1.6 Validation d'un algorithme

Lorsque nous construisons un algorithme, nous devons répondre aux trois questions suivantes :

a) l'algorithme donne t-il toujours un résultat ? c'est la *terminaison*

b) l'algorithme donne t-il le bon résultat ? c'est la *correction*

c) l'algorithme donne t-il le bon résultat en temps raisonnable ? c'est la *complexité*

Les deux premières questions répondent à la notion de *validation* de l'algorithme.

Remarque 1.6.1 *Pourquoi chercher à valider un algorithme au lieu de simplement le tester ?*

> *" Tester un algorithme montre la présence d'erreurs et non leur absence"*
>
> *[E.W.Dijkstra 1969]*

En effet :

le test :

1. valide une instance ;
2. permet une vérification rapide qui est utile au cours du développement d'un programme ;
3. fait apparaître les limites de la solution.

la preuve :

1. donne une garantie incontestable sur le principe de l'algorithme ;
2. nécessite l'utilisation d'outils formels
3. ne garantit pas contre les erreurs de programmation

1.6.1 Types d'algorithmes

Un algorithme est soit *itératif*, soit *récursif*.

L'algorithme itératif est constitué d'actions simples et de structures de contrôle (les boucles et les choix).
Son analyse de complexité consiste à estimer le nombre de répétitions des itérations d'actions (voir les exemples précédents).

Les algorithmes récursifs consistent à ramener la solution d'un problème de taille n à celle d'un nombre réduit de sous-problèmes de même type que le problème initial mais chacun de taille m, n (dans une stratégie "*Diviser pour régner*".
Son analyse de complexité consiste à estimer le temps d'exécution de tous les sous-problèmes et leur composition pour résoudre entièrement le problème. La combinaison des analyses de chaque sous-problème fournit une relation de récurrence qui sera transformée en une equation qui relie le nombre d'opérations total à la taille du problème initial.

Exemple 1.6.2

$$n! = \begin{cases} 1 & si\ n = 0, 1 \\ n * (n-1)! & si\ n > 0 \end{cases}$$

Si on note $T(n)$ le temps d'exécution de cette fonction alors on peut écrire :

$$T(n) = \begin{cases} 0 & si\ n = 0, 1 \\ 1 + T(n-1) & si\ n > 0 \end{cases}$$

La résolution de cette équation de récurrence donne $T(n) = \Theta(n)$

1.6.2 La validation : terminaison et correction

Rappelons qu'un algorithme est une suite finie d'actions qui opèrent sur un ensemble fini de variables et les modifient de leur état initial vers un état final. Un sous-ensemble de ces variables de l'état final constitue le résultat de l'algorithme.

Un algorithme résout un problème P si pour tout ensemble de valeurs des variables d'entrée, la suite des opérations exécutées est finie, c'est ce qu'on appelle la *condition de terminaison*, et lorsque l'algorithme termine le sous-ensemble des variables en sortie contient le résultat recherché, c'est *la correction* de l'algorithme.

Ainsi, avant d'évaluer la quantité de ressources nécessaire à l'exécution de l'algorithme, il est nécessaire de prouver qu'il est *valide*.

Prouver qu'un algorithme est valide consiste à :

1. démontrer qu'il *termine* quelles que soit les données d'entrée ;
2. démontrer qu'il est *correcte*, c'est-à-dire qu'il calcule bien ce qui est demandé.

1.6.3 La terminaison d'un algorithme

Pour prouver qu'un traitement termine, il suffit de trouver une mesure (fonction) qui prend des valeurs dans $\mathbb{N}$ et qui décroit strictement à chaque appel itération ou appel récursif.
La terminaison est garantie puisqu'il est impossible de construire une suite d'entiers infinie et strictement décroissante.

Quelques règles pratiques

1. La terminaison d'un algorithme sans boucle ni récursion, ayant un nombre fini de lignes est évidente.
2. La terminaison d'une boucle "***Pour***" dont le compteur ne varie pas dans le corps de la boucle est aussi évidente. (Donc il ne faut jamais modifier la variable du compteur de la boucle dans le corps de cette boucle).

