, $2n^3 + 3n^2 + 4 = \Theta(n^3)$. On aurait aussi pu écrire, de manière moins précise, que ça vaut $\Omega(n^3)$, $o(n^{10})$, ou encore $O(n^3)$ ou $O(n^{10})$, mais pas $o(n^3)$ ni $\Omega(n^4)$.

Notez que le symbole "=" n'est pas vraiment une égalité. En particulier, il n'est pas symétrique, par exemple $a = O(b)$ n'implique pas que $b = O(a)$. Pour éviter les confusions, on utilise parfois le symbole $\in$ à la place (avec la même signification), p.ex. $a \in O(b)$.

Adjectifs fréquents

Constant	$O(1)$	Polynomial	$O(n^c) = n^{O(1)}$
Logarithmique	$O(\log n)$	Quasi-polynomial	$n^{log^{O(1)}n}$
Linéaire	$O(n)$	Exponentiel	$2^{n^{O(1)}}$ ou parfois juste $O(2^n)$
Quasi-linéaire	$O(n \log n)$	Factoriel	$O(n!) = O(n^n)$
Quadratique	$O(n^2)$		

Les mêmes adjectifs sont utilisés pour Θ et Ω (en disant "au moins ..." pour Ω). Pour o et ω, on parlera plutôt de super-X et de sous-X (où X est l'adjectif de votre choix), par exemple, $\omega(n)$ est super-linéaire et $o(n^2)$ est sous-quadratique. Ces adjectifs peuvent s'appliquer aux algorithmes aussi bien qu'aux problèmes qu'ils résolvent. Notez que les bases des logarithmes ne sont pas précisées, car les logs en bases différentes ne diffèrent que par des facteurs constants (c'est bien pratique !).

5.2 Analyse d'algorithmes et de problèmes

On utilise beaucoup les notations asymptotiques pour analyser la complexité des algorithmes ou des problèmes qu'ils résolvent, le plus souvent pour désigner le temps (nombre d'opérations) ou l'espace (quantité de mémoire) utilisés, en fonction de la taille de l'entrée n. En général, on s'intéresse à la complexité dans le pire cas, c'est à dire au temps ou à l'espace maximum que l'algorithme est susceptible d'utiliser.

5.2.1 Exemples

Dans ces exemples, nous nous éloignons un peu des machines de Turing, en supposant un modèle plus proche de nos ordinateurs, pour lesquels on peut accéder à n'importe quel adresse mémoire directement (sans avoir à déplacer une tête de lecture).

Recherche d'élément : On souhaite chercher si une liste de n élements contient une certaine valeur. On peut supposer que chaque élément de la liste a une taille constante $\Theta(1)$.

```
def contains(liste, element):
   for elem in liste:
      if elem == element:
         return True
   return False
```

Dans le pire des cas, on parcourt n éléments et la comparaison à effectuer pour chacun se fait en temps constant. Cet algorithme a donc une complexité en temps de $\Theta(n)$. Souvent, on se contente d'écrire $O(n)$ au lieu de $\Theta(n)$, même si c'est moins précis (p.ex. dans la doc des langages de programmation), car l'important est de savoir que cela *ne dépasse pas* un certain ordre de grandeur.

Recherche dans une liste triée : Même problème que précédemment, mais en supposant que la liste est déjà triée.

```
def contains(liste, element):
   min = 0
   max = len(liste)-1
   while max > min:
      i = round((min + max)/2)
      if element > liste[i]:
         min = i+1
      elif element < liste[i]:
         max = i-1
      else:
         return True
   return False
```

Chaque tour de boucle se fait en temps constant $\Theta(1)$. Combien de tours de boucle peut-on faire dans le pire des cas pour une liste de taille n ? Remarquez qu'à chaque tour de boucle, on élimine la moitié de l'espace de recherche restant. Cet algorithme a donc une complexité en temps de $\Theta(\log n)$, ce qui est *beaucoup* mieux que $\Theta(n)$.

Recherche de doublon : Étant donné une liste de n élements non triés, on souhaite détecter si cette liste contient deux fois le même élément. Voici deux propositions d'algorithmes quasi-identiques.

```
def doublons_v1(liste, element):
   n = len(liste)
   for i1 in range(n):
      for i2 in range(n):
         if i1 != i2:
            if liste[i1] == liste[i2]:
               return True
   return False
```

```
def doublons_v2(liste, element):
   n = len(liste)
   for i1 in range(n-1):
      for i2 in range(i1 + 1, n):
         if liste[i1] == liste[i2]:
            return True
   return False
```

La première version de l'algorithme parcourt tous les élements, et pour chacun, reparcourt tous les élements pour les comparer. Cet algorithme a une complexité en temps

de $\Theta(n^2)$. La deuxième version est un peu meilleure, puisqu'elle compare chaque élément avec les éléments qui sont *à sa droite* dans la liste. (En effet, il est inutile de comparer un élément avec les éléments *à sa gauche*, car cela a déjà été fait.) L'algorithme est donc plus rapide, mais de combien ? Le premier élément sera comparé avec $n-1$ éléments, le second avec $n-2$ éléments, *etc.* et le dernier avec 0 éléments. Cela donne $\Sigma_0^{n-1} = n \cdot (n-1)/2 = (n^2-n)/2$. Supprimons les facteurs constants $(1/2)$ et les termes dominés $(-n)$, on aboutit à nouveau à une complexité de $\Theta(n^2)$. C'est donc toujours quadratique, même si l'algorithme est *un peu* plus rapide.

