Introduction

Problème: Multiplier 2 nombres entiers à n chiffres

Algo 1: Algorithme classique → au tableau

Algo 2: Algorithme dit “à la russe”

on écrit les deux nombres à multiplier côte à côte

tant que le nombre de gauche n’est pas égale à 1 faire

on divise par deux le nombre de gauche et on ne garde que la partie entière (ignorer les fractions)

on double le nombre de la colonne de droite en l’additionnant à lui-même

fin tant que

éliminer toutes les lignes dont le nombre de gauche est pair

additionner les nombres restant de la colonne de droite

Introduction

Problème: Multiplier 2 nombres entiers à n chiffres

Algo 3: Algorithme diviser-pour-régner

On a une table des valeurs pour la multiplication entre chiffres. On divise les nombres à multiplier en nombres de plus petite taille pour pouvoir éventuellement consulter la table

Soit n le nombre de chiffre des entiers à multiplier

Si n = 1 on consulte la table

Si n > 1 on sépare nos deux nombres x et y en 4 nombres s, t, u et v de taille n/2 chacun et on reprend l'algorithme sur ces nombres de taille plus petite

Introduction

Étant donné un problème ⟶ plusieurs algorithmes

Comment trouver le “meilleur”?

Analyse de complexité

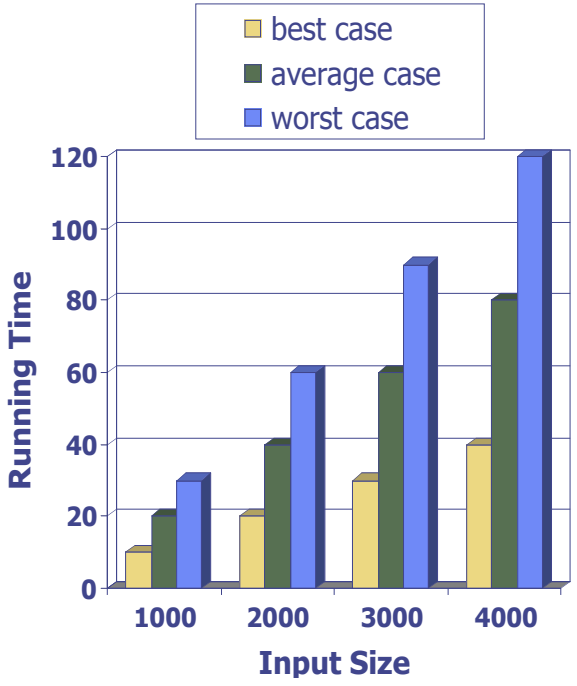
Algorithme optimal: On ne peut faire mieux!

Tout autre algorithme pour le même problème prendra au moins le même temps que l’algorithme optimal.

Pour pouvoir déterminer l’optimalité, on doit pouvoir comparer les algorithmes.

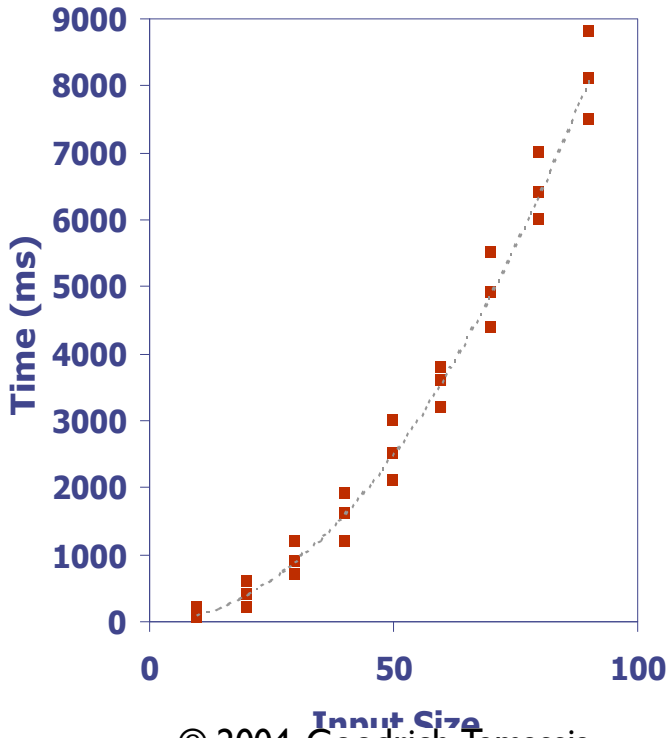
Complexité en temps des algorithmes

- 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...



