

Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
`dominique.poulalhon@irif.fr`

Université Paris Diderot
L2 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2023-2024

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

Question : peut-on l'utiliser pour traiter une entrée de taille
 $n = 10\,000$ ($= 100 \times 100 = 100^2 = 100 + 9\,900$) ?

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

Question : peut-on l'utiliser pour traiter une entrée de taille $n = 10\,000$ ($= 100 \times 100 = 100^2 = 100 + 9\,900$) ?

Le temps (approximatif) nécessaire dépend de sa complexité :

$C(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
temps	0,02 s	1 s	2 s	2 min	3 h	10^{2960} ans

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

Question : peut-on l'utiliser pour traiter une entrée de taille $n = 10\,000$ ($= 100 \times 100 = 100^2 = 100 + 9\,900$) ?

Le temps (approximatif) nécessaire dépend de sa complexité :

$C(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
temps	0,02 s	1 s	2 s	2 min	3 h	10^{2960} ans

- si $C(n) = \alpha \log n$:

$$C(10\,000) = C(100^2) = \alpha \log(100^2) = \alpha \cdot 2 \cdot \log 100 = 2 \cdot C(100)$$

\Rightarrow peu importe α

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

Question : peut-on l'utiliser pour traiter une entrée de taille $n = 10\,000$ ($= 100 \times 100 = 100^2 = 100 + 9\,900$) ?

Le temps (approximatif) nécessaire dépend de sa complexité :

$C(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
temps	0,02 s	1 s	2 s	2 min	3 h	10^{2960} ans

- si $C(n) = \alpha n$:

$$C(10\,000) = C(100 \times 100) = \alpha \times 100 \times 100 = 100 \cdot C(100)$$

\implies peu importe α

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

Question : peut-on l'utiliser pour traiter une entrée de taille $n = 10\,000$ ($= 100 \times 100 = 100^2 = 100 + 9\,900$) ?

Le temps (approximatif) nécessaire dépend de sa complexité :

$C(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
temps	0,02 s	1 s	2 s	2 min	3 h	10^{2960} ans

- si $C(n) = \alpha n^2$:

$$C(10\,000) = C(100 \times 100) = \alpha \times 100^2 \times 100^2 = 100^2 \cdot C(100)$$

\Rightarrow peu importe α

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

Question : peut-on l'utiliser pour traiter une entrée de taille $n = 10\,000$ ($= 100 \times 100 = 100^2 = 100 + 9\,900$) ?

Le temps (approximatif) nécessaire dépend de sa complexité :

$C(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
temps	0,02 s	1 s	2 s	2 min	3 h	10^{2960} ans

- si $C(n) = \alpha 2^n$:

$$C(10\,000) = C(9\,900 + 100) = \alpha \times 2^{9\,900} \times 2^{100} = 2^{9\,900} \cdot C(100) \\ \Rightarrow \text{peu importe } \alpha$$

COMPLEXITÉ ET ORDRES DE GRANDEUR

Considérons un algorithme (ou plutôt un programme, dans un langage donné, sur une machine donnée), qui met

1 centième de seconde à traiter les entrées de taille $n = 100$

Question : peut-on l'utiliser pour traiter une entrée de taille $n = 10\,000$ ($= 100 \times 100 = 100^2 = 100 + 9\,900$) ?

Le temps (approximatif) nécessaire dépend de sa complexité :

$C(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
temps	0,02 s	1 s	2 s	2 min	3 h	10^{2960} ans

Autre manière de voir les choses : en une heure, ce programme peut traiter des entrées de taille au plus...

$C(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
n_{\max}	$10^{720\,000}$	36 000 000	10 000 000	60 000	7100	118

COMMENT MONTRER QUE $f \in \Theta(g)$?

Exemple : soit $f(n) = 5n^3 + 2n^2$

On veut montrer que $f(n) \in \Theta(n^3)$, c'est-à-dire :

$$\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \quad \forall n \geq n_0, \quad c_1 n^3 \leq f(n) \leq c_2 n^3$$

COMMENT MONTRER QUE $f \in \Theta(g)$?

Exemple : soit $f(n) = 5n^3 + 2n^2$

On veut montrer que $f(n) \in \Theta(n^3)$, c'est-à-dire :

$$\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \quad \forall n \geq n_0, \quad c_1 n^3 \leq f(n) \leq c_2 n^3$$

Il suffit donc de *trouver/deviner* un n_0 et des constantes c_1, c_2 satisfaisants, puis de prouver/vérifier qu'ils le sont.

COMMENT MONTRER QUE $f \in \Theta(g)$?

Exemple : soit $f(n) = 5n^3 + 2n^2$

On veut montrer que $f(n) \in \Theta(n^3)$, c'est-à-dire :

$$\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \quad \forall n \geq n_0, \quad c_1 n^3 \leq f(n) \leq c_2 n^3$$

Il suffit donc de *trouver/deviner* un n_0 et des constantes c_1, c_2 satisfaisants, puis de prouver/vérifier qu'ils le sont.

- ici, $c_1 = 5$ convient, puisque pour tout $n \geq 0$,

$$f(n) = 5n^3 + \underbrace{2n^2}_{\geq 0} \geq 5n^3$$

COMMENT MONTRER QUE $f \in \Theta(g)$?

