Chap 4. Diviser pour régner

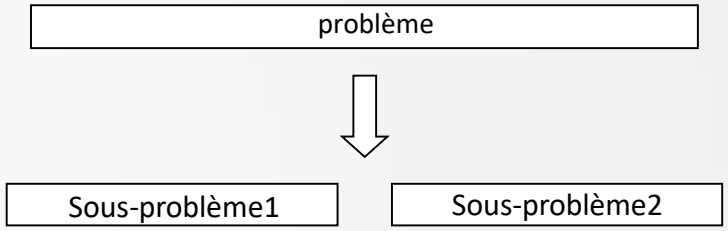
Année universitaire :2020/2021

1 Diviser pour régner : Principe

- De nombreux algorithmes ont une structure récursive: pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres, résolvent les sous problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.

- Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :
- **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.

2 Exemple_1 : Recherche du maximum d'un tableau




```
Maximum (x, left, right)
If ( left = right)
   return left;
   Else
m= (left+right)/2          // division du problème en 2 sous-problèmes
k1 = maximum (x, left, m)      // régner sur le 1er sous-problème
k2 = maximum (x, m+1, right) // régner sur le 2ème sous-problème
If(x (k1) > x (k2))                              // combiner les solutions
          Return k1
          Else
          Return k2
End_maximum
```

3 Eexemple_2 : multiplication naïve de matrices

Nous nous intéressons ici à la multiplication de matrices carrées de taille *n*.

Algorithme naïf

- L'algorithme classique est le suivant :

➢ Soit *n* la taille des matrices carrés *A* et *B*

➢ Soit *C* une matrice carrée de taille *n*

Multiplier (*A*, *B,C*)
For i = 1 to n do
  For j=1 to n do
  C(i,j)=0
For k=1 to n do
C(i,j)= C(i,j) +A(i,k)*B(k,j)
End_Multiplier

Cet algorithme effectue **$O(n^3)$** multiplications et autant d'additions.

3 Eexemple_2 : multiplication naïve de matrices

Algorithme « diviser pour régner » naïf

- Dans la suite nous supposerons que n est une puissance de 2. Décomposons les matrices A, B et C en sous-matrices de taille $n/2$ x $n/2$. L'équation $C = AB$ peut alors se récrire :

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

- En développant cette équation, nous obtenons :

  $r = ae+bf$; $s = ag+bh$; $t = ce+df$ et $u = cg+dh$:

- Chacune de ces quatre opérations correspond à :

➢ deux multiplications de matrices carrées de taille $n/2$ ➔ $2T(n/2)$

➢ et une addition de telles matrices ➔ $O(n^2)$

- A partir de ces équations on peut aisément dériver un algorithme « diviser pour régner » dont la complexité est donnée par la récurrence :

  $T(n) = 8T(n/2)+O(n^2)$

4 Analyse des algorithmes «diviser pour régner» (1)

- Lorsqu'un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une équation de récurrence qui décrit le temps d'exécution global pour un problème de taille *n* en fonction du temps d'exécution pour des entrées de taille moindre.

4 Analyse des algorithmes «diviser pour régner» (2)

- La récurrence définissant le temps d'exécution d'un algorithme « diviser pour régner » se décompose suivant les trois étapes du paradigme de base :

1. Si la taille du problème est suffisamment réduite, $n <= c$ pour une certaine constante c, la résolution est directe et consomme un temps constant O(1).
2. Sinon, on divise le problème en a sous-problèmes chacun de taille $1/b$ de la taille du problème initial. Le temps d'exécution total se décompose alors en trois parties :

   - $D(n)$: le temps nécessaire à la division du problème en sous-problèmes.

   - $aT(n/b)$: le temps de résolution des a sous-problèmes.

   - $C(n)$: le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

- La relation de récurrence prend alors la forme :

$$T(n) = \begin{cases} \mathrm{O}(1) & si \quad n \le c \\ aT(n/b) + D(n) + C(n) & \sin on \end{cases}$$

Notations de Landau (1)

- Quand nous calculerons la complexité d'un algorithme, nous ne calculerons généralement pas sa complexité exacte, mais son ordre de grandeur.

