Rappel : Complexité

Pour plus de détails consulter le cours de L1
http://dept-info.labri.fr/ENSEIGNEMENT/algoprog/support.html

Complexité

Définition 1.1 L'efficacité d'un algorithme est mesurée par son coût (complexité) en temps et en mémoire.

La complexité d'un algorithme se mesure en calculant :

- le nombre d'opérations élémentaires,
- la taille de la mémoire nécessaire,

pour traiter une donnée de taille n.

Complexité

On considèrera dans ce cours que la complexité des instructions élémentaires les plus courantes sur un ordinateur a un temps d'exécution constant égal à 1.

Centre d'intérêt pour l'algorithmique c'est l'ordre de grandeur au voisinage de l'infini de la fonction qui exprime le nombre d'instructions ou la taille de la mémoire.

Question : L'infini de quoi ?

Complexité

Définition 1.2 On définit les trois complexités suivantes :

- Complexité dans le pire des cas :
  $C^{>}_A(n)=\max\{C_A(d), d \text{ donnée de taille } n\}$
- Complexité dans le meilleur des cas :
  $C^{<}_A(n)=\min\{C_A(d), d \text{ donnée de taille } n\}$
- Complexité en moyenne :

$$\overline{C}_A(n) = \sum_{d \text{ instance de } A} \Pr(d) C_A(d)$$

où Pr(d) est la probabilité d'avoir en entrée une instance d parmi toutes les données de taille n.

Complexité

Cas Particulier : Problème NP-complet

C'est un problème pour lequel on ne connait pas d'algorithme correct efficace : réalisable en temps et en mémoire.

L'ensemble des problèmes NP-complets ont les propriétés suivantes :

- Si on trouve un algorithme efficace pour un problème NP complet alors il existe des algorithmes efficaces pour tous,
- Personne n'a jamais trouvé un algorithme efficace pour un problème NP-complet,
- Personne n'a jamais prouvé qu'il ne peut pas exister d'algorithme efficace pour un problème NP-complet particulier.

Le plus célèbre est le problème du voyageur de commerce.

Temps d'exécution : Comment ?

- 2ème idée : Utiliser une méthode théorique qui :
- Utilise l'algorithme et est indépendante de l'implémentation et du langage de programmation.
- Définit le temps d'exécution comme une fonction dépendante de la taille des données d'entrée.
- Prends en compte toutes les entrées possibles.
- Est indépendante des environnements matériel et logiciel.

Complexité temporelle : Définition

- Le temps de calcul (ou complexité) d'un algorithme est la fonction qui à un entier n associe le nombre maximal d'instructions élémentaires que l'algorithme effectue, lorsqu'on travaille sur des objets de taille *n*.
- La fonction compte les instructions élémentaires au lieu de mesurer le temps.
- On s'intéresse en particulier à la complexité dans le pire des cas.
- Exemples d'opérations élémentaires :
  - additionner, soustraire, multiplier ou diviser deux nombres,
  - tester si une valeur est égale à une autre valeur,
  - affecter une valeur à une variable.

Complexité temporelle : Définition

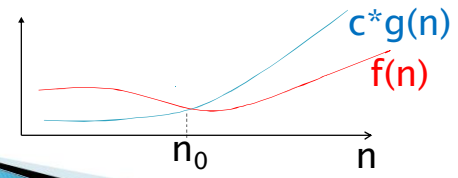
- Le décompte des instructions peut être fastidieux si on compte aussi les accès aux structures de données, les E/S, appels de fonctions, etc
- Même en se limitant à une seule opération on peut obtenir une expression qu'il faudra approximer pour obtenir une solution
- Même si les opérations élémentaires ont des temps d'exécution constants sur une machine donnée, ils varient d'une machine à l'autre.
- Donc pour simplifier le calcul sans perdre sa pertinence on négligera les constantes.
- En pratique, on se contente d'un ordre de grandeur asymptotique.

Ordres de grandeur

Limite asymptotique supérieure – O (Grand O)

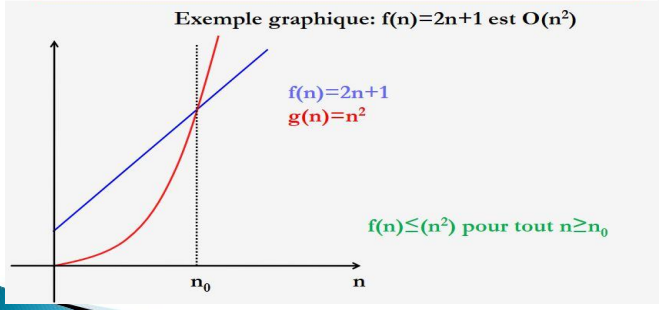
Soient 2 fonctions $f(n)$ et $g(n)$ de $\mathbb{N}$ dans $\mathbb{R}$.
On dit que $f(n)$ est $O(g(n))$ ou $f(n) = O(g(n))$
si et seulement si :

$$\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^* \,/\, f(n) \leq c * g(n) \; \forall \, n \geq n_0$$



