UFR d'Informatique

Algorithmique M1

2024–2025

F. Laroussinie

plan

- Diviser pour régner
- Programmation dynamique
- Gloutons

- Analyse amortie

Il faut programmer les algorithmes !

objectifs

Apprendre à manipuler les algorithmes.

- concevoir
- analyser
- chercher dans la littérature
- comprendre
- modifier

Fonctionnement du cours

Cours et TD

et : «exprime une addition, une liaison, un rapprochement...»
(le Petit Robert)

Cours: Amphi 4C, mardi 8h30 → 10h30
TD1: 1003 (SG), F. L., lundi 10h45→12h45; [MIDS +...]
TD2: 1003 (SG), G. Aubian, lundi 16h15→18h15; [GENIAL +...]
TD3: 1005(SG), R. Mantaci, mercredi 8h30→10h30;
[...]

francoisl@irif.fr

https://www.irif.fr/~francoisl/m1algo.html

Constitution des groupes

Cours: Amphi 4C, mardi 8h30 → 10h30
TD1: 1003 (SG), F. L., lundi 10h45→12h45; [MIDS +...]
TD2: 1003 (SG), R. Mantaci, lundi 16h15→18h15; [GENIAL +...]
TD3: 1005(SG), G. Aubian, mercredi 8h30→10h30; [...]

Pour les étudiant-e-s des parcours DATA, LP, MPRI et IMPAIRS: indiquer vos préférences pour les **3** groupes…

→ feuille à remplir ici et maintenant !

Les TD

Cours: Amphi 4C, mardi 8h30 → 10h30
TD1: 1003 (SG), F. L., lundi 10h45→12h45; [MIDS +...]
TD2: 1003 (SG), R. Mantaci, lundi 16h15→18h15; [GENIAL +...]
TD3: 1005(SG), G. Aubian, mercredi 8h30→10h30; [...]

Des séances de TD +
des séances de TP: avec **vos** machines.

Contrôle des connaissances

Un examen + du contrôle continu

CC = (1 TD noté +1 partiel)

Partiel : mardi 22 octobre (à confirmer) 8h30-10h30

Ce qu'il ne faut pas faire

Ecrire algorythme

Ecrire algorithmie

Croire que cela vient du grec *algos* qui signifie douleur

Aller en cours et pas en TD

Aller en TD et pas en cours

N'aller ni en cours ni en TD

(Arriver en retard, partir en avance, entrer et sortir pendant le cours.)

C’est parti !

Algorithmes *corrects* : y en-a-t-il ?

Algorithmes *efficaces* : notions de complexité (pire cas, en moyenne, amortie…)

Algorithmes *optimaux*: peut-on faire mieux ?

Quand on propose un algorithme, on doit prouver sa correction et donner sa complexité.

Un mot sur la complexité des algorithmes…

Attention aux vielles légendes estudiantine…

Non, la complexité n'a pas été inventée pour torturer les étudiant-e-s !

Non, la complexité n'est pas un truc théorique déconnecté de la réalité de l'informatique…

Mesurer la complexité…

d'un algorithme:

- efficace ou pas ?
- comparer deux algos.

d'un problème:

- difficile ou facile ?

Les deux notions combinées permettent de savoir si un algorithme est **optimal** ou non.

L'efficacité d'un algorithme

- On ne veut pas mesurer le temps nécessaire en minutes ou en microsecondes.

  → On veut une notion robuste: indépendante d'un ordinateur, d'un compilateur, d'un langage de programmation, *etc.*

- On va évaluer le nombre d'"opérations élémentaires" dans le pire cas (ou en moyenne,…) en fonction de la taille des données.
  (on se contente d'un ordre de grandeur).

On s'intéresse parfois aussi à la quantité de mémoire nécessaire pour l'exécution d'un algorithme.

In fine, le temps de calcul dépend…

- de l'algorithme
- de l'ordinateur
- du langage

On se concentre d'abord sur l'algo…

Evaluer l'efficacité d'un algorithme

Pire cas, en moyenne, amortie…

$C_A(x)$: nombre d'*opérations élémentaires* nécessaires pour l'exécution de l'algorithme A sur la donnée x.

Complexité (coût) dans le **pire cas**:

$$C_A(n) = \max_{x.|x|=n} C_A(x)$$

Complexité en **moyenne**:

$$C_A^{moy}(n) = \sum_{x.|x|=n} p(x).C_A(x)$$

distribution de probabilités sur les données de taille n

Complexité **amortie**:

Evaluation du coût cumulé de *n* opérations (dans le pire cas).

