Université de Bejaia.

Faculté des Sciences Exactes.

Département Informatique

Cours
Programmation Avancée

Master 1 en Informatique

Options : RN (RS & SIA)

Prof. BOUALLOUCHE Louiza

Chap. 5
Le paradigme « Diviser pour Régner »

Principe du paradigme « Diviser pour Régner »

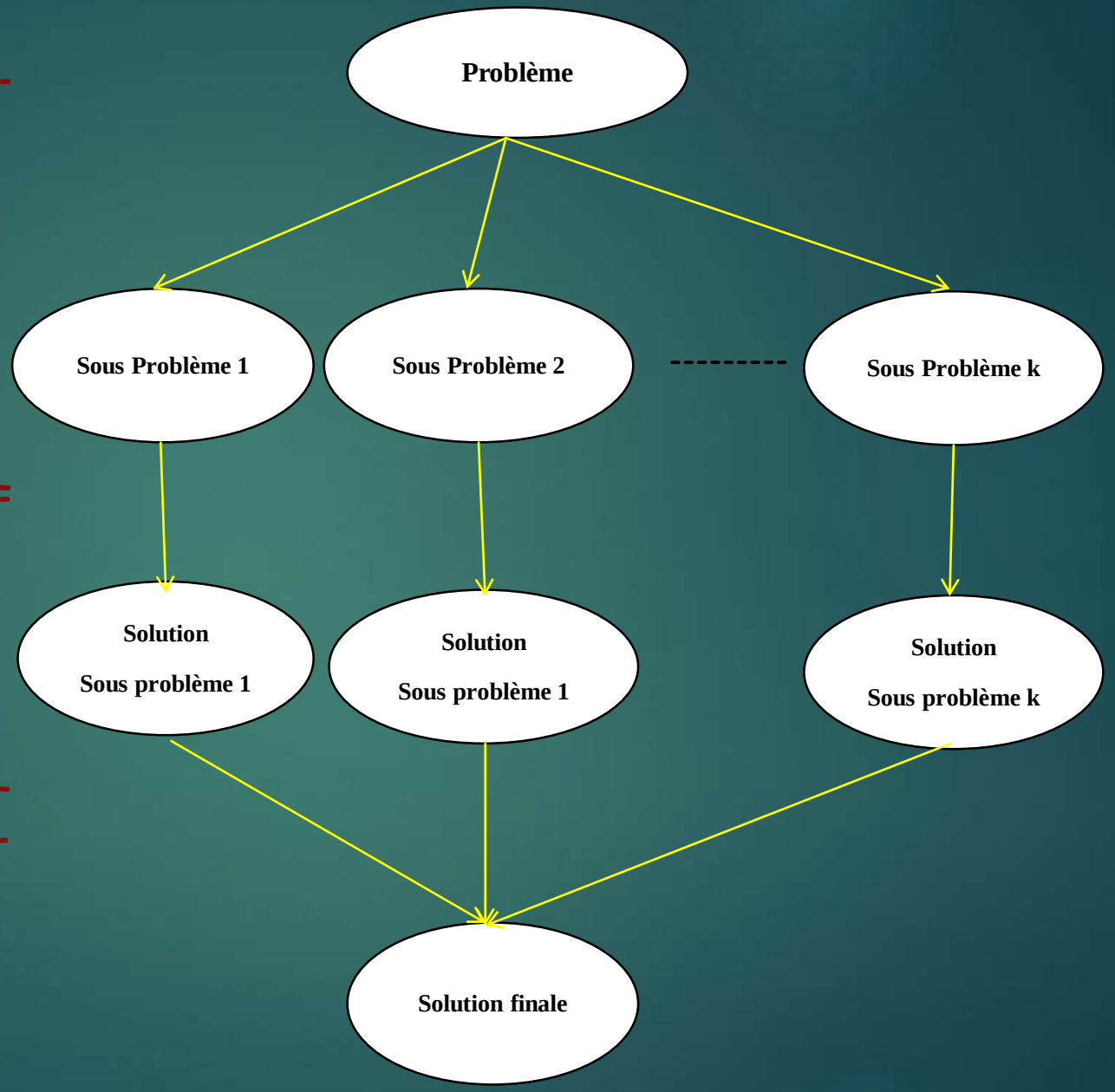
Il s'agit de résoudre un problème en résolvant deux ou plusieurs problèmes de même type. Le paradigme Diviser pour Régner (DpR) donne alors lieu à trois étapes à chaque niveau de récursivité. Il s'agit de :