- Pour ce faire, nous avons besoin de notations asymptotiques connues sous le nom de « notation de Landau » (Mathématicien Allemand).

Notations de Landau (2)

$$\mathrm{O}: \quad f = \mathrm{O}(g) \quad \Leftrightarrow \quad \exists n0, \exists c \geq 0, \quad / \quad \forall n \geq n0, \quad f(n) \leq c.g(n)$$

- On dit aussi que f « est dominée asymptotiquement par g »
- Ou que g « est un majorant presque partout de f »

Exemples

- $n = O(n)$, $2n = O(3n)$, $n+2 = O(n)$ (Il suffit de prendre $n0 = 2$ et $c = 2$)
- sqrt(n) = $O(n)$, $\log(n) = O(n)$, $n = O(n^2)$.

Notations de Landau (3)

$$\Omega: \quad f = \Omega(g) \quad \Leftrightarrow \quad g = \mathrm{O}(f)$$

- Cela signifie que g « est majorée par f »

$$o: \quad f = o(g) \quad \Leftrightarrow \quad \forall c \geq 0, \exists n0 \quad / \quad \forall n \geq n0, \quad f(n) \leq c.g(n)$$

- Ceci signifie que : f « est négligeable devant g »

Exemples

- sqrt(n) = $o(n)$, log(n) = $o(n)$, $n = o(n^2)$, log(n) = o(sqrt(n)).

Notations de Landau (4)

$$\Theta : \quad f = \Theta(g) \Leftrightarrow f = \mathrm{O}(g) \quad et \quad g = \mathrm{O}(f)$$

- Ceci signifie que :

$$f = \Theta(g) \Leftrightarrow \quad \exists c, d \in R^{+}, \exists n0 / \forall n > n0 \quad d.g(n) \leq f(n) \leq c.g(n)$$

- Ce qui veut dire que chaque fonction est un majorant de l'autre ou encore que les deux fonctions sont « de même ordre de grandeur asymptotique »

Exemples : n+log(n) = Θ (n+sqrt(n)).

Remarques

- On peut considérer que :
- $\mathrm{O}(g)$ est une borne supérieure
- $\Omega(g)$ est une borne inférieure
- $\Theta(g)$ est une borne exacte. Celle-ci est donc plus précise que les précédentes

Résolution des récurrences

5 Équations de récurrence linéaires (1)

Exemple : La suite de Fibonacci est définie par :

$$u_n = \begin{cases} 1 & si \quad n < 2 \\ u_{n-1} + u_{n-2} \sin on \end{cases}$$

- Un algorithme récursif pour calculer le n^{ème} terme de la suite est :

```
Fibonacci ( n)
   If ( n=0  or n =1 )
            Return 1
            Else
            Return (Fibonacci (n-2) + Fibonacci (n-1))
End_Fibonacci
```

5 Équations de récurrence linéaires (2)

- Soit T(n) le nombre d'additions effectuées par cet algorithme. T(n) vérifie les équations suivantes :

$$T(n) = \begin{cases} 0 & si \quad n = 0 \\ 0 & si \quad n = 1 \\ 1 + T(n-1) + T(n-2) & si \quad n \geq 2 \end{cases}$$

- Cette équation est linéaire et d'ordre 2 car chaque terme de rang >1 dépend uniquement des deux termes qui le précèdent.

5 Équations de récurrence linéaires (3)

Définition : Une équation récurrente est dite linéaire d'ordre k si chaque terme s'exprime comme combinaison linéaire des k termes qui le précèdent plus une certaine fonction de n.

$$u_n = \alpha_1 . u_{n-1} + \alpha_2 . u_{n-2} + \ldots + \alpha_k . u_{n-k} + f(n)$$

- Il faut bien sûr connaître les k premiers termes.

