Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2019-2020

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	100 s	10 000 s	$2^{9\,900}/100$ s

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	$0,02\,s$	$1\,s$	$2\,s$	$2\,\text{min}$	$3\,\text{h}$	10^{2960} ans

- si $C(n) = \alpha \log n$:
  $C(10\,000) = C(100^2) = \alpha \log(100^2) = \alpha \cdot 2 \cdot \log 100 = 2 \cdot C(100)$
  $\Longrightarrow$ *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	$0,02\,s$	$1\,s$	$2\,s$	$2\,\text{min}$	$3\,\text{h}$	10^{2960} ans

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

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

  $\Longrightarrow$ *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 $C(n) = \alpha\, n^2$:
  $C(10\,000) = C(100 \times 100) = \alpha \times 100^2 \times 100^2 = 100^2 \cdot C(100)$
  $\Longrightarrow$ *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 $C(n) = \alpha 2^n$:

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

$\Longrightarrow$ *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

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 :

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

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 :

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

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 :

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

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 $n \geqslant 0$,

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

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 :

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

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 $n \geqslant 0$,

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

- 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 $n \geqslant 1$, $n^2 \leqslant n^3$, donc

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

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 :

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

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 $n \geqslant 0$,

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

- 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 $n \geqslant 1$, $n^2 \leqslant n^3$, donc

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

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 :

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

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 $n \geqslant 0$,

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

- 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 $n \geqslant 1$, $n^2 \leqslant n^3$, donc

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

Autre option, raisonner « *à la limite* » : si $\lim \frac{f}{g} = c$ 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 polynômes [1] 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 $\Theta(n^d)$ $(d \geqslant 0)$ sont strictement ordonnées en fonction du degré (y compris d non entier)

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 $\Theta(n^d)$ $(d \geqslant 0)$ sont strictement ordonnées en fonction du degré (y compris d non entier)
- 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 $\Theta(n^d)$ $(d \geqslant 0)$ sont strictement ordonnées en fonction du degré (y compris d non entier)
- 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 $\Theta(n^d)$ $(d \geqslant 0)$ sont strictement ordonnées en fonction du degré (y compris d non entier)
- 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

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 $\Theta(n^d)$ $(d \geqslant 0)$ sont strictement ordonnées en fonction du degré (y compris d non entier)
- 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
- 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

Complexité des calculs de F_n

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

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

Complexité des calculs de F_n

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

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

Preuve de terminaison : par récurrence (forte)

- cas de base : si $n \leqslant 2$, `fibo_1(n)` termine
- hérédité : soit $n \geqslant 2$ t.q. `fibo_1(k)` termine pour tout $k < n$
  alors `fibo_1(n-1)` et `fibo_1(n-2)` terminent, donc `fibo_1(n)` termine

donc `fibo_1(n)` termine pour tout n

Complexité des calculs de F_n

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

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

Preuve de correction : par récurrence (forte)

- cas de base : si $n \leqslant 2$, `fibo_1(n)` retourne F_n
- hérédité : soit $n \geqslant 2$ t.q. `fibo_1(k)` retourne F_k pour tout $k < n$
  alors `fibo_1(n)` retourne
  `fibo_1(n − 1)` + `fibo_1(n − 2)` $= F_{n-1} + F_{n-2} = F_n$

donc `fibo_1(n)` retourne F_n pour tout n

Complexité des calculs de F_n

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

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

(donc $A(n) + 1 = F_n$ pour tout $n \geqslant 1$)

donc $A(n) \geqslant A(n-1)$ pour tout n, donc A est croissante,

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

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

Complexité des calculs de F_n

calcul itératif des n premières valeurs $\Longrightarrow$ $\Theta(n)$ 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 $\Longrightarrow$ $\Theta(n)$ 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 terminaison : $n - 1$ tours de boucle

Complexité des calculs de F_n

calcul itératif des n premières valeurs $\Longrightarrow$ $\Theta(n)$ 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 $\Longrightarrow$ $\Theta(n)$ 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 $a^k \cdot a^\ell$, $k \in \{1, \ell\}$

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 $a^k \cdot a^\ell$, $k \in \{1, \ell\}$

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

- $b^{\lfloor \log_b n \rfloor} \leqslant n < b^{\lfloor \log_b n \rfloor + 1}$,
- 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 $a^k \cdot a^\ell$

