INFO601_CMI, L3 Informatique, Algorithmique III
Leçon 4, Complexité des algorithmes récursifs

Jacques-Olivier Lachaud
LAMA, Université Savoie Mont blanc
jacquesolivierlachaud@github.io
(suivre teaching/INFO601_CMI)

27 novembre 2025

1 Complexité des fonctions récursives

On abordera ici essentiellement la complexité en pire cas des algorithmes récursifs, dans les cas relativement simples (récurrences linéaires, méthodes diviser pour régner).

1.1 Complexité en pire cas des fonctions récursives

La complexité en pire cas d'une fonction récursive $f(\chi)$ s'étudie en général en associant (au moins) un paramètre n au temps d'exécution T. Les *paramètres* ou *contexte* χ de f sont réduits à ce paramètre n de manière à simplifier l'étude du temps d'exécution de f. Ce paramètre n est construit de manière à être décroissant à chaque appel de f. De plus, lorsque $n \leq cst$, la récursion doit se terminer. On pourra alors exprimer le temps d'exécution en pire cas de f comme une suite T, où $T(0), \dots T(cst)$ ont des valeurs données, et $T(n)$ s'exprime en fonction de valeurs de la suite T plus petites.

Quelques exemples :

```
// Calcul de la factorielle d'un entier m
Fonction FACT( E m : entier ) : entier ;                    // Ici, on choisit simplement n = m
début
    si m = 0 alors Retourner 1                              /* On a facilement: T(0) = Θ(1) */;
    sinon Retourner m*FACT(m − 1)                           /* et T(n) = Θ(1) + T(n − 1) */;
```

```
// Calcul du m-ième terme de la suite de Fibonacci
Fonction FIB( E m : entier ) : entier ;                     // Ici, on choisit simplement n = m
début
    si m = 0 alors Retourner 0                              /* On a facilement: T(0) = Θ(1) */;
    sinon si m = 1 alors Retourner 1                        /* On a facilement: T(1) = Θ(1) */;
    sinon Retourner FIB(m − 1)+FIB(m − 2) /* et T(n) = Θ(1) + T(n − 1) + T(n − 2) */;
```

```
// Calcul du maximum d'un tableau T entre deux bornes a ≤ b
Fonction MAX( E T : tableau de Elem, E a,b :entier ) : Elem      /* Ici, on choisit
 n = b − a */ début
    si a = b alors Retourner T[a]                           /* On a facilement: T(0) = Θ(1) */;
    sinon Retourner Max2(T[a], MAX(T, a + 1, b)             /* et T(n) = Θ(1) + T(n − 1) */;
```

```
// Recherche dichotomique de x dans un tableau T entre deux bornes a ≤ b
Fonction DICHO( E T : tableau de Elem, E a,b :entier, x : Elem ) : booléen ;
 // Ici, on choisit n = b − a
début
    si a = b alors Retourner T[a] = x                    /* On a facilement: T(0) = Θ(1) */;
    sinon
        m ← ⌊(a+b)/2⌋ ;
        si x ≤ T[m] alors Retourner DICHO(T, a, m, x)    /* et T(n) = Θ(1) + T(⌊n/2⌋) */;
        sinon Retourner DICHO(T, m + 1, b, x)            /* et T(n) = Θ(1) + T(⌊n/2⌋) */;
```

```
// Tours de Hanoi: déplacer m disques de rayons croissants du plot i vers le
   plot j, en utilisant un plot intermédiaire k
Action HANOI( E m, i, j, k : entiers ;                   // Ici, on choisit n = m
début
    si m ≠ 0 alors
        // On déplace m − 1 disques du plot i vers le plot k.
        HANOI( m − 1, i, k, j ) ;                        // soit un temps T(m − 1)
        // On déplace le premier disque sur le plot i vers le plot j.
        AFFICHE( "Déplace disque sur plot ", i, " vers plot ", j) ;        // soit Θ(1)
        // On déplace m − 1 disques du plot i vers le plot k.
        HANOI( m − 1, k, j, i ) ;                        // soit un temps T(m − 1)
```

On se ramène à exprimer la complexité en pire cas des fonctions récursives sous forme de suites mathématiques.

Fonction	n_0	$T(0),\dots,T(n_0)$	récurrence
FACT	0	$T(0)=\mathcal{O}(1)$	$T(n+1)=T(n)+\mathcal{O}(1)$
FIB	1	$T(0)=T(1)=\mathcal{O}(1)$	$T(n+2)=T(n+1)+T(n)+\mathcal{O}(1)$
MAX	0	$T(0)=\mathcal{O}(1)$	$T(n+1)=T(n)+\mathcal{O}(1)$
DICHO	0	$T(0)=\mathcal{O}(1)$	$T(n+1)=T(\lfloor\frac{n+1}{2}\rfloor)+\mathcal{O}(1)$
HANOI	0	$T(0)=\mathcal{O}(1)$	$T(n+1)=2T(n)+\mathcal{O}(1)$

1.2 Récurrences linéaires simples

Dans beaucoup de cas, on peut se ramener à une formule du genre :

$$\begin{cases} T(0)=\mathcal{O}(1), T(1)=\mathcal{O}(1), \text{etc}, T(n_0)=\mathcal{O}(1) \\ \forall n \geq n_0, T(n+n_0+1)=c_{n_0}T(n+n_0)+c_{n_0-1}T(n+n_0-1)+\ldots+c_0T(n)+f(n) \end{cases} \quad (1)$$

Exemples : FACT, FIB, MAX, HANOI.
On voit que DICHO est différente des précédentes.