3. La terminaison d'une boucle " ***Tant que*** " ou d'une fonction récursive nécessite une preuve.

Exemple 1.6.3 *La conjecture de Syracuse*

```
Tant que  n>1
faire
   si n pair  Alors   n<-- n/2
              Sinon   n<-- 3*n+1
    fsi

 fait
 retourner (n)
```

Il n'existe pas encore de preuve que Syracuse(n) s'arrête $\forall n \in \mathbb{N}$

La démarche

On prouve la terminaison de beaucoup de boucle " ***Tant que***" en mettant en évidence l'existence d'une suite finie d'entiers naturels qui décroit à chaque itération de la boucle.

Exemple 1.6.4

```
i <--- 0;
Tant que i< N
faire
   i <--- i+1
fait
```

Si $U_i = N - i > 0$*, alors* $N, N-1, N-2, \ldots, N-i, \ldots, 0$ *est une suite décroissante. Ce cas décrit par une boucle "Pour" est évident.*

Exemple 1.6.5 *Calcul de la somme* $S = a + b$ *par itération* $(a, b > 0)$

```
S  <--- a;
Tant que b>0
faire
   S <--- S+1;
```

```
   b <--- b-1;
fait
retourner S;
```

La suite décroissante est : $b, b-1, b-2, \ldots, 1, 0$

Exemple 1.6.6 *Le calcul de la division euclidienne* $div(a,b) = (q,r)$

```
q <--- 0;
Tant que a>= b
faire
  a <--- a-b;
  q <--- q+1
fait;

retourner (a, q)
```

La suite décroissante est : $a, a-b, a-2*b, a-3*b, \ldots a-q*b-r=0$*, ce qui donne :* $a = b*q+r$*, l'algorithme s'arrête avec* $q > 0$ *et* $r >= 0$*.*

1.6.4 La correction d'un algorithme itératif

L'algorithme itératif a la structure générale suivante :

```
Algorithme Iteratif
debut
Var  declaration des variables;

Initialisations;
Tant que (condition)
Faire
  Suite d'actions
Fait

FinAlgo
```

Rappelons que prouver qu'un algorithme est valide consiste à prouver sa terminaison puis sa correction.

1.6.5 Invariant de boucle

La preuve de correction est basée sur la notion d'*invariant de boucle*.

Définition 1.6.7 *L'invariant d'une boucle est une propriété de la boucle qui est :*

1. *vérifiée avant l'entrée dans la boucle ;*
2. *vérifiée à la fin de chaque itération de la boucle*

Ceci induit que la propriété est vérifiée à la fin de la dernière itération en quittant la boucle, c-a-d à la fin de l'exécution de l'algorithme, étant donné que l'algorithme termine.

La terminaison est une condition dont il faut s'assurer avant la preuve de correction.

1.6.6 Quelques exemples de l'invariant de boucle

Exemple 1.6.8 *Calcul de* 2^n

```
Fonction DeuxPuissN (E/ n:entier):entier
Var
 i, P: entier;

 P <--- 1;
 i <--- 1;
 Tant que i<= n Faire
   P <--- 2*P;
Fait;

retourner P;
Finfonction
```

L'invariant de la boucle est :

> *À la fin de l'itération* i*, l'algorithme calcule* $P_i = 2^i$*.*

Cet invariant permet de démontrer (par récurrence) qu'à l'arrêt de l'algorithme $P = 2^n$*.*

Exemple 1.6.9 *Calcul du maximum des éléments d'un tableau de n entiers*

```
Fonction Maxim (E/ T:tab, n:entier):entier
debut

Var i, max: entier;

i <--- 1; max <--- T[1];

Tant que i<n
Faire

   i<--- i+1;
   Si T[i} > max Alors max <--- T[i}
   fsi;

Fait;
retourner (max)
FinFonction;
```

L'invariant de cette boucle est :

> *À la fin de l'itération i, la variable max_i contient le $Maximum(T[1], T[2], \ldots, T[i])$*

Exemple 1.6.10 *L'algorithme de la division Euclidienne de a par b de l'exemple 1.6.6*

> *À la fin de l'itération i, $a = q_i * b + r_i$ avec $r_i = r_{i-1} - b$ et $q_i = q_{i-1} + 1$*

1.6.7 EXEMPLE

Écrire un algorithme qui calcule la somme des éléments d'un tableau de n entiers ($n > 0$).

