Chapitre 3

Analyse et Complexité des Algorithmes Récursifs

I. Introduction & Définitions:

Un algorithme (ou Procédure/fonction) est dit récursif s'il est défini en fonction de lui-même.

La solution est construite en recommençant l'exécution de la même fonction/procédure un certain nombre de fois.

On parle de plusieurs types de récursivité:

- Récursivité Simple: la fonction récursive contient un seul appel récursif

```
procédure proc(XX)
début
        ⋮
        proc(YY)
        ⋮
fin
```

Exemple

```
Fonction   Puissance ( x, n : entier ): entier
Début
        si  n = 0     alors   retourner ( 1 )
        sinon         retourner ( x * Puissance ( x, n-1) )
        fsi
Fin
```

- Récursivité Multiple: la fonction récursive contient plus d'un appel récursif.

```
procédure proc(XX)
début
        ⋮
        proc(YY)
        proc(ZZ)
        ⋮
fin
```

Exemple

```
Fonction  Combinaison ( n, p : entier ): entier

Début

si  p = 0     alors   retourner ( 1 )

sinon
   retourner ( Combinaison ( n-1,p) + Combinaison ( n-1, p-1) )
Fsi
Fin
```

- **Récursivité Mutuelle (ou croisée)**: deux fonctions mutuellement récursives si l'une fait appel à l'autre et réciproquement.

```
procédure proc1(XX)
début
      :
      proc2(YY)
      :
fin
procédure proc2(XX)
début
      :
      proc1(YY)
      :
fin
```

```
Fonction  Pair ( n: entier ): Booléen

Début

si  n = 0     alors   retourner ( Vrai )

sinon
   retourner ( Impair (n-1) )
Fsi
Fin
----------------------------------------
Fonction  Impair ( n: entier ): Booléen

Début

si  n = 0     alors   retourner ( Faux )

sinon
   retourner ( Pair (n-1) )
Fsi
Fin
```

Définitions

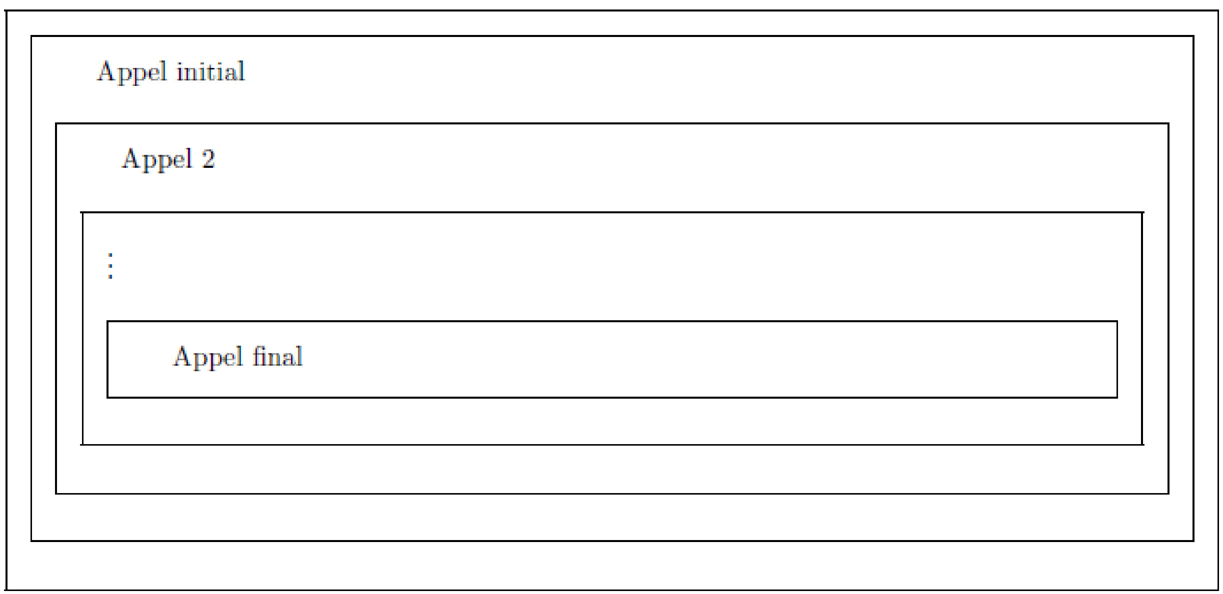
- Arité d'une procédure récursive

C'est le nombre de fois que la procédure/fonction fait appel à elle-même. S'il fait appel à lui-même p fois il est dit p-aire.