Exemple : soit $f(n) = 5n^3 + 2n^2$

On veut montrer que $f(n) \in \Theta(n^3)$, c'est-à-dire :

$$\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \quad \forall n \geq n_0, \quad c_1 n^3 \leq f(n) \leq c_2 n^3$$

Il suffit donc de *trouver/deviner* un n_0 et des constantes c_1, c_2 satisfaisants, puis de prouver/vérifier qu'ils le sont.

- ici, $c_1 = 5$ convient, puisque pour tout $n \geq 0$,

$$f(n) = 5n^3 + \underbrace{2n^2}_{\geq 0} \geq 5n^3$$

- pour c_2 et n_0 , c'est un peu plus compliqué ; on peut par exemple prendre $c_2 = 7$ et $n_0 = 1$ car pour tout $n \geq 1$, $n^2 \leq n^3$, donc

$$f(n) = 5n^3 + \underbrace{2n^2}_{\leq 2n^3} \leq 7n^3.$$

COMMENT MONTRER QUE $f \in \Theta(g)$?

Exemple : soit $f(n) = 5n^3 + 2n^2$

On veut montrer que $f(n) \in \Theta(n^3)$, c'est-à-dire :

$$\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N}, \quad \forall n \geq n_0, \quad c_1 n^3 \leq f(n) \leq c_2 n^3$$

Il suffit donc de *trouver/deviner* un n_0 et des constantes c_1, c_2 satisfaisants, puis de prouver/vérifier qu'ils le sont.

- ici, $c_1 = 5$ convient, puisque pour tout $n \geq 0$,

$$f(n) = 5n^3 + \underbrace{2n^2}_{\geq 0} \geq 5n^3$$

- pour c_2 et n_0 , c'est un peu plus compliqué ; on peut par exemple prendre $c_2 = 7$ et $n_0 = 1$ car pour tout $n \geq 1$, $n^2 \leq n^3$, donc

$$f(n) = 5n^3 + \underbrace{2n^2}_{\leq 2n^3} \leq 7n^3.$$

Autre option, raisonner « à la limite » : si $\lim_{n \rightarrow \infty} \frac{f}{g} = c$ avec $c > 0$, alors pour n'importe quelles constantes $c_1 < c < c_2$, il existe n_0 (éventuellement très grand) avec la bonne propriété

RÉSULTATS À RETENIR

Remarque : les fonctions de complexité sont par définition positives, et en général de limite infinie

- tous les polynômes¹ de degré d sont dans la classe $\Theta(n^d)$

1. à coefficient dominant positif

RÉSULTATS À RETENIR

Remarque : les fonctions de complexité sont par définition positives, et en général de limite infinie

- tous les **polynômes**¹ de degré d sont dans la classe $\Theta(n^d)$
- les classes $\Theta(n^d)$ ($d \geq 0$) sont **strictement ordonnées** en fonction du **degré** (y compris d non entier) :
si $d_1 < d_2$, $n^{d_1} \in O(n^{d_2})$, mais $n^{d_1} \notin \Theta(n^{d_2})$

1. à coefficient dominant positif

RÉSULTATS À RETENIR

Remarque : les fonctions de complexité sont par définition positives, et en général de limite infinie

- tous les **polynômes**¹ de degré d sont dans la classe $\Theta(n^d)$
- les classes $\Theta(n^d)$ ($d \geq 0$) sont **strictement ordonnées** en fonction du **degré** (y compris d non entier) :
si $d_1 < d_2$, $n^{d_1} \in O(n^{d_2})$, mais $n^{d_1} \notin \Theta(n^{d_2})$
- tous les **logarithmes** (de n'importe quelle base) de polynômes (non constants) sont dans la classe $\Theta(\log n)$

1. à coefficient dominant positif

RÉSULTATS À RETENIR

Remarque : les fonctions de complexité sont par définition positives, et en général de limite infinie

- tous les **polynômes**¹ de degré d sont dans la classe $\Theta(n^d)$
- les classes $\Theta(n^d)$ ($d \geq 0$) sont **strictement ordonnées** en fonction du **degré** (y compris d non entier) :
si $d_1 < d_2$, $n^{d_1} \in O(n^{d_2})$, mais $n^{d_1} \notin \Theta(n^{d_2})$
- tous les **logarithmes** (de n'importe quelle base) de polynômes (non constants) sont dans la classe $\Theta(\log n)$
- pour tout $d > 0$, $\log n \in O(n^d)$, mais $\log n \notin \Theta(n^d)$

1. à coefficient dominant positif

RÉSULTATS À RETENIR

Remarque : les fonctions de complexité sont par définition positives, et en général de limite infinie

- tous les **polynômes**¹ de degré d sont dans la classe $\Theta(n^d)$
- les classes $\Theta(n^d)$ ($d \geq 0$) sont **strictement ordonnées** en fonction du **degré** (y compris d non entier) :
si $d_1 < d_2$, $n^{d_1} \in O(n^{d_2})$, mais $n^{d_1} \notin \Theta(n^{d_2})$
- tous les **logarithmes** (de n'importe quelle base) de polynômes (non constants) sont dans la classe $\Theta(\log n)$
- pour tout $d > 0$, $\log n \in O(n^d)$, mais $\log n \notin \Theta(n^d)$
- les fonctions **exponentielles** sont strictement ordonnées en fonction de la **base**
si $b_1 < b_2$, $b_1^n \in O(b_2^n)$, mais $b_1^n \notin \Theta(b_2^n)$

1. à coefficient dominant positif

RÉSULTATS À RETENIR

Remarque : les fonctions de complexité sont par définition positives, et en général de limite infinie