→ complexité en temps.
[on peut aussi considérer sa complexité en espace mémoire.]

Complexité des algorithmes

Obj: avoir un ***ordre de grandeur*** du nombre d'opérations…

Notations: O( ), Ω( ) et Θ( ):

$O(g(n)) = \{f(n) \mid \exists c>0, n_0 \geq 0 \text{ tq } 0 \leq f(n) \leq c.g(n) \ \forall\ n \geq n_0\}$

→ ensemble des fonctions *majorées* par c.g(n)

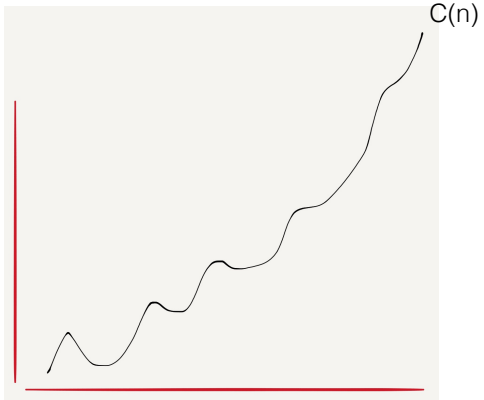
$\Omega(g(n)) = \{f(n) \mid \exists c>0, n_0 \geq 0 \text{ tq } 0 \leq c.g(n) \leq f(n) \ \forall\ n \geq n_0\}$

→ ensemble des fonctions *minorées* par c.g(n)

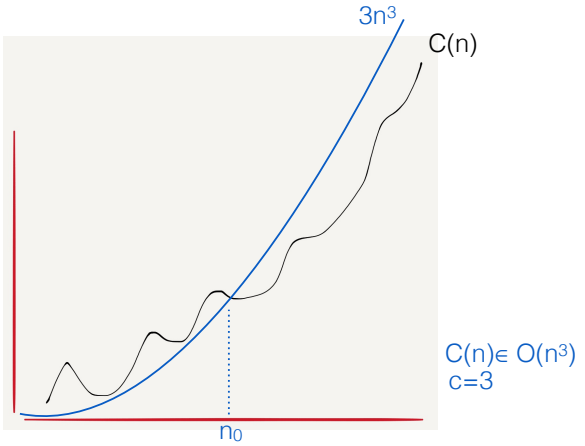
$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2>0, n_0 \geq 0 \text{ tq } 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \ \forall\ n \geq n_0\}$

→ ensemble des fonctions *encadrées* par c_1.g(n) et c_2.g(n)

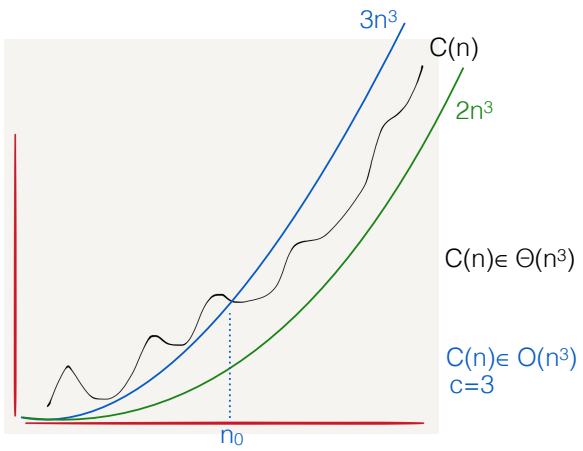
Notations: O,Ω,Θ



Notations: O,Ω,Θ



Notations: O,Ω,Θ



Les algorithmes efficaces… et les autres.

On s'intéresse à des grandes familles de fonctions:

- les algorithmes sous-linéaires.
  Par ex. en O(log n)
- les algorithmes linéaires: O(n)
  ou "quasi-linéaires" comme O(n.log n)
- les algorithmes polynomiaux $O(n^k)$

- les algorithmes exponentiels: $O(2^n)$
- …

n^2 vs 2^n

n^2 → $n \cdot n$

2^n → $2 \cdot 2 \cdot \ldots \cdot 2$ (n fois)

Les algorithmes efficaces… et les autres.

