La Résolution des équations de récurrence

I) Introduction

Généralement le temps d'exécution d'une fonction de récurrence s'exprime par une équation de récurrence.
La résolution des problèmes par la stratégie « Diviser pour régner » conduit souvent à des fonctions qui s'écrivent sous la forme générale suivante :

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

Cette forme traduit le principe de résolution suivant :

- Le problème initial est décomposé en ***a*** sous problèmes de même nature que le problème initial
- Chaque sous problème est de taille **n/b**
- La décomposition en sous problèmes puis la recomposition des solutions des sous problèmes a un coût **f(n)**

Exemple1 *: Le tri Fusion*

$$T(n) = \begin{cases} O(1) & \text{si } n<=1 \\ 2 * T(n/2)+O(n) & \text{si } n>1 \end{cases}$$

Donc a=2, b=2, f(n)=O(n)

La fonction récursive du Tri-Fusion est la suivante :

```
Action Tri-Fusion(T : tableau ; i, j :entier)
   // i positionnné sur le 1er élément et  j sur le dernier élément
Debut
k : entier ; // milieu
si  (i<j)  alors
                k ← [i+j/2] ;
                Tri-Fusion(T, i, k) ;
                Tri-Fusion(T, k+1, j) ;
                fusionner (T, i, j, k) ;
Finsi ;
Fin ;
```

La procédure fusionner consiste à fusionner deux tableaux triés en un tableau trié. Elle a une complexité linéaire.

La complexité du tri-fusion dans le pire cas est *O(nlog n)* (on exécute n fois l'action intermédiaire) que nous allons montrer :

$$T(n) = \begin{cases} O(1) & si\ n<=1 \\ 2 * T(\frac{n}{2}) + O(n) & si\ n>1 \end{cases} \qquad (1)$$

$\exists c>0$ *et* $n_0 \geq 0$ | $T(n) \leq c.\ n \log n \quad \forall n \geq n_0$

La propriété (1) est vraie pour n=1 $O(1) \leq c.n\log n$

On suppose que (1) est vraie pour k<n et on montre qu'elle est vraie pour k=n

Si n est pair : $\left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil = \frac{n}{2} < n$

On peut écrire par récurrence : $T(\frac{n}{2}) = \frac{n}{2} log(\frac{n}{2})$

$T(n) \leq (c^* \frac{n}{2} log(\frac{n}{2})) * 2 + d*n$

$\leq 2*c * \frac{n}{2}\ (\log n - \log 2) + d*n \quad (\log 2=1)$

$\leq c*n\ (\log n - 1) + d*n$

$\leq c*n\log n - c*n + d*n$

$\leq c*n\log n + n(d-c)$

Il suffit de choisir un c et d | d-c<0 pour avoir (nlogn –n < à nlogn)

d=1 c=2 et $n_0=2$

$T(n) \leq 2*n*\log n - n$

On obtient $T(n) \leq 2n \log n$

Si n est impair :

$\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2} et \left\lceil \frac{n}{2} \right\rceil = \frac{n+1}{2}$

Par un raisonnement similaire on obtient à nouveau que $T(n) \leq c.n\log n$

Donc T(n) est en O(nlogn)

***Exemple2** : La recherche dichotomique*

```
Fonction RechDicho (A :tableau ; val, deb, fin :entier) :entier ;
milieu : entier ;
Debut
  si (deb>fin)  alors   retourner -1
  sinon  milieu=(deb+fin)/2 ;
         si (val=A[milieu]) retourner milieu
        sinon si (val<A[milieu])  alors   retourner(RechDicho(A, deb, milieu-1, val)
                                  sinon  retourner(RechDicho(A, milieu+1, fin, val) ;
         finsi ;
finsi ;
Fin ;
```

Le problème initial est divisé en 2 sous problèmes, mais on ne fait qu'un seul appel donc : a=1, b=2, f(n)=Θ(1) (la comparaison dans le cas particulier quand deb=fin).

$$\Rightarrow \boldsymbol{T(n)= T(n/2) + \Theta(1)}$$

Remarque : La complexité de la recherche dichotomique est = Θ(logn)
L'algorithme « diviser pour régner » illustrant les paramètres d'une équation de récurrence générique.

```
Action fonct(P :problème ; n : taille) : solution
                                (entier)
debut
   si (n=1)  alors  solution ← résoudre le cas évident (direct)
            sinon
                (P1, …, Pa)  ← décompose P en a sous-problèmes de taille n/b
                pour i ← 1 à  a faire
                    solutioni ← fonct (Pi, n/b) ;
                fait ;
               solution ← composer (solution1, …, solutiona) ;
  finsi ;
fin ;
```

<u>*Remarque :*</u> il est généralement (beaucoup plus simple) de travailler en supposant que $n=b^k$ dans le cas où l'équation est de la forme : T(n)=a T(n/b)+f(n).