Complexité de l'exponentiation binaire

Complexité

$\Theta(\log_2 n)$ multiplications de la forme $a^k \cdot a^\ell$

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 utilisées usuellement : 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 $a^k \cdot a^\ell$

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

sinon il faut tenir compte du coût de ces multiplications ; par exemple en arithmétique exacte sur des entiers :

valeur	taille (en bits)	coût du calcul naïf du carré
a	$\log_2 a$	$\Theta((\log_2 a)^2)$
a^k	$k \cdot \log_2 a$	$\Theta(k^2 \cdot (\log_2 a)^2)$

Complexité des calculs de F_n

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

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

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

Complexité des calculs de F_n

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

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

calcul de $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ $\Longrightarrow$ $\Theta(\log_2 n)$ multiplications...
de matrices 2×2
(chacune impliquant 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)

$\Longrightarrow$ 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'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 $O(n^2 \log n)$ (coût de l'algo utilisant 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 $O(n^2 \log n)$ (coût de l'algo utilisant l'algo de multiplication usuel)

Or...

- les résultats des expérimentations montrent bien une complexité en $\Theta(n^2)$ pour `fibo_3`
- mais `fibo_4` semble nettement plus efficace que `fibo_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 $O(n^2 \log n)$ (coût de l'algo utilisant l'algo de multiplication usuel)

Or...

- les résultats des expérimentations montrent bien une complexité en $\Theta(n^2)$ pour `fibo_3`
- mais `fibo_4` semble nettement plus efficace que `fibo_3`

Conclusion : l'algorithme de multiplication usuel *n'est pas optimal*

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 :

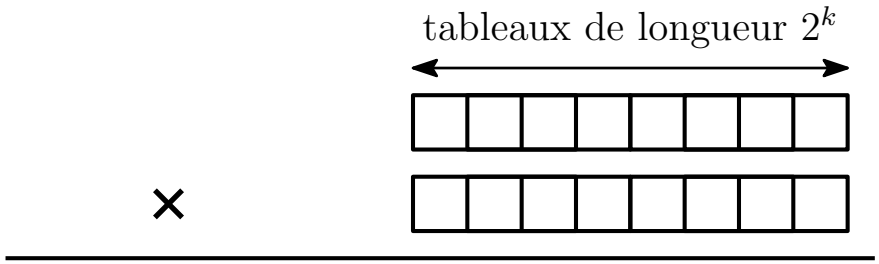
$$P \cdot Q = P^{(0)}Q^{(0)} + (P^{(0)}Q^{(1)} + P^{(1)}Q^{(0)})X^{2^{k-1}} + P^{(1)}Q^{(1)}X^{2^k}$$

Ou encore :

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

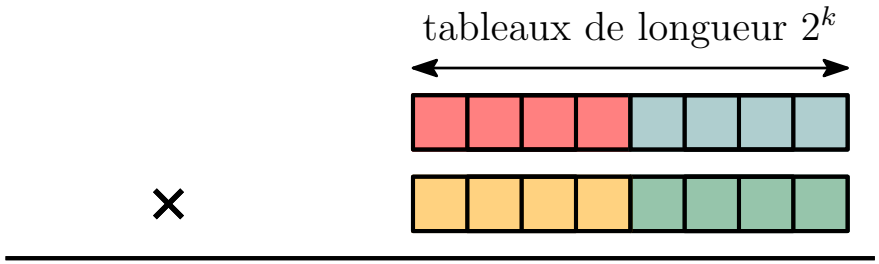
Multiplication par la méthode de Karatsuba

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



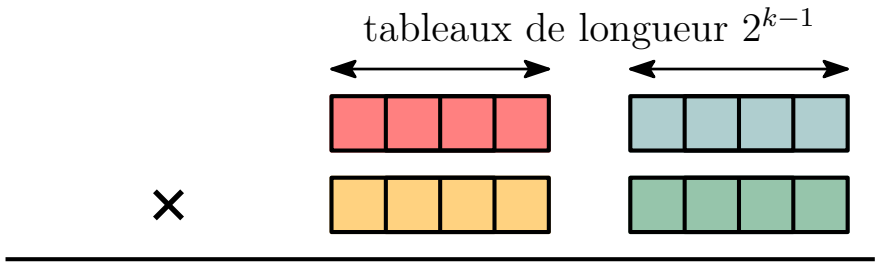
Multiplication par la méthode de Karatsuba

