

Cours n° 3 : diviser pour régner

Christophe Gonzales



But du cours n° 3

Diviser pour régner

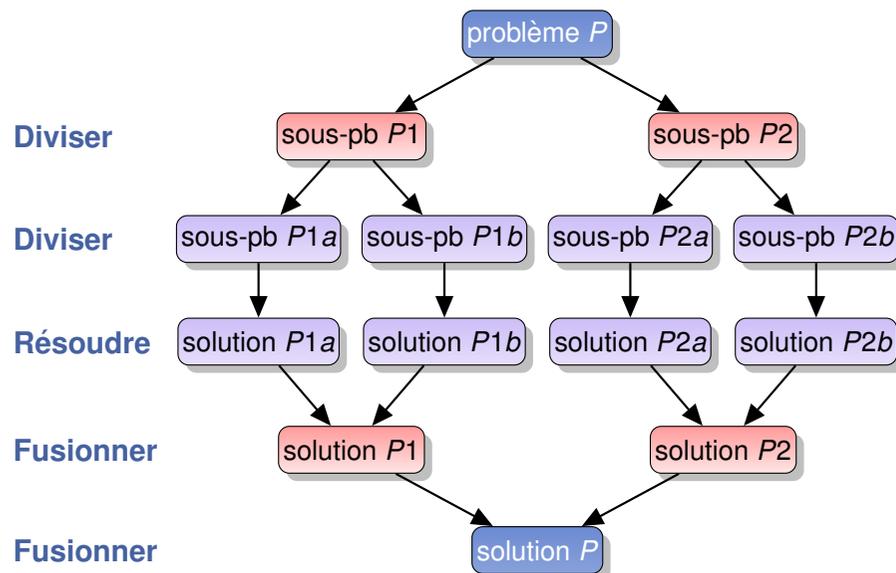
Schéma de résolution d'un problème qui utilise 3 étapes :

- 1 **Diviser** le problème en plusieurs sous-pbs plus petits
- 2 **Résoudre** les sous-problèmes récursivement et séparément
- 3 **Fusionner** les résultats pour obtenir la solution du pb d'origine

Illustrations du principe :

- ▶ Multiplication de matrices (Strassen)
- ▶ Tri rapide de tableaux
- ▶ Transformée de Fourier rapide (Cooley-Tukey)
- ▶ Géométrie algorithmique : recherche du couple de points les plus proches

Illustration graphique



Qu'est-ce qu'un bon « diviser pour régner » ?

Quelques conditions pour un « diviser pour régner » rapide :

- ▶ Séparations en sous-problèmes \approx de mêmes tailles
- ▶ Fonctions « terminales » (sans divisions) : rapides
- ▶ Fusion pas trop coûteuse

Théorème Maître

Complexité : théorème maître

- ▶ f fonction qui résout un problème de taille n en :
 - ▶ Divisant en $a > 0$ sous-problèmes de tailles n/b
 - ▶ Résolvant ces a sous-problèmes séparément
 - ▶ Fusionnant les résultats
 - ▶ Complexité de division + fusion en $O(n^d)$, $d \geq 0$
- ▶ Alors # d'opérations de $f = T(n) = a \times T(\lfloor n/b \rfloor) + O(n^d)$
- ▶ De plus, $T(n) \in \begin{cases} O(n^d) & \text{si } d > \log_b(a) \\ O(n^d \log n) & \text{si } d = \log_b(a) \\ O(n^{\log_b(a)}) & \text{si } d < \log_b(a) \end{cases}$

Exemple :

- ▶ Multiplication de 2 matrices carrées de taille n :
 - ▶ Algo naïf : $O(n^3)$ (cf. cours 1)
 - ▶ Strassen : $a = 7, b = 2, d = 2, \log_2(7) \approx 2,8 > d$
 \implies Complexité = $O(n^{\log_2(7)}) = O(n^{2,8})$

Multiplication de matrices de Strassen (1969)

Principe de Strassen

- ▶ Entrée : deux matrices A et B de tailles $n \times n$
- ▶ Sortie : la matrice produit $C = A \times B$
- ▶ Idée : scinder les matrices en 4 sous-blocs :



$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

\implies Sous-blocs A_{ij} et B_{ij} de tailles environ $\frac{n}{2} \times \frac{n}{2}$

- ▶ **Diviser** : Créer 7 matrices produits de certains de ces blocs
- ▶ **Résoudre** : Calculer ces 7 matrices M_i
- ▶ **Fusionner** : 6 additions et 2 soustractions des $M_i \implies C$

