

ALGORITHMIQUE II

NOTION DE COMPLEXITE

- ❑ Comment choisir entre différents algorithmes pour résoudre un même problème?
- ❑ Plusieurs critères de choix :
 - Exactitude
 - Simplicité
 - **Efficacité** (but de ce chapitre)

- L'évaluation de la complexité d'un algorithme se fait par l'analyse relative à deux ressources de l'ordinateur:
 - Le temps de calcul
 - L'espace mémoire, utilisé par un programme, pour transformer les données du problème en un ensemble de résultats.

L'analyse de la complexité consiste à mesurer ces deux grandeurs pour choisir l'algorithme le mieux adapté pour résoudre un problème.(le plus rapide, le moins gourmand en place mémoire)

On ne s'intéresse, ici, qu'à la **complexité temporelle** c.à d. qu'au temps de calcul
(par opposition à la complexité spatiale)

- Le temps d'exécution est difficile à prévoir, il peut être affecté par plusieurs facteurs:
 - la machine
 - la traduction (interprétation, compilation)
 - l'environnement (partagé ou non)
 - L'habileté du programmeur
 - Structures de données utilisées

- Pour pallier à ces problèmes, une notion de complexité plus simple, mais efficace, a été définie pour un **modèle de machine** . Elle consiste à compter les **instructions de base** exécutées par l'algorithme. Elle est exprimée en fonction de la taille du problème à résoudre.
- Une instruction de base (ou élémentaire) est soit: une affectation, un test, une addition, une multiplication, modulo, ou partie entière.

- La complexité dépend de la taille des données de l'algorithme.
- Exemples :
 - Recherche d'une valeur dans un tableau
→ taille (= nombre d'éléments) du tableau
 - Produit de deux matrices
→ dimension des matrices
 - Recherche d'un mot dans un texte
→ longueur du mot et celle du texte

On note généralement:

n la taille de données, $T(n)$ le temps (ou le coût) de l'algorithme.

$T(n)$ est une fonction de $\mathbb{N}$ dans $\mathbb{R}^+$

- Dans certains cas, la complexité ne dépend pas seulement de la taille de la donnée du problème mais aussi de la donnée elle-même.
 - ➔ Toutes les données de même taille ne génèrent pas nécessairement le même temps d'exécution.
 - (Ex. la recherche d'une valeur dans un tableau dépend de la position de cette valeur dans le tableau)

- Une donnée particulière d'un algorithme est appelée **instance** du problème.
- On distingue trois mesures de complexité:

1. Complexité dans le meilleur cas

$$T_{\text{Min}}(n) = \min \{T(d) ; d \text{ une donnée de taille } n \}$$

2. Complexité dans le pire cas

$$T_{\text{Max}}(n) = \max \{T(d) ; d \text{ une donnée de taille } n \}$$

3. dans la cas moyen

$$T_{\text{MOY}}(n) = \sum_{d \text{ de taille } n} p(d).T(d)$$

$p(d)$: probabilité d'avoir la donnée d

$$T_{\text{MIN}}(n) \leq T_{\text{MOY}}(n) \leq T_{\text{MAX}}(n)$$

- Exemple. Complexité de la recherche d'un élément x dans un tableau A à n valeurs.

```

i := 1
tantque (i ≤ n) et (A[i] ≠ x) faire
    i := i+1;
fsi;
si i > n alors retourner(faux);
sinon retourner(vrai);

```

On note par:

a : le cout d'une affectation

t : cout d'un test

d : cout d'une addition

- **Cas le plus favorable.** x est le premier élément du tableau:

$$T_{\min}(n) = 1a + 3t$$

- **Pire des cas.** x n'est pas dans le tableau:

$$T_{\max}(n) = (n+1)a + (2n+2)t + nd$$

- **En moyenne :**

- **Complexité en moyenne:**

- On note par :

D_i ($1 \leq i \leq n$) : ensemble de données (de taille n) où x est présent à la i^{eme} position

D_{n+1} : ensemble de données où x n'est pas présent

- On suppose que la probabilité de présence de x dans une donnée est q . De plus, dans le cas où x est présent, on suppose que sa probabilité de présence dans l'une des positions est de $1/n$

