Introduction à la complexité algorithmique

Complexité algorithmique

Introduction à la complexité algorithmique

Comment déterminer quel est l'algorithme le plus performant pour résoudre un problème donné ?

Avec quels critères ?

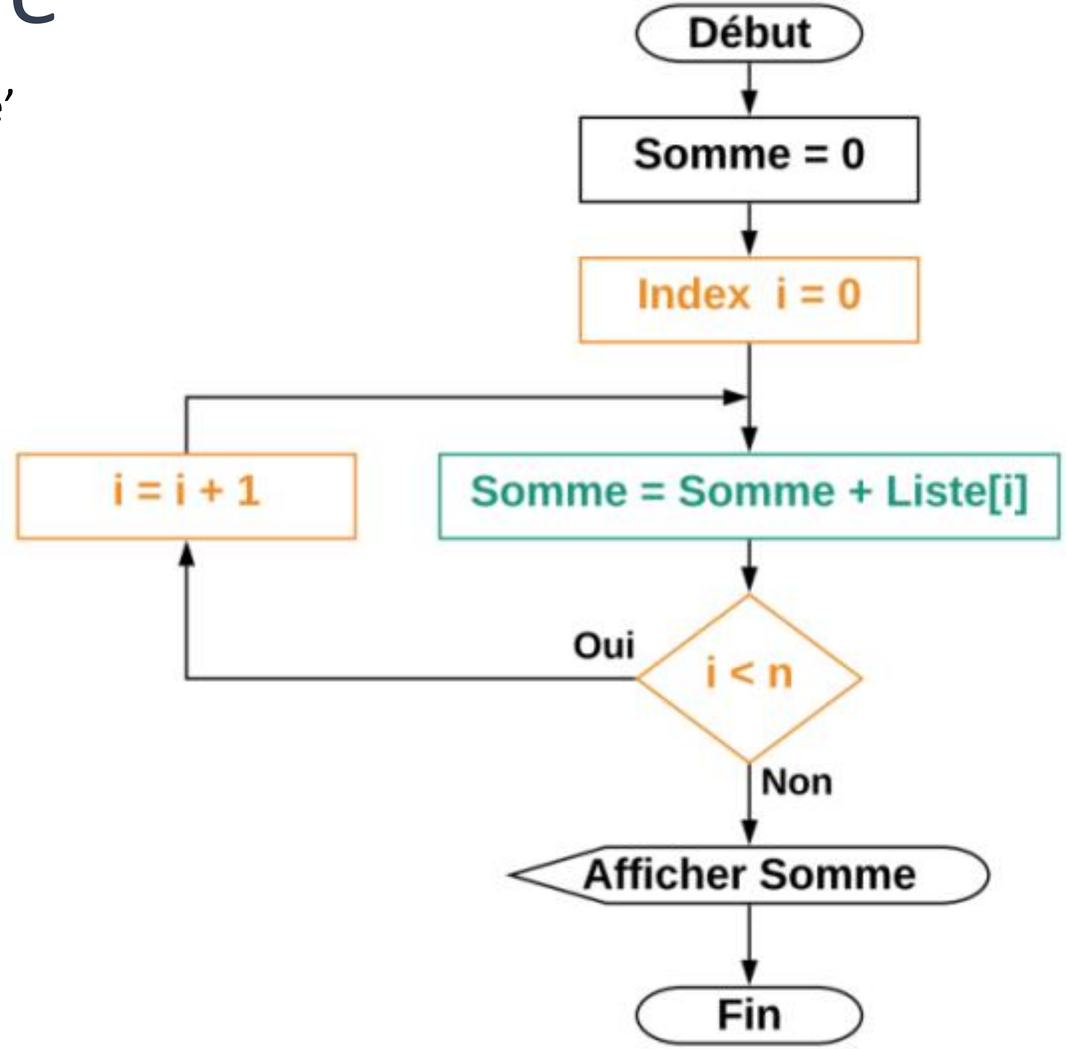
- ❑ La rapidité
- ❑ L'occupation mémoire
- ❑ La bande passante utilisée

Un exemple pour comprendre

Calculons la somme de tous les éléments d'une liste 'Liste' de n nombres.