- **Diviser le** problème en un certain nombre de sous-problèmes ;
- **Régner** sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;
- **Combiner** les solutions des sous-problèmes en une solution complète du problème initial.

Diviser: le Problème de taille *n* en *k* Sous Problèmes de taille *n/m*.

Régner: résoudre chaque Sous Problème récursivement.

Combiner: les solutions obtenues.



Intérêt du paradigme DpR. L'intérêt de l'application de ce paradigme, réside dans le fait de trouver le moyen de minimiser le nombre de sous-problèmes par rapport à la version DpR naïve ; ce qui minimiserait, de manière significative, le temps d'exécution de l'algorithme de résolution du problème initial.

Exercices d'application

Exercice 1. La somme des éléments d'un tableau.

Soit à écrire une fonction qui délivre la somme des éléments d'un tableau T de n éléments. On présente 2 versions, l'une itérative classique et l'autre récursive basée sur le paradigme DpR.

La fonction itérative

```
Fonction Som (T : Tab, n : entier): réel ;
 Var i: entier ;    S : réel ;
 Début
     S ← T[1] ;
     Pour i allant de 2 à n faire
             S ← S + T[i];
      Fin Pour
     Retourner S ;
Fin.
```

La complexité T(n) de la fonction en nombre d'additions est : $T(n) = n - 1$.

La fonction de type DpR qui calcule cette somme.

Principe.

On définit la fonction *SomDpR(T, i, j)* qui retourne la somme des éléments de *T* entre les positions *i* et *j*.

Le calcul de *SomDpR(T, i, j)* est effectué en divisant le tableau *T[i..j]* en deux moitiés; puis en calculant les sommes des moitiés ; et enfin en additionnant ces deux sommes.

La valeur recherchée est alors *SomDpR(T, 1, n)*. Autrement dit, les paramètres de la fonction au premier appel (dans le programme principal) sont *T*, 1, *n*

```
Fonction SomDpR(T: tab; i, j: entier): réel
  Var m: entier, SomG, SomD: réel ;
  Début
      Si i = j alors
        retourner T[i] ;
      Sinon
          m ← (i+j) div 2 ;
          SomG ← SomDpR (T, i, m) ;
          SomD ← SomDpR (T, m+1,j) ;
        Retourner (SomG + SomD) ;
  Fin si
Fin.
```

Calcul de la complexité *T(n)* en nombre d'additions, en utilisant la récurrence.

Il s'agit de résoudre le système de récurrence suivant

$T(1) = 0$

$T(n) = 2*T(n/2) + 1$

(on considère uniquement l'addition de *SomG* + *SomD*, les autres peuvent être négligées).

$$T(n) = 2*T(n/2) + 1 = 2(2*T(n/2^2)+1)+1 = 2^2*T(n/2^2)+2+1 = \ldots = 2^k*T(n/2^k)+2^{k-1}+2^{k-2}+\ldots+2^0$$

On s'arrête lorsque $n/2^k = 1$; *ce qui correspond à* $T(n/2^k) = 0$.

D'où $T(n) = 2^{k-1}+2^{k-2}+\ldots+2^0 = (1-2^k)/(1-2) = 2^k - 1 = \mathbf{n-1}$.

Conclusion. On ne gagne rien en appliquant l'algorithme DpR. au contraire la version récursive génère plus de calcul et occupe plus d'espace mémoire.