Rappel : Complexité en $O(n^{2,8})$ au lieu de $O(n^3)$

- ▶ $n = 1000 \implies n^3 \approx 4 \times n^{2,8}$

Les matrices M_i de Strassen et la matrice C

Les matrices M_i :

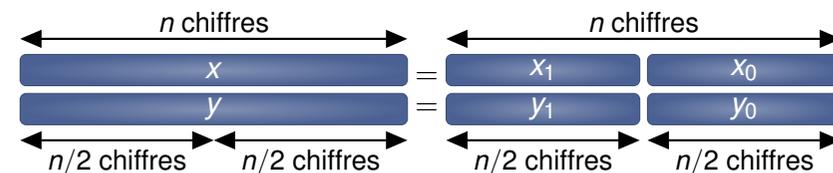
- ▶ $M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
- ▶ $M_2 = (A_{21} + A_{22}) \times B_{11}$
- ▶ $M_3 = A_{11} \times (B_{12} - B_{22})$
- ▶ $M_4 = A_{22} \times (B_{21} - B_{11})$
- ▶ $M_5 = (A_{11} + A_{12}) \times B_{22}$
- ▶ $M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
- ▶ $M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

$$C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix}$$

Multiplication de nombres sur n chiffres en base B

Principe de division :

$$B=10, x=1234 \implies x_1=12, x_0=34$$



- ▶ $x = x_1 B^{n/2} + x_0 \quad y = y_1 B^{n/2} + y_0$
- ▶ $xy = (x_1 B^{n/2} + x_0) \times (y_1 B^{n/2} + y_0) = z_2 B^n + z_1 B^{n/2} + z_0$
 - ▶ $z_2 = x_1 y_1$
 - ▶ $z_1 = x_1 y_0 + x_0 y_1$
 - ▶ $z_0 = x_0 y_0$

- ▶ 4 multiplications de tailles $n/2 \implies$ pas intéressant :

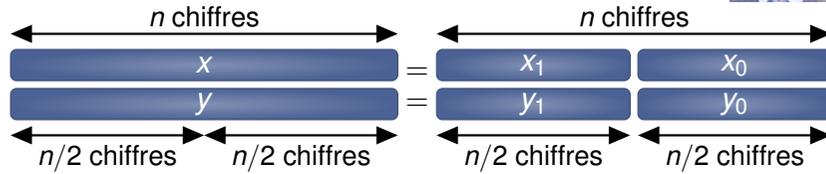
Théorème maître : $\#mult(n) = 4 \#mult(n/2) + \Theta(n)$

\implies complexité en $O(n^2)$

= complexité de la multiplication « classique »

Algorithme de multiplication de Karatsuba (1962)

Principe de division :



- ▶ $x = x_1 B^{n/2} + x_0$ $y = y_1 B^{n/2} + y_0$
- ▶ $xy = (x_1 B^{n/2} + x_0) \times (y_1 B^{n/2} + y_0) = z_2 B^n + z_1 B^{n/2} + z_0$
 - ▶ $z_2 = x_1 y_1$
 - ▶ $z_1 = (x_1 + x_0) \times (y_1 + y_0) - z_2 - z_0$
 - ▶ $z_0 = x_0 y_0$
 - ▶ Théorème maître : $\#mult(n) = 3 \#mult(n/2) + \Theta(n)$
 \implies complexité en $O(n^{\log_2(3)}) \approx 2^{1,58}$