-Profondeur d'une procédure récursive

La profondeur d'un appel récursif est son rang depuis l'appel initial.

La profondeur de la récursivité est le nombre d'appels nécessaires pour aboutir à l'appel final (qui ne génère par d'autre appel récursifs).



-Procédure récursive Terminale

Nous parlons d'une procédure récursive terminale quand l'appel se fait en dernière position du traitement.

Ce type de procédure a la spécificité de trouver facilement une version itérative équivalente.

Exemples

```
fonction factoriel(N)
début
      si (N = 1)alors retourner (1)
                sinon retourner (N*factoriel(N - 1))
      finsi
fin
```

```
fonction PGCD(A, B)
début
      si (A = B)alors retourner (A)
                sinon retourner (PGCD(|A - B|, min(A, B)))
      finsi
fin
```

II. Approche « Diviser Pour Régner »

II. 1 Principe:

pour résoudre un problème donné, De nombreux algorithmes de structure récursive découpent le problème initial en un certain nombre de sous-problèmes similaires au problème initial mais de tailles inférieures, résolvent les sous-problèmes de façon récursive puis combinent ces solutions pour retrouver la solution au problème initial.

L'approche appliquée par de tels algorithmes est connue sous le nom « Diviser pour régner » (Divide and conquer en anglais)

Le paradigme Diviser pour régner donne lieu à trois étapes à chaque niveau :

— <u>Diviser</u> le problème en un certain nombre de sous-pbs de taille plus faible.

— <u>Régner</u> sur les sous-pbs de taille plus faible en les résolvant récursivement (si la taille d'un sous-pbs est assez réduite, on peut le résoudre directement).

— <u>Combiner</u> les solutions des sous-problèmes en une solution complète du problème initial.

Un des avantages des algorithmes basés sur ce paradigme est que leurs temps d'exécution sont faciles à déterminer à l'aide de récurrences.

Une récurrence est une équation ou une inégalité qui décrit une fonction à partir de sa valeur sur des entiers plus petits.

fonction $\mathcal{P}(N)$
début

$\vdots$

Décomposition de N en $N1$, $N2$,...,N_a
$\mathcal{P}_1(N1)$
$\mathcal{P}_2(N2)$

$\vdots$

$\mathcal{P}_a(N_a)$
Combinaison des solutions
fin

Coût total de la résolution = $\left\{ \begin{array}{l} \Sigma \text{ Coût de résolution des sous-pbs} \\ + \\ \text{Coût de la combinaison des solutions} \end{array} \right.$

Exemple: (Produit de matrices carrées d'ordre n) C= A* B

On suppose que $n = 2^k, k \geq 1$. On décompose les matrices en sous-matrices de taille $\frac{n}{2}$

$$\left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{2_1} & C_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{2_1} & A_{22} \end{array}\right) \times \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{2_1} & B_{22} \end{array}\right)$$

$$\begin{cases} C_{11} = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} = A_{21}A_{12} + A_{22}A_{22} \end{cases}$$

On aboutit à 8 opérations de $\bigotimes$ de matrices $(\frac{n}{2}, \frac{n}{2})$ et 4 $\bigoplus$ de matrices $(\frac{n}{2}, \frac{n}{2})$.

$$T(n) = 4 \times 2T(\tfrac{n}{2}) + 4A(\tfrac{n}{2}) = 8T(\tfrac{n}{2}) + 4(\tfrac{n}{2})^2 = 8T(\tfrac{n}{2}) + \mathcal{O}(n^2)$$

II. 2 Analyse des algorithmes « Diviser pour Régner »

Quand 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.

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, $\mathbf{n \leq c}$ la résolution est directe et consomme un temps O(1)

2. Sinon, on divise le problème en **a** sous problèmes, chacun de taille **n/b** de la taille du problème initial. Le temps d'exécution total se décompose en deux parties :

(a) aT(n/b) : temps de résolution des a sous-pbs.

(b) f(n) : temps d'exécution nécessaire pour construire la solution finale à partir des solutions aux sous-pbs.

La relations de récurrence s'écrit alors :

$$\boxed{T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq c \\ aT(\frac{n}{b}) + f(n) & \text{sinon} \end{cases}}$$

où a ≥ 1, b > 1 sont des constantes et f(n) : une fonction positive dépendante de n.