II) Quelques méthodes de résolution

a) Devinez la fonction puis la démontrer

Une première façon de trouver une solution à une équation de récurrence est de deviner la solution puis de prouver, à l'aide de l'induction mathématique, que cette solution est valable (méthode dite « guess-and-test »).
La difficulté est qu'il n'existe pas de méthode générale pour deviner une borne autre que l'expérience de cas similaires.

Exemple1 : $\begin{cases} T(n)= T(n/2) + 1 & \text{pour } n>1 \\ T(1)=1 \end{cases}$

Quelques calculs nous indiquent une solution possible (on utilise des n puissance 2 à cause de l'expression n/2 :

- $T(1)=1 \Rightarrow \mathbf{T(2^0)=1}$
- $T(2)=T(1) + 1=2 \Rightarrow \mathbf{T(2^1)=2}$
- $T(4)=T(2) + 1=3 \Rightarrow \mathbf{T(2^2)=3}$
- $T(8)=T(4) + 1=4 \Rightarrow \mathbf{T(2^3)=4}$
- $T(16)=T(8) + 1=5 \Rightarrow \mathbf{T(2^4)=5}$
- $T(32)=T(16) + 1=6 \Rightarrow \mathbf{T(2^5)=6}$
- …

$\Rightarrow \mathbf{T(2^k)=T(2^{k-1}) + 1} \quad \Rightarrow \mathbf{T(2^k)=k+1}$

Montrons que :
$T(n)= \log n+ 1$ donc que $T(n) = \Theta(\log n)$ pour $n=2^k$ $\Rightarrow$ *$\log n = \log 2^k$*
$= k \log 2$ $(\log 2=1)$
$=k$
ainsi $T(n)=k+1$ et $k=\log n$

- ***Cas de base :*** n=1 $\Rightarrow$ *$T(1)=1$*
  $T(1)=1=O+1=\log 1 + 1$ $\Rightarrow$ vrai $(\log 1=0)$
- ***Cas inductif (induction généralisée)***
  Supposons $T(m)=\log(m) + 1$ est vrai pour $m<n$
  $T(n)= T(n/2) + 1$
  $=(\log(n/2)+1) + 1$
  $= \log n - \log 2 + 1 + 1$
  $= \log n + 1$

On peut donc conclure que pour tout $n>=1$, $T(n)=\log_2 n + 1 \Rightarrow$ ***$T(n)=\Theta(\log_2 n)$***

- *Comment avoir la bonne intuition ?*

Si T(n) = 2 (T(n/2) + 17) + n*
Puisqu'on considère ces fonctions à l'infini, il peut sembler intuitif, qu'à l'infini : T(n/2) et T(n/2)+17 soient équivalents. On peut donc poser la solution T(n)=O(nlogn)

- *On fait un changement de variables*

si $T(n) = 2T(\sqrt{n}) + \log n$ *qui semble difficile.*
On fait un changement de variable $m=\log_2 n \Rightarrow n=2^m$
D'où : $T(2^m)= 2.\ T(2^{m/2}) + m$

On peut renommer $T(2^m)$ *par* $S(m) \Rightarrow S(m) = 2.S(m/2) + m$ *qui est la même récurrence que le tri-fusion*

donc S(m)=O(m log m)

$S(m)= T(2^m)=T(n)=O(\log n \ .\ \log\log n)$ *(m étant = logn)*
$\Rightarrow$ $T(n)=O(\log n \ .\ \log\log n)$

b) Résolution par substitution (ou itérative)

Il s'agit de substituer de manière répétitive la définition de la fonction dans le membre droit de l'équation jusqu'à obtenir une forme simple, le cas évident.

Exemple :

$$T(n) = \begin{cases} T(n-1) + n & \text{si } n>1 \\ 1 & \text{si } n=1 \end{cases}$$

On itère la formule :
T(n)=T(n-1)+n
T(n-1)=T(n-2) + (n-1)
T(n-2)=T(n-3) + (n-2)
T(n-3)=T(n-4) + (n-3)
…
T(2)=T(1) + 2

T(1) =1

T(n)=T(n-2) + (n-1) + n
= T(n-3) + (n-2) + (n-1) + n
= T(n-4) + (n-3) + (n-2) + (n-1) + n
…
T(n) = T(1) + 2 + 3+ … +(n-3) + (n-2) + (n-1) + n
= 1 + 2 + 3+ … +(n-3) + (n-2) + (n-1) + n

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$$

A la fin de l'itération il est important de retrouver des séries connues.