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



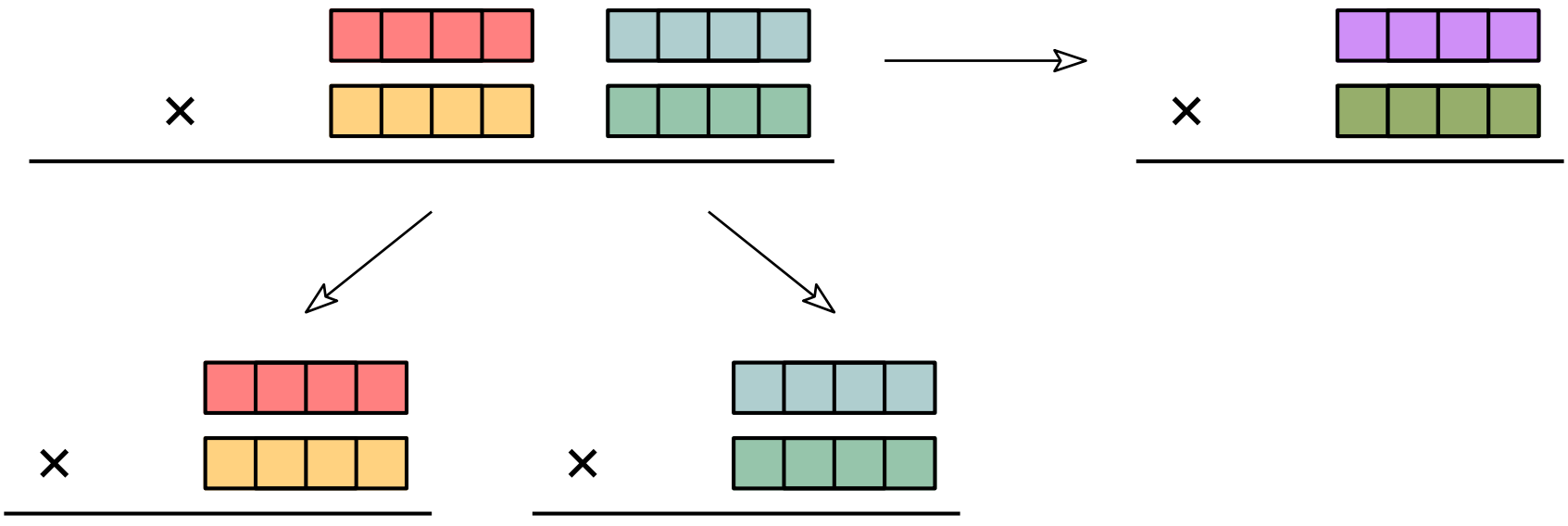
Multiplication par la méthode de Karatsuba

