UFR d'Informatique

Algorithmique M1

2023–2024

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 6C, mardi 8h30 → 10h30
TD1: 279F, Raphael Cosson, lundi 10h45→12h45; [MIDS +...]
TD2: 279F, Roberto Mantaci, lundi 16h15→18h15; [GENIAL +...]
TD3: 279F, F.L., vendredi 14h→16h; [...]

francoisl@irif.fr

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

Constitution des groupes

Cours: Amphi 6C, mardi 8h30 → 10h30
TD1: 279F, Raphael Cosson, lundi 10h45→12h45; [MIDS +...]
TD2: 279F, Roberto Mantaci, lundi 16h15→18h15; [GENIAL +...]
TD3: 279F, F.L., vendredi 14h→16h; [...]

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 6C, mardi 8h30 → 10h30
TD1: 279F, Raphael Cosson, lundi 10h45→12h45; [MIDS +...]
TD2: 279F, Roberto Mantaci, lundi 16h15→18h15; [GENIAL +...]
TD3: 279F, F.L., vendredi 14h→16h; [...]

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 24 octobre ou 7 novembre, 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 à une vielle légende estudiantine…

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

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.

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**:

distribution de probabilités sur les données de taille n

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

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)

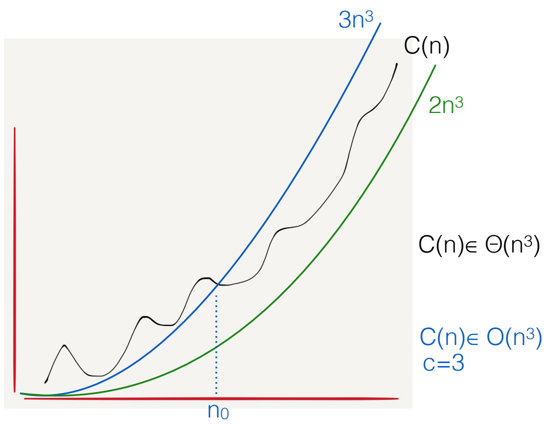
$\Omega(g(n)) = \{f(n) \mid \exists c>0, n_0\geq 0 \text{ tq } 0\leq c.g(n)\leq f(n)\ \forall\ n\geq n_0\}$

→ 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



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)
- …

Les algorithmes efficaces… et les autres.

fct \ 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.

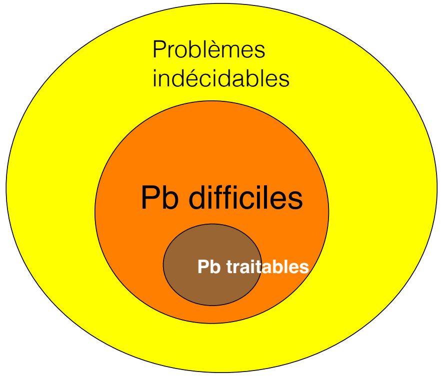
Supposons qu'aujourd'hui, on puisse résoudre un problème de taille K en une heure...

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.

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 » !

Exemple

Voir J. Bentley
«Pearls of programming»

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 }

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

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

Algo 1

Enumérer tous les sous-tableaux...

Complexité en $O(n^3)$

```
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 }

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

Algo 2

idée: beaucoup de calculs inutiles dans algo1...

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

sum=21

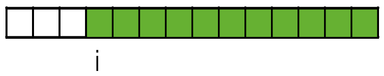
: 35

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
```