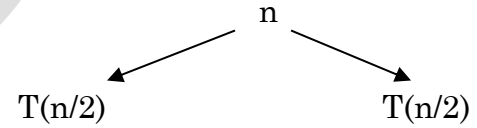
c) Arbre des appels récursifs

Les arbres des appels récursifs sont un moyen qui permet de visualiser l'itération d'une récurrence en particulier pour les algorithmes du type « diviser pour régner » : T(n)=a*T(n/b) + f(n)

Exemple 1:
T(n)=2*T(n/2) + n si n>1
T(n) = Θ (1) si n=1

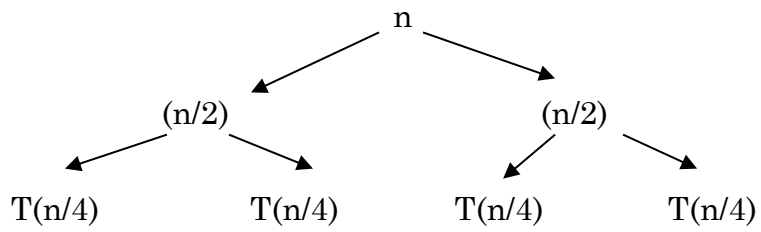
On représente l'arbre ainsi :

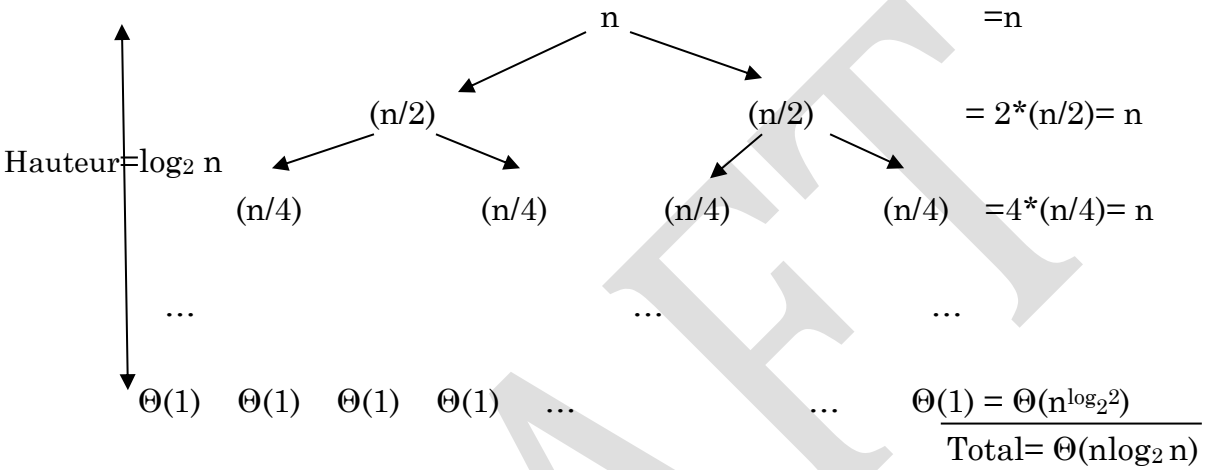
T(n) =



Deux appels récursifs de taille n / 2

On développe par la même définition les deux sous arbres, puis leurs descendants. Lorsque un tableau est de taille 1 (il est trié)n la complexité au niveau de chaque feuille est Θ (1). La figure suivante montre cette décomposition.





La hauteur d'une arborescence est toujours bornée par log(n).

Remarque : On écrit log(n) au lieu de $log_2(n)$.

Partant de $n=2^k$ le nombre de niveaux de l'arbre est exactement log(n)+1=k+1 (La hauteur de l'arbre est k).

Il suffit d'additionner le cout de chaque nœud à chaque niveau de l'arbre

Racine=n

n+ 2(n/2) + 4(n/4) + …

Le niveau de profondeur i pour $0 \le i \le k-1$ a un cout total de 2^i x $\frac{n}{2^i}$=n ce niveau comporte 2^i nœuds $\Rightarrow$ T(n) = k x n = n log(n)

T(n)=2T(n/2)+n

=2 (2T(n/2^2) + n/2) + n = 2^2 T(n/2^2) + 2(n/2) + n

= 2(2(2 T(n/2^3)+ n/2^2) + n = 2^3 T(n/2^3) + 2^2(n/2^2) + 2(n/2) + n

…

= 2^k T(n/2^k) + (k-1) * n avec $n=2^k$, T(2^k/2^k) = T(1) = 1

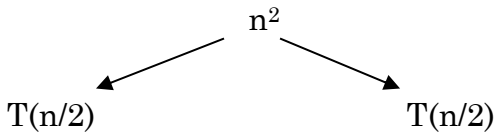
$\Rightarrow$ = n + (k-1)*n

= n*k $n=2^k$ log(n)=log(2^k) $\Rightarrow$ log(n) = k log(2) = k

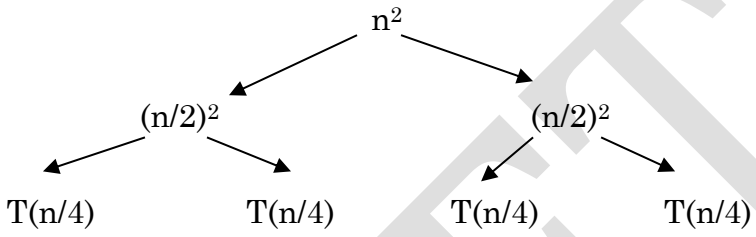
= n*log(n)