- tous les **polynômes**¹ de degré d sont dans la classe $\Theta(n^d)$
- les classes $\Theta(n^d)$ ($d \geq 0$) sont **strictement ordonnées** en fonction du **degré** (y compris d non entier) :
si $d_1 < d_2$, $n^{d_1} \in O(n^{d_2})$, mais $n^{d_1} \notin \Theta(n^{d_2})$
- tous les **logarithmes** (de n'importe quelle base) de polynômes (non constants) sont dans la classe $\Theta(\log n)$
- pour tout $d > 0$, $\log n \in O(n^d)$, mais $\log n \notin \Theta(n^d)$
- les fonctions **exponentielles** sont strictement ordonnées en fonction de la **base**
si $b_1 < b_2$, $b_1^n \in O(b_2^n)$, mais $b_1^n \notin \Theta(b_2^n)$
- pour tous $d > 0$ et $b > 1$, $n^d \in O(b^n)$, mais $n^d \notin \Theta(b^n)$

1. à coefficient dominant positif

QUELQUES ÉLÉMENTS DE PREUVE

- pour toutes bases a et b , $\log_a n \in \Theta(\log_b n)$: en effet, par définition

$$\forall n, \quad n = a^{\log_a n} = b^{\log_b n} \quad (*)$$

QUELQUES ÉLÉMENTS DE PREUVE

- pour toutes bases a et b , $\log_a n \in \Theta(\log_b n)$: en effet, par définition

$$\forall n, \quad n = a^{\log_a n} = b^{\log_b n} \quad (*)$$

en particulier, $b = a^{\log_a b}$ (et symétriquement $a = b^{\log_b a}$)

QUELQUES ÉLÉMENTS DE PREUVE

- pour toutes bases a et b , $\log_a n \in \Theta(\log_b n)$: en effet, par définition

$$\forall n, \quad n = a^{\log_a n} = b^{\log_b n} \quad (*)$$

en particulier, $b = a^{\log_a b}$ (et symétriquement $a = b^{\log_b a}$)

en substituant dans (*), on obtient :

$$\forall n, \quad n = a^{\log_a n} = (a^{\log_a b})^{\log_b n} = a^{(\log_a b) \cdot (\log_b n)}$$

QUELQUES ÉLÉMENTS DE PREUVE

- pour toutes bases a et b , $\log_a n \in \Theta(\log_b n)$: en effet, par définition

$$\forall n, \quad n = a^{\log_a n} = b^{\log_b n} \quad (*)$$

en particulier, $b = a^{\log_a b}$ (et symétriquement $a = b^{\log_b a}$)

en substituant dans (*), on obtient :

$$\forall n, \quad n = a^{\log_a n} = (a^{\log_a b})^{\log_b n} = a^{(\log_a b) \cdot (\log_b n)}$$

donc :

$$\forall n, \quad \log_a n = (\log_a b) \cdot (\log_b n)$$

(et en particulier, $(\log_a b) \cdot (\log_b a) = 1$)

QUELQUES ÉLÉMENTS DE PREUVE

- pour toutes bases a et b , $\log_a n \in \Theta(\log_b n)$: en effet, par définition

$$\forall n, \quad n = a^{\log_a n} = b^{\log_b n} \quad (*)$$

en particulier, $b = a^{\log_a b}$ (et symétriquement $a = b^{\log_b a}$)

en substituant dans (*), on obtient :

$$\forall n, \quad n = a^{\log_a n} = (a^{\log_a b})^{\log_b n} = a^{(\log_a b) \cdot (\log_b n)}$$

donc :

$$\forall n, \quad \log_a n = (\log_a b) \cdot (\log_b n)$$

(et en particulier, $(\log_a b) \cdot (\log_b a) = 1$)

- pour tout $b > 1$, et tout $d > 0$, $\log_b(n^d) \in \Theta(\log_b n)$; en effet,

$$\forall n, \quad \log_b(n^d) = d \cdot \log_b n$$

QUELQUES ÉLÉMENTS DE PREUVE

- pour toutes bases a et b , $\log_a n \in \Theta(\log_b n)$: en effet, par définition

$$\forall n, \quad n = a^{\log_a n} = b^{\log_b n} \quad (*)$$

en particulier, $b = a^{\log_a b}$ (et symétriquement $a = b^{\log_b a}$)

en substituant dans $(*)$, on obtient :

$$\forall n, \quad n = a^{\log_a n} = (a^{\log_a b})^{\log_b n} = a^{(\log_a b) \cdot (\log_b n)}$$

donc :

$$\forall n, \quad \log_a n = (\log_a b) \cdot (\log_b n)$$

(et en particulier, $(\log_a b) \cdot (\log_b a) = 1$)

- pour tout $b > 1$, et tout $d > 0$, $\log_b(n^d) \in \Theta(\log_b n)$; en effet,

$$\forall n, \quad \log_b(n^d) = d \cdot \log_b n$$

donc tous les **logarithmes** (de n'importe quelle base) de polynômes non constants sont dans la même classe, notée $\Theta(\log n)$ sans préciser la base

COMPLEXITÉ DES CALCULS DE F_n

utilisation naïve de la récurrence

$\implies \Theta(\varphi^n)$ additions

```
def fibo_1(n) :  
    if n <= 2 : return 1  
    return fibo_1(n-1) + fibo_1(n-2)
```