Voilà l'algorithme écrit en pseudo code :

```
Somme <— 0
Pour i parcourant tous les éléments de la liste
Faire
        Somme <— Somme + Liste[i]
Fin pour
Afficher Somme
```



Voilà l'algorithme sous forme d'algorigramme

Introduction à la complexité algorithmique

Analyse de l'exemple

Quels sont les opérations présentes :

Des affectations de valeurs notées =

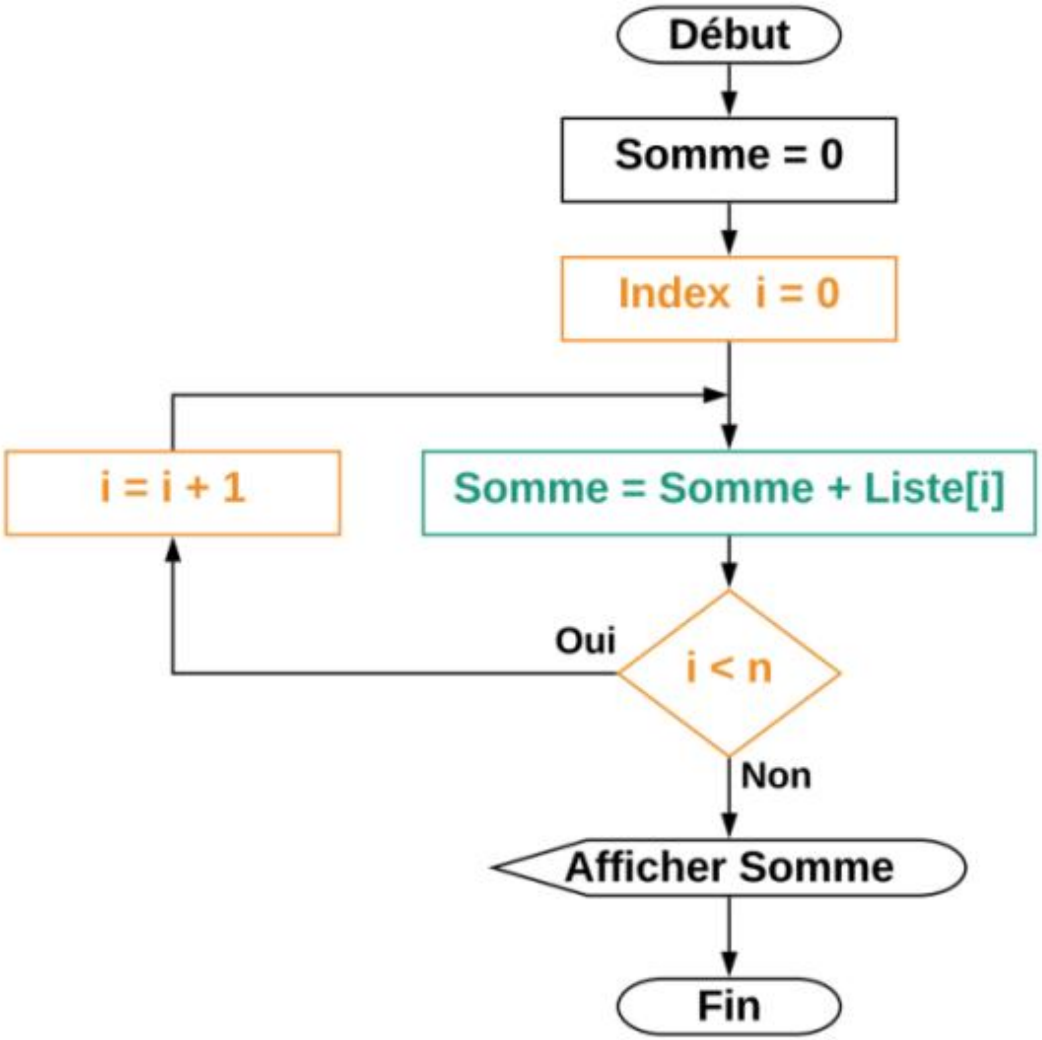
Des additions notées +

Des comparaisons ici <

Étude de l'algorithme de la boucle

La boucle est constituée de deux parties :