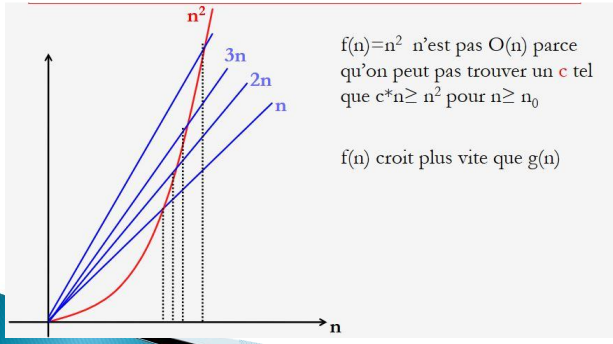
Ordres de grandeur

Limite asymptotique supérieure O (Grand O)



Ordres de grandeur

Limite asymptotique supérieure O (Grand O)



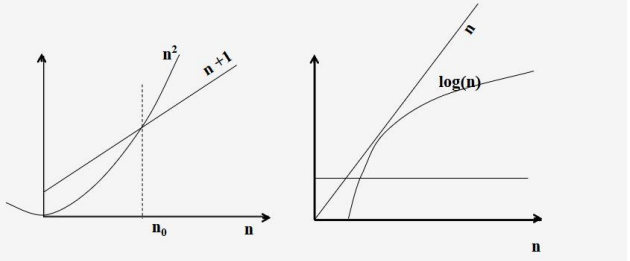
f(n)=n² n'est pas O(n) parce qu'on peut pas trouver un c tel que c*n≥ n² pour n≥ n0

f(n) croit plus vite que g(n)

Ordres de grandeur

Limite asymptotique supérieure O (Grand O)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)\ldots$

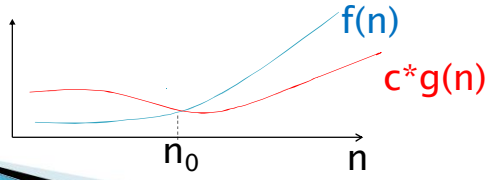


Ordres de grandeur

Limite asymptotique inférieure- Ω (Grand Oméga)

Soient 2 fonctions $f(n)$et $g(n)$de $\mathbb{N}$ dans $\mathbb{R}$.
On dit que $f(n)$est $\Omega(g(n))$ ou $f(n) = \Omega(g(n))$
si et seulement si :

$$\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^* / f(n) \geq c * g(n) \; \forall \; n \geq n_0$$



Ordres de grandeur

Limite asymptotique - Θ (Grand thêta)

Soient 2 fonctions $f(n)$et $g(n)$de $\mathbb{N}$ dans $\mathbb{R}$.
On dit que $f(n)$est $\Theta(g(n))$ ou $f(n)$
$= \Theta(g(n))$
si et seulement si :

$f(n)$ est $O(g(n))$ et $f(n)$ est $\Omega(g(n))$

Ordres de grandeur : comportements asymptotiques

Intuitivement :

- Grand O : f(n) est O(g(n)) si f(n) est plus petite ou égale à g(n) quand n est grand.
- Grand Oméga : f(n) est Ω (g(n)) si f(n) est plus grande ou égale à g(n) quand n est grand.
- Grand thêta : f(n) est ? (g(n)) si f(n) est à peu près égale g(n) quand n est grand.

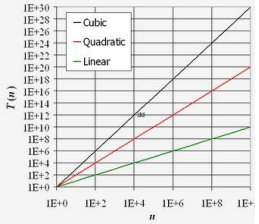
Complexité : ordre de grandeur

n =	2	16	256	1024
log (log n)	0	2	3	3.32
log n	1	4	8	10
n	2	16	256	1024
n log n	2	64	2048	10 240
n^2	4	256	65 536	1 048 576
n^3	8	4 096	16 777 216	$1.07 * 10^9$
2^n	4	65 536	$1.15 * 10^{77}$	$1.79 * 10^{308}$

Complexité : vocabulaire

Terminologie: Types de complexité

Constante:	O(1)
Logarithmique:	O(log(n))
Linéaire:	O(n)
Quasi-linéaire:	O(n.log(n))
Quadratique:	O(n^2)
Cubique:	O(n^3)
Polynomiale:	O(n^k), k >0
Quasi-polynomiale	O($n^{\log(n)}$)
Exponentielle:	O(a^n), n > 1
Factorielle	O(n!)



Pour mémoire

Propriétés des logarithmes:

$\log_b(x*y) = \log_b x + \log_b y$

$\log_b (x/y) = \log_b x - \log_b y$

$\log_b x^a = a*\log_b x$

$\log_b a = \log_x a / \log_x b$