Complexité algorithmique

Introduction

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

18 janvier 2018

Motivation : efficacité

- Les ressources lors de l'exécution d'un programme sont limitées
  - Temps
  - Mémoire
- Une exigence dans la conception d'algorithmes est qu'ils soient « efficaces »

Question

- Que signifie « efficace » ?
  - Temps ? Mémoire ?
  - En moyenne ? Pire cas ?

- On a besoin d'une notion permettant de caractériser cela

Motivation : sécurité

- Souvent, la sécurité repose sur des problèmes mathématiques complexes

Exemple

- Il doit être « facile » de calculer telle fonction
- Il doit être « difficile » d'effectuer telle action

- Intuition
  - Facile = « rapide » à calculer
  - Difficile = « lent » à calculer
- On a besoin d'une notion permettant de caractériser cela

Contexte

- À plusieurs endroits, les librairies détaillent la complexité des algorithmes
  - Complexité en temps, dans le pire des cas
  - Comportement asymptotique
- À plusieurs endroits, on parle de problèmes « faciles » ou « difficiles »

Exemple

- « Dans le pire des cas, `max` examine un nombre d'éléments linéairement proportionnel à la taille du conteneur »
- Il est difficile de calculer le nombre chromatique d'un graphe

- Cette section est une simple introduction, non mathématique

Compter les opérations élémentaires

- Caractériser la complexité d'un algorithme passe par une étape de comptage « d'opérations élémentaires » exécutées
- Opérations atomiques, « de même temps d'exécution »
  - Pour simplifier, un $+$ prend le même temps qu'un $\star$
- On compte les opérations arithmétiques et logiques, assignations, accès mémoire, etc.

Exemple : algorithme de recherche de maximum

```
int max = -∞;
for(int i = 0; i < v.size(); i++)
  if(v[i] > max)
    max = v[i];
```

Analyse détaillée

- Analysons la complexité de cet algorithme
  1. 2 instruction requises : une pour l'accès, l'autre pour l'assignation
  2. Il faut différencier les itérations
     - Première itération : 2 instructions (une pour `i = 0`, une autre pour `i < n`)
     - À chaque autre itération : 2 instructions (*n* fois)
  3. Le `if` est toujours considéré (pire cas) : 2 instructions (*n* fois)
  4. Le corps du `if` comprend également 2 instructions (*n* fois)
- Au final, cet algorithme exécute dans le pire des cas

$$2 + 2 + 2n + 2n + 2n = 6n + 4$$

opérations élémentaires

Comportement asymptotique

- Compter les instructions comme ci-dessus pour tous les algorithmes est fastidieux
  - A-t-on vraiment besoin de compter à une instruction près ?
  - Pertinence par rapport à la définition d'une instruction élémentaire ?
- On s'intéresse au comportement asymptotique
  - Quand *n* est grand
- On « laisse tomber » le 4
  - C'est une « constante d'initialisation »
  - En `Java`, on a besoin de temps pour initialiser la VM
  - Pourquoi considérer cette constante ?
- On « laisse tomber » le 6
  - En `Java`, on a un contrôle de borne
- Complexité dans le pire des cas : $\mathcal{O}(n)$

La vraie définition

Définition

- $\mathcal{O}(f(n)) = \{g(n) \mid \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geqslant n_0, 0 \leqslant g(n) \leqslant cf(n)\}$.

- Cette définition n'est pas l'objet de ce cours

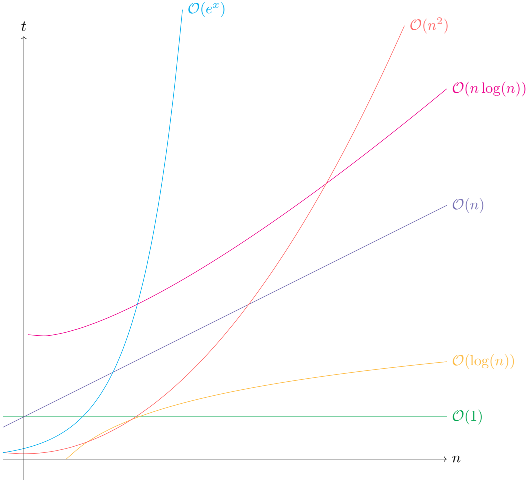
Ce qui importe

- « $f(n)$ se comporte asymptotiquement comme $g(n)$ »

- Cela prend « autant de temps » de rechercher le maximum d'une liste que d'en imprimer le contenu

Classes de complexité et exemples

- $\mathcal{O}(1)$: accès dans un tableau
- $\mathcal{O}(\log(n))$: recherche dichotomique
- $\mathcal{O}(n)$: maximum
- $\mathcal{O}(n\log(n))$: trier efficacement
- $\mathcal{O}(n^2)$: tri inefficace
- $\mathcal{O}(e^n)$: Fibonacci naïf
- « Facile » : borné par un polynôme



$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n\log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(e^n)$$