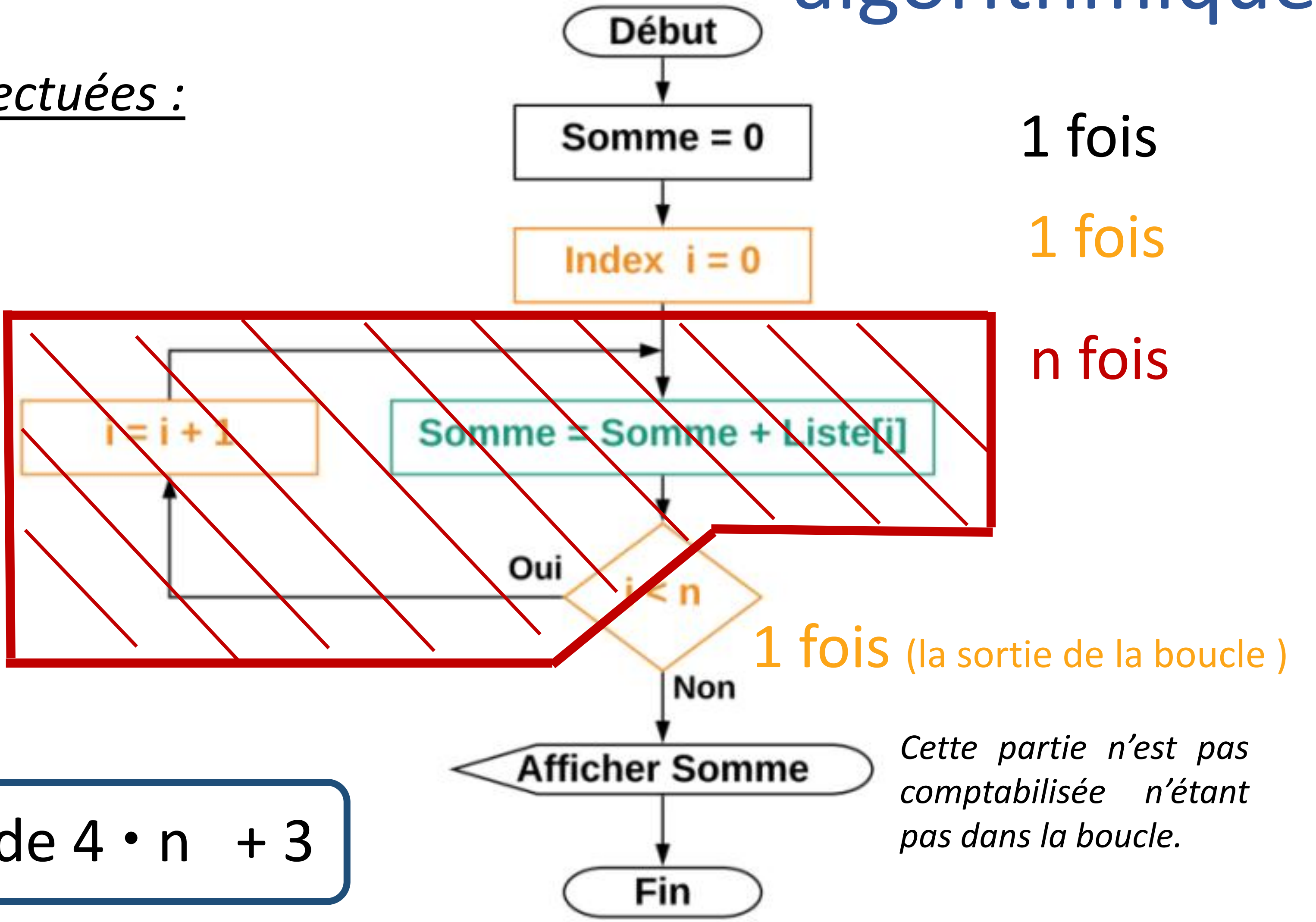
1) La gestion de la boucle permettant de la réaliser n fois avec :
   - l'initialisation **index i = 0**
   - Le test de fin de boucle **i < n**
   - L'incrémentation de l'index **i = i + 1**
2) Le corps de la boucle comprenant l'action effectuée à chaque tour
   somme = somme + Liste[i]



Analyse de l'exemple

Calcul du nombre d'opérations effectuées :

Pour faciliter l'analyse nous identifions les différentes parties de l'algorithme en indiquant le nombre de fois où chacune est exécutée.



Cette partie n'est pas comptabilisée n'étant pas dans la boucle.

Bilan du décompte :

Quantité d'instructions	Nombre d'exécutions
1	1
1	1
4	n
1	1

Le nombre d'opérations est de $4 \cdot n + 3$

Introduction à la complexité algorithmique

Complexité de la boucle

Bilan du décompte :

Le nombre d'opérations est de $4 \cdot n + 3$

Conclusion sur la complexité de l'algorithme :

Quand le nombre de termes n devient très grand la constante est négligeable. Le nombre d'opérations est de l'ordre de grandeur de $4 \cdot n$.

Pour ces études on ne garde que la forme générale et on s'intéresse aux ordres de grandeur, donc le 4 disparaît, et on aboutit à la conclusion que cet algorithme est linéaire en n.

Autrement dis si la quantité de données n est doublée la durée de calcul est doublée également.

Notation mathématique de la complexité :

Il existe une notation mathématique pour indiquer la complexité d'un algorithme : la notation O (lire grand O).

Dans notre exemple la complexité est en : O(n).

Résultats expérimentaux

<u>Résultats obtenus pour différentes valeurs de n :</u>

La complexité en $O(n)$ est-elle vérifiée ?