1. Écrire un algorithme itératif et prouver sa validité

2. Déterminer sa complexité
3. Écrire un algorithme récursif pour le même problème et prouvez-le ;
4. déterminer sa complexité

Solution

On suppose que ***Tab*** est la définition d'un tableau d'entiers et ***n*** sa taille. La fonction prend en entrée un tableau A et sa taille et retourne la somme des éléments de A.

Algorithme itératif

```
Fonction Som ( A: Tab, n: entier): entier
debut

 Var
  Somme, i : entier;

  Somme <--- 0;
  Pour i <--- 1 a n Faire
   S <--- S + A[i]
  Fait;

 retourner Somme;

FinFonction
```

Prouver la validité de cet algorithme consiste à démontrer :

1. sa terminaison ;
2. sa correction.

La terminaison : L'invariant $N-i$ de la boucle "Pour" donne une suite décroissante $N, N-1, \ldots, 0$. La terminaison est triviale car la boucle est exécutée n fois et le corps de la boucle ne contient que l'addition et l'affectation qui sont des opérations qui s'exécutent chacune en temps fini.

La correction : Nous la démontrons par récurrence sur la taille des données n. Nous montrons qu'à chaque itération le résultat est bien celui qui est recherché. Il faut formaliser l'expression du résultat.

Considérons la propriété suivante :

> **Propriété 1.6.11** *"à la fin de l'itération i, la variable $Somme$ contient la somme des i premiers éléments du tableau A."*

Cette propriété est l'invariant de boucle de l'algorithme, ce qui est calculé à la fin de chaque itération de la boucle.

Étapes de la preuve Notons $Somme_i$ la valeur de cette somme à la fin de l'étape i. Nous avons alors :

— $Somme_0 = 0$;

— Pour $i = 1$, nous avons $Somme_1 = Somme_0 + A[1] = A[1]$.

La propriété (1.6.11) est donc vraie pour $i = 0$ et $i = 1$.

— Hypothèse de Récurrence :

Supposons la propriété (1.6.11) vraie à l'ordre i, soit :

$$Somme_i = \sum_{j=1}^{i} A[j]$$

et calculons l'itération $i+1$.

À la fin de l'itération $i+1$, la variable $Somme$ contient ce qu'elle contenait à l'étape i plus l'élément $A[i+1]$, autrement dit :

$Somme_{i+1} = Somme_i + A[i+1]$

$$Somme_{i+1} = \sum_{j=1}^{i} A[j] + A[i+1] \tag{1.1}$$

$$= \sum_{j=1}^{i+1} A[j] \tag{1.2}$$

La propriété (1.6.11) est donc vraie à l'ordre $i+1$.

L'algorithme termine à la fin de l'itération n et on aura donc calculé :

$$Somme_n = \sum_{j=1}^{n} A[j]$$

L'algorithme est donc valide.

La complexité Nous comptons le nombre d'opérations dans la boucle : « une addition + une affectation" ». Notons par $T(n)$ le nombre d'opérations lorsque l'algorithme termine.

$$T(n) = \mathcal{O}(n) \text{ et plus précisément } T(n) = \Theta(n)$$

Algorithme récursif

```
Fonction SomRec ( A: Tab, n: entier): entier

debut

 si (n<=0)  Alors retourner 0
           Sinon retourner (SomRec (A, n-1) + A[n])

 fsi;

FinFonction
```

La terminaison : On peut remarquer immédiatement que dans l'appel récursif, la valeur de l'argument dont dépend la récursion (ici, la valeur du paramètre n) diminue à chaque appel et finit par atteindre le cas de base. Néanmoins on peut prouver la terminaison par récurrence sur i de la manière suivante :

— si $i = 0$, le cas est évident, l'algorithme termine sans appel récursif.