Analyse de la complexité (démonstration d'un résultat un peu moins fort mais suffisant) : soit $A(n)$ le nombre d'additions effectuées

$$A(n) = \begin{cases} 0 & \text{si } n \leq 2 \\ 1 + A(n-1) + A(n-2) & \text{si } n > 2 \end{cases}$$

(donc en fait, $A(n) + 1 = F_n$ pour tout $n \geq 1$)

donc $A(n) \geq A(n-1)$ pour tout n , c'est-à-dire que A est croissante
(et même strictement croissante sauf entre 0 et 2)

d'où l'encadrement : $\forall n > 3, 2A(n-2) \leq A(n) \leq 2A(n-1)$,

qui entraîne : $A(n) \in \Omega(\sqrt{2}^n)$ et $A(n) \in O(2^n)$

COMPLEXITÉ DES CALCULS DE F_n

calcul itératif des n premières valeurs

$\Rightarrow \Theta(n)$ additions

```
def fibo_3(n) :  
    previous, last = 0, 1  
    for i in range(2, n+1) :  
        previous, last = last, previous + last  
    return last
```

COMPLEXITÉ DES CALCULS DE F_n

calcul itératif des n premières valeurs

$\Rightarrow \Theta(n)$ additions

```
def fibo_3(n) :  
    previous, last = 0, 1  
    for i in range(2, n+1) :  
        previous, last = last, previous + last  
    return last
```

Preuve de correction : à l'aide de l'invariant :

« après le tour de boucle d'indice i , $previous = F_{i-1}$ et $last = F_i$ »

- c'est vrai « après le tour d'indice 1 », *i.e.* avant le 1^{er} tour de boucle (d'indice 2)
- si $previous = F_{i-1}$ et $last = F_i$ au début d'un tour de boucle,
 $previous = F_i$ et $last = F_{i-1} + F_i = F_{i+1}$ après ce tour

donc à la sortie de la boucle, $previous = F_{n-1}$ et $last = F_n$, donc
`fibo_3(n)` retourne F_n pour tout n

COMPLEXITÉ DES CALCULS DE F_n

calcul itératif des n premières valeurs

$\Rightarrow \Theta(n)$ additions

```
def fibo_3(n) :  
    previous, last = 0, 1  
    for i in range(2, n+1) :  
        previous, last = last, previous + last  
    return last
```

Analyse de la complexité : $n - 1$ tours de boucle, avec une addition (de grands entiers) par tour, donc :

$$A(n) = n - 1 \in \Theta(n)$$

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

```
def puissance(a, n) :  
    if n == 0 : return 1  
    if n == 1 : return a  
    tmp = puissance(a, n//2)  
    carre = tmp * tmp           # une multiplication  
    if n%2 == 0 : return carre  
    else : return a * carre     # une multiplication
```

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

```
def puissance(a, n) :  
    if n == 0 : return 1  
    if n == 1 : return a  
    tmp = puissance(a, n//2)  
    carre = tmp * tmp           # une multiplication  
    if n%2 == 0 : return carre  
    else : return a * carre    # une multiplication
```

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

Rappel : $\log_b n$ est défini par l'égalité $n = b^{\log_b n}$, donc :

- $b^{\lfloor \log_b n \rfloor} \leq n < b^{\lfloor \log_b n \rfloor + 1}$,
- n s'écrit avec $\lfloor \log_b n \rfloor + 1$ chiffres en base b ,
- la division euclidienne de n par b itérée $\lfloor \log_b n \rfloor + 1$ fois donne 0

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

si ces multiplications ont un coût constant, *i.e.* si les opérandes ont une taille constante, complexité en $\Theta(\log_2 n)$

c'est le cas avec l'arithmétique modulaire ou l'arithmétique flottante usuelles : tous les nombres sont codés sur exactement 32 (ou 64) bits, donc le coût d'une multiplication est constant

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

si ces multiplications ont un coût constant, complexité en $\Theta(\log_2 n)$

sinon, il faut tenir compte du coût de ces multiplications

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

si ces multiplications ont un coût constant, complexité en $\Theta(\log_2 n)$

sinon, il faut tenir compte du coût de ces multiplications

par exemple en arithmétique exacte sur des entiers, même en considérant que a est de taille bornée, a^k est de taille $\Theta(k)$, donc :

- la multiplication $a \times a^k$ par l'algo naïf a un coût $\Theta(k)$
- le calcul du carré de a^k par l'algo naïf a un coût $\Theta(k^2)$

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

si ces multiplications ont un coût constant, complexité en $\Theta(\log_2 n)$

sinon, il faut tenir compte du coût de ces multiplications

par exemple en arithmétique exacte sur des entiers, même en considérant que a est de taille bornée, a^k est de taille $\Theta(k)$, donc :

- la multiplication $a \times a^k$ par l'algo naïf a un coût $\Theta(k)$
- le calcul du carré de a^k par l'algo naïf a un coût $\Theta(k^2)$

$\Theta(\log n)$ multiplications, chacune en $O(n^2) \implies$ cumul en $O(n^2 \log n)$

COMPLEXITÉ DE L'EXPONENTIATION BINAIRE

Complexité : $\Theta(\log_2 n)$ multiplications de la forme $a \times a^k$ ou $a^k \times a^k$

si ces multiplications ont un coût constant, complexité en $\Theta(\log_2 n)$

sinon, il faut tenir compte du coût de ces multiplications

par exemple en arithmétique exacte sur des entiers, même en considérant que a est de taille bornée, a^k est de taille $\Theta(k)$, donc :