II. 3 Résolution des récurrences

Théorème:

Soient $a \geq 1$ et $b > 1$ deux constantes.

Soit $f(n)$ et $T(n)$ deux fonctions telles que

$$T(n) = aT(\frac{n}{b}) + f(n)$$

T(n) peut être bornée asymptotiquement comme suit :

— Si $f(n) = \mathcal{O}(n^{(\log_b a)-\epsilon})$ pour une constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$;

— Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$;

— Si $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ pour une constante $\epsilon > 0$, et si $a.f(\frac{n}{b}) \leq c.f(n)$ pour une constante $c < 1$ et n suffisamment grand, alors $T(n) = \Theta(f(n))$;

Exemple 1: Multiplication de Matrices Carrées

$T(n) = aT(\frac{n}{b}) + f(n)$

où $a = 8, b = 2$ et $f(n) = n^2$

$\log_2 8 = \log_2 2^3 = 3$

$f(n) = \mathcal{O}(n^2) = \mathcal{O}(n^{3-1})$

Nous sommes dans le premier cas du théorème.

> Si $f(n) = \mathcal{O}(n^{(\log_b a)-\epsilon})$ pour une constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$;

La complexité de la version « `Diviser pour régner` » de la multiplication matricielle

$T(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_2 8}) \Rightarrow T(n) = \Theta(n^3)$

$\Rightarrow$ Nous remarquons que cette version n'apporte pas d'améliorations de point de vue complexité.

Exemple 2: Tri Fusion

Fonction Tri-Fusion(S, n)
Début
  Si $n \leq 1$ **alors** retourner (S)

  Sinon

  /*Décomposer S en S_1 et S_2*/
  $S_1 \longleftarrow$ Tri-Fusion (S_1, n/2)
  $S_2 \longleftarrow$ Tri-Fusion (S_2, n/2)

  $S \longleftarrow$ Fusion (S_1, S_2)

  Retourner (S)

  Fsi

Fin

$$T(n) = \begin{cases} 0 \text{ si } n \leq 1 \\ 2\,T(n/2) + (n-1) \end{cases}$$

On a $f(n) = n-1 \Rightarrow f(n) = O(n)$

$a = 2$ et $b = 2 \Rightarrow Log_b a = Log_2 2 = 1$

$F(n) = O(n^{\log_2 2})$ on est dans le 2ème cas:

D'où $T(n) = \Theta\,(n^{\log_b a} \, Log\ n)$
$= \Theta\,(n \, Log\ n)$

II. 4 Algorithme de Strassen pour la multiplication matricielle

En 1969 *Volker Strassen* imagine un algorithme permettant de descendre le nombre de multiplication en dessous de n^3.

C’est un algorithme qui obéit à la démarche « Diviser pour régner » qui n’effectue que 7 multiplications de sous-matrices, contrairement à l’algorithme qui en effectue 8.

Mais il réalise plus d’additions et de soustractions, ce qui a toujours un moindre coût par rapport à la multiplication de matrices.

Principe:

Soient A, B et C trois matrices carrées ordre n.

$$\left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{2_1} & C_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{2_1} & A_{22} \end{array}\right) \times \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{2_1} & B_{22} \end{array}\right)$$

Fonction Strassen (A, B : Matrice, n: entier): Matrice
Début
  Si n = 1 **alors** retourner (A*B)
  Sinon
  /*On partitionne A et B chacune en 4 matrices de taille n/2 */

$$\left.\begin{array}{l}
E_1 \longleftarrow \text{Strassen}\,(A_{12} - A_{22},\ B_{21} + B_{22},\ n/2\,) \\
E_2 \longleftarrow \text{Strassen}\,(A_{11} + A_{22},\ B_{11} + B_{22},\ n/2\,) \\
E_3 \longleftarrow \text{Strassen}\,(A_{11} - A_{21},\ B_{11} + B_{12},\ n/2\,) \\
E_4 \longleftarrow \text{Strassen}\,(A_{11} + A_{12},\ B_{22},\ n/2\,) \\
E_5 \longleftarrow \text{Strassen}\,(A_{11}\,,\ B_{12} - B_{22},\ n/2\,) \\
E_6 \longleftarrow \text{Strassen}\,(A_{22},\ B_{21} + B_{11},\ n/2\,) \\
E_7 \longleftarrow \text{Strassen}\,(A_{21} + A_{22},\ B_{11}\,,\ n/2\,)
\end{array}\right\} \quad 7\,S(n/2) + 10\,(n/2)^2$$

$$+$$

$$\left.\begin{array}{l}
C_{11} \longleftarrow E_1 + E_2 - E_4 + E_6 \\
C_{12} \longleftarrow E_4 + E_5 \\
C_{21} \longleftarrow E_6 + E_7 \\
C_{22} \longleftarrow E_2 - E_3 + E_5 - E_7
\end{array}\right\} \quad 8\,(n/2)^2$$