Peut-on faire mieux?

Exercice 2. Multiplication de deux matrices carrées

Algorithme naïf

```
Var i, j, k: entier ; A, B, C: matrice;
Début
  Pour i allant de 1 à n faire
    Pour j allant de 1 à n faire
      C[i,j] ←0 ;
      Pour k allant de 1 à n faire
        C[i,j] ← C[i,j] + A[i,k]* B[k,j]
      Fin Pour
    Fin Pour
  Fin Pour
Fin
```

Dans cet algorithme le nombre de multiplications est $T(n)= n^3 = O(n^3)$.

La complexité en nombre de multiplications et d'additions est aussi $T(n)= 2*n^3 = O(n^3)$.

Algorithme « Diviser pour Régner » classique naïf

On suppose que n est une puissance de 2. En fait, on peut toujours se ramener à ce cas en ajoutant des 0.

On décompose A, B et C en sous matrices de taille (n/2) X (n/2)

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \begin{bmatrix} e & g \\ f & h \end{bmatrix}$$

$C \qquad A \qquad B$

$$r = a \otimes e + b \otimes f$$

$$s = a \otimes g + b \otimes h$$

$$t = c \otimes e + d \otimes f$$

$$u = c \otimes g + d \otimes h$$

$C = A \otimes B$ où le symbole $\otimes$ représente la multiplication matricielle.

Aussi, les symboles $+$ et $-$ représentent l'addition et la soustraction matricielles respectivement.

La relation de récurrence de la complexité T(n) de cet algorithme en nombre d'additions et de multiplications?

Le nombre d'additions est de l'ordre $\Theta(n^2)$. La relation de récurrence associée à T(n) est alors

$$T(n)=8*T(n/2)+\Theta(n^2)$$

À résoudre…

Analyse de la complexité des algorithmes « Diviser pour Régner »

Il y a une relation de récurrence pour le temps d'exécution

1. Si la taille *n* du problème est suffisamment réduite, $n \leq c$ pour une certaine constante *c*, la résolution est directe et consomme un temps constant *O (1)*.

2. Sinon, on divise le problème en *k* sous-problèmes chacun de taille *n/m*. Le temps d'exécution total se décompose alors en trois parties :

   a. ***D(n)*** : Temps nécessaire à la division du problème en sous-problèmes.
   b. ***K*T(n/m)*** : Temps de résolution des *k* sous-problèmes.
   c. ***C(n)*** : Temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

La relation de récurrence prend alors la forme :

$$T(n) = \begin{cases} \Theta(1) & si\ n \leq c \\ k * T(n/m) + D(n) + C(n) & sinon \end{cases}$$

Où *n*/m est soit comme $\lfloor n/m \rfloor$ soit comme$\lceil n/m \rceil$.

Résolution des récurrences « Diviser pour Régner »

Théorème.

Soient k >= 1 et m > 1 deux constantes, soient f (n) et T(n) des fonctions définies pour les entiers positifs par la récurrence :

$$T(n) = k*T(n/m)+ f(n);$$

n/m est interprété soit comme $\lfloor n/m \rfloor$ *soit comme* $\lceil n/m \rceil$.

T(n) peut être borné asymptotiquement comme suit :

1. *Si* $f(n) = O(n^{\log_m k - \xi})$ *pour une certaine constante* $\xi > 0$, *alors* $T(n) = \Theta(n^{\log_m k})$.
2. *Si* $f(n) = \Theta(n^{\log_m k})$, *alors* $T(n) = \Theta(n^{\log_m k} * \log n)$.
3. *Si* $f(n) = \Omega(n^{\log_m k+\xi})$ *pour une certaine constante* $\xi > 0$, *et si* $k* f(n/m) \leq c* f(n)$ *pour une constante* $c < 1$ *et n suffisamment grand, alors* $T(n) = \Theta(f(n))$.