- la multiplication $a \times a^k$ par l'algo naïf a un coût $\Theta(k)$
- le calcul du carré de a^k par l'algo naïf a un coût $\Theta(k^2)$

$\Theta(\log n)$ multiplications, chacune en $O(n^2) \implies$ cumul en $O(n^2 \log n)$

Mais on peut être plus précis : si on néglige le coût des multiplications par a et qu'on considère seulement les calculs successifs de carrés, cela fait (à peu près, en partant du dernier calculé) : $c(\frac{n}{2})^2 + c(\frac{n}{4})^2 + c(\frac{n}{8})^2 + c(\frac{n}{16})^2 + \dots$

d'où un coût cumulé en $\Theta(n^2)$

(par l'algo de multiplication naïf)

COMPLEXITÉ DES CALCULS DE F_n

utilisation naïve de la récurrence $\implies \Theta(\varphi^n)$ additions (d'entiers)

calcul itératif des n premières valeurs $\implies \Theta(n)$ additions (d'entiers)

calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ $\implies \Theta(\log_2 n)$ multiplications
(de matrices 2×2)
(chacune implique 4 additions et 8 multiplications d'entiers)

COMPLEXITÉ DES CALCULS DE F_n

utilisation naïve de la récurrence $\Rightarrow \Theta(\varphi^n)$ additions (d'entiers)

calcul itératif des n premières valeurs $\Rightarrow \Theta(n)$ additions (d'entiers)

calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ $\Rightarrow \Theta(\log_2 n)$ multiplications
(de matrices 2×2)
(chacune implique 4 additions et 8 multiplications d'entiers)

comme $F_n \in \Theta(\varphi^n)$, les opérations arithmétiques se font sur des *entiers de taille $\Theta(n)$* (c'est-à-dire de $\Theta(n)$ chiffres dans la base choisie)

\Rightarrow additions en $\Theta(n)$ opérations élémentaires,
multiplications en $O(n^2)$ (coût de l'algo naïf)

COMPLEXITÉ DES CALCULS DE F_n (ET DU PRODUIT D'ENTRIERS)

Conclusion provisoire

- le calcul itératif des n premières valeurs est en $\Theta(n^2)$
- le calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ est en $O(n^2)$
(coût de l'algo basé sur l'algo de multiplication usuel)

COMPLEXITÉ DES CALCULS DE F_n (ET DU PRODUIT D'ENTIERS)

Conclusion provisoire

- le calcul itératif des n premières valeurs est en $\Theta(n^2)$
- le calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ est en $O(n^2)$
(coût de l'algo basé sur l'algo de multiplication usuel)

Or...

- les résultats des expérimentations montrent bien une complexité en $\Theta(n^2)$ pour `fibonacci_3`
- mais `fibonacci_4` semble nettement plus efficace que `fibonacci_3`

COMPLEXITÉ DES CALCULS DE F_n (ET DU PRODUIT D'ENTIERS)

Conclusion provisoire

- le calcul itératif des n premières valeurs est en $\Theta(n^2)$
- le calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ est en $O(n^2)$
(coût de l'algo basé sur l'algo de multiplication usuel)

Or...

- les résultats des expérimentations montrent bien une complexité en $\Theta(n^2)$ pour `fibonacci_3`
- mais `fibonacci_4` semble nettement plus efficace que `fibonacci_3`

Conclusion : l'algorithme de multiplication usuel *n'est pas optimal*
(et ce n'est pas lui qui est utilisé par PYTHON)

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

(présentée sur des polynômes pour éviter les retenues)

Hypothèse : P, Q de degré (au plus) $2^k - 1$,
 $P^{(0)}$ et $P^{(1)}$ les polynômes de degré (au plus) $2^{k-1} - 1$ tels que :

$$P = P^{(0)} + P^{(1)} \cdot X^{2^{k-1}}$$

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

(présentée sur des polynômes pour éviter les retenues)

Hypothèse : P, Q de degré (au plus) $2^k - 1$,
 $P^{(0)}$ et $P^{(1)}$ les polynômes de degré (au plus) $2^{k-1} - 1$ tels que :

$$P = P^{(0)} + P^{(1)} \cdot X^{2^{k-1}}$$

Alors :

$$P \cdot Q = P^{(0)}Q^{(0)} + (P^{(0)}Q^{(1)} + P^{(1)}Q^{(0)})X^{2^{k-1}} + P^{(1)}Q^{(1)}X^{2^k}$$

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

(présentée sur des polynômes pour éviter les retenues)

Hypothèse : P, Q de degré (au plus) $2^k - 1$,
 $P^{(0)}$ et $P^{(1)}$ les polynômes de degré (au plus) $2^{k-1} - 1$ tels que :

$$P = P^{(0)} + P^{(1)} \cdot X^{2^{k-1}}$$

Alors :

$$P \cdot Q = P^{(0)}Q^{(0)} + (P^{(0)}Q^{(1)} + P^{(1)}Q^{(0)})X^{2^{k-1}} + P^{(1)}Q^{(1)}X^{2^k}$$

Ou encore :

$$\begin{aligned} P \cdot Q &= P^{(0)}Q^{(0)} + P^{(1)}Q^{(1)}X^{2^k} \\ &+ \left[(P^{(0)} + P^{(1)})(Q^{(0)} + Q^{(1)}) - P^{(0)}Q^{(0)} - P^{(1)}Q^{(1)} \right] X^{2^{k-1}} \end{aligned}$$