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



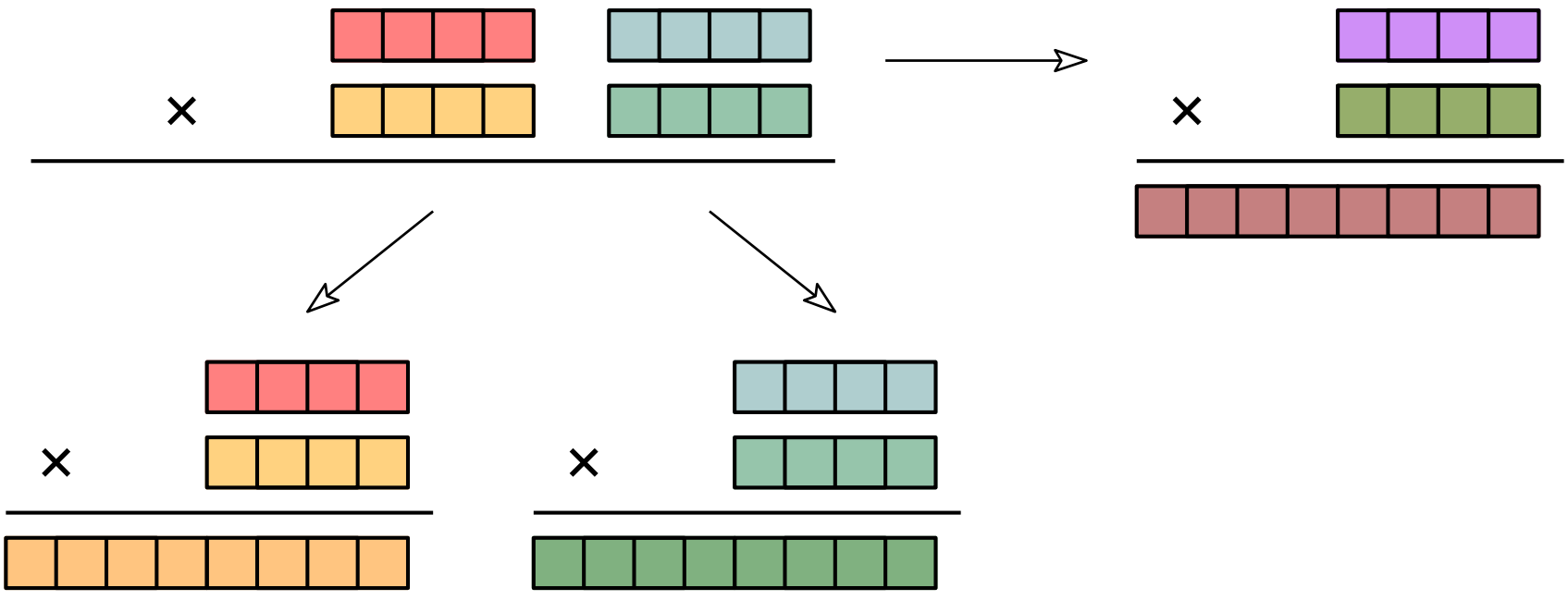
Multiplication par la méthode de Karatsuba

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



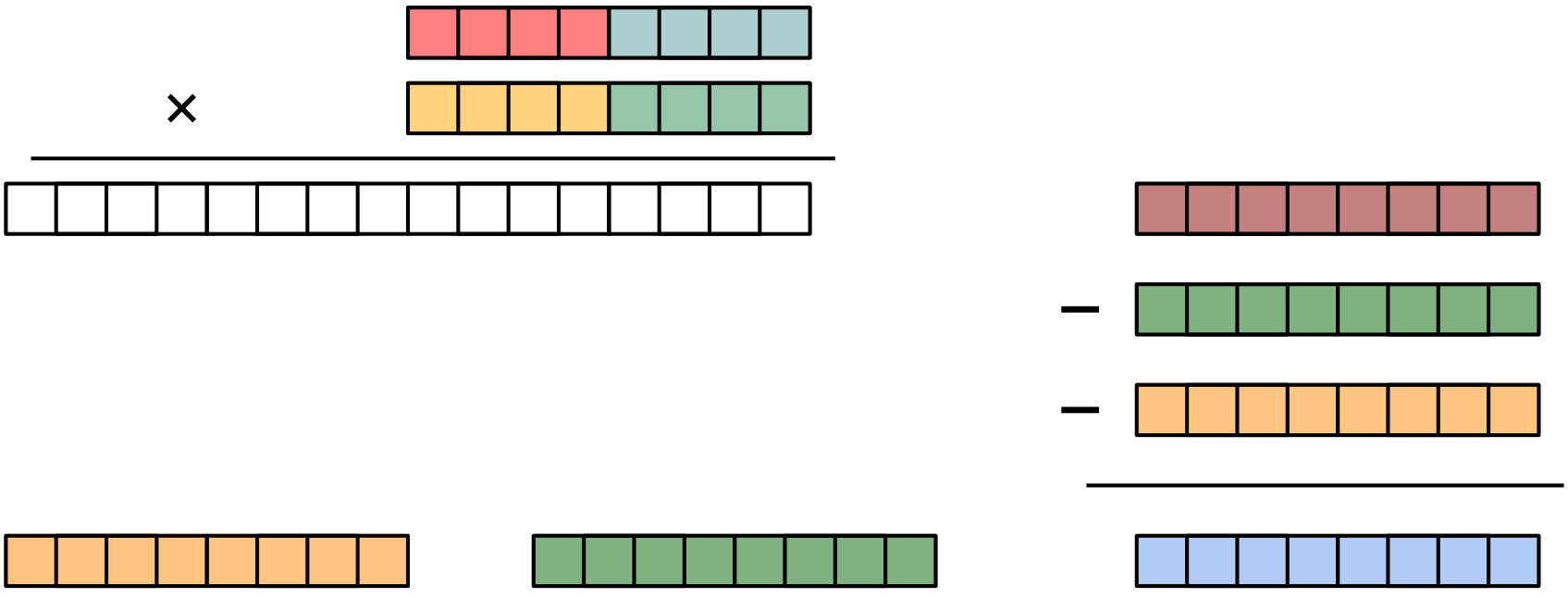
Multiplication par la méthode de Karatsuba