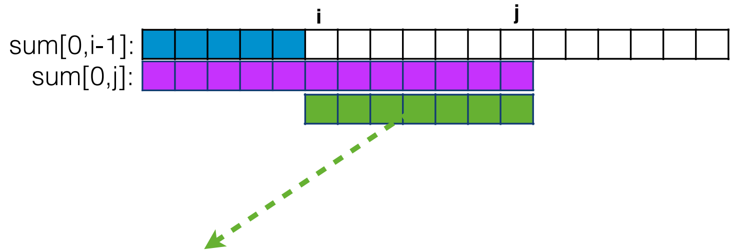
itération i ∈ {0,..,n-1}



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

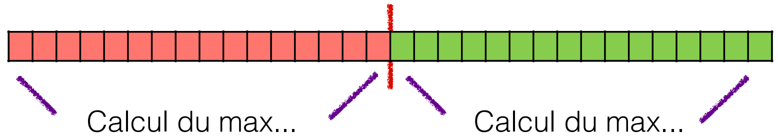
ici sum[i] = «sum[0,i]»

```
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 $O(n^2)$

Algo 4

Un diviser pour régner ?

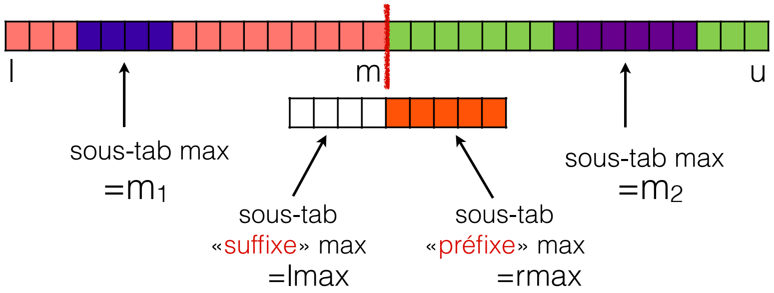


+ «traitement»

↓

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

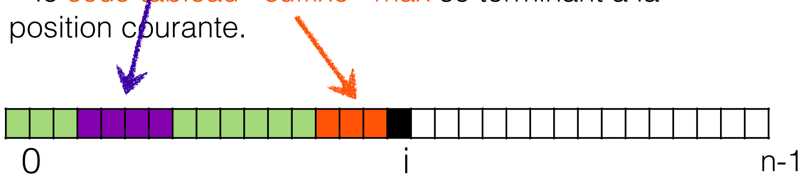
Complexité en O(n·log n)

```
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))
```

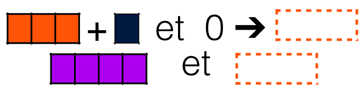
Algo 5

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.

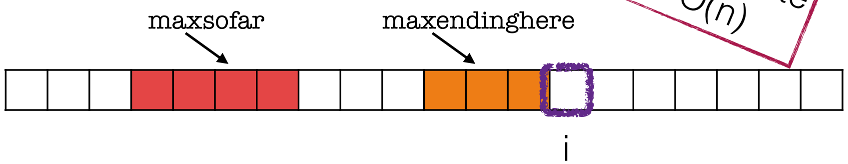


Mise à jour:

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

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)

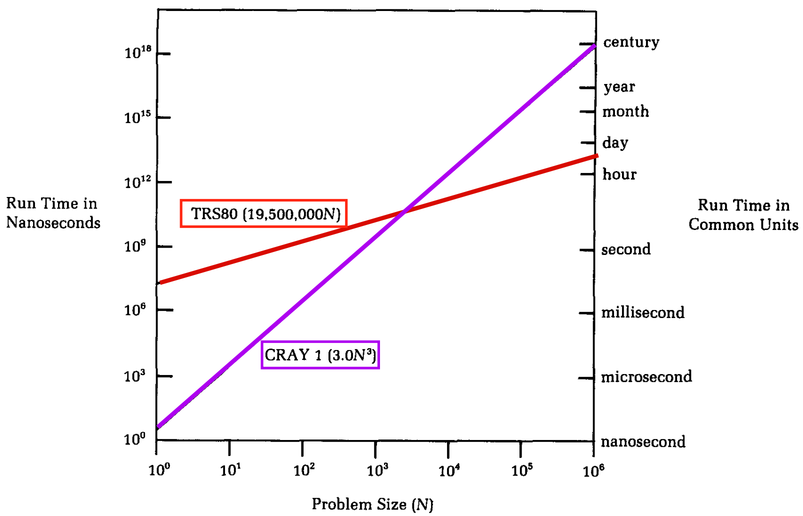


FIGURE 1. The Run Times of Two Programs

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

Et aujourd'hui ?

2012: Macbook Air, i7...

(en secondes)

	Algo 1	Algo 2	Algo 3	Algo 4	Algo 5
100	0,0213	0,0018	0,0023	0,0006	0,0001
500	2,0193	0,0423	0,0533	0,0031	0,0003
1 000	16,2401	0,1736	0,2239	0,007	0,0005
2 000	125,9673	0,7059	0,883	0,0154	0,0011
10 000		17,5162	21,31	0,0788	0,0052
30 000		158,801	192,8002	0,2664	0,0174
100 000				1,0107	0,0657
1 000 000				10,1928	0,5407

rappel 1984: 1h 22min 15min 33sec

2021: Macbook Pro, i5 (16Gb).

(en secondes)

	Algo 1	Algo 2	Algo 3	Algo 4	Algo 5
100	0,0145	0,0015	0,0016	0,0006	0,0000
500	1,6278	0,0345	0,0359	0,0018	0,0002
1 000	14,4006	0,1341	0,1532	0,0042	0,0005
2 000	111.9258	0.5274	0,6124	0,0096	0,0010
10 000		13.718	16.3547	0.052	0.0047
30 000		127.7064	146.9065	0.1731	0.0151
100 000				0.6253	0.0525
1 000 000				7.0648	0.5043

rappel 1984: 1h 22min 15min 33sec

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	2 mins
1,000,000	95 yrs	5.4 hrs

2023:
Sur un MacBook Pro
(3,1 GHz Intel Core i5 double cœur, 16Go):
N=10000
Algo en n^3 : **2789** s (**46**')

Algo linéaire: **284** µs

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++ etc.

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 !

Conclusion

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

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

Rechercher de meilleurs algorithmes est important.

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