Application sur l'algorithme DpR naïf de la multiplication de 2 matrices

La complexité de l'algorithme DpR naïf de C = A∗B est $T(n)=8* T(n/2)+ \Theta(n^2)$

$K = 8,\ m = 2,\ f(n) = \Theta(n^2)$.

Vérification du cas 1 du théorème.

Si $f(n) = O(n^{\log_m k-\xi})$ pour une certaine constante $\xi > 0$, alors $T(n) = \Theta(n^{\log_m k})$.

$\log_2 8 = 3$, Si $\xi = 1$ $f(n) = \Theta(n^{\log_2 8-1}) = \Theta(n^2)$

$T(n) = \Theta(n^{\log_m k}) = \Theta(n^{\log_2 8})$

$\boldsymbol{T(n) = \Theta(n^3)}$

Conclusion. La complexité de cet algorithme est la même que la complexité de l'algorithme classique naïf.

Algorithme de Strassen pour la multiplication de matrices

Pr. Volker Strassen propose une algorithme « Diviser pour Régner » qui n'effectue que 7 multiplications de matrices, au lieu de 8 dans l'algorithme DpR naïf, mais qui effectue plus d'additions et de soustractions de matrices (l'addition ou la soustraction est négligeable devant une multiplication).

La relation de récurrence de la complexité de l'algorithme de Strassen est alors

$T(n) = 7 * T(n/2) + \Theta(n^2)$

Algorithme de Strassen

L'algorithme de Strassen se décompose en quatre étapes :

1. Diviser les matrices A et B en matrices carrés de taille n/2.
2. Calculer 14 matrices A_1, ..., A_7, B_1, ..., B_7 carrés de taille n/2, au moyen de $\Theta(n^2)$ additions et soustractions scalaires.
3. Calculer récursivement les 7 produits de matrices $M_i = A_i \otimes B_i$, $i \in [1..7]$.
4. Calculer les sous-matrices désirées r, s, t et u en additionnant et/ou soustrayant les combinaisons idoines des matrices *Ad hoc* avec $\Theta(n^2)$ additions et soustractions scalaires.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad B = \begin{bmatrix} e & g \\ f & h \end{bmatrix}, \quad C = \begin{bmatrix} r & s \\ t & u \end{bmatrix}$$

$M1 = A_1 \circledast B_1 = (a + d) \circledast (e + f)$

$M2 = A_2 \circledast B_2 = (c + d) \circledast e$

$M3 = A_3 \circledast B_3 = a \circledast (g - h)$

$M4 = A_4 \circledast B_4 = d \circledast (f - e)$

$M5 = A_5 \circledast B_5 = (a + b) \circledast h$

$M6 = A_6 \circledast B_6 = (c - a) \circledast (e + g)$

$M7 = A_7 \circledast B_7 = (b - d) \circledast (f + h)$

Finalement

$r = M1 + M4 - M5 + M7$

$s = M3 + M5$

$t = M2 + M4$

$u = M1 - M2 + M3 + M6$

Résolution de la relation de récurrence de la complexité :

$T(n)= 7* T(n/2)+ \Theta(n^2)$ $K = 7,\ m = 2,\ f(n) = \Theta(n^2)$.

Si $f(n) = O(n^{\log_m k-\xi})$ *pour une certaine constante* $\xi > 0$, *alors* $T(n) = \Theta(n^{\log_m k})$.

$\log_2 7 = 2.807$, Si $\xi = 0.807$ $f(n) = \Theta(n^{\log_2 2.807-0.807}) = \Theta(n^2)$

$T(n)= \Theta(n^{\log_m k}) = \Theta(n^{\log_2 7})$

$\boldsymbol{T(n)= \Theta(n^{2.807})}$

Conclusion. Cet algorithme est plus efficace que les 2 versions précédentes (Classique et DpR naïf) de complexité $\boldsymbol{T(n)= \Theta(n^3)}$