fct $\backslash n$	10	50	100	300	1000
$5n$	50	250	500	1500	5000
$n \cdot \log_2(n)$	33	282	665	2469	9966
n^2	100	2500	10000	90000	$10^6(7c)$
n^3	1000	125000	$10^6(7c)$	$27 \cdot 10^6(8c)$	$10^9(10c)$
2^n	1024	$\ldots(16c)$	$\ldots(31c)$	$\ldots(91c)$	$\ldots(302c)$
$n!$	$3.6 \cdot 10^6(7c)$	$\ldots(65c)$	$\ldots(161c)$	$\ldots(623c)$	…!!!
n^n	$10 \cdot 10^9(11c)$	$\ldots(85c)$	$\ldots(201c)$	$\ldots(744c)$	…!!!!

notation: (Xc) -> "s'écrit avec X chiffres en base 10"

NB: le nombre de nano-secondes depuis le big-bang comprend **27** chiffres...

(voir « Algorithmics, the spirit of computing », D. Harel)

Les algorithmes efficaces… et les autres.

Avec un ordinateur exécutant 10^9 instructions par seconde…

Fonc.\ n	20	40	60	100	300
n^2	1/2500 milliseconde	1/625 milliseconde	1/278 milliseconde	1/100 milliseconde	1/11 milliseconde
n^5	1/300 seconde	1/10 seconde	78/100 seconde	10 secondes	40,5 minutes
2^n	1/1000 seconde	18,3 minutes	36,5 année	400.10^9 siècles	(72c) siècles
n^n	$3,3.10^9$ années	(46c) siècles	(89c) siècles	(182c) siècles	(725c) siècles

On situe le big-bang à environ $13,8.10^9$ années !

(voir « Algorithmics, the spirit of computing », D. Harel)

Les algorithmes efficaces… et les autres.

Supposons qu'aujourd'hui, on puisse résoudre un problème de taille K en une heure...

Si l'algorithme a une complexité **n**, alors...
- un ordinateur 100 fois plus rapide, résoudra des pb de taille **100xK**.
- un ordinateur 1000 fois plus rapide, résoudra des pb de taille **1000xK**.

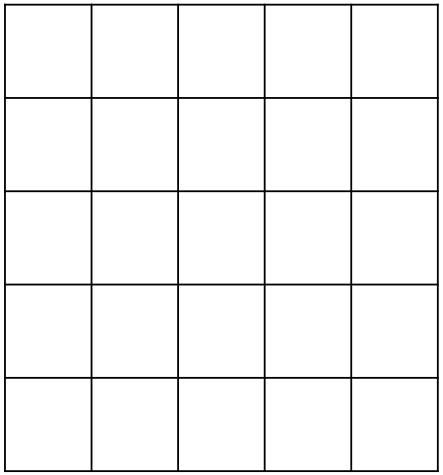
Si l'algorithme a une complexité n^2, alors...
- un ordinateur 100 fois plus rapide, résoudra des pb de taille **10xK**.
- un ordinateur 1000 fois plus rapide, résoudra des pb de taille **32xK**.

Si l'algorithme a une complexité 2^n, alors...
- un ordinateur 100 fois plus rapide, résoudra des pb de taille **7+K**.
- un ordinateur 1000 fois plus rapide, résoudra des pb de taille **10+K**.

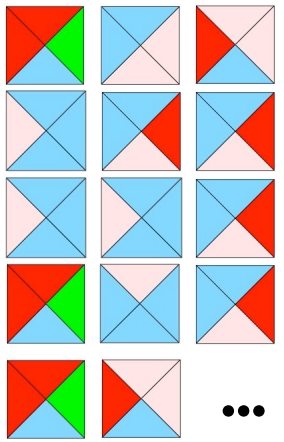
La fonction n! n'est pas terrible non plus…

Un puzzle...

Une grille 5x5

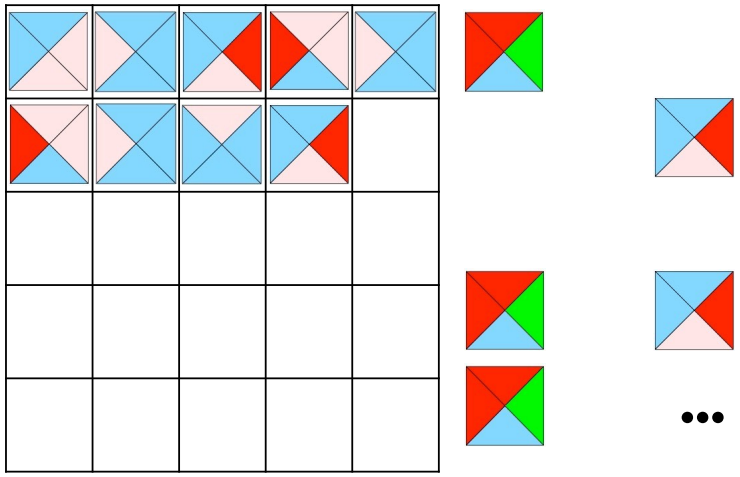


25 tuiles



Un puzzle...