On a :

$$p(D_i) = q/n, \quad T(D_i) = i a + (2i+1)t + (i-1)d \quad ; \quad (1 \leq i \leq n)$$

$$p(D_{n+1}) = 1 - q, \quad T(D_{n+1}) = T_{\max} = (n+1)a + (2n+2)t + nd$$

$$p(D_i) = q/n, \quad T(D_i) = i a + (2i+1)t + (i-1)d \quad ; \quad (1 \leq i \leq n)$$

$$p(D_{n+1}) = 1 - q, \quad T(D_{n+1}) = T_{\max}(n) = (n+1)a + (2n+2)t + nd$$

$$T_{\text{MOY}}(n) = \sum_{d \text{ de taille } n} p(d) \cdot T(d)$$

$p(d)$: probabilité d'avoir la donnée d

$$T_{\text{moy}}(n) = \sum_{i=1}^n p(D_i) T(D_i) + p(D_{n+1}) T(D_{n+1})$$

$$T_{\text{moy}}(n) = \frac{q}{n} \sum_{i=1}^n (ia + (2i+1)t + (i-1)d) + (1-q)[(n+1)a + (2n+2)t + nd]$$

$$T_{\text{moy}}(n) = q \left[\frac{n+1}{2} a + (n+2)t + \frac{n-1}{2} d \right] + (1-q)[(n+1)a + (2n+2)t + nd]$$

Cas où $q=1$, i.e. x est toujours présent: $T_{\text{moy}}(n) = \frac{1}{2} [(n+1)a + (2n+4)t + (n-1)d]$

On remarque que la complexité de cet algo. est de la forme: $\alpha n + \beta$, α et β son des constates

$$= \frac{n}{2} (a + 2t + d) + \frac{1}{2} (a + 4t - d)$$

Complexité asymptotique

Comportement de $T(n)$

12

- Pour mesurer la complexité d'un algorithme, il ne s'agit pas de faire un décompte exact du nombre d'opérations $T(n)$, mais plutôt de donner un ordre de grandeur de ce nombre pour n assez grand.

- **Notation de Landau**

- ❖ **"grand O"**

$$T(n) = O(f(n)) \text{ ssi}$$

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 \quad T(n) \leq c.f(n)$$

- ❖ **"grand oméga"**

$$T(n) = \Omega(f(n)) \text{ ssi}$$

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 \quad T(n) \geq c.f(n)$$

- ❖ **"grand théta"**

$$T(n) = \Theta(f(n)) \text{ ssi}$$

$$\exists c_1 > 0 \exists c_2 > 0 \exists n_0 > 0 \forall n > n_0 \quad c_1 f(n) \leq T(n) \leq c_2 f(n)$$

Remarque: les constantes c , c_1 , c_2 et n_0 sont indépendantes de n

- D'une manière générale, $f : \mathbb{R} \rightarrow \mathbb{R}$
 - ▣ $f(x) = O(g(x))$ s'il existe un voisinage V de x_0 et une constante $k > 0$ tels que $|f(x)| \leq k|g(x)|$, ($x \in V$)
 - Si la fonction g ne s'annule pas, il revient au même de dire que le rapport $\left| \frac{f(x)}{g(x)} \right|$ est borné pour $x \in V$.
 - Exemple: au voisinage de 0, on a:

$$x^2 = O(x), \ln(1+x) = O(x)$$
 - Au voisinage de l'infini (comme pour le cas de la complexité), il existe $\alpha > 0$ ($V =]\alpha, +\infty[$) et $k > 0$ t.q

$$|f(x)| \leq k|g(x)|, \forall x > \alpha$$
 et on dit que f est dominée asymptotiquement par g .
 (Au voisinage de $+\infty$, on a: $x = O(x^2)$, $\ln x = O(x)$)

□ Remarques

1. $f = O(g) \Leftrightarrow$ le quotient $\left| \frac{f(x)}{g(x)} \right|$ est bornée au voisinage de l'infini.
2. $\lim_{x \rightarrow \infty} \left| \frac{f(x)}{g(x)} \right| = a \rightarrow f = O(g)$
3. si $a = 1$, on écrit $f \sim g$ et on a $f = \Theta(g)$.
4. $f = \Theta(g)$ ne signifie pas que le quotient $f(x)/g(x)$ tend vers une limite (1 en particulier).

Exemple. $f(x) = x(2 + \sin x)$. On a $x \leq f(x) \leq 3x$
 $\forall x > 0$, donc $f(x) = \Theta(x)$. En revanche, le quotient $f(x)/x$ ne tend vers aucune limite lorsque $x \rightarrow +\infty$

□ Abus de notation:

On a par définition:

$$O(f) = \{g: \mathbb{N} \rightarrow \mathbb{R} / \exists c > 0 \exists n_0 > 0 \quad g(n) \leq c.f(n), \forall n > n_0\}$$

$$\Omega(f) = \{g: \mathbb{N} \rightarrow \mathbb{R} / \exists c > 0 \exists n_0 > 0 \quad g(n) \geq c.f(n), \forall n > n_0\}$$

$$\Theta(f) = O(f) \cap \Omega(f)$$

La difficulté, dans la familiarisation avec ces concepts, provient de la convention de notation (de Landau) qui veut que l'on écrive :

$g = O(f)$, ou encore $g(n) = O(f(n))$ au lieu de $g \in O(f)$

De manière analogue, on écrit $O(f) = O(g)$ lorsque $O(f) \subset O(g)$

(il en est de même pour les notations Ω ou Θ)