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



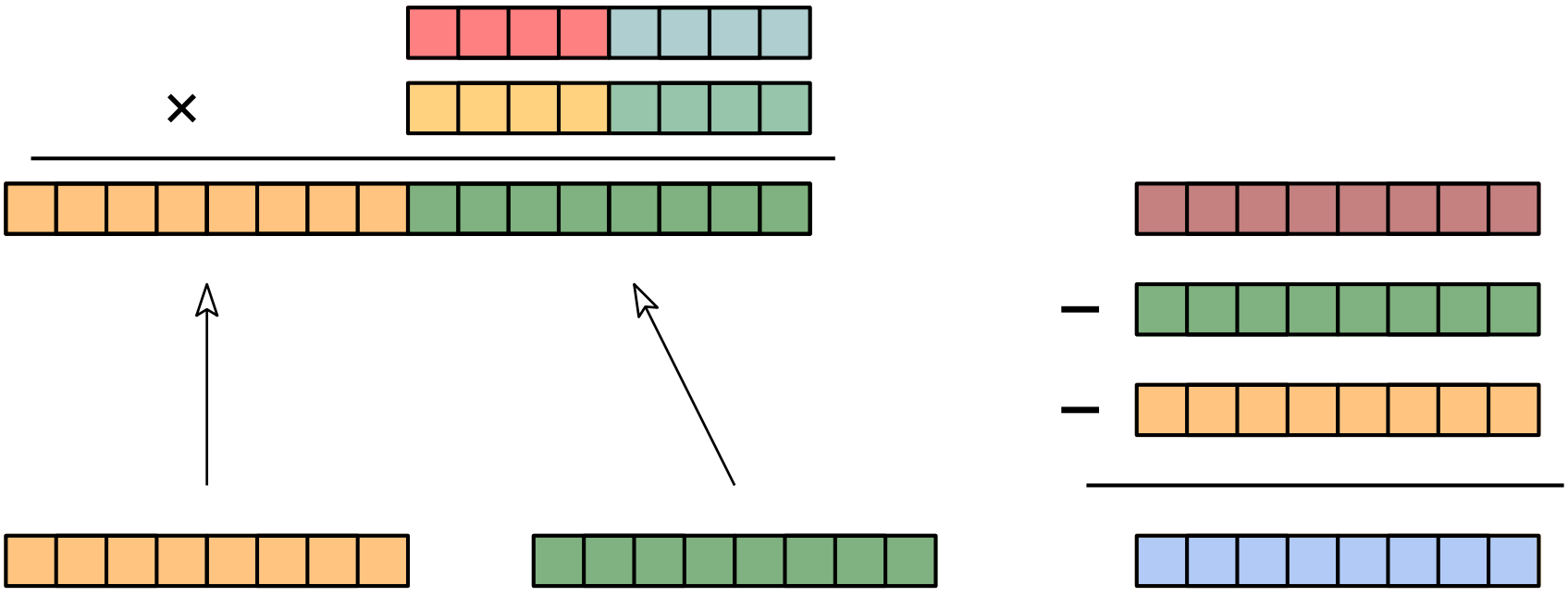
Multiplication par la méthode de Karatsuba