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 pour obtenir une mesure réelle du temps d'exécution
- Dessiner le graphique des résultats



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(algo,d) = temps d'exécution de l'algorithme **algo** appliqué aux données **d**

Complexité au pire:

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

Complexité au mieux:

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

Complexité en moyenne:

$$T_{MOY}(\textbf{algo},\textbf{n}) = \sum_{\textbf{d taille n}} p(d)\textbf{T(algo,d)}$$

où p(d) = probabilité d'avoir l'entrée **d**

LUO(CAML, MAPLE): Automatic asymptotic average-case analysis of algorithms
Bruno Salvy et Paul Zimmermann
http://algo.inria.fr/libraries/

Opérations élémentaires

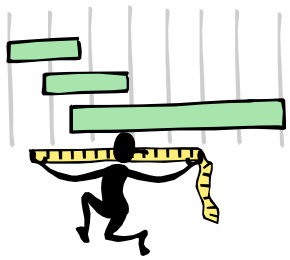
- Opérations de base effectuées par l'algorithme
  - Évaluer une expression
  - Assigner une valeur à une variable
  - Appeler une méthode
  - ..... etc...
- Indépendantes du langage de programmation choisi
- On assume que leur temps d'exécution est **constant**

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

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	2
for *i* ← 1 **to** *n* – 1 **do**	2 (n-1)
if *A*[*i*] > *currentMax* **then**	2 (n-1)
currentMax ← *A*[*i*]	2 (n-1)
return *currentMax*	1
	total = 6n - 3

Estimer le temps d'exécution



- L'algorithme **arrayMax** exécute 6n-3 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

  $$\mathbf{a}(6n-3) \leq T(n) \leq \mathbf{b}(6n-3)$$

- Le temps d'exécution est donc borné par deux fonctions

Taux de croissance du temps d'exécution

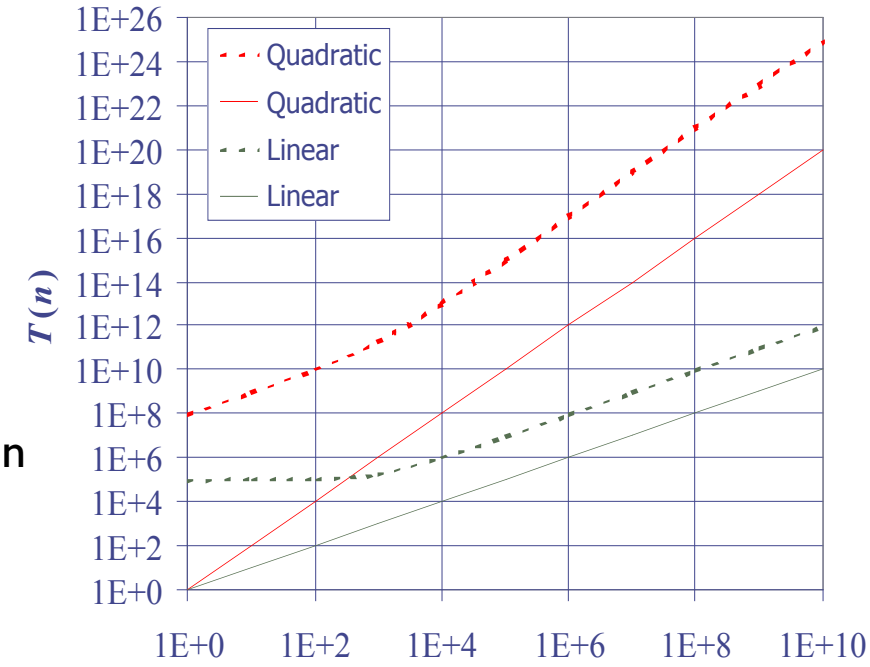
- Changer d'environnement (hardware, software)
  - Affecte **T(n)** d'un facteur constant
  - N'affecte pas le taux de croissance de **T(n)**

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



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	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

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, Sedgewick

Temps de résolution très grands problèmes

Opérations par secondes	Taille du problème: 1 million			Taille du problème 1 milliard		
	n	$n \log(n)$	n^2	n	$n \log(n)$	n^2
10^6	Secondes	Secondes	Semaines	Heures	Heures	Jamais
10^9	Immédiat	Immédiat	Heures	Secondes	Secondes	Décennies
10^{12}	Immédiat	Immédiat	Secondes	Immédiat	Immédiat	Semaines