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

(présentée sur des polynômes pour éviter les retenues)

Hypothèse : P, Q de degré (au plus) $2^k - 1$,
 $P^{(0)}$ et $P^{(1)}$ les polynômes de degré (au plus) $2^{k-1} - 1$ tels que :

$$P = P^{(0)} + P^{(1)} \cdot X^{2^{k-1}}$$

Alors :

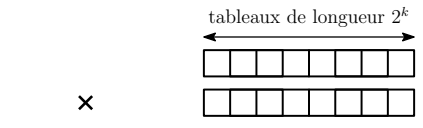
$$P \cdot Q = P^{(0)}Q^{(0)} + (P^{(0)}Q^{(1)} + P^{(1)}Q^{(0)})X^{2^{k-1}} + P^{(1)}Q^{(1)}X^{2^k}$$

Ou encore :

$$\begin{aligned} P \cdot Q &= P^{(0)}Q^{(0)} + P^{(1)}Q^{(1)}X^{2^k} \\ &+ \left[(P^{(0)} + P^{(1)})(Q^{(0)} + Q^{(1)}) - P^{(0)}Q^{(0)} - P^{(1)}Q^{(1)} \right] X^{2^{k-1}} \end{aligned}$$

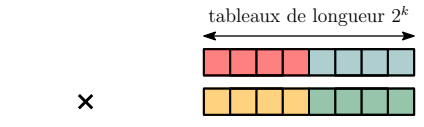
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Polynômes de degré $2^k - 1 \iff$ tableaux de longueur 2^k



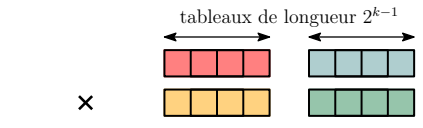
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Découpage en tableaux de longueur 2^{k-1}



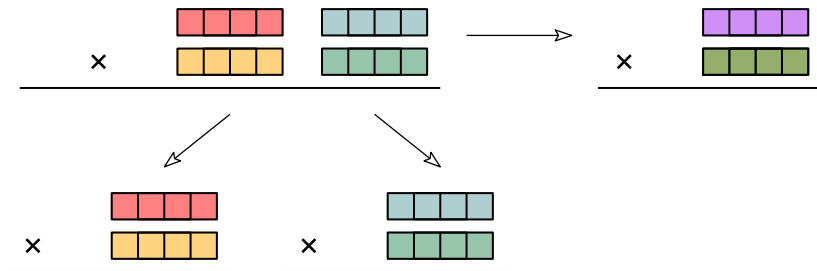
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Découpage en tableaux de longueur 2^{k-1}



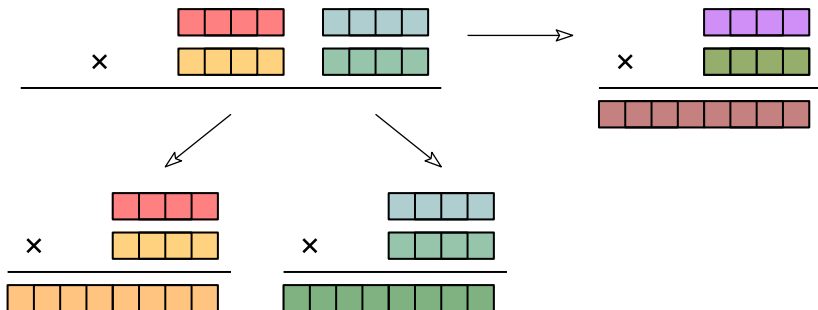
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Trois appels *récur­sifs* sur des tableaux de longueur 2^{k-1}



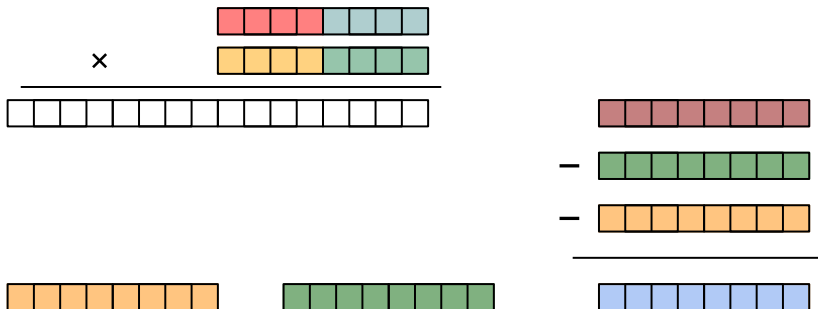
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Trois appels *récurifs* sur des tableaux de longueur 2^{k-1}



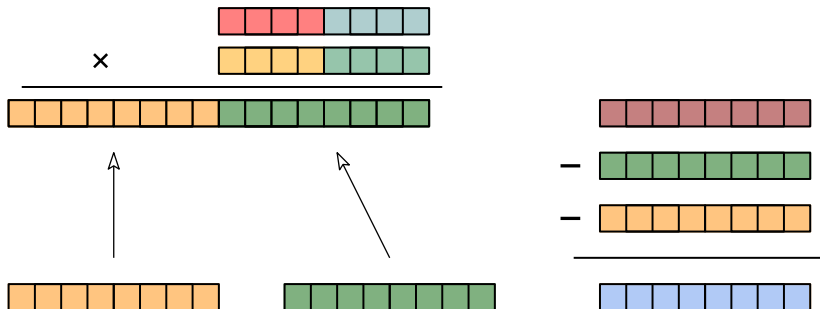
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Regroupement des résultats des appels récurrents