- Regroupement des résultats des appels récursifs



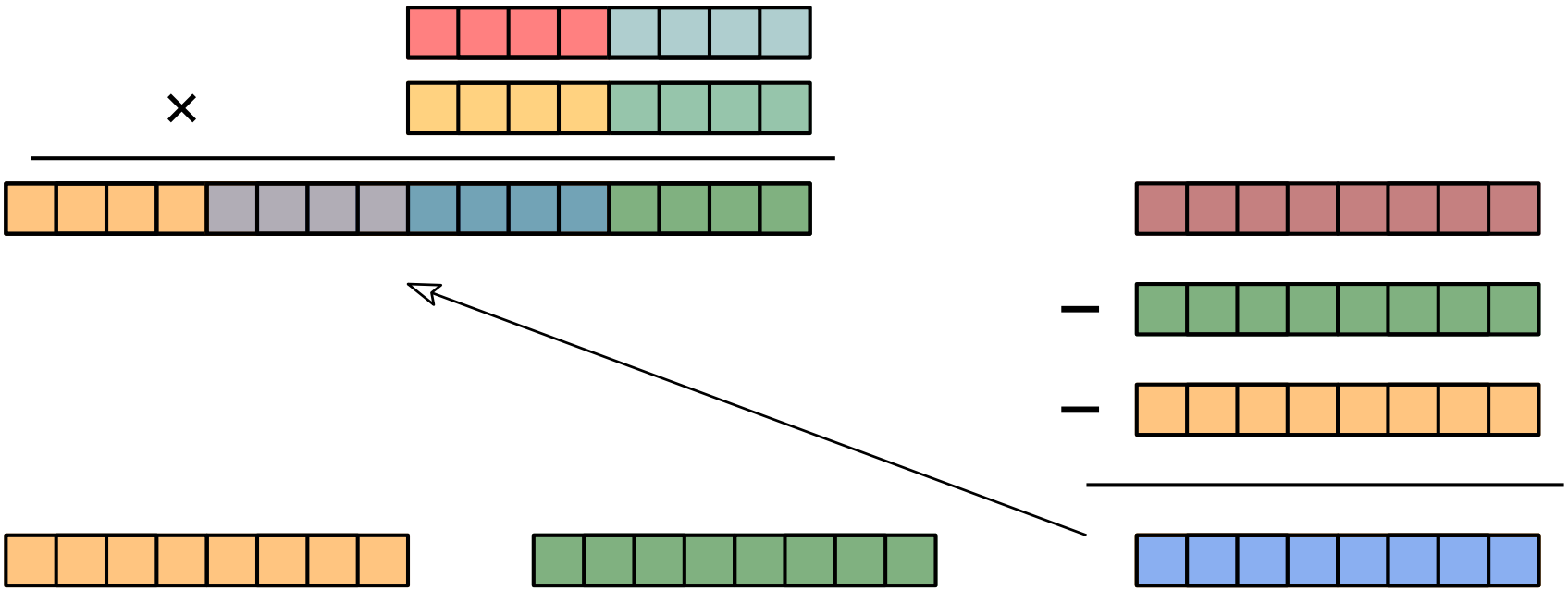
Multiplication par la méthode de Karatsuba

- Regroupement des résultats des appels récursifs



Multiplication par la méthode de Karatsuba

- Regroupement des résultats des appels récursifs



Multiplication par la méthode de Karatsuba

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

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

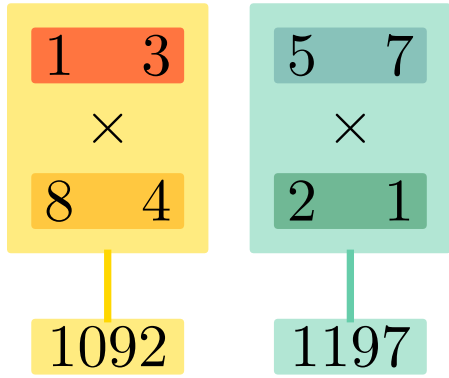
Multiplication par la méthode de Karatsuba

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

$$\begin{array}{rcl} & 1\ 3 \quad 5\ 7 & = 13 \cdot 100 + 57 \\ \times & 8\ 4 \quad 2\ 1 & = 84 \cdot 100 + 21 \\ \hline \end{array}$$

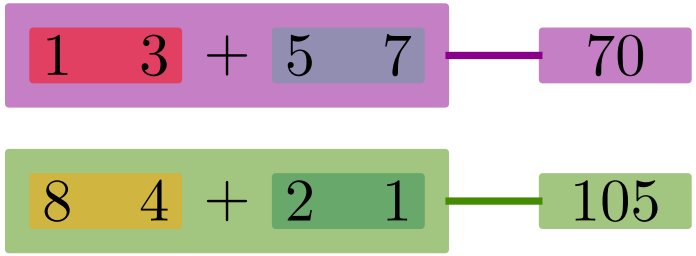
Multiplication par la méthode de Karatsuba