Il suffit de vérifier que la durée mesurée double bien quand n double. Plus particulièrement pour les grandes valeurs de n.

Qu'observez-vous ?

Votre conclusion ?

```
n =     100 valeurs  le temps d'éxécution :  0.000046 secondes
n =     200 valeurs  le temps d'éxécution :  0.000061 secondes
n =     400 valeurs  le temps d'éxécution :  0.000142 secondes
n =    1000 valeurs  le temps d'éxécution :  0.000332 secondes
n =    2000 valeurs  le temps d'éxécution :  0.000716 secondes
n =     400 valeurs  le temps d'éxécution :  0.000122 secondes
n =    1000 valeurs  le temps d'éxécution :  0.000335 secondes
n =    2000 valeurs  le temps d'éxécution :  0.000708 secondes
n =    4000 valeurs  le temps d'éxécution :  0.001314 secondes
n =   10000 valeurs  le temps d'éxécution :  0.003236 secondes
n =   20000 valeurs  le temps d'éxécution :  0.006700 secondes
n =   40000 valeurs  le temps d'éxécution :  0.014664 secondes
n =  100000 valeurs  le temps d'éxécution :  0.034079 secondes
n =  200000 valeurs  le temps d'éxécution :  0.068755 secondes
n =  400000 valeurs  le temps d'éxécution :  0.141120 secondes
n = 1000000 valeurs  le temps d'éxécution :  0.348514 secondes
n = 2000000 valeurs  le temps d'éxécution :  0.739935 secondes
n = 4000000 valeurs  le temps d'éxécution :  1.497242 secondes
```

Définition d’un algorithme

Un algorithme peut se comprendre comme un ensemble de règles permettant de résoudre un problème donné.

Exemple avec la multiplication R = M•N

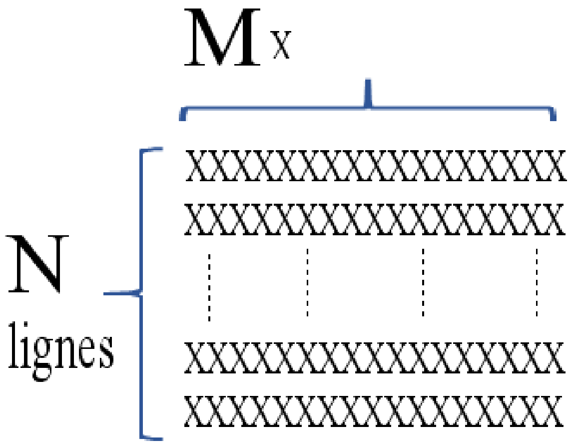
$$\begin{array}{r} M \\ \times\ N \\ \hline R \end{array}$$

Comme à l’école

Boulier chinois

37	129
18	258
9	516
4	1032
2	2064
1	4128
	4773

A la russe



Compter les croix

Pourquoi comparer les algorithmes ?

- ❑Pour trouver le plus rapide
- ❑Pour trouver le plus économe en mémoire
- ❑Pour trouver le meilleur compromis mémoire / vitesse

Les algorithmes doivent être justes

- ❑La terminaison du programme doit être vérifiée dans tous les cas
- ❑Ils doivent être sans ‘bugs’ ………. impossible ??
- ❑Ils doivent produire les résultats attendus

Comment comparer les algorithmes ?

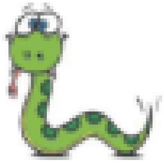
Un exemple : déterminer la liste des décompositions possibles d'un entier n comme somme de deux carrés

N = 1000 solutions possibles :

- 10, 30 => $10^2 + 30^2 = 1000$
- 18, 26 => $18^2 + 26^2 = 1000$

Comparaison de quatre algorithmes

Durée de calcul en mS en fonction de **n**	Algorithme A1	Algorithme A2	Algorithme A3	Algorithme A4
1000	980	489	0,498	0,060
2000	3948	2017	0,953	0,080
4000	15835	7888	1,87	0,109



A vérifier avec le script : Comparaison de complexite.py

Complexité d'un algorithme

Comparaison de quatre algorithmes

Bilan pour la rapidité	Algorithme A1	Algorithme A2	Algorithme A3	Algorithme A4
Complexité théorique	$O(n^2)$	$O(n^2)$	$O(n)$	$O(\sqrt{n})$

