Complexité algorithmique

pierre.coucheney@uvsq.fr

MIN17102 : complément d'algorithmique et de complexité, Septembre 2025

Organisation du cours

- 6 séances de cours et 5 TDs
- Evaluation :
  - une note d'examen (90%)
  - 2 mini contrôles en cours vendredi 19 et mardi 23 septembre (10%)
- ressources sur e-campus
- emploi du temps :

  `https://edt.uvsq.fr/cal?vt=month&dt=2025-09-01&et=module&eid=-1742255826:1477018062:16:5436263:5&fid0=MIN17102`

Ressources bibliographiques

- **Introduction à l'algorithmique** de Cormen, Leiserson et Rivest.
- **Programmation Efficace** de Durr et Vie.
- Sites avec des défis de programmation, par exemple :

  `https://adventofcode.com/`

Contenus du cours

Rappels de notions étudiées en Licence.

- complexité d’un algorithme
- classes de complexité
- structures de données :
  - tableaux, listes (trié, non trié)
  - piles, files
  - arbres
- conception d’algorithmes :
  - méthodes de type diviser pour régner
  - méthodes énumératives
  - programmation dynamique

Compétences attendues

1. appliquer un algorithme à la main sur un exemple
2. montrer la validité d'un algorithme, estimer sa complexité (analyse d'algorithme)
3. écrire un algorithme qui résout un problème donné (conception d'algorithme)
4. écrire un algorithme ayant des bonnes performances

Un programme est l'implémentation d'un algorithme ce qui permet de le tester.

Compétences attendues

1. appliquer un algorithme à la main sur un exemple
2. montrer la validité d'un algorithme, estimer sa complexité (analyse d'algorithme)
3. écrire un algorithme qui résout un problème donné (conception d'algorithme)
4. écrire un algorithme ayant des bonnes performances

Un programme est l'implémentation d'un algorithme ce qui permet de le tester.

Langage de description d’un algorithme

Plusieurs niveaux de description :

- par un **exemple** (un humain pour un humain à l’oral)
- par un **programme** (un humain pour une machine)
- par du **pseudo-code** (un humain pour un humain à l’écrit)
- par un **code binaire** (une machine pour une machine)

Pseudo-langage du cours

- l'ensemble des valeurs (≈ type) des entrées et des sorties de l'algorithme est précisé au début
- idem pour les variables locales
- un algorithme peut retourner plusieurs valeurs
- à moins que ce ne soit précisé dans la partie réservée aux variables d'entrées, le passage des paramètres qui ne sont pas des tableaux se fait par valeur : **les variables ne sont pas modifiées quand elles sont passées à une fonction**.
- le passage des tableaux se fait par référence.
- notations :
  - affectation $x \leftarrow 3$
  - test d'égalité si $x = 3$
  - échange de valeurs entre deux variables $x \leftrightarrow y$

Pseudo-langage du cours (suite)

- types de base : entiers, flottants, booléens
- types complexes obtenus en définissant
  - des structures

    ```
    struct toto{
    a: entier
    x: flottant
    }
    ```
  - des tableaux

    ```
    t[i]: element d'indice i du tableau
    t.nmax: nombre maximum d'éléments dans le tableau
    t.n: taille effective du tableau ( < t.nmax )
    ```
  - des références : $\uparrow x$ désigne la référence à la variable x.

Un exemple simple

Algorithme : $f(n : \text{entier})$:

Entrées : n entier

Sorties :

tant que $n \neq 0$
    si $n\%2 = 0$
        $n \leftarrow n/2$
    sinon
        $n \leftarrow n+1$

Cet algorithme termine-t'il?

Coût d'un algorithme

- Exécuter un algorithme utilise des ressources : temps, espace mémoire, énergie, bande passante d'un réseau...
- le temps d'exécution croît avec la quantité d'opérations élémentaires à effectuer
  - affectations
  - comparaison de valeurs
  - opérations arithmétique
- pour avoir de l'information sur l'ensemble des entrées, il faut des outils théoriques

Exemple de coût (i)

Algorithme : $somme1(n : \text{entier}) : \text{entier}$

Entrées : n entier positif

Sorties : entier

Variables locales : res : entier

```
res ← 0
pour i de 1 à n
    res ← res + i
retourner res
```

Nombre d'additions ?