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



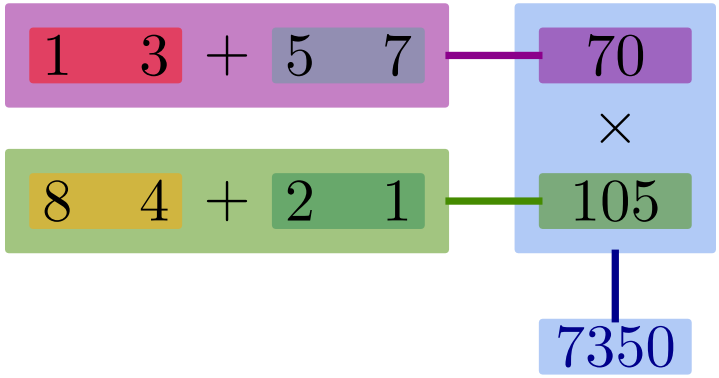
Multiplication par la méthode de Karatsuba

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



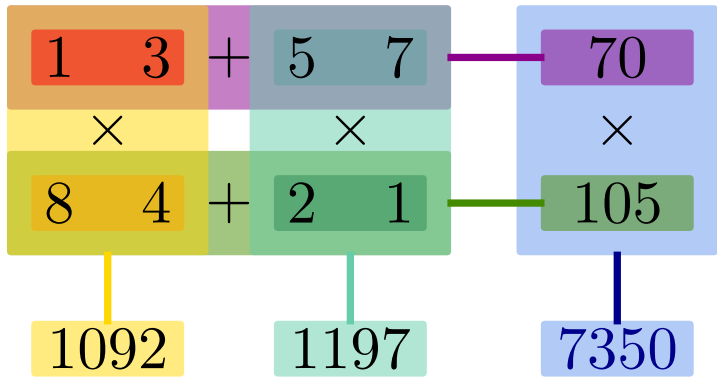
Multiplication par la méthode de Karatsuba

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 *récursivement* 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 $\frac{n}{2} = 2^{k-1}$ $\Longrightarrow$ $P^{(0)}, P^{(1)}, Q^{(0)}, Q^{(1)}$
- calculer $P^{(0)} + P^{(1)}$ et $Q^{(0)} + Q^{(1)}$ (2 sommes de taille $\frac{n}{2}$)

- résoudre *récursivement* 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 $\frac{n}{2} = 2^{k-1}$ $\Longrightarrow$ $P^{(0)}, P^{(1)}, Q^{(0)}, Q^{(1)}$
- calculer $P^{(0)} + P^{(1)}$ et $Q^{(0)} + Q^{(1)}$ (2 sommes de taille $\frac{n}{2}$)

- appeler récursivement `karatsuba` sur :
  - $(P^{(0)}, Q^{(0)})$ $\Longrightarrow$ $R^{(0)}$ (de taille n)
  - $(P^{(1)}, Q^{(1)})$ $\Longrightarrow$ $R^{(1)}$ (de taille n)
  - $(P^{(0)} + P^{(1)}, Q^{(0)} + Q^{(1)})$ $\Longrightarrow$ $R^{(2)}$ (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 $\frac{n}{2} = 2^{k-1}$ $\Longrightarrow$ $P^{(0)}, P^{(1)}, Q^{(0)}, Q^{(1)}$
- calculer $P^{(0)} + P^{(1)}$ et $Q^{(0)} + Q^{(1)}$ (2 sommes de taille $\frac{n}{2}$)

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

$$\Longrightarrow \quad C(n) = 3 \cdot C(\frac{n}{2}) + \Theta(n)$$

Multiplication par la méthode de Karatsuba

Complexité : elle se décompose en 2 parties :

- le coût des 3 appels récursifs 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)

$$\Longrightarrow \quad C(n) = 3 \cdot C(\frac{n}{2}) + \Theta(n)$$

Hypothèse : additions négligeables devant 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 récursifs 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)

$$\Longrightarrow \quad C(n) = 3 \cdot C(\frac{n}{2}) + \Theta(n)$$

Hypothèse : additions négligeables devant 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, puisqu'elles ont un coût cumulé en $O(n^{\log_2 3})$

Conclusion : complexité des calculs de F_n

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

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

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

Complexité pour deux entiers de n bits

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

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