□ Exemple. Soit la fonction $T(n) = \frac{1}{2} n^2 + 3n$

- $T(n) = \Omega(n)$ ($n_0 = 1, c = \frac{1}{2}$)
- $T(n) = \Theta(n^2)$ ($n_0 = 1, c_1 = \frac{1}{2}, c_2 = 4$)
- $T(n) = O(n^3)$ ($n_0 = 1, c = 4$)
- $T(n) \neq O(n)$

Supposons que $T(n) = O(n)$

$$\exists c > 0, \exists n_0 > 0 : \frac{1}{2} n^2 + 3n \leq cn \quad \forall n \geq n_0$$

donc $c \geq \frac{1}{2} n$, contradiction. (la constante c ne peut dépendre de n)

□ Remarques

1. Si $T(n)$ est un polynôme de degré k alors $T(n) = \Theta(n^k)$
2. $O(n^k) \subset O(n^{k+1}) \quad \forall k \geq 0$ (idem pour Θ)
3. $\Theta(f(n)) \subset O(f(n))$ pour toute fonction f positive
4. $O(1)$ utilisé pour signifier « en temps constant »

□ Remarques pratiques:

- Le cas le plus défavorable est souvent utilisé pour analyser un algorithme.
- La notation O donne une borne supérieure de la complexité pour toutes les données de même taille(suffisamment grande). Elle est utilisée pour évaluer un algorithme dans le cas le plus défavorable.
- $T(n) \leq cf(n)$ signifie que le nombre d'opérations ne peut dépasser $cf(n)$ itérations, pour n'importe quelle donnée de longueur n .
- Pour évaluer la complexité d'un algorithme, on cherche un majorant du nombre d'opérations les plus dominantes.
- Dans les notations asymptotiques, on ignore les constantes.

L'algorithme de recherche dans un tableau à n éléments, cité précédemment, est en $O(n)$

□ Ordre de grandeur courant

- $O(1)$: complexité constante
- $O(\log(n))$: complexité logarithmique
- $O(n)$: complexité linéaire
- $O(n^2)$: complexité quadratique
- $O(n^3)$: complexité cubique
- $O(2^n)$: complexité exponentielle

- Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme.

On suppose que l'ordinateur utilisé peut effectuer 10^6 opérations à la seconde (une opération est de l'ordre de la μs)

$n \backslash T(n)$	$\log n$	n	$n \log n$	n^2	2^n
10	3 μs	10 μs	30 μs	100 μs	1000 μs
100	7 μs	100 μs	700 μs	1/100 s	10^{14} siècles
1000	10 μs	1000 μs	1/100 μs	1 s	astrono mique
10000	13 μs	1/100 μs	1/7 s	1,7 mn	astrono mique
100000	17 μs	1/10 s	2 s	2,8 h	astrono mique

-

➤ Un algorithme est dit **polynomial** si sa complexité est en $O(n^p)$.

➤ Un algorithme est dit praticable s'il est polynomial ($p \leq 3$).

Les algorithmes polynomiaux où $p > 3$ sont considérés comme très lents
(un algorithme polynomial de l'ordre de n^5 prendrait environ 30 ans pour $n=1000$)

➤ Un algorithme est dit exponentiel si sa complexité est supérieure à tout polynôme.

➤ Deux grandes classes de la complexité :

- $\mathbb{P}$ classe des algorithmes polynomiaux

- $\mathbb{NP}$ classe des algorithmes « Non déterministe polynomiale »

On a :

$$(O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(e^n) \subset O(n!))$$

□ Propriétés

En utilisant la notation de Landau (pour les fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$), on a :

1. $f + O(g) = \{f + h \mid h \in O(g)\}$
 $(h = f + O(g) \Leftrightarrow h - f \in O(g)).$
 $f O(g) = \{f h \mid h \in O(g)\}$
2. $f = O(f)$
3. $f = O(g), g = O(h) \Rightarrow f = O(h)$
4. $c O(f) = O(c f) = O(f) \quad (c > 0)$
5. $O(f) + O(g) = O(f + g) = O(\max(f, g))$
6. $O(f) + O(f) = O(f)$
7. $O(f) O(g) = O(fg)$
8. $f = O(g) \Leftrightarrow g = \Omega(f)$

□ Calcul de la complexité: règles pratiques

1. la complexité d'une suite d'instructions est la somme des complexités de chacune d'elles.
2. Les opérations élémentaires telle que l'affectation, test, accès à un tableau, opérations logiques et arithmétiques, lecture ou écriture d'une variable simple ... etc, sont en $O(1)$.
3. $T(\text{si } C \text{ alors } A1 \text{ sinon } A2) = \max(T(C), \max(T(A1), T(A2)))$