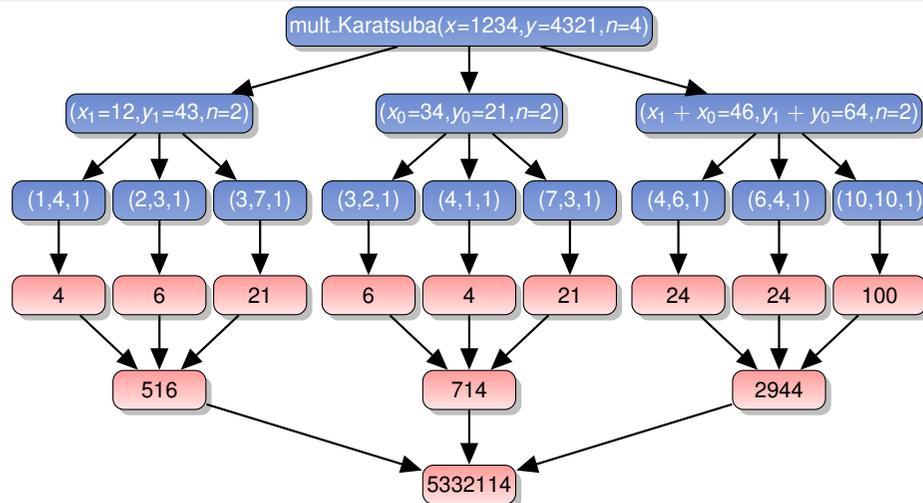
Algorithme de Karatsuba

Karatsuba en base $B = 10$:

```

1 fonction mult_Karatsuba (x, y, n) :
2   si n = 1 alors
3     retourner x × y
4   sinon
5     m ← n/2
6     x1 ← x / 10m # décalages de bits
7     x0 ← x % 10m # masquage de bits
8     y1 ← y / 10m
9     y0 ← y % 10m
10
11    z2 ← mult_Karatsuba (x1, y1, m)
12    z0 ← mult_Karatsuba (x0, y0, m)
13    t1 ← mult_Karatsuba (x1 + x0, y1 + y0, m)
14    z1 ← t1 - z2 - z0
15
16    retourner z2 × 10n + z1 × 10m + z0 # x : décalages de bits
17  finsi
    
```

Application de l'algorithme de Karatsuba



- ▶ $516 = 4 \times 10^2 + (21 - 6 - 4) \times 10^1 + 6$
- ▶ $714 = 6 \times 10^2 + (21 - 4 - 6) \times 10^1 + 4$
- ▶ $2944 = 24 \times 10^2 + (100 - 24 - 24) \times 10^1 + 24$
- ▶ $5332114 = 516 \times 10^4 + (2944 - 516 - 714) \times 10^2 + 714$

Recherche de l'élément maximal d'un tableau

```

1 fonction recherche_max (tableau T) : algorithme itératif
2   n ← longueur(T)
3   elt_max = T[0]
4   pour i variant de 1 à n-1 faire
5     si T[i] > elt_max alors
6       elt_max = T[i]
7   finsi
8   fait
9   retourner elt_max
    
```

Complexité pire cas =

```

1 fonction recherche_max_DPR (tableau T) : diviser pour régner
2   n ← longueur(T)
3   si n = 1 alors
4     retourner T[0]
5   sinon
6     m ← ⌊n/2⌋
7     elt1 ← recherche_max_DPR (sous-tableau T[0..m])
8     elt2 ← recherche_max_DPR (sous-tableau T[m+1..n-1])
9     retourner max (elt1, elt2)
10  finsi
    
```

Complexité pire cas =

\implies Diviser pour régner n'est pas toujours intéressant !

Tri de tableaux

- ▶ Idée générale des algorithmes de tri : permuter des éléments

⚠ Ne pas utiliser de tableau temporaire !
Trop coûteux en mémoire

⇒ Algorithmes par comparaisons des cellules du tableau

Algorithme naïf : tri par sélection

```
1 fonction tri_selection(tableau T) :
2   n ← longueur(T)
3   pour i variant de 0 à n-2 faire
4     min ← i # recherche du min des éléments de T[i..n-1]
5     pour j variant de i+1 à n-1 faire
6       si t[j] < t[min] alors
7         min ← j
8     finsi
9     fait
10    échanger t[i] et t[min] # placer le min dans T[i]
11  finfait
```

Complexité pire cas : $O(n^2)$

Théorème : Tout algo de tri par comparaisons est en $\Omega(n \log(n))$

Tri rapide de tableaux (quicksort)

Principe du tri rapide – Hoare 1961

- ▶ Partitionnement par comparaisons/permutations
- ▶ **Diviser** : placer par permutations un élément du tableau (*pivot*) à sa place définitive :
 - ▶ les éléments inférieurs sont à sa gauche
 - ▶ les éléments supérieurs sont à sa droite
- ⇒ diviser en 2 sous-tableaux
- ▶ **Résoudre** : trier les 2 sous-tableaux
- ▶ **Fusionner** : plus rien à faire

⚠ Plein de variations sur le choix du pivot !

```
1 fonction tri_rapide (tableau T,
2   index_min, # 1er elt du tableau
3   index_max) : # dernier elt du tableau
4   si index_min < index_max alors
5     pivot ← partition (T, index_min, index_max)
6     tri_rapide (T, index_min, pivot-1)
7     tri_rapide (T, pivot+1, index_max)
8   finsi
```