$2n$ additions

Exemple de coût (i)

Algorithme : *somme*1(n : entier) : entier

Entrées : n entier positif

Sorties : entier

Variables locales : *res* : entier

```
res ← 0
pour i de 1 à n
    res ← res + i
retourner res
```

Nombre d'additions ?

$2n$ additions

Exemple de coût (ii)

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Algorithme : *somme*2(n : entier) : entier

Entrées : n entier positif

Sorties : entier

Variables locales : *res* : entier

$res \leftarrow n+1$

$res \leftarrow res \times n$

$res \leftarrow res/2$

retourner *res*

Nombre d'opérations arithmétiques ?

1 addition, 1 multiplication, 1 division

Exemple de coût (ii)

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Algorithme : $somme2(n : \text{entier}) : \text{entier}$

Entrées : n entier positif

Sorties : entier

Variables locales : res : entier

$res \leftarrow n+1$

$res \leftarrow res \times n$

$res \leftarrow res/2$

retourner res

Nombre d'opérations arithmétiques ?

1 addition, 1 multiplication, 1 division

Notion de complexité d'un algorithme

Objectif : mesure approximative du coût d'un algorithme qui

- ne dépend pas de son implémentation,
- permet la comparaison entre différents algorithmes pour un même problème.

Différentes **mesures** de la performance d'un algorithme :

- calculer le coût précis d'un algorithme pour une entrée ou un ensemble d'entrées données
- pire cas : calculer le coût de l'entrée qui nécessite le plus d'opérations pour une taille d'entrées données
- en moyenne : calculer la moyenne des coûts sur l'ensemble des entrées d'une taille donnée

Approximation : on cherche un ordre de grandeur quand la taille des entrées est grande (complexité asymptotique).

Notion de complexité d’un algorithme

Objectif : mesure approximative du coût d’un algorithme qui

- ne dépend pas de son implémentation,
- permet la comparaison entre différents algorithmes pour un même problème.

Différentes **mesures** de la performance d’un algorithme :

- calculer le coût précis d’un algorithme pour une entrée ou un ensemble d’entrées données
- pire cas : calculer le coût de l’entrée qui nécessite le plus d’opérations pour une taille d’entrées données
- en moyenne : calculer la moyenne des coûts sur l’ensemble des entrées d’une taille donnée

Approximation : on cherche un ordre de grandeur quand la taille des entrées est grande (complexité asymptotique).

Notion de complexité d'un algorithme

Objectif : mesure approximative du coût d'un algorithme qui

- ne dépend pas de son implémentation,
- permet la comparaison entre différents algorithmes pour un même problème.

Différentes **mesures** de la performance d'un algorithme :

- calculer le coût précis d'un algorithme pour une entrée ou un ensemble d'entrées données
- pire cas : calculer le coût de l'entrée qui nécessite le plus d'opérations pour une taille d'entrées données
- en moyenne : calculer la moyenne des coûts sur l'ensemble des entrées d'une taille donnée

Approximation : on cherche un ordre de grandeur quand la taille des entrées est grande (complexité asymptotique).

Taille des entrées

Pour donner du sens à la complexité, il faut déterminer ce qu'est la taille d'une entrée

- **Règle 1 :** toujours préciser ce qu'on considère comme la taille de l'entrée
- **Règle 2 :** c'est le nombre d'objets dans l'entrée
  - ex. un tableau de n éléments est de taille n
- parfois plusieurs paramètres
  - ex. calculer a^b
- pour un entier, on considère sa valeur ou sa taille ?

Ordre de grandeur : définition informelle

- on note la complexité $O(f(n))$ pour dire qu'elle est de l'ordre de $f(n)$ quand n est grand
- on retire les constantes multiplicatives et additives
- on utilise l'échelle des fonctions usuelles :

$$n!, \quad 2^n, \quad n^3, \quad n^2, \quad n\log(n), \quad n, \quad \log(n), \quad 1$$

Exemples :

- $3n^3 + \sqrt{2n} + \log(n) \in O(n^3)$
- $2^n \log(n) + 4n! \in O(n!)$
- $4\log(n^2) + \log(2n) \in O(\log(n))$

Ordre de grandeur : définition informelle

- on note la complexité $O(f(n))$ pour dire qu'elle est de l'ordre de $f(n)$ quand n est grand
- on retire les constantes multiplicatives et additives
- on utilise l'échelle des fonctions usuelles :

$$n!, \quad 2^n, \quad n^3, \quad n^2, \quad n\log(n), \quad n, \quad \log(n), \quad 1$$