Outil théorique de comparaison des complexités

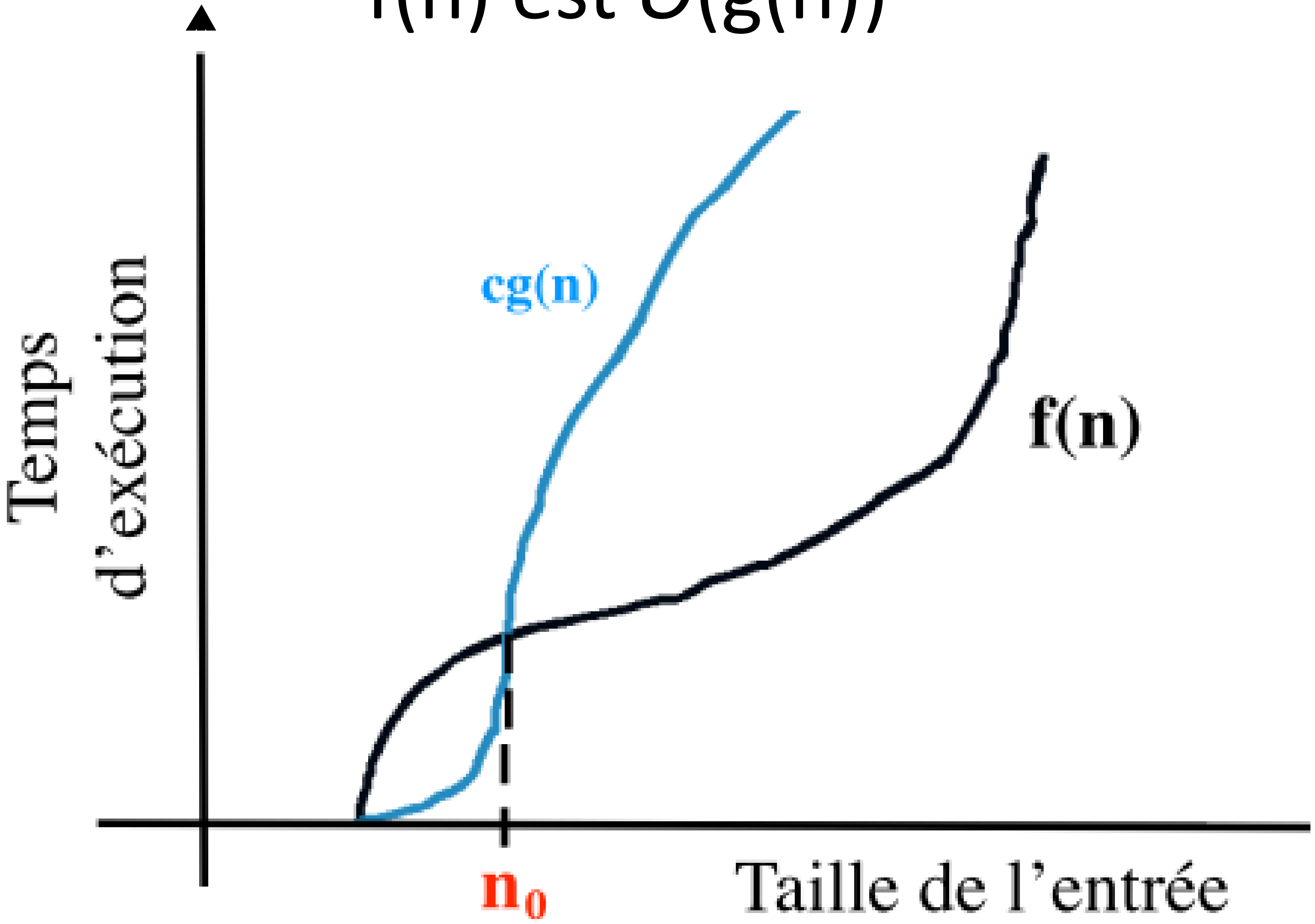
La notation O (lire grand O)

Complexité algorithmique

La notation grand *O*

f(n) est *O*(g(n))

$\exists n_0 \geqq 0, \exists C > 0$
tel que $\forall\, n \geqq n_0$
on a $f(n) \leqq C \cdot g(n)$



Le taux de croissance de f(n) est plus petit ou égal au taux de croissance de g(n)

Comparaison de taux de croissance de fonctions

$$1 \leqq \log_2(n) \leqq n \leqq n \cdot \log_2(n) \leqq n^2 \leqq 2^n \leqq n^n$$

Donc

$$O(1) \leqq O\ \log_2(n)$$

$$O\ (\log_2 n) \leqq O\ (n \cdot \log_2 n)$$

....

Propriétés des logarithmes

$\log_a(x) = \log(x)/\log(a) = \ln(x)/\ln(a)$ logarithme en base a de x

$\log_e(x) = \ln(x)$ logarithme népérien

$\log_{10}(x) = \log(x)$ logarithme décimal

$\log_b(x^a) = a \log_b(x)$

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

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

Opération fondamentale

Pour connaître le temps de calcul, nous choisissons une opération fondamentale et nous calculons le nombre d'opérations fondamentales exécutées par l'algorithme.