Une grille 5x5 25 tuiles



Puzzle

On essaie toutes les possibilités ?

25 tuiles à placer sur 25 cases:

- 25 possibilités pour la première,
- 24 pour la seconde,
- 23 pour la troisième,
- ...

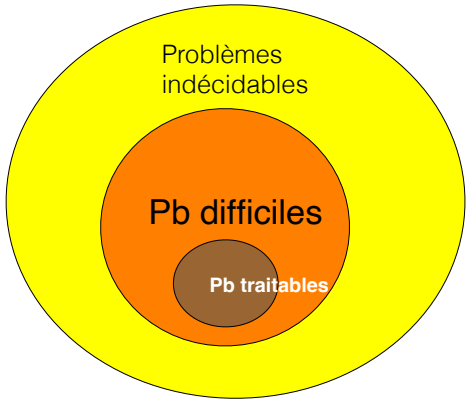
25x24x23x...x2 = **25! possibilités**

Avec un ordinateur qui évalue 1 milliard de cas par seconde, il faudrait ...

490 millions d'années !

(25! s'écrit avec 26 chiffres...)

Vue globale



Pb traitables = algo. polynomial

Attention:
Il y a algorithmes efficaces… et algorithmes efficaces !

n^3, n^2, $n.\log n$, ou n ce n'est pas « pareil » !

Un problème:
Etant donné un tableau de nombres (positifs ou négatifs) de taille n, **calculer** la somme maximale des éléments d'un sous-tableau.

suite de cases consécutives

Communications of the ACM, Sept. 1984, Vol. 27, Nb 9.

programming pearls

by Jon Bentley

ALGORITHM DESIGN TECHNIQUES

The September 1983 column described the "everyday" impact that algorithm design can have on programmers: an algorithmic view of a problem gives insights that may make a program simpler to understand and to write. In this column we'll study a contribution of the field that is less frequent but more impressive: sophisticated algorithmic methods sometimes lead to dramatic performance improvements.

This column is built around one small problem, with an emphasis on the algorithms that solve it and the techniques used to design them. Some of the algorithms are a little complicated, but the complication is justified; while the first algorithm we'll study takes 39 days to solve a problem of size 10,000, the final algorithm solves the same problem in just a third of a second.

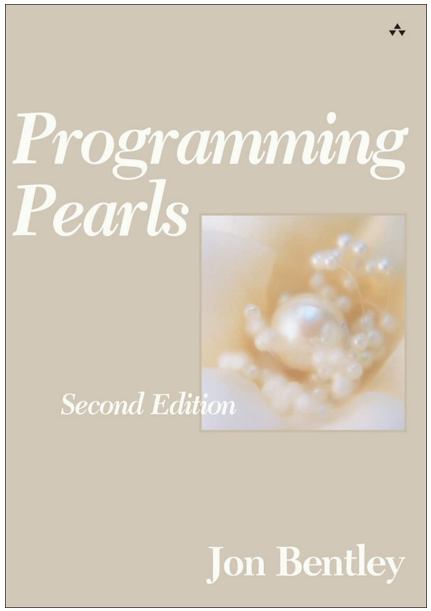
The Problem and a Simple Program

The problem arose in one-dimensional pattern recognition; I'll describe its history later. The input is a vector X of N real numbers; the output is the maximum sum found in any *contiguous* subvector of the input. For instance, if the input vector is

if N is 1,000 and 39 days if N is 10,000 (we'll get to timing details later).

Those times are anecdotal; we get a different kind of feeling for the algorithm's efficiency using "big-oh" notation.[1] The statements in the outermost loop are executed exactly N times, and those in the middle loop are executed at most N times in each execution of the outer loop. Multiplying those two factors of N shows that the four lines contained in the middle loop are executed $O(N^2)$ times. The loop in those four lines is never executed more than N times, so its cost is $O(N)$. Multiplying the cost per inner loop times its number of executions shows that the cost of the entire program is proportional to N cubed, so we'll refer to this as a cubic algorithm.