Exemples :

- $3n^3 + \sqrt{2n} + \log(n) \in O(n^3)$
- $2^n \log(n) + 4n! \in O(n!)$
- $4\log(n^2) + \log(2n) \in O(\log(n))$

Notation grand O

Définition

Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction. $O(f)$ est l'**ensemble de fonctions** suivant :

$$\{g \text{ tel que } \exists C, n_0 \text{ tels que } \forall n \geq n_0, g(n) \leq Cf(n)\}$$

Attention, la notation grand O est une classe de fonctions à laquelle appartient la fonction de complexité. On peut donner une complexité sans O et on peut utiliser des O ailleurs qu'en complexité.

Quelques chiffres

Effet de la multiplication de la puissance d'une machine par 10, 100 et 1000 sur la taille maximale N des problèmes que peuvent traiter des algorithmes de complexité donnée en un temps donné :

complexité	$\times 10$	$\times 100$	$\times 1000$
$O(\log(n))$	N^{10}	N^{100}	N^{1000}
$O(n)$	$10N$	$100N$	$1000N$
$O(n\log(n))$	$< 10N$	$< 100N$	$< 1000N$
$O(n^2)$	$\approx 3N$	$10N$	$\approx 32N$
$O(n^3)$	$\approx 2N$	$\approx 5N$	$\approx 10N$
$O(2^n)$	$\approx N+3$	$\approx N+7$	$\approx N+10$

Complexité d'un problème (informel)

Complexité du meilleur algorithme qui le résout (borne inférieure).

- classe P : ensemble des problèmes qui sont résolus par un algorithme de complexité **polynomiale** ;
- classe EXP : ensemble des problèmes qui sont résolus par un algorithme de complexité **exponentielle** ;
- classe NP : ensemble des problèmes pour lesquels on peut **vérifier** une solution en temps polynomial

$$P \subseteq NP \subseteq EXP$$

Complexité d'un problème (informel)

Complexité du meilleur algorithme qui le résout (borne inférieure).

- classe P : ensemble des problèmes qui sont résolus par un algorithme de complexité **polynomiale** ;
- classe EXP : ensemble des problèmes qui sont résolus par un algorithme de complexité **exponentielle** ;
- classe NP : ensemble des problèmes pour lesquels on peut **vérifier** une solution en temps polynomial

$$P \subseteq NP \subseteq EXP$$

Exemple : calcul de la valeur minimale d'un tableau d'entiers

Algorithme : min(t : tableau d'entiers) : entier

Entrées : t tableau d'entiers non vide

Sorties : entier

$m \leftarrow t[0]$
pour *i de 1 à* $t.n - 1$
    $m \leftarrow \min(m, t[i])$
retourner m

Calcul de la valeur minimale d'un tableau d'entiers : version récursive

---

Algorithme : min_rec(t : tableau d'entiers, i : entier) : entier

---

Entrées : t tableau d'entiers non vide

Sorties : entier

Résultat : retourne l'entier minimum jusqu'à l'indice i du tableau

```
si i = 0
    retourner t[i]
retourner min(t[i], min_rec(t, i − 1))
```

Calcul de la valeur minimale d'un tableau d'entiers : version récursive 2

Algorithme : min_rec2(t : tableau d'entiers, g, d : entiers) : entier

Entrées : t tableau d'entiers non vide, $0 \leq g \leq d < t.n$

Sorties : entier

Résultat : retourne l'entier minimum entre les indices g et d du tableau

si $g = d$

    retourner *t[g]*

$m \leftarrow (g+d)/2$

retourner $\min(min\_rec2(g, m), min\_rec2(m+1, d))$

Master theorème (version faible)

Soit une fonction de complexité définie de la manière suivante :

$$c(n) = ac(n/b) + f(n)$$

où f est une fonction entière positive.

- si $f(n) \in O(n^{\log_b a - \epsilon})$ avec $\epsilon > 0$, alors

$$c(n) \in O(n^{\log_b a})$$

- sinon si $f(n) \in O(n^{\log_b a})$, alors

$$c(n) \in O(n^{\log_b a} \log n)$$

- sinon si $af(n/b) \leq kf(n)$ avec $k < 1$ alors

$$c(n) \in O(f(n))$$