Cela signifit-il que ce problème lui-même est quadratique ? Non ! Car il pourrait exister de meilleurs algorithmes. Par exemple, on pourrait créer une troisième version `doublons_v3` qui commence par *trier* la liste, puis qui détecte en une seule passe s'il y a des doublons (deux éléments identiques d'affilée). Le tri coûte $O(n \log n)$ et la passe finale $O(n)$, ce qui donne $O(n \log n + n) = O(n \log n)$, donc quasi-linéaire, c'est beaucoup mieux !

Définition 5.1. La complexité d'un problème est la complexité du meilleur algorithme qui le résoud.

Pour les problèmes, on utilise généralement la notation O plutôt que la notation Θ, car souvent, on ne sait pas s'il existe de meilleurs algorithmes que ceux que l'on connaît.

5.3 Classes de complexité

Il existe de nombreuses classes de complexité, qui correspondent à des *ensembles de problèmes* dont la résolution requiert une certaine quantité de ressources (en général, temps ou espace). Elles se définissent en fonction de la taille n de l'entrée à traiter.

Les deux classes principales sont TIME et SPACE. La plupart des autres classes sont des cas particulier de celles-ci. Formellement, ces classes sont définies en termes de langages formels (problèmes de décision), bien que leur définition s'étende naturellement à d'autres types de problèmes (recherche, optimisation, dénombrement, etc.).

- $\mathsf{TIME}(f(n))$: Langages qui peuvent être décidés en temps $O(f(n))$, sans se soucier de l'espace.
- $\mathsf{SPACE}(f(n))$: Langages qui peuvent être décidés en espace $O(f(n))$, sans se soucier du temps.

La définition exacte de ces classes requiert de spécifier le modèle de machine utilisé. Par exemple, machine de Turing à une bande ou plusieurs bandes, machine RAM, etc. (La bonne nouvelle est que pour beaucoup de classes importantes, cela ne change rien.)

Retour aux exemples

Soit FIND le problème de décider si une liste de longueur n contient un certain élément (premier problème ci-dessus). Nous avons vu qu'il existe un algorithme en $O(n)$ pour résoudre ce problème, donc FIND $\in \mathsf{TIME}(n)$. Qu'en est-il de l'espace ? Hormis la taille de l'entrée, qui n'est pas comptabilisée, ce problème n'utilise aucun espace, il est donc résoluble en espace constant $O(1)$. On a donc FIND $\in \mathsf{SPACE}(1)$. Puisque les classes TIME et SPACE décrivent des *ensembles* de problèmes, on peut aussi écrire FIND $\in \mathsf{TIME}(n) \cap \mathsf{SPACE}(1)$.

Pour le problème des doublons, un phénomène intéressant se produit : les versions 1 et 2 montrent que DOUBLONS $\in \mathsf{SPACE}(1)$. En effet, l'algorithme n'utilise aucune mémoire (hormis l'entrée elle-même, qui ne compte pas). La version 3 montre que DOUBLONS $\in \mathsf{TIME}(n \log n)$. On a donc bien DOUBLONS $\in \mathsf{TIME}(n \log n) \cap \mathsf{SPACE}(1)$. Cependant (attention à la subtilité), il est possible qu'aucun algorithme ne puisse satisfaire ces deux grandeurs en même temps, puisque le tri requiert $O(n)$ espace. On parle bien ici d'ensemble de *problèmes* et non d'ensemble d'algorithmes. Bienvenue dans l'espace-temps informatique !

5.4 Classes incontournables

Les classes suivantes sont des cas particuliers très utilisés de TIME et SPACE :

- $\mathsf{LOGSPACE} = \mathsf{SPACE}(\log n)$ Langages décidables en espace logarithmique $O(\log n)$.
- $\mathsf{P} = \mathsf{TIME}(n^{O(1)})$ Langages décidables en temps polynomial $n^{O(1)}$.
- $\mathsf{PSPACE} = \mathsf{SPACE}(n^{O(1)})$ Langages décidables en espace polynomial $n^{O(1)}$.
- $\mathsf{EXP} = \mathsf{TIME}(2^{n^{O(1)}})$ Langages décidables en temps exponentiel $2^{n^{O(1)}}$.

Notez que nous n'avons pas défini de classe particulière pour le temps logarithmique. La raison est que pour la grande majorité des problèmes usuels, un algorithme (ou une machine de Turing) doit au moins lire l'intégralité de son entrée, ce qui prend déjà un temps $O(n)$. Il y a des exceptions, comme chercher un élément dans une liste triée que l'on peut adresser directement [1].

La semaine prochaine, nous montrerons que $\mathsf{LOGSPACE} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$. Nous montrerons aussi que $\mathsf{P} \subsetneq \mathsf{EXP}$ (inclusion *stricte*) et $\mathsf{LOGSPACE} \subsetneq \mathsf{PSPACE}$.

1. Adresser directement = consulter les éléments à une certaine position sans parcourir les précédents. Ce n'est pas possible pour une machine de Turing, mais ça l'est pour une machine RAM (nos ordinateurs).