Those simple steps illustrate the technique of "big-oh" analysis of run time and many of its strengths and weaknesses. Its primary weakness is that we still don't really know the amount of time the program will take for any particular input; we just know that the number of steps it executes is $O(N^3)$. Two strong points of the method often compensate for that weakness. "Big-oh" analyses are usually easy to perform (as above), and the



1999

Voir J. Bentley «Pearls of programming»

Exemple

suite de cases consécutives

Un problème:
Etant donné un tableau de nombres (positifs ou négatifs) de taille n, calculer la somme maximale des éléments d'un sous-tableau.

8	-10	10	4	-19	40	0	5	-9	14	2	3	78	7	-24	6	9	-18	7	2

somme max:= 140

somme de tous les éléments = 115

Les cas simples...

8	0	10	4	19	40	0	5	9	14	2	3	78	7	2	6	9	8	7	2

solution ? la somme totale... 223

-8	-10	-10	-4	-19	-40	-10	-5	-9	-14	-2	-3	-78	-7	-24	-6	-9	-18	-7	-2

solution ? 0 ! (i.e. le sous-tableau vide)

Le problème

Donnée: un tableau T[0..n-1]

Résultat: Max { sum[i,j] | $0 \leq i,j \leq n-1$ }

$$sum[i,j] = \sum_{k \in [i;j]} T[k]:$$

intervalle: i,i+1,...,j

Algo 1

Complexité en $O(n^3)$

Enumérer tous les sous-tableaux...

```
def algo1(T[0...n-1]) :
    maxsofar = 0
    for i = 0,...,n-1 :
        for j = i...,n-1 :
            sum = 0
            for k = i,...,j :
                sum += T[k]
            maxsofar = max(maxsofar,sum)
    return maxsofar
```

Max { sum[i,j] | $0 \leq i,j \leq n-1$ }

$$sum[i,j] = \sum_{k \in [i;j]} T[k]$$

Algo 2

Complexité en $O(n^2)$

```
def algo2(T[0...n-1]) :
    maxsofar = 0
    for i = 0,...,n-1 :
        sum = 0
        for j = i,...,n-1 :
            sum += T[j]
            maxsofar = max(maxsofar,sum)
    return maxsofar
```

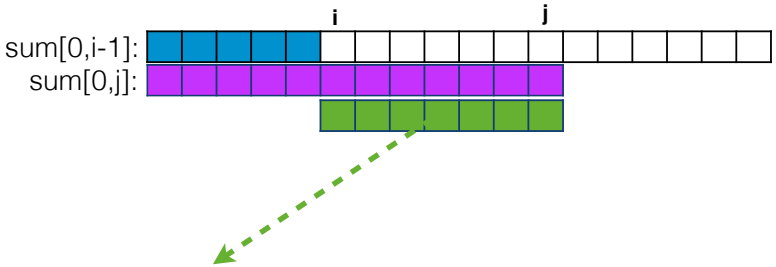
itération i ∈ {0,..,n-1}

i

calcul de tous les sous-tableaux commençant en i.

Algo 3

Autre idée:



sum[i,j] = sum[0,j] - sum[0,i-1]

Algo 3

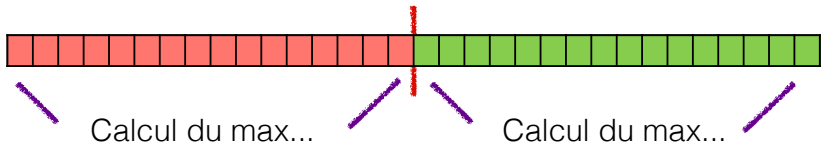
ici sum[i] = «sum[0,i]»

```
def algo3(T[0..n-1]) :
   sum[i] = 0      ∀ i = -1,0,...,n
   for i = 0..n-1 :
      sum[i] = sum[i-1]+T[i]
   maxsofar = 0
   for i = 0..n-1 :
      for j = i..n-1 :
         sumij = sum[j] - sum[i-1]
         maxsofar = max(maxsofar,sumij)
   return maxsofar
```

Complexité en $O(n^2)$

Un diviser pour régner ?