— Hypothèse de Récurrence :

On suppose qu'à l'étape n l'algorithme termine soit (**HR**) : $SomRec(A,n)$ termine et évaluons $SomRec(A,n+1)$.

$$SomRec(A,n+1) \quad \text{appelle} \quad \underbrace{SomRec(A,n)}_{\nearrow \text{ termine par HR}} \quad + \quad \underbrace{A[n+1]}_{\uparrow \text{ termine}}$$

Donc $SomRec(A,n+1)$ termine pour tout n.

La correction : Notons $Somme_n$ la valeur retournée par l'appel $SomRec(A,n)$. La fonction reproduit la relation de récurrence suivante :

$$\begin{aligned} Somme_0 &= 0 \\ Somme_n &= Somme_{n-1} + A[n], \quad n > 0 \end{aligned}$$

La preuve est similaire à celle du cas de la solution itérative (voir le paragraphe (1.6.11) où il est montré que la fonction calcule bien la somme des éléments du tableau, soit :

$$Somme_n = \sum_{j=1}^{n} A[j]$$

Quelques exemples

Exemple 1.6.12 *Soit la fonction récursive suivante :*

```
fonction F (E/n: entier): entier
debut

  Si n = 0 Alors  retourner 2
           Sinon retourner F(n-1)* F(n-1)

  Fsi
FinFonction;
```

1. *Que calcule F ? Prouvez-le.*
2. *Calculer la complexité de F.*
3. *Comment améliorer cette complexité ?*

Solution

1. **Que fait la fonction F ?**

$$\begin{aligned} F(0) &= 2 \\ F(1) &= F(0) * F(0) = 2 * 2 = 2^2 \\ F(2) &= F(1) * F(1) = 2^2 * 2^2 = 2^4 = 2^{2^2} \\ F(3) &= F(2) * F(2) = 2^{2^2} * 2^{2^2} = 2^8 = 2^{2^3} \\ F(4) &= F(3) * F(3) = 2^{2^3} * 2^{2^3} = 2^{16} = 2^{2^4} \end{aligned}$$

Affirmation : $F(n) = 2^{2^n}$

2. **Preuve de validité**

— **La terminaison** : Par récurrence sur n

Pour $n = 0$: F termine (évident)

Pour $n > 0$: Supposons que $F(n)$ termine (Hypothèse de récurrence).

$$F(n+1) = \underset{\substack{\nearrow \\ \text{termine} \\ \text{par HR}}}{F(n)} \quad \underset{\substack{\uparrow \\ \text{termine}}}{*} \quad \underset{\substack{\nwarrow \\ \text{termine} \\ \text{par HR}}}{F(n)}$$

— **La correction** :

Notons F_n la valeur retournée par la fonction F à la fin de l'appel avec n. Donc la fonction calcule :

$$\begin{cases} F_0 = 2 \\ F_n = F_{n-1} * F_{n-1} \end{cases}$$

Montrons que : $\boxed{F(n) = 2^{2^n} \quad \forall n}$

(a) Pour $n = 0$ $F_0 = 2 = 2^{2^0} = 2^1 = 2$ donc (vraie) ;

(b) Supposons que $F(n) = 2^{2^n}$ (HR), et calculons F_{n+1}.

$$\begin{aligned} F_{n+1} &= F_{n-1} * F_{n-1} \\ &= 2^{2^n} * 2^{2^n} \text{(HR)} \\ &= (2^{2^n})^2 \\ &= 2^{2^{n+1}} \text{CQFD} \end{aligned}$$

— **La complexité** :

Nous comptons le nombre de multiplications effectuées pour calculer F_n. Soit $T(n)$ ce nombre. Nous avons :

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + 2 * T(n-1) & \text{si } n > 0 \end{cases}$$

La résolution de cette equation donne une complexité $T(n) = \mathcal{O}(2^n)$.

