

# Analyse et complexité des algorithmes

## Algorithmique

### → Conception de méthodes pour la résolution de problèmes

- On a une description des données d'un problème (entrées, spécification en mots du résultat cherché)
- On décrit des méthodes pour résoudre le problème → **ALGORITHMES**
- On montre que les méthodes répondent au problème

### → Complexité des méthodes

- Efficacité: temps de calcul, espace mémoire utilisé

### → Réalisation

- Organisation des données → **CHOIX D'UNE STRUCTURE**
- Implémentation

## Algorithme

**Historique:** Le terme **algorithme** vient d'**Al Khowarizmi**, mathématicien arabe du IXe siècle

- Le livre d'Al Khowarizmi constitue la base de la notation décimale moderne.
- Au départ, le mot "**algorisme**" désignait les règles nécessaires pour effectuer des calculs arithmétiques en utilisant la notation décimale.
- Le terme **algorithme** apparaît au XVIIIe siècle.

## Algorithme

### Définitions:

#### 1) Le petit Larousse

Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations.

#### 2) Encyclopedia Universalis

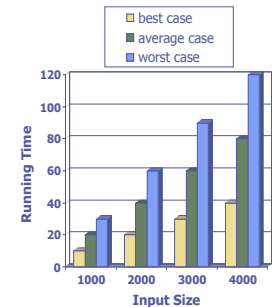
Spécification d'un schéma de calcul sous forme d'une suite fini d'opérations élémentaires obéissant à un enchaînement déterminé.

# Algorithme

- Propriétés:**
- **Les entrées:** un algorithme prend des valeurs d'entrées à partir d'ensembles définis
  - **La sortie:** constitue la solution du problème de départ
  - ➔ **La finitude:** l'algorithme doit produire la sortie souhaitée en un **nombre fini** (mais peut-être très grand) d'**étapes**, quelque soit l'entrée
  - ➔ **L'efficacité:** Chaque étape de l'algorithme doit pouvoir s'exécuter dans un temps fini
  - ➔ **La généralité:** l'algorithme s'applique à tous les problèmes d'une forme désirée

# Complexité en temps des algorithmes (§2)

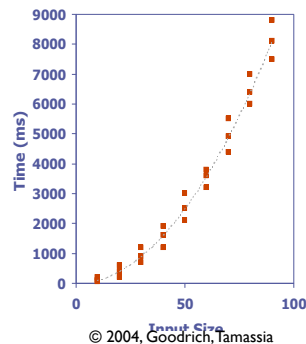
- La plupart des algorithmes transforment des entrées en une sortie
- La complexité en temps d'un algorithme est habituellement fonction de la taille des entrées
- La complexité en moyenne est souvent difficile à obtenir
- On étudie plutôt la complexité dans le pire des cas:
  - plus facile à analyser
  - crucial dans beaucoup d'applications: jeux, finance...



© 2004, Goodrich, Tamassia

# Méthode 1: Études expérimentales

- Implémenter l'algorithme en Java (ou autre)
- Faire fonctionner le programme avec des entrées de taille et de composition différentes
- Utiliser une méthode comme `System.currentTimeMillis()` pour obtenir une mesure réelle du temps d'exécution
- Dessiner le graphique des résultats



© 2004, Goodrich, Tamassia

# Limitation de cette méthode

- On doit implémenter l'algorithme
  - On veut connaître la complexité en temps d'un algorithme avant de l'implémenter, question de sauver du temps et de l'\$\$\$\$
- Les résultats trouvés ne sont pas représentatifs de toutes les entrées
- Pour comparer 2 algorithmes différents pour le même problème, on doit utiliser le même environnement (hardware, software)

## Méthode 2: Analyse Théorique

- Se fait à partir du pseudo-code de l'algorithme et non de l'implémentation
- Caractérise le temps d'exécution comme une fonction de  $n$ , la taille de l'entrée
- Prend en considération toutes les entrées
- Indépendant de l'environnement utilisé (hardware, software)

## Complexité en temps

$T(\text{algo}, \mathbf{d})$  = temps d'exécution de l'algorithme **algo** appliqué aux données **d**

**Complexité au pire:**

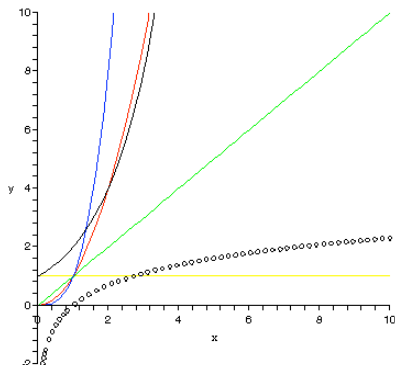
$$T_{\text{MAX}}(\text{algo}, n) = \max \{T(\text{algo}, \mathbf{d}), \mathbf{d} \text{ de taille } n\}$$

**Complexité au mieux:**

$$T_{\text{MIN}}(\text{algo}, n) = \min \{T(\text{algo}, \mathbf{d}), \mathbf{d} \text{ de taille } n\}$$