Problème	Opération fondamentale
Recherche d'un élément dans une liste	Comparaison
Tri d'une liste, d'un fichier, …	Comparaisons, déplacements
Multiplication des matrices réelles	Multiplications et additions
Addition des entiers binaires	Opération binaire

Complexité de certains problèmes courants

Complexité	Exemple
$O(1)$	Accès à un élément de tableau
$O(\log(n))$	Recherche dichotomique
$O(n)$	Recherche dans un tableau non trié
$O(n\log(n))$	Tri rapide
$O(n^2)$	Tri à bulles
$O(n^3)$	Multiplication de matrices
$O(2^n)$	Algorithme du "voyageur de commerce"

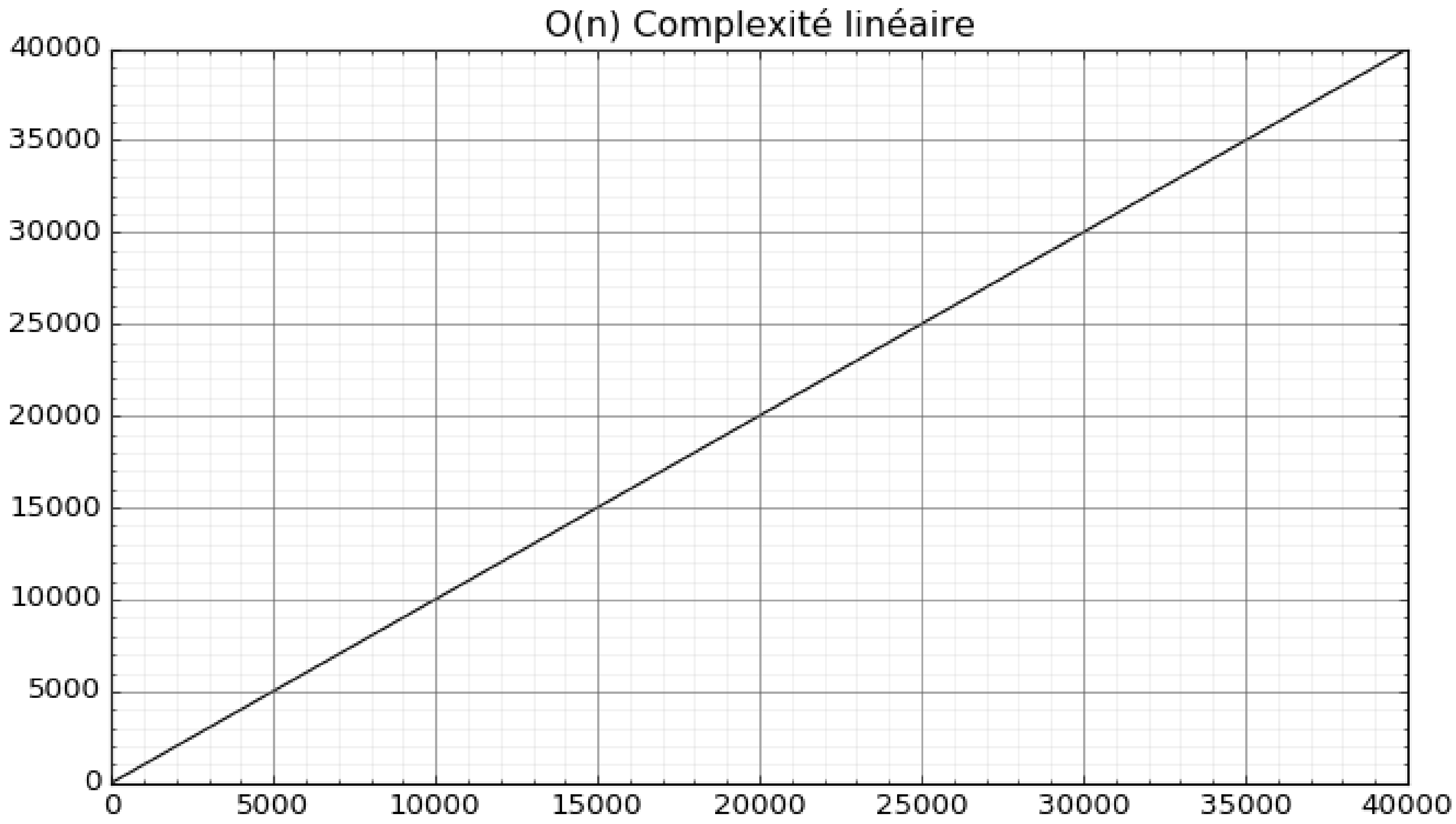
Que peut-on faire en fonction de la complexité

Quelle valeur de n est réaliste pour réaliser un calcul ?