Partitionnement : algorithme de Lomuto

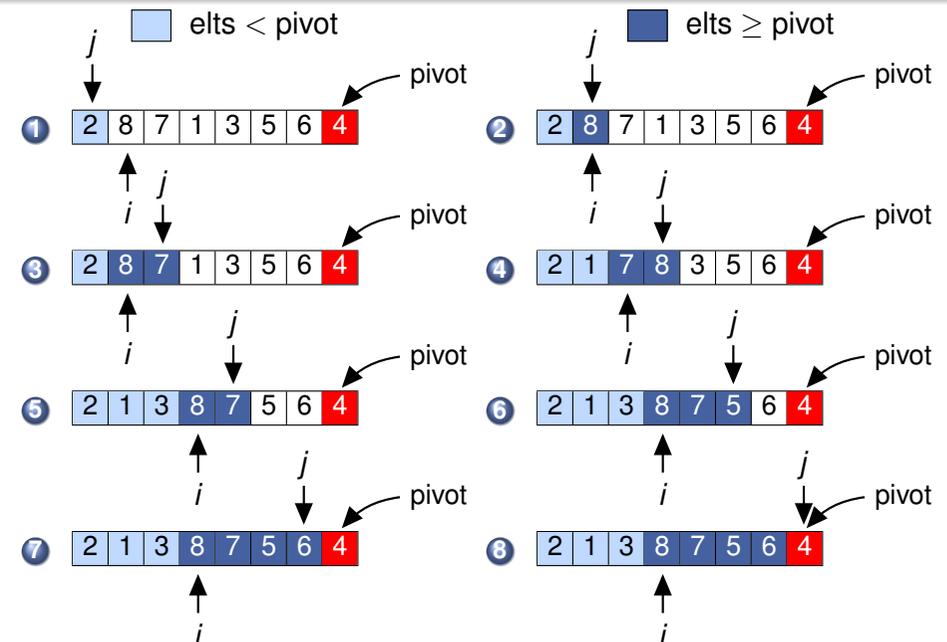
```
1 fonction partition(tableau T,
2   index_min, # 1er élément du tableau
3   index_max) : # dernier élément du tableau
4   pivot ← T[index_max]
5   i ← index_min
6   pour j variant de index_min à index_max faire
7     si T[j] < pivot alors
8       échanger T[i] et T[j]
9       i ← i + 1
10  finsi
11  fait
12  échanger T[i] et T[index_max]
13  retourner i
```

▶ Invariant de boucle :

(propriété vraie à la fin de chaque itération de la boucle)

- ▶ À gauche de i : elts < pivot
- ▶ Entre i et j : elts \geq pivot

Fin des itérations de la boucle de partitionnement



Complexité : pire cas

▶ Tableau déjà trié

⇒ partition génère :

- ▶ 1 sous-tableau de $n - 1$ cellules
- ▶ 1 sous-tableau de 1 cellule

$$T_{max}(n) = T(n-1) + \Theta(n) = \sum_{k=1}^n \Theta(k) = \Theta(n^2).$$

↑
partition

▶ Complexité pire cas = $\Theta(n^2)$

Complexité : cas moyen

$$T_{moy}(n) = \Theta(n \log(n)).$$

▶ Quicksort :

- ▶ Complexités : pire cas : $O(n^2)$, moyenne : $O(n \log(n))$
- ▶ Utilisations : `qsort` du langage C
`Array.sort` de Java 6

▶ Merge sort :

- ▶ Complexités : pire cas : $O(n \log(n))$, moyenne : $O(n \log(n))$
- ▶ Utilisations : `sort` de Python et Java 7 (Timsort)
`stable_sort` du C++

▶ Heap sort :

- ▶ Complexités : pire cas : $O(n \log(n))$, moyenne : $O(n \log(n))$

▶ Intro sort :

- ▶ Quicksort, se rabattant sur un heap sort si complexité augmente
- ▶ Complexités : pire cas : $O(n \log(n))$, moyenne : $O(n \log(n))$
- ▶ Utilisations : `sort` du C++

Conclusion

Diviser pour régner

- ▶ Technique très utilisée qui consiste à :
 - ▶ diviser un problème de taille n en sous-problèmes similaires plus petits
 - ▶ résoudre ces sous-problèmes
 - ▶ fusionner les solutions pour résoudre le problème initial
- ▶ Choix de division important
 - ⇒ conditionne la performance de l'algorithme
 - ⇒ découpage en sous-problèmes de tailles quasi-identiques
- ▶ Complexité : théorème maître
- ▶ Souvent plus simple à écrire en récursif
- ▶ Se prête bien à la parallélisation