- A l'ordre 1, l'équation devient : $u_n = a . u_{n-1} + f(n)$

- Par itération et sommation on trouve : $u_n = a^n (u_0 + \sum_{i=1}^{n} \frac{f(i)}{a^i})$

Exemple : Problème des tours de Hanoi

- Le coût de l'algorithme récursif est :

$$T(n) = \begin{cases} 0 & si \quad n = 0 \\ 2T(n-1)+1 & si \quad n \geq 1 \end{cases}$$

T(n) est donc de la forme : $u_n = 2.u_{n-1} + 1$

- Cette équation peut être résolue par la méthode itérative :

$$u_n = 2.u_{n-1} + 1$$

$$u_n = 2(2u_{n-2}+1)+1 = 2^2 u_{n-2} + 2 + 1 = 2^2 u_{n-3} + 2^2 + 2 + 1$$

...

$$u_n = 2^n u_0 + 2^{n-1} + \ldots + 2^2 + 2 + 1 = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

- Si on reprend la formule :

$$u_n = a^n (u_0 + \sum_{i=1}^{n} \frac{f(i)}{a^i})$$

et on remplace a par 2 et f(i) par 1 on trouve le même résultat (car la somme d'une suite géométrique=u0(q^n-1)/(q-1))

$$u_n = 2^n (u_0 + \sum_{i=1}^{n} \frac{1}{2^i}) = 2^n (\sum_{i=0}^{n} \frac{1}{2^i} - 1) = 2^n - 1$$

6 Equation de récurrence linéaire sans second membre (1)

- Une équation de récurrence est dite sans second membre si f(n)=0

$$u_n - \alpha_1 . u_{n-1} - \alpha_2 . u_{n-2} - \ldots - \alpha_k . u_{n-k} = 0$$

- A une telle équation, on peut associer un polynôme caractéristique :

$$P(x) = x^k - \alpha_1 x^{k-1} - \alpha_2 x^{k-2} - \ldots - \alpha_k$$

- La résolution de ce polynôme nous donne m racines r_i (avec m<=k).
- La solution de l'équation de récurrence est ainsi donnée par :

$$u_n = Q_1(n) r_1^n + Q_2(n) r_2^n + \ldots + Q_m(n) r_m^n$$

- Cette solution est en général exponentielle.

6 Equation de récurrence linéaire sans second membre (2)

Exemple 1: Reprenons l'exemple de la suite de Fibonacci. Son coût est donné par :

$$T(n)=\begin{cases}0 & si \quad n=0\\ 0 & si \quad n=1\\ 1+T(n-1)+T(n-2) & si \quad n\geq 2\end{cases}$$

- En posant : $S(n)=T(n)+1$ on obtient :

$$S(n)=\begin{cases}1 & si \quad n=0\\ 1 & si \quad n=1\\ S(n-1)+S(n-2) & si \quad n\geq 2\end{cases}$$

- Donc une équation de récurrence sans second membre. $S(n)-S(n-1)-S(n-2)=0$
- Son polynôme caractéristique est : $x^2-x-1=0$
- Il possède deux racines : $\frac{1+\sqrt{5}}{2}$ et $\frac{1-\sqrt{5}}{2}$

- La solution de l'équation de récurrence est donc : $S(n)=a\left(\frac{1+\sqrt{5}}{2}\right)^n+b\left(\frac{1-\sqrt{5}}{2}\right)^n$

- Les coefficients a et b peuvent être déterminés à l'aide des conditions aux limites : $S(0)=1=a+b$ $S(1)=1=\frac{a+b}{2}+\frac{(a-b)\sqrt{5}}{2}$

d'où : $a=\frac{5+\sqrt{5}}{10} \quad et \quad b=\frac{5-\sqrt{5}}{10}$

et enfin : $S(n)=\frac{5+\sqrt{5}}{10}\left(\frac{1+\sqrt{5}}{2}\right)^n+\frac{5-\sqrt{5}}{10}\left(\frac{1-\sqrt{5}}{2}\right)^n$

Donc : $S(n)=T(n)=\mathrm{O}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$