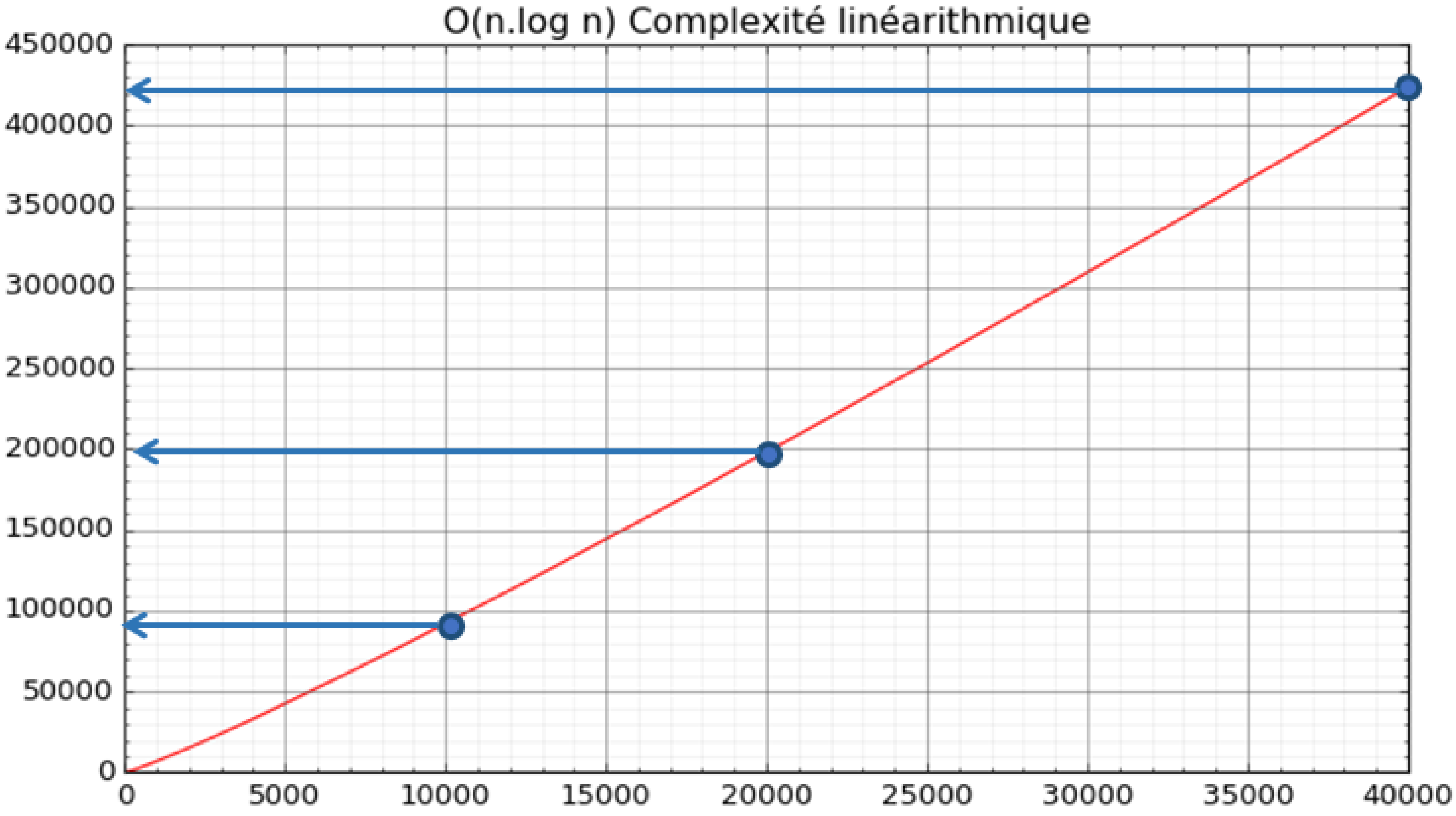
Complexité	Nom courant	Temps quand on double la taille de l'entrée	Max n
$O(n)$	linéaire	prend 2 fois plus de temps	10^{12}
$O(1)$	constant	prend le même temps	pas de limite
$O(n^2)$	quadratique	prend 4 fois plus de temps	10^6
$O(n^3)$	cubique	prend 8 fois plus de temps	10 000
$O(\log n)$	logarithmique	prend seulement une étape de plus	$10^{10^{12}}$
$O(n \log n)$	linearithmique	prend deux fois plus de temps $+ \log n$	10^{11}
$O(2^n)$	exponentiel	prend tellement de temps que c'est inconcevable	30