3. **Comment améliorer l'algorithme ?**

On peut réduire la complexité en évitant les deux appels récursifs pour un même calcul.

```
Fonction F_new (E/n:entier):entier
debut

Si n=0 Alors retourner 2
       Sinon retourner (F(n-1))^2

Fsi
FinFonction;
```

La complexité devient :

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + T(n-1) & \text{si } n > 0 \end{cases}$$

qui a pour solution $T(n) = \Theta(n)$.

Fonction itérative Soit la fonction :

```
Fonction G(E/n : entier):entier
debut
Var R: entier;

R <--- 2;
 Pour i<1 a n Faire
   R <--- R*R
 Fait
retourner R
FinFonction;
```

— **Que calcule G ?**

1. $\underline{n = 0}$: $R = 2$;
2. $\underline{n = 1}$: $R = 2 * 2 = 2^2$;
3. $\underline{n = 2}$: $R = 2^2 * 2^2 = 2^4 = 2^{2^2}$;
4. $\underline{n}$: $G(n) = 2^{2^n}$, la même valeur que $F(n)$.

— **La terminaison** : Elle est évidente.

Si $n = 0$ la fonction G retourne 2 et termine ; si $n > 0$ la fonction termine une itération finie (la boucle **Pour**).

— **La correction** :
Notons R_i la valeur que calcule G 'a la fin de l'itération i.

L'invariant de boucle :

> À la fin de l'itération i, $R_i = 2^{2^i}$

$$\begin{cases} R_0 = 2 \\ R_n = 2^{2^n} \end{cases}$$

Comme pour F on peut prouver cette invariant par récurrence sur n.
Donc $G(n) = 2^{2^n}$.

— **La complexité** : $T(n) = \Theta(n)$

Chapitre 2

La résolution des équations de récurrence

2.1 Introduction

"*Diviser pour régner*" est une stratégie de résolution d'algorithmes qui consiste à décomposer un problème initial $\mathcal{P}$ de taille n, généralement complexe, en plusieurs sous-problèmes, chacun de taille $m < n$ plus simples, puis à résoudre l'ensemble de ces sous-problèmes et *composer* leur solution afin d'obtenir la solution du problème $\mathcal{P}$.

L'algorithme obtenu est généralement un algorithme récursif. La complexité (temporelle) de $\mathcal{P}$ hérite de la structure récursive de l'algorithme.

2.1.1 Forme générale de la complexité

L'expression $T(n)$ de la complexité d'un algorithme de type "*Diviser pour régner*" a la forme suivante :

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

Cette formule traduit le principe de résolution suivant :

1. Le problème initial est décomposé en a sous-problèmes de même nature que le problème initial ;
2. chaque sous-problème est de taille n/b ;

3. la décomposition en sous-problèmes puis la recomposition des solutions des sous-problèmes a un coût $f(n)$

Exemple 2.1.1 *Algorithme du Tri-Fusion*

```
Action Tri-Fusion (A, deb, fin)
debut
si deb < fin Alors
              m <--- (deb+fin)/2;
              Tri-Fusion (A, deb, m);
              Tri-Fusion (A, m+1, fin);
              Fusionner (A, deb, m, fin);
fsi;
Fin;
```

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

$$Donc\ a = 2; b = 2; f(n) = \Theta(n)$$

Exemple 2.1.2 *Algorithme de la recherche dichotomique dans un tableau* $A[deb..fin]$

```
Action RecherchDicho(A, x, deb, fin)
debut
si deb = fin Alors
       si A[deb] = x Alors retourner deb
                     Sinon retourner -1
        fsi
       Sinon
             m <--- (deb+fin)/2;
             si A[m] = x Alors retourner m
                         Sinon
                          si A[m} < x Alors
                                       retourner RechDicho (A, x, m+1, fin)
                                      Sinon
                                       retourner RechDicho (A, x, deb, m-1)
                          fsi
            fsi
fsi
Fin;
```