4. $T(\text{pour } i:=e_1 \text{ à } e_2 \text{ faire } A_i \text{ fpour}) = \sum_{i=e_1}^{e_2} T(A_i)$
 (si A_i ne contient pas de boucle dépendante de i
 et si A_i est de complexité $O(m)$ alors la complexité
 de cette boucle « pour » est $O((e_2 - e_1 + 1)m).$)

5. La difficulté, pour la boucle tantque, est de
 déterminer le nombre d'itération Nb_iter (ou
 donner une borne supérieure de ce nombre)

$$T(\text{tantque } C \text{ faire } A \text{ ftantque}) = O(Nb_iter \times (T(C) + T(A)))$$

□ Exemples

1. Calcul de la somme $1+2+\dots+n$

$S:=0; \text{ // } O(1)$

Pour $i:=1$ à n faire

$s:=s+i; \text{ // } O(1)$

fpour;

$$\sum_{i=1}^n O(1)$$

$O(n)$

$$O(1) + O(n) = O(n)$$

$$\boxed{T(n) = O(n)}$$

2. Calcul de : $T[i] = \sum_{j=1}^i j$ pour $i = 1, 2, \dots, n$

pour $i := 1$ à n faire

$s := 0$; // $O(1)$

pour $j := 1$ à i faire

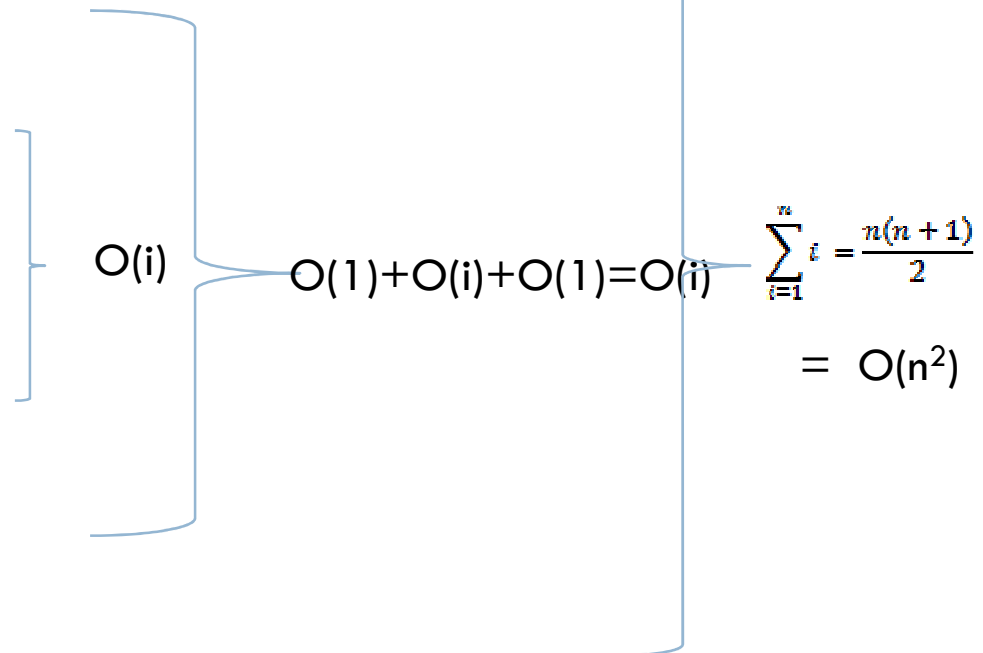
$s := s + j$; // $O(1)$

fpour;

$T[i] := s$; // $O(1)$

fpour;

$$T(n) = O(n^2)$$



$$O(i) \left\{ \begin{array}{l} O(1) + O(i) + O(1) = O(i) \\ \sum_{i=1}^n i = \frac{n(n+1)}{2} \\ = O(n^2) \end{array} \right.$$

3. Analyse de l'algo. Suivant :

Donnée n ; ($n > 0$)

Résultat cpt ;

début

$\text{cpt} := 1$;

tantque $n \geq 2$ faire

$n := n \text{ div } 2$;

$\text{cpt} := \text{cpt} + 1$;

ftantque

fin

Que calcule cet algo?

Quelle est sa complexité?

- $\text{cpt} = 1 + \text{le nombre d'itérations}$
- Le nombre d'itérations = nombre de division de n par 2.
- Soit p ce nombre.
 - si n est une puissance de 2, i.e. $n = 2^p$ alors $p = \log_2(n)$.
 - p vérifie: $2^p \leq n < 2^{p+1}$
 - $p \leq \log_2(n) < p+1 \Rightarrow p = E(\log_2(n))$
- $\text{cpt} = 1 + E(\log_2(n))$, cette expression de cpt correspond au nombre de bits nécessaires pour représenter l'entier n .
- $T(n) = O(\log(n))$