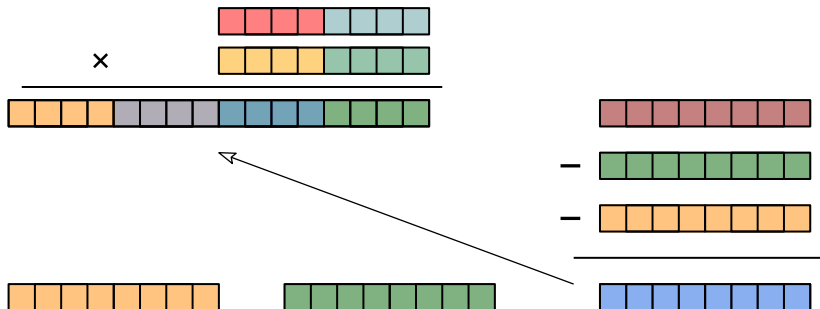
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Regroupement des résultats des appels récurrents



MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

- Regroupement des résultats des appels récurrents



MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Exemple (sur des entiers écrits en base 10, unités à droite) :

$$\begin{array}{r} 1 \ 3 \ 5 \ 7 \\ \times \ 8 \ 4 \ 2 \ 1 \\ \hline \end{array}$$

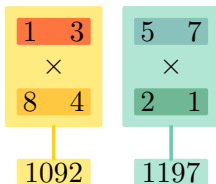
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Exemple (sur des entiers écrits en base 10, unités à droite) :

$$\begin{array}{r} \begin{array}{cc} 1 & 3 \\ \times & 8 & 4 \end{array} \quad \begin{array}{cc} 5 & 7 \\ 2 & 1 \end{array} = 13 \cdot 100 + 57 \\ \hline \end{array}$$

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Exemple (sur des entiers écrits en base 10, unités à droite) :



MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Exemple (sur des entiers écrits en base 10, unités à droite) :

A diagram illustrating the Karatsuba multiplication of 13 and 57. The number 13 is represented by a red box with '1' and a light red box with '3'. The number 57 is represented by a light blue box with '5' and a blue box with '7'. These are enclosed in a larger light purple box. A purple line connects this box to a purple box containing the result '701'.

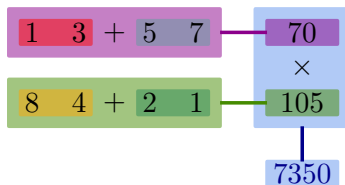
$$\begin{array}{|c|c|} \hline 1 & 3 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 5 & 7 \\ \hline \end{array} = 701$$

A diagram illustrating the Karatsuba multiplication of 84 and 21. The number 84 is represented by a yellow box with '8' and a light yellow box with '4'. The number 21 is represented by a light green box with '2' and a green box with '1'. These are enclosed in a larger light green box. A green line connects this box to a green box containing the result '1764'.

$$\begin{array}{|c|c|} \hline 8 & 4 \\ \hline \end{array} + \begin{array}{|c|c|} \hline 2 & 1 \\ \hline \end{array} = 1764$$

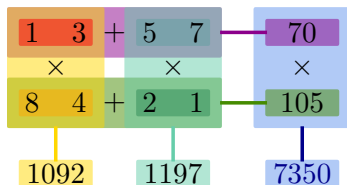
MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Exemple (sur des entiers écrits en base 10, unités à droite) :



MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Exemple (sur des entiers écrits en base 10, unités à droite) :



$$1092 \cdot 10000 + (7350 - 1092 - 1197) \cdot 100 + 1197$$

$$= 11\,427\,297$$

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

stratégie « *diviser-pour-régner* » :

- découper le problème en sous-problèmes de taille inférieure
- résoudre *récurivement* le ou les sous-problèmes
- résoudre le problème initial à l'aide des résultats des sous-problèmes

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

stratégie « *diviser-pour-régner* » :

- scinder chaque polynôme de longueur $n = 2^k$ en deux polynômes de longueur $\frac{n}{2} = 2^{k-1} \implies P^{(0)}, P^{(1)}, Q^{(0)}, Q^{(1)}$
- calculer $P^{(0)} + P^{(1)}$ et $Q^{(0)} + Q^{(1)}$ (*2 sommes de taille $\frac{n}{2}$*)
- résoudre *récurivement* le ou les sous-problèmes
- résoudre le problème initial à l'aide des résultats des sous-problèmes

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

stratégie « *diviser-pour-régner* » :

- scinder chaque polynôme de longueur $n = 2^k$ en deux polynômes de longueur $\frac{n}{2} = 2^{k-1} \implies P^{(0)}, P^{(1)}, Q^{(0)}, Q^{(1)}$
- calculer $P^{(0)} + P^{(1)}$ et $Q^{(0)} + Q^{(1)}$ (*2 sommes de taille $\frac{n}{2}$*)
- appeler récursivement **karatsuba** sur :
 - $(P^{(0)}, Q^{(0)}) \implies R^{(0)}$ (de taille n)
 - $(P^{(1)}, Q^{(1)}) \implies R^{(1)}$ (de taille n)
 - $(P^{(0)} + P^{(1)}, Q^{(0)} + Q^{(1)}) \implies R^{(2)}$ (de taille n)
- résoudre le problème initial à l'aide des résultats des sous-problèmes

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

stratégie « *diviser-pour-régner* » :