Le problème initial est divisé en deux sous problèmes, mais on ne fait qu'un seul appel récursif (dans un cas ou dans l'autre).

$$T(n) = \begin{cases} 1 & si\ n \leq 1 \\ T(n/2) + \Theta(1) & si\ n > 1 \end{cases}$$

Donc $a = 1; b = 2; f(n) = \Theta(1)$

2.2 Méthodes de résolution des équations de récurrence

Généralement le temps d'exécution d'une fonction récursive s'exprime par une équation de récurrence. L'équation de récurrence décrit une fonction à partir de sa valeur sur des entrées plus petites.

2.2.1 Résolution par induction mathématique

Dans cette méthode nous faisons l'hypothèse d'une borne que nous devinons puis nous la démontrons par récurrence. La difficulté de cette méthode réside dans le fait qu'il n'existe pas de méthode générale pour deviner une borne autre que l'apprentissage et l'expérience de cas similaires.

Premier exemple

Considérons l'équation de récurrence de l'algorithme récursif de la recherche dichotomique.

Exemple 2.2.1 *Nous calculons quelques étapes pour nous guider à proposer une formule :*

$$\begin{array}{lcll} T(1) & = & 1 & \Longrightarrow \quad T(2^0) = 1 \\ \\ T(2) & = & T(1)+1 & \Longrightarrow \quad T(2^1) = 2 \\ T(4) & = & T(2)+1 & \Longrightarrow \quad T(2^2) = 3 \\ T(8) & = & T(4)+1 & \Longrightarrow \quad T(2^3) = 4 \\ T(16) & = & T(8)+1 & \Longrightarrow \quad T(2^4) = 5 \\ T(32) & = & T(16)+1 & \Longrightarrow \quad T(2^5) = 6 \\ \vdots & & \vdots & \\ T(2^k) & = & T(2^{k-1})+1 \Longrightarrow & T(2^k) = k+1 \end{array}$$

Lorsque $n = 2^k \implies \log n = k * \log 2$*, d'où* $k = \log_2 n$*. Nous proposons de montrer que* $T(n) = \log_2 n + 1$ *vraie.*

Preuve

1. *Cas de base :* $n = 1$ *et* $T(1) = 1 = \log_2 1 + 1 = 1$*, donc vrai.*

2. *Supposons* $T(m) = \log_2 m + 1$ *pour* $m < n$ *vrai.*

$$\begin{array}{rcll} T(n) & = & T(n/2)+1 & n/2 < n \quad \textit{donc} \\ & = & (\log_2(n/2)+1)+1 & \\ & = & \log_2 n - \log_2 2 + 1 + 1 & \\ & = & \log_2 n - \cancel{\log_2 2} + \cancel{1} + 1 & \\ & = & \log_2 n + 1 & \textit{CQFD.} \end{array}$$

Donc :

$$\forall n \geq 1, T(n) = \log_2 n + 1 \implies T(n) = \Theta(\log_2 n))$$

Second exemple

Sachant que pour le Tri-Fusion, la complexité est $T(n) = 2 * T(n/2) + n$ dont la solution proposée est $T(n) = \mathcal{O}(n * \log n)$, peut être prouvée par récurrence.

Comment avoir la bonne intuition lorsque :

1. $T(n) = 2 * (T(n/2) + 17) + n$
   Puisque nous considérons ces fonctions à l'infini, il est possible d'avoir l'intuition qu'à l'infini $T(n/2)$ et $T(n/2) + 17$ soient équivalentes. Nous pourrons alors proposer la solution $T(n) = \mathcal{O}(n * \log n)$ aussi comme solution.