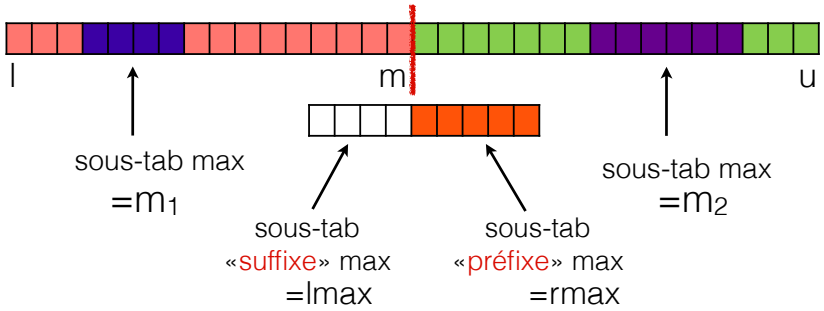
Algo 4



+ «traitement»

Résultat pour T

Algo 4



résultat = max(m_1, m_2, lmax+rmax)

Calcul de lmax : tester tous les sous-tab finissant en m.
Calcul de rmax : tester tous les sous-tab commençant en m+1.

Algo 4

```
def algo4(T,l,u) :
  if (l>u) : return 0
  if (l==u) : return max(0,T[l])

  m = (l+u)/2
  lmax = sum = 0
  for i = m ... l :      // l ≤ m  [calcul de lmax]
     sum += T[i]
     lmax = max(lmax,sum)
  rmax = sum = 0
  for i = m+1 ... u :   // m ≤ u  [calcul de rmax]
     sum += T[i]
     rmax = max(rmax,sum)

  return max(lmax+rmax, algo4(T,l,m),algo4(T,m+1,u))
```

Complexité en $O(n \cdot \log n)$

Algo 5

Idée: on parcourt le tableau de gauche à droite en gardant:

- le sous-tableau max rencontré dans la partie gauche parcourue, et
- le sous-tableau «suffixe» max se terminant à la position courante.

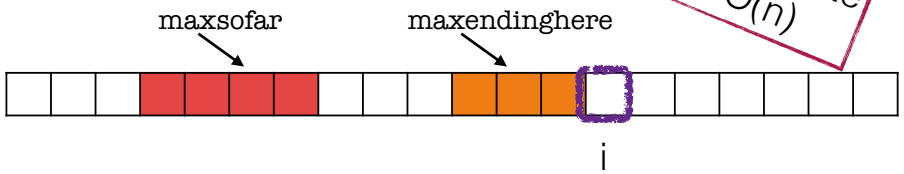
Mise à jour:

1) comparer  et 0 ➔ 

2) comparer  et 

Algo 5

Complexité en $O(n)$




```
def algo5(T[0..n-1]) :
  maxsofar = 0
  maxendinghere = 0

  for i = 0..n-1 :
     maxendinghere = max(maxendinghere + T[i],0)
     maxsofar = max(maxsofar,maxendinghere)

  return maxsofar
```

Bilan

Algo 1	Algo 2 Algo 3	Algo 4	Algo 5
$O(n^3)$	$O(n^2)$	$O(n \cdot \log n)$	$O(n)$
naif...	prog. dyn.	diviser-pour-régner	«scan»

Y a-t-il une différence en pratique ?

J. Bentley «Pearls of programming», Addison-Wesley (1986)

TABLE I. Summary of the Algorithms

Algorithm		1	2	~~3~~ 4	~~4~~ 5
Lines of C Code		8	7	14	7
Run time in microseconds		$3.4N^3$	$13N^2$	$46N \log N$	$33N$
Time to solve problem of size	10^2	3.4 secs	130 msecs	30 msecs	3.3 msecs
	10^3	.94 hrs	13 secs	.45 secs	33 msecs
	10^4	39 days	22 mins	6.1 secs	.33 secs
	10^5	108 yrs	1.5 days	1.3 min	3.3 secs
	10^6	108 mill	5 mos	15 min	33 secs
Max problem solved in one	sec	67	280	2000	30,000
	min	260	2200	82,000	2,000,000
	hr	1000	17,000	3,500,000	120,000,000
	day	3000	81,000	73,000,000	2,800,000,000
If N multiplies by 10, time multiplies by		1000	100	10+	10
If time multiplies by 10, N multiplies by		2.15	3.16	10–	10

1984... machine: VAX-11/750 Langage C

J. Bentley «Pearls of programming», Addison-Wesley (1986)

Pour illustrer $O(n^3) < O(n^2)$:
(et le fait que les constantes sont négligeables…)

CRAY-1 vs TRS 80 ?

algo cubique:
~ 3 n^3 (ns)

algo linéaire:
~19.500.000 n (ns)

CRAY-1 vs TRS 80 ?