Dans ce tableau log n représente $\log_2 n$ pour une analyse de complexité ils sont équivalents

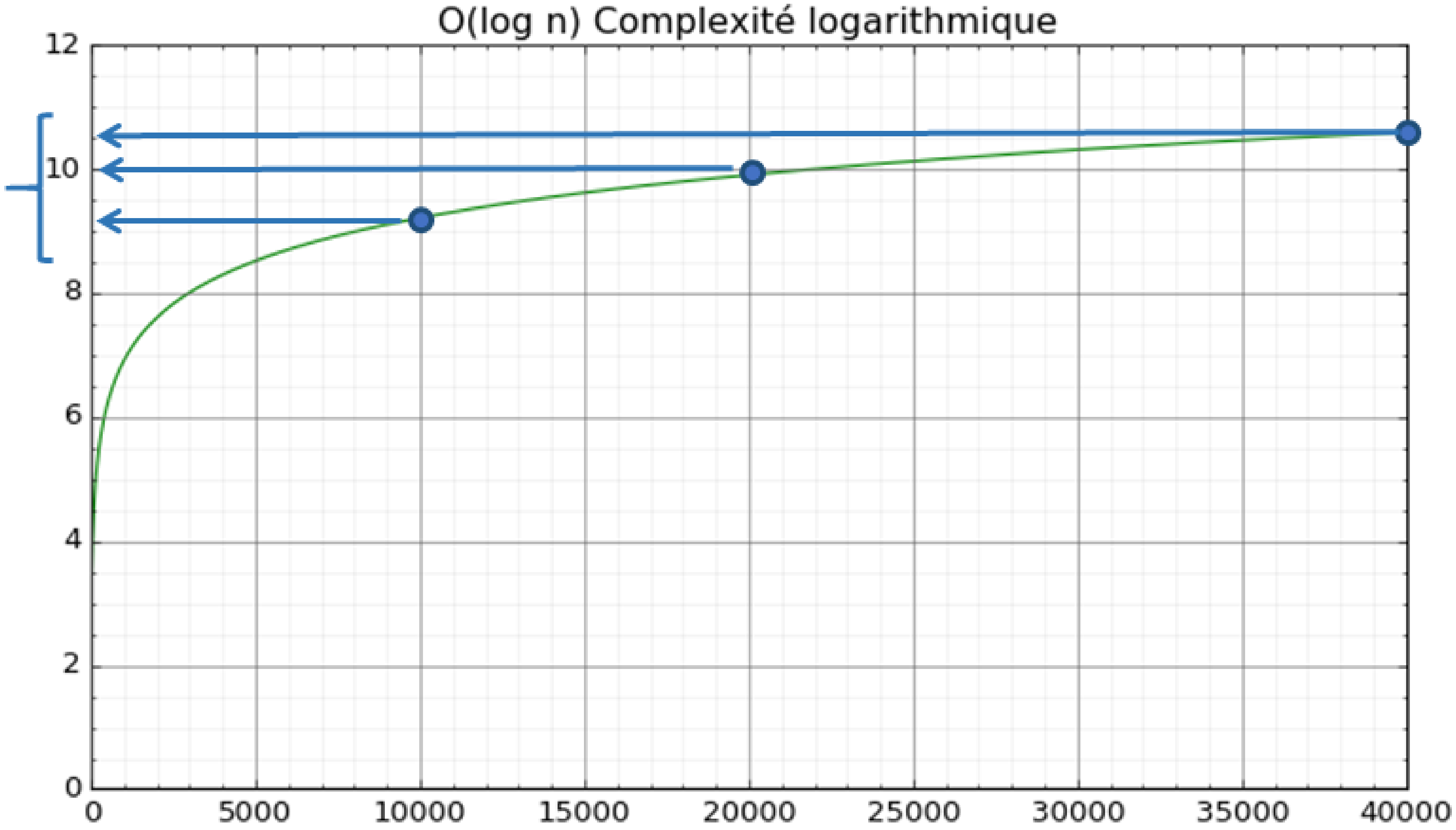
Analyser une complexité par le graphique



Analyser une complexité par le graphique



Analyser une complexité par le graphique



Revenir sur la comparaison de quatre algorithmes

Durée de calcul en mS en fonction de **n**	Algorithme A1	Algorithme A2	Algorithme A3	Algorithme A4
1000	980	489	0,498	0,060
2000	3948	2017	0,953	0,080
4000	15835	7888	1,87	0,109
Complexité théorique	$O(n^2)$	$O(n^2)$	$O(n)$	$O(\sqrt{n})$