Exemple-2

- Soit la suite définie par : $\begin{cases} u_n = 3u_{n-1} + 4u_{n-2} & si \quad n \geq 2 \\ u_0 = 0 \quad et \quad u_1 = 1 \end{cases}$
- Son polynôme caractéristique est : $P(x) = x^2 - 3x - 4$
- Ses racines sont : $r_1 = 4 \quad et \quad r_2 = -1$
- La solution de l'équation de récurrence est donc : $u_n = a4^n + b(-1)^n$
- Les conditions aux limites donnent :

$$\left.\begin{cases} u(0) = a + b = 0 \\ u(1) = 4a - b = 1 \end{cases}\right\} \Rightarrow a = \frac{1}{5} \quad et \quad b = -\frac{1}{5}$$

- d'où : $u(n) = \frac{1}{5}(4^n - (-1)^n) = \mathrm{O}(4^n)$

7 Résolution des récurrences « diviser pour régner »

Théorème 1 (Résolution des récurrences « diviser pour régner »)

- *Soient a* >=1 *et b* > 1 *deux constantes,*

  soit f (*n*) *une fonction et soit T*(*n*) *une fonction définie pour les entiers positifs par la récurrence :*

$$T(n) = a.T(n/b) + f(n)$$

- *T*(*n*) *peut alors être bornée asymptotiquement comme suit :*

	Si la forme de f(n) est :	Alors le coût est :
1	$f(n) = O(n^{(\log_b a)-\varepsilon})$ pour un $\varepsilon > 0$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
3	$f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ pour un $\varepsilon > 0$, et $af(n/b) \le cf(n)$ pour un $c < 1$ et n suffisamment grand	$T(n) = \Theta(f(n))$

Version-2 du théorème

$T(n) = a.T(n/b) + f(n)$ OU $T(n) = a.T(n/b) + c.n^k$

	Si on a :	Alors le coût est :
1	$a > b^k$	$T(n) = \Theta(n^{\log_b a})$
2	$a = b^k$	$T(n) = \Theta(n^k \log n)$
3	$a < b^k$	$T(n) = \Theta(n^k)$

Signification intuitive du théorème

- Dans chaque cas, on compare $f(n)$ $avec$ $n^{\log_b a}$. La solution de la récurrence est déterminée par la plus grande des deux.

Exemple-1 : Retour sur l'exemple de la multiplication de matrices

- La relation de récurrence est : $T(n) = 8T(n/2)+O(n^2)$

- Donc : $a = 8$,
- $b = 2$ ➔ $\log_b a = 3$
- $f(n) = \Theta(n^2) = \Theta(n^{\log_b a - 1})$
- On est donc dans le cas 1 du théorème (avec $\varepsilon = 1$), l'algorithme a donc une complexité en $\Theta(n^3)$

Exemple-2

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

- On est dans le cas : a=9 , b=3 et f(n)=n

→ $n^{\log_b a} = n^{\log_3 9} = n^2 = \Theta(n^2)$ → $f(n) = n = n^{2-1} = n^{\log_3 9 - \varepsilon}$ $avec$ $\varepsilon = 1$

→ On applique le cas 1 :

$$T(n) = \Theta(n^2)$$

Exemple-3

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

- On est dans le cas : a=1 , b=3/2 et f(n)=1

→ $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = 1$ → $f(n) = \Theta(n^{\log_b a})$

- On applique donc le cas 2 :

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$$

Exemple-4

$$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$$

- On est dans le cas : a=3 , b=4 et f(n)=nlogn

→ $n^{\log_b a} = n^{\log_4 3} = \mathrm{O}(n^{0,793})$ → $f(n) = \Omega(n^{\log_4 3+\varepsilon})$ $avec$ $\varepsilon \cong 0.2$

- Pour n suffisamment grand on a :

$$af\left(\frac{n}{4}\right) = 3\frac{n}{4}\log\left(\frac{n}{4}\right) \le \frac{3}{4}n\log n = c.f(n) \quad avec \quad c = \frac{3}{4}$$

- On applique donc le cas 3 :

$$T(n) = \Theta(n\log n)$$

8 Autres récurrences

- D’autres récurrences peuvent ne pas trouver de solution avec les techniques présentées. Dans ce cas, on peut procéder par itération :

  - calculer les quelques premières valeurs de la suite
  - chercher une régularité
  - poser une solution générale en hypothèse
  - la prouver par induction