- scinder chaque polynôme de longueur $n = 2^k$ en deux polynômes de longueur $\frac{n}{2} = 2^{k-1} \implies P^{(0)}, P^{(1)}, Q^{(0)}, Q^{(1)}$
- calculer $P^{(0)} + P^{(1)}$ et $Q^{(0)} + Q^{(1)}$ (*2 sommes de taille $\frac{n}{2}$*)

- appeler récursivement **karatsuba** sur :
 - $(P^{(0)}, Q^{(0)}) \implies R^{(0)}$ (de taille n)
 - $(P^{(1)}, Q^{(1)}) \implies R^{(1)}$ (de taille n)
 - $(P^{(0)} + P^{(1)}, Q^{(0)} + Q^{(1)}) \implies R^{(2)}$ (de taille n)

- calculer $R^{(3)} = R^{(2)} - R^{(0)} - R^{(1)}$ (*2 sommes de taille n*)
- calculer $R = R^{(0)} + R^{(3)} X^{\frac{n}{2}} + R^{(1)} X^n$ (*2 sommes de taille $\frac{n}{2}$*)

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Complexité : elle se décompose en 2 parties :

- le coût des 3 appels récurifs sur des paramètres de taille $\frac{n}{2}$
- le coût des additions : $\Theta(n)$ additions élémentaires
i.e. de coefficients (pour les polynômes) ou de chiffres (pour les entiers)

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Complexité : elle se décompose en 2 parties :

- le coût des 3 appels récurifs sur des paramètres de taille $\frac{n}{2}$
- le coût des additions : $\Theta(n)$ additions élémentaires
i.e. de coefficients (pour les polynômes) ou de chiffres (pour les entiers)

$$\Rightarrow C(n) = 3 \cdot C\left(\frac{n}{2}\right) + \Theta(n)$$

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Complexité : elle se décompose en 2 parties :

- le coût des 3 appels récurifs sur des paramètres de taille $\frac{n}{2}$
- le coût des additions : $\Theta(n)$ additions élémentaires
i.e. de coefficients (pour les polynômes) ou de chiffres (pour les entiers)

$$\Rightarrow C(n) = 3 \cdot C\left(\frac{n}{2}\right) + \Theta(n)$$

Hypothèse : on peut se contenter de compter les multiplications
soit $M(n)$ le nombre de multiplications élémentaires (de chiffres),

$$M(1) = 1 \quad \text{et} \quad M(2^k) = 3 \cdot M(2^{k-1})$$

donc $M(2^k) = 3^k$, et plus généralement

$$M(n) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$$

avec $\log_2 3 \approx 1.585$

MULTIPLICATION PAR LA MÉTHODE DE KARATSUBA

Complexité : elle se décompose en 2 parties :

- le coût des 3 appels récurifs sur des paramètres de taille $\frac{n}{2}$
- le coût des additions : $\Theta(n)$ additions élémentaires
i.e. de coefficients (pour les polynômes) ou de chiffres (pour les entiers)

$$\Rightarrow C(n) = 3 \cdot C\left(\frac{n}{2}\right) + \Theta(n)$$

Hypothèse : on peut se contenter de compter les multiplications
soit $M(n)$ le nombre de multiplications élémentaires (de chiffres),

$$M(1) = 1 \quad \text{et} \quad M(2^k) = 3 \cdot M(2^{k-1})$$

donc $M(2^k) = 3^k$, et plus généralement

$$M(n) = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$$

avec $\log_2 3 \approx 1.585$

Remarque : cela valide *a posteriori* le choix de négliger les additions,
puisque l'on peut montrer qu'elles ont donc un coût cumulé en $O(n^{\log_2 3})$

CONCLUSION : COMPLEXITÉ DES CALCULS DE F_n

utilisation naïve de la récurrence $\Rightarrow \Theta(n\varphi^n)$

calcul itératif des n premières valeurs $\Rightarrow \Theta(n^2)$

calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ $\Rightarrow O(n^{\log_2 3})$

CONCLUSION : COMPLEXITÉ DES CALCULS DE F_n

utilisation naïve de la récurrence $\implies \Theta(n\varphi^n)$

calcul itératif des n premières valeurs $\implies \Theta(n^2)$

calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ $\implies O(n^{\log_2 3})$

(la complexité du calcul de puissance par exponentiation rapide est du même ordre de grandeur que la multiplication utilisée ; la preuve est la même que pour le cas de la multiplication naïve ; donc $\Theta(n^{\log_2 3})$ par Karatsuba)

Complexité pour deux entiers de n bits

???	par itération d'additions	$\Theta(n \cdot 2^n)$
???	méthode scolaire <i>version binaire utilisée dès l'Égypte ancienne en général, nécessite une numération de position</i>	$\Theta(n^2)$
1960	conjecture de Kolmogorov : complexité intrinsèque	$\Omega(n^2)$
1962	algorithme de Karatsuba <i>utilisé par Python pour les grands entiers</i>	$\Theta(n^{\log_2 3})$
1971	algorithme de Schönhage et Strassen <i>à base de « Transformée de Fourier Rapide » utilisé par la bibliothèque GMP pour $n > 100\,000$</i>	$\Theta(n \log n \log \log n)$
2019	algorithme de Harvey et van der Hoeven <i>(mais seulement pour $n \geq 2^{1729^{12}}$...)</i>	$O(n \log n)$

Conjecture (1971) complexité intrinsèque du problème en $\Theta(n \log n)$