  Fsi
Fin

On a $T(n) = M(n) + A(n)$

$$= 7\,T(n/2) + 18\,(n/2)^2$$

$$= 7\,(M(n/2) + A(n/2)) + 18\,(n/2)^2$$

$$= \underbrace{7\,M(n/2)}_{\text{Nbre de multiplications}} + 7\,\underbrace{A(n/2) + 18\,(n/2)^2}_{\text{Nbre d'additions}}$$

$$\Rightarrow T(n) = \begin{cases} M(1) = 1 & \text{si } n=1 \\ 7M(n/2) & \text{sinon} \end{cases} + \begin{cases} A(1) = 0 \\ 7\,A(n/2) + 18\,(n/2)^2 \end{cases}$$

$1^{ère}$ **Méthode** : Développement des récurrences

1/ Calcul du nbre d'additions:

$$A(n) = 7\,A(n/2) + 18\,(n/2)^2$$

$$= 7\,(7\,A(n/2^2) + 18\,(n/2^2)^2) + 18\,(n/2)^2$$

$$= 7\,(7\,(7\,A(n/2^3) + 18\,(n/2^3)^2) + 18\,(n/2^2)^2) + 18\,(n/2)^2$$

$$= \underbrace{7\;(7\;(7\;(\ldots..(7}_{\text{P fois}}\underbrace{A(n/2^p)}_{A(1)} + 18\;(n/2^p)^2) + 18(n/2^{p-1})^2) + \ldots\ldots)$$
$$+18\,(n/2^2)^2) + 18\,(n/2)^2$$

$$= 7^p\,\cancel{A(1)} + 7^{p-1}\,18\,(n/2^p)^2 + \ldots\ldots. + 7^1 18\,(n/2^2)^2 + 7^0 18\,(n/2^1)^2$$

$$= \sum_{i=0}^{p-1} 7^i * 18 * \left(\frac{n}{2^{i+1}}\right)^2 = 18 * \left(\frac{n}{2}\right)^2 \sum_{i=0}^{p-1} (7/4)^i$$

or $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$ $\Rightarrow$ $= \frac{9}{2} n^2 \left(\frac{(7/4)^p - 1}{(7/4) - 1}\right)$

$$= 6\cancel{n^2} * \frac{7^p - 4^p}{\cancel{4^p}}$$

avec $4^p = (2^p)^2 = n^2$

$$= 6(7^p - 4^p)$$

ramenons cette expression en fonction de n :

$$= 6(7^p - 4^p) = 6(7^{log_2 n} - n^2) = 6(n^{log_2 7} - n^2)$$

car,

$$a^{log(b)} = 2^{log(a^{log(b)})} = 2^{log(b).log(a)} = b^{log(a)}$$

et $n = 2^p \Rightarrow \log_2 n = \log_2 2^p = P \log_2 2 = P$

2/ Calcul du nbre de multiplications:

$M(n) = 7\,M(n/2) = 7(\,7M(n/4)) = 7(\,7(\,7\,M(n/8)))$

$= \ldots\ldots\ldots\ldots\ldots = 7(\,7(\,7\,(\ldots\ldots(7M(1))))) = 7^p\,M(1) = 7^p$

$= 7^{log_2 n} = n^{log_2 7}$

Finalement $S(n) = n^{log_2 7} + 6(n^{log_2 7} - n^2)$ or $\log_2 7 = 2.807$

d'où $\boxed{S(n) = O(n^{2.807})}$

2ème Méthode : Application du théorème

Avec le théorème : $T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2 = 7T(\frac{n}{2}) + \frac{9}{2}n^2$

Application numérique : $a = 7$, $b = 2$ et $\log_2 7 = 2.807\ldots$

$f(n) = \mathcal{O}(n^{2.807-0.807}) \Rightarrow T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$