TABLE II. The Tyranny of Asymptotics

N	Cray-1, FORTRAN, Cubic Algorithm	TRS-80, BASIC, Linear Algorithm
10	3.0 microsecs	200 millisecs
100	3.0 millisecs	2.0 secs
1000	3.0 secs	20 secs
10,000	49 mins	3.2 mins
100,000	35 days	32 mins
1,000,000	95 yrs	5.4 hrs

J. Bentley «Pearls of programming», Addison-Wesley (1986)

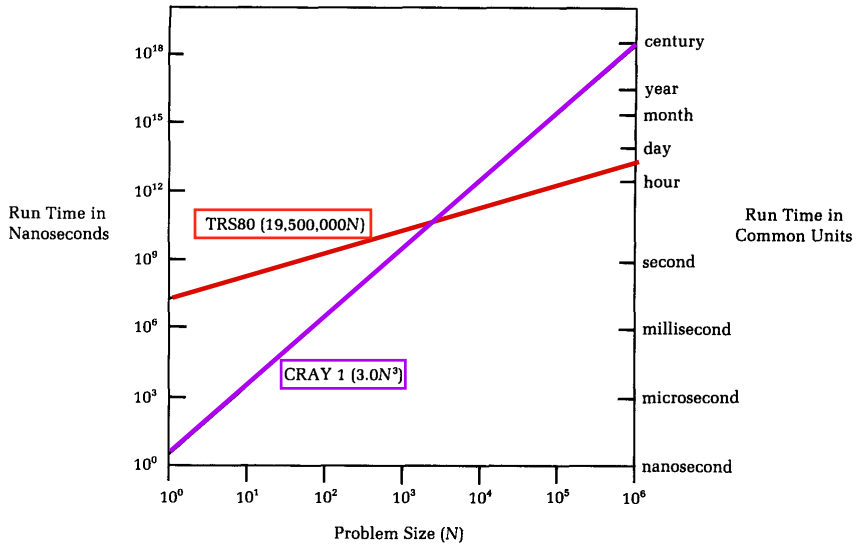


FIGURE 1. The Run Times of Two Programs

J. Bentley «Pearls of programming», Addison-Wesley (1986)

Et aujourd'hui ?

2012: Macbook Air, i7...

Prog. Python

(en secondes)

	$O(n^3)$ Algo 1	$O(n^2)$ Algo 2	$O(n^2)$ Algo 3	$O(n.\log n)$ Algo 4	$O(n)$ Algo 5
100	0,02	0,002	0,0023	0,0006	0,0001
500	2	0,04	0,05	0,0031	0,0003
1 000	16	0,17	0,22	0,007	0,0005
2 000	126	0,7	0,9	0,02	0,0011
10 000		17	21	0,08	0,0052
30 000		159	193	0,3	0,0174
100 000				1	0,0657
1 000 000				10	0,5407

rappel 1984: 1h 22min 15min 33sec

2024: Macbook Pro, M3 pro (18Go)

Prog. Python

(en secondes)

	$O(n^3)$ Algo 1	$O(n^2)$ Algo 2	$O(n^2)$ Algo 3	$O(n.\log n)$ Algo 4	$O(n)$ Algo 5
100	0,005	0,0005	0,0006	0,0001	0
500	0,6	0,01	0,015	0,0008	0,0001
1 000	5	0,05	0,06	0,002	0,0002
2 000	40 (126)	0.2	0,2	0,004	0,0004
10 000	1h22	5	6	0.02	0.0017
30 000		46 (159)	53 (193)	0.06	0.0052
100 000		9 min		0.2	0.0175
1 000 000				2.6 (10)	0.1704

rappel 1984: ~1h 22min 15min 33sec (2012)

J. Bentley «Pearls of programming», Addison-Wesley (1986)

TABLE II. The Tyranny of Asymptotics

N	Cray-1, FORTRAN, Cubic Algorithm	TRS-80, BASIC, Linear Algorithm
10	3.0 microsecs	200 millisecs
100	3.0 millisecs	2.0 secs
1000	3.0 secs	20 secs
10,000	49 mins	3.2 mins
100,000	35 days	32 mins
1,000,000	95 yrs	5.4 hrs

2024:
Sur un MacBook Pro - (M3 Pro, 18Go):
N=10000
Algo en n^3 : **1h22'** **Python**