2. $T(n) = 2 * T(\sqrt{n}) + \log n$ **Faire des changements de variables**
   Cette fonction semble difficile à priori. Si nous faisons le changement de variable suivant : $m = \log_2 n \implies n = 2^m$. Nous obtenons :

$$T(2^m) = 2 * T(2^{m/2}) + m$$

   Nous renommons $T(2^m)$ par $S(m)$, ce qui donne :

$$S(m) = 2 * S(m/2) + m$$

qui est la recurrence du Tri-Fusion donc sa solution est :

$$S(m) = \mathcal{O}(m \log m)$$

Puisque $S(m) = T(2^m) = T(n) = \mathcal{O}(\log n * \log\log n)$. Donc

$$T(n) = \mathcal{O}(\log n * \log\log n)$$

2.2.2 La méthode par substitution 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, celle du cas évident.

Par exemple :

Exemple 2.2.2 *Par exemple, considérons l'équation de récurrence suivante :*

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

Nous itérons la formule :

$$\begin{array}{lcl} T(n) & = & T(n-1) + n \\ T(n-1) & = & T(n-2) + n - 1 \Longleftrightarrow T(n) = T(n-2) + (n-1) + n \\ T(n-2) & = & T(n-3) + n - 2 \Longleftrightarrow T(n) = T(n-3) + (n-2) + (n-1) + n \\ T(n-3) & = & T(n-4) + n - 3 \Longleftrightarrow T(n) = T(n-4) + (n-3) + (n-2) + (n-1) + n \\ \vdots & & \\ T(n) & = & T(1) + 2 + 3 + \cdots + (n-2) + (n-1) + n \end{array}$$

$$T(n) = \sum_{i=1}^{n} i = n * (n+1)/2$$

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

Cette méthode transforme la récurrence en une sommation qu'il s'agit de borner. Elle fait appel à l'algèbre, car à la fin du processus d'itération il est important de retrouver des séries mathématiques connues.

2.2.3 Les arbres récursifs

Les arbres récursifs sont un moyen qui permet de visualiser l'itération d'une equation en particulier pour les algorithmes de type « *Diviser pour régner* ».

Exemple 2.2.3 *Considérons l'équation de récurrence suivante :*

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

Nous représentons l'arbre sous la forme suivante :

Dessin de l'arbre à compléter

2.2.4 La méthode générale

Elle donne des bornes pour les récurrences de la forme suivante : $T(n) = a * T(n/b) + f(n)$ avec $a \geq 1, b > 1$ et $f(n)$ est une fonction donnée. Elle est basée sur le théorème suivant où trois cas sont envisagés.

Théorème

Soient $a \geq 1$ et $b > 1$ deux constantes, et soit $f(n)$ une fonction et $T(n)$ l'équation de récurrence suivante :

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

Alors $T(n)$ peut être bornée asymptotiquement ainsi :

1. Si $f(n) = \mathcal{O}(n^{log_b a - \epsilon})$ pour une constante $\epsilon > 0$ alors $T(n) = \Theta(n^{\log_b a})$
2. Si $f(n) = \Theta(n^{\log_b a})$ alors $T(n) = \Theta(n^{\log_b a} \log_2 n)$
3. Si $f(n) = \Omega(n^{log_b a + \epsilon})$ pour une constante $\epsilon > 0$ et si $a * f(n/b) \leq c * f(n)$ pour une constante $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$.

Dans les trois cas il y a comparaison entre $f(n)$ et $(n^{\log_b a})$

1. Dans le premier cas : $f(n) \leq (n^{\log_b a})$, $f(n)$ est plus petite que $(n^{\log_b a})$ du facteur polynomial n^{ϵ}.
2. Dans le troisième cas : $f(n) \geq (n^{\log_b a})$ du facteur polynomial n^{ϵ}.