## Plusieurs fonctions importantes (§2.3)

- fonction constante,  $f(x)=1$
- fonction linéaire,  $f(x)=x$
- fonction logarithmique,  $f(x)=\log(x)$
- fonction quadratique,  $f(x)=x^2$
- fonction cubique,  $f(x)=x^3$
- fonction exponentielle,  $f(x)=2^x$



## Comparaison des fonctions

$\log(n)$	$\sqrt{n}$	$n$	$n \log(n)$	$n^2$
3	3	10	33	100
7	10	100	664	10 000
10	32	1000	9966	1 000 000
13	100	10 000	132 877	100 000 000
17	316	100 000	1 660 964	10 000 000 000
20	1000	1 000 000	19 931 569	1 000 000 000 000

Valeurs des fonctions communément rencontrées, tableau modifié de © 2004, Sedgwick

## Temps de résolution très grands problèmes

Opérations par secondes	Taille du problème: 1 million			Taille du problème 1 milliard		
	n	n log(n)	n <sup>2</sup>	n	n log(n)	n <sup>2</sup>
10 <sup>6</sup>	Secondes	Secondes	Semaines	Heures	Heures	Jamais
10 <sup>9</sup>	Immédiat	Immédiat	Heures	Secondes	Secondes	Décennies
10 <sup>12</sup>	Immédiat	Immédiat	Secondes	Immédiat	Immédiat	Semaines

© 2004, Sedgewick

n = 1 million  
 n log n = ~ 20 millions  
 n<sup>2</sup> = 1000 milliards

## Opérations élémentaires

- Opérations de base effectuées par l'algorithme
  - Évaluer une expression
  - Assigner une valeur à une variable
  - Appeler une méthode
  - ..... etc...
- Indépendantes du langage de programmation choisi
- On assume que leur temps d'exécution est **constant**

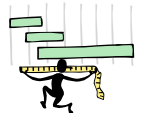
## Compter les opérations élémentaires

En inspectant le pseudocode d'un algorithme, on peut déterminer le nombre maximum d'opérations élémentaires exécuté par un algorithme, comme une fonction de la taille de l'entrée

<b>Algorithm</b> <i>arrayMax</i> ( <i>A</i> , <i>n</i> )	# operations
<i>currentMax</i> ← <i>A</i> [0]	2
<b>for</b> <i>i</i> ← 1 to <i>n</i> - 1 <b>do</b>	2 <i>n</i>
<b>if</b> <i>A</i> [ <i>i</i> ] > <i>currentMax</i> <b>then</b>	2( <i>n</i> - 1)
<i>currentMax</i> ← <i>A</i> [ <i>i</i> ]	2( <i>n</i> - 1)
{ increment counter <i>i</i> }	2( <i>n</i> - 1)
<b>return</b> <i>currentMax</i>	1
	Total 8 <i>n</i> - 2

© 2004, Goodrich, Tamassia

## Estimer le temps d'exécution



- L'algorithme **arrayMax** exécute  $8n-2$  opérations élémentaires dans le pire des cas. Soit
  - a** = temps d'exécution le plus rapide d'une opération élémentaire
  - b** = temps d'exécution le plus lent d'une opération élémentaire
- Soit **T(n)** la complexité dans le pire des cas de **arrayMax**. Alors
 
$$a(8n-2) \leq T(n) \leq b(8n-2)$$
- Le temps d'exécution est donc borné par deux fonctions

## Taux de croissance du temps d'exécution

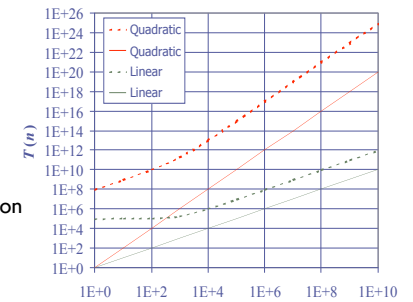
- Changer d'environnement (hardware, software)
  - Affecte  $T(n)$  d'un facteur constant
  - N'affecte pas le taux de croissance de  $T(n)$
- Le taux de croissance linéaire de  $T(n)$  est une propriété intrinsèque de l'algorithme.

## Facteurs constants

- Le taux de croissance d'une fonction n'est pas affecté par
  - les facteurs constants
  - les termes d'ordre plus petit

### Exemples

- $100n + 10^5$  est une fonction linéaire
- $10^5 n^2 + 10^8 n$  est une fonction quadratique



© 2004, Goodrich, Tamassia

## Facteurs constants

**FIGURE 2.1** The growth rate of all terms of function  $f(n) = n^2 + 100n + \log_{10}n + 1,000$ .

$n$	$f(n)$		$n^2$		$100n$		$\log_{10}n$		1,000	
	Value	%	Value	%	Value	%	Value	%	Value	%
1	1,101	0.1	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	4.76	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	47.6	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	90.8	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	99.0	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	99.9	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

© 2005, Drozdek