Algo linéaire: **1,7** ms

mais...
2024: GO
N=10000
Algo en n^3 : **42 s**
Algo linéaire: **15** µs

2024:
Sur un MacBook Pro - **Go**
M3 Pro 18Go
Algo en n^3 :

N=10000 -> **42 s** → **49'** en Fortran sur le CRAY1
→ **1h21** en Python (MBP)
N=20000 -> **5 min 43 s**
N=30000 -> **20 min 37s**
N=32000 -> **25 min 11s**
N=35000 -> **33 min 5 s**

Rappel:
TRS-80, Basic, algo linéaire:
N=10000 -> 3.2 minutes
N=100000 -> 32 minutes

Et bien sûr:
GO+ Algo linéaire:
N=10000 : 9 µs
N=100000: 140 µs

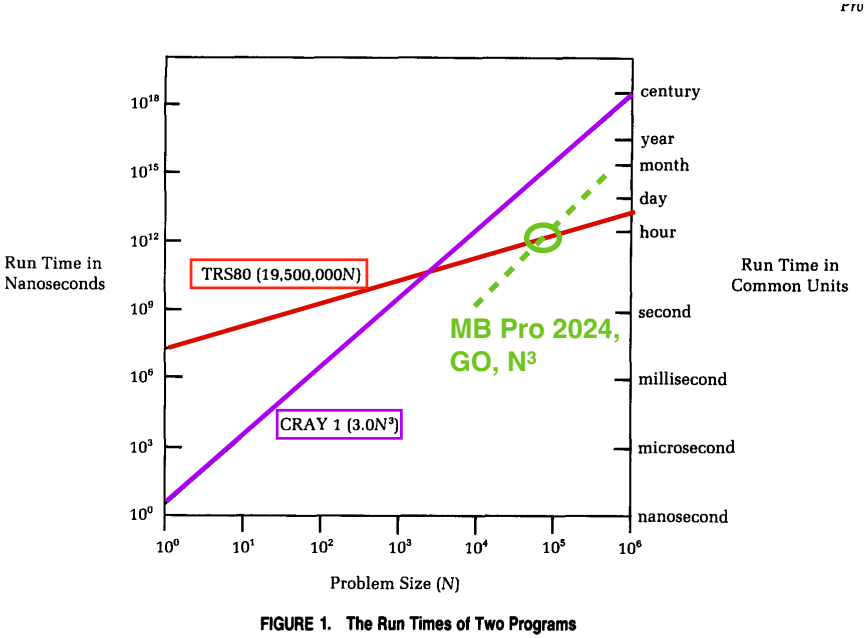


FIGURE 1. The Run Times of Two Programs

J. Bentley «Pearls of programming», Addison-Wesley (1986)

Conclusion…

1984:
Sur un TRS-80 (Z80 qq Ko) - Langage **Basic**
Algo en n :
N=100 000 -> **32 min**

2024:
Sur un MacBook Pro (M3 Pro 18Go) - Langage **Go**
Algo en n^3 :
N=35 000 -> **32 min 34s**

Remarque n°1

La notion de complexité (ou de coût, d'efficacité…) d'un algorithme est robuste…

Elle ne s'attaque pas en achetant (ou en attendant !) un ordinateur avec plus de mémoire, un CPU++, un meilleur langage etc.

(mais mieux vaut avoir un algo top, une machine rapide et programmer avec un langage adapté…)

Remarque n°2

Il y a de fortes différences en pratique entre ces algorithmes !

Et pourtant... ce sont tous des algorithmes dits «*efficaces*» !

il y a beaucoup de problèmes pour lesquels
- il n'y a que des algorithmes très inefficaces… ou
- il n'y a même pas d'algorithme !

Remarque n°3

les constantes, on s'en fout… au début.

Améliorer son code, choisir un langage efficace et une bonne machine (…), c'est nécessaire…

Gagner un facteur 2 (ou plus), c'est appréciable !

Conclusion

⇒ Il y a une vraie différence entre $O(n^3)$, $O(n^2)$, et $O(n)$!

⇒ Il y en a encore plus entre **$O(n^3)$** et **$O(2^n)$**,... !

Rechercher de meilleurs algorithmes est important.

Un problème peut s'attaquer de plusieurs manières: les idées sous-jacentes aux algorithmes peuvent être très différentes !

plan

- Diviser pour régner
- Programmation dynamique
- Gloutons

- Analyse amortie