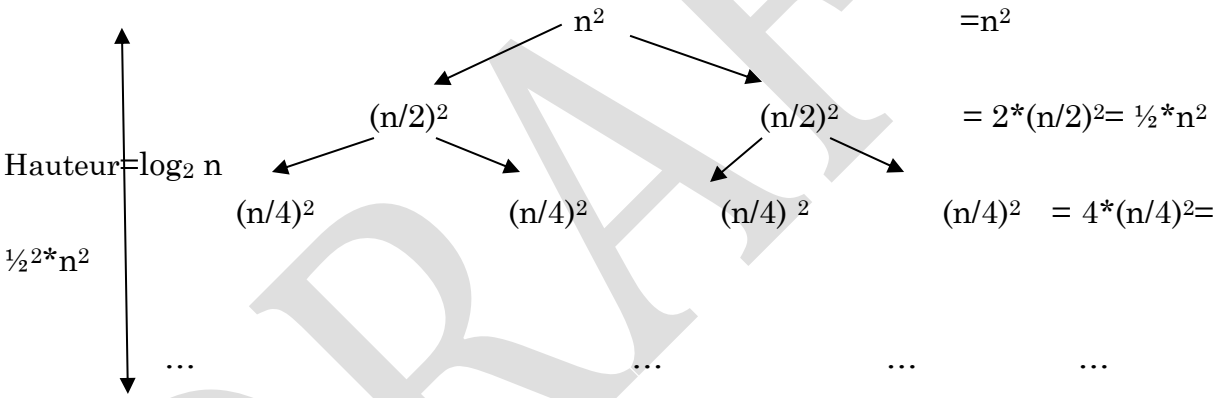
Exemple 2 : $T(n)=2T(n/2) + n^2$ On représente l'arbre ainsi :

T(n) =



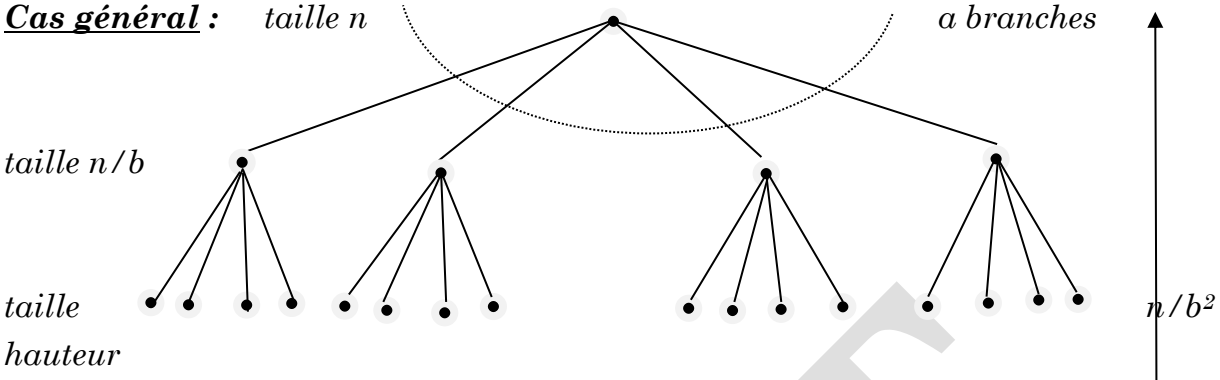
Deux appels récursifs de taille n / 2





Total = $O(n^2)$

Cas général : *taille n* *a branches*



avec $a^{\log_b n} = n^{\log_b a}$

Théorème fondamental pour la complexité

Si $T(n)=aT(n/b) + O(n^d)$ avec $a>0$, $b>1$, $d>=0$, alors

$$T(n) = \begin{cases} O(n^d) & \text{si } d>\log_b a \\ O(n^d \log_b n) & \text{si } d=\log_b a \\ O(n^{\log_b a}) & \text{si } d<\log_b a \end{cases}$$

Exemples :

- $T(n)= 3T(n/4)+\Theta(n^2)$ hauteur de l'arbre $log_4 n$ $\Longrightarrow$ *complexité*$=O(n^2)$
- $T(n)=T(n/3)+T(2n/3)+n$ hauteur de l'arbre $log_{3/2} n$ $\Longrightarrow$ *complexité*$= O(n \log n)$