Exemple d'application du théorème

$T(n) = 3 * T(n/4) + n * log n$

$$\begin{cases} a = 3 \\ b = 4 \\ f(n) = n * log n \end{cases}$$

$$\begin{aligned} \log_b a &= \log_4 3 \simeq 0.793 \\ n * \log n &\neq \mathcal{O}(n^{0.793-\epsilon}) \\ n * \log n &\neq \Theta(n^{0.793}) \end{aligned}$$

Mais $n * \log n = \Omega(n^{0.793+\epsilon}), \epsilon > 0$ (borne inf)
On prend $\epsilon = 1 - 0.793 = 0.207 \simeq 0.2$ $(n * \log n > n)$
et

$$\begin{aligned} a * f(n/b) &= 3 * f(n/4) \\ &= 3 * \tfrac{n}{4} \log \tfrac{n}{4} \\ &\leq \tfrac{3}{4} * n \log n \end{aligned}$$

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

2.3 Exercices d'application

2.3.1 Résoudre l'équation de récurrence suivante par substitutions itératives et par le théorème général

$$T(n) = \begin{cases} 9 * T(n/3) + n^2 & si\, n > 2 \\ 1 & \text{si } n = 1 \end{cases}$$

Substitutions itératives

$$\begin{aligned}
T(n) &= 9 * T(n/3) + n^2 \\
&= 9 * \left[9 * T(\tfrac{n}{3^2}) + (\tfrac{n}{3})^2\right] + n^2 \\
&= 9^2 * T(\tfrac{n}{3^2}) + 9 * (\tfrac{n}{3})^2 + n^2 \\
&= 9^2 * \left[9 * T(\tfrac{n}{3^3}) + (\tfrac{n}{3^2})^2\right] + 9 * (\tfrac{n}{2})^2 + n^2 \\
&= 9^3 * T(\tfrac{n}{3^3}) + 9^2 * (\tfrac{n}{3^2})^2 + 9 * (\tfrac{n}{3})^2 + n^2 \\
&= 9^3 * \left[9 * T(\tfrac{n}{3^4}) + (\tfrac{n}{3^3})^2\right] + 9^2 * (\tfrac{n}{3^2})^2 + 9 * (\tfrac{n}{3})^2 + n^2 \\
&= 9^4 * T(\tfrac{n}{3^4}) + 9^3 * (\tfrac{n}{3^3})^2 + 9^2 * (\tfrac{n}{3^2})^2 + 9 * (\tfrac{n}{3})^2 + n^2 \\
&= 9^4 * T(\tfrac{n}{3^4}) + 9^3 * (\tfrac{n}{3^3})^2 + 9^2 * (\tfrac{n}{3^2})^2 + 9 * (\tfrac{n}{3})^2 + n^2 \\
&= 9^4 * T(\tfrac{n}{3^4}) + n^2 * \left[\tfrac{\cancel{9^3}}{\cancel{(3^3)^2}}\right] + n^2 * \left[\tfrac{\cancel{9^2}}{\cancel{(3^2)^2}}\right] + n^2 * \left[\tfrac{\cancel{9}}{\cancel{(3^2)}}\right] + n^2 \\
&= 9^4 * T(\tfrac{n}{3^4}) + n^2 + \cancel{n^2} + n^2 + n^2 \\
&= \vdots \\
T(n) &= 9^k * T(\tfrac{n}{3^k}) + k * n^2
\end{aligned}$$

On pose $\boxed{n = 3^k \Longrightarrow k = \log_3 n}$.

D'autre part : $9^k = (3*3)^k = 3^k * 3^k = n^2$

Donc : $T(n) = n^2 * T(1) + k * n^2 = (k+1) * n^2 = (\log n + 1) * n^2 = n^2 + n^2 * \log n$

$$\boxed{T(n) = \Theta(n^2 * \log n)}$$

Application du théorème général

$T(n) = a * T(n/b) + f(n)$ avec $a \geq 1, b > 1$

$$a = 9; b = 3; f(n) = n^2$$

$\log_b a = \log_3 9 = \log_3 3^2 = 2 * \log_3 3 = 2$

$n^{\log_b a} = n^2$ et $f(n) = n^2$. Donc nous sommes dans le cas $f(n) = \Theta(n^{\log_b a})$. Le theorme donne :

$$\boxed{T(n) = \Theta(n^{\log_b a} * \log n) = \Theta(n^2 * \log n)}$$