1.2.1 La forme $T(n+1)=cT(n)+1$

Ici, $n_0=0$, et on suppose $T(0)=1$ sans perte de généralité. On reconnait une suite géométrique de raison c :

$$\begin{aligned} T(n+1) =& cT(n)+1 = c^2T(n-1)+c+1 = c^3T(n-2)+c^2+c+1 = \cdots \\ =& c^n+c^{n-1}+\cdots+c+1. \end{aligned}$$

— Si $c=1$, on a immediatement $\forall n \geq 1, T(n)=1+\ldots+1=n$.
— sinon $T(n+1)=\frac{c^{n+1}-1}{c-1}$.
Donc, si $c=1$, la fonction récursive est de complexité linéaire $\Theta(n)$. Lorsque $c>1$, l'algorithme prend un temps $\mathcal{O}(c^n)$. C'est par exemple le cas des algorithmes récursifs :

— calcul des coefficients binomiaux par relation de récurrence
— de l'algorithme du sac-à-dos.
— résolution des tours de Hanoi HANOI.

1.2.2 La forme $T(n+1) = cT(n) + f(n)$

Cela dépend de $f(n)$ et doit être traité au cas par cas.

1.2.3 La forme $T(n+2) = dT(n+1) + eT(n) + 1$

Ici, $n_0 = 1$. On la découpe en deux en regardant seulement $T'(n+2) = dT'(n+1) + eT'(n)$. En fait, si $\max(d, e) \geq 1$, on a $T(n) = \Theta(T'(n))$.

On va chercher les racines r_1 et r_2 du polynôme $X^2 - dX - e$.

— Si $r_1 \neq r_2$ et réels, il est facile de vérifier que $\lambda r_1^n + \mu r_2^n$ satisfont la relation de récurrence. On détermine alors λ et μ avec les valeurs $T(0)$ et $T(1)$.
— les autres cas ne se produisent pas *a priori* du fait que ce sont des temps de calcul (d et e sont positifs).

On peut par exemple vérifier la formule sur FIB où $T(n+2) = T(n+1) + T(n) + 1$.

$$\begin{gathered} T'(n+2) - T'(n+1) - T'(n) = 0 \\ \text{discriminant } \Delta = 1 - 4 \times 1 \times (-1) = 5 \\ \text{solutions } r_1 = \tfrac{1+\sqrt{5}}{2} \quad r_2 = \tfrac{1-\sqrt{5}}{2} \end{gathered}$$

Le discriminant vaut $\sqrt{5}$ d'où les deux racines $r_1 = \frac{1+\sqrt{5}}{2}$ et $r_2 = \frac{1-\sqrt{5}}{2}$. On reconnait r_1 comme étant le nombre d'or ϕ. Comme $|r_2| < 1$, r_2^n tend vers 0 et r_1^n domine. On peut déjà conclure que

$$T_{\text{FIB}} = \Theta(\phi^n).$$

On peut chercher aussi les constantes. En prenant $T(0) = T(1) = 1$, on écrit les égalités initiales :

$$\begin{array}{l} \lambda r_1^0 + \mu r_2^0 = 1 \\ \lambda r_1^1 + \mu r_2^1 = 1 \end{array} \Leftrightarrow \begin{array}{c} \lambda + \mu = 1 \\ \lambda \frac{1+\sqrt{5}}{2} + \mu \frac{1-\sqrt{5}}{2} = 1 \end{array} \Leftrightarrow \begin{array}{c} \lambda + \mu = 1 \\ \lambda - \mu = \frac{1}{\sqrt{5}} \end{array} \Leftrightarrow \begin{array}{l} \lambda = \frac{1}{2} + \frac{\sqrt{5}}{10} \\ \mu = \frac{1}{2} - \frac{\sqrt{5}}{10} \end{array}$$

Pour conclure

$$T'(n) = \left(\frac{1}{2} + \frac{\sqrt{5}}{10}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1}{2} - \frac{\sqrt{5}}{10}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n$$

On voit alors que le deuxième terme tend vers 0. On déduit que $T'(n) = \mathcal{O}\left(\frac{1+\sqrt{5}}{2}\right)^n$.

NB : Le nombre d'or $\phi \approx 1,618$ est plus petit que 2. On aurait pu montrer très facilement que $T(n) = \mathcal{O}(2^n)$ avec la section précédente. Là, on montre que $T(n) = \Theta(\phi^n)$ ce qui est bien plus précis.

1.3 La forme "diviser pour régner"

Il est très fréquent d'avoir des algorithmes de la forme "diviser pour régner". Ceci est par exemple le cas du tri fusion (voir Algorithme 1). On observe une récurrence de la forme $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$. En réalité les arrondis inférieurs ou supérieurs ne changent pas la complexité finale.

On dispose du théorème général suivant :

Théorème 1 *Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour les entiers non négatifs par la récurrence*

$$T(n) = aT(n/b) + f(n),$$

où n/b peut être interprétée comme $\lceil \frac{n}{b} \rceil$ ou $\lfloor \frac{n}{b} \rfloor$. De plus $T(0) = O(1)$. Alors $T(n)$ peut être bornée asymptotiquement ainsi :

Algorithme 1 : Tri fusion.

```
// Divise en O(n)
Action DIVISE( E T : tableau de Elem, E n : entier, S T1,T2 : tableau de Elem, S n1,n2 ) ;
Var : i : entier ;
début
    i ← 0, n1 ← 0, n2 ← 0 ;
    tant que i < n faire
        T1[n1] ← T[i] ;
        n1 ← n1 + 1, i ← i + 1 ;
        si i < n alors
            T2[n2] ← T[i] ;
            n2 ← n2 + 1, i ← i + 1 ;
// Fusionne en O(n1) + O(n2) = O(n)
Action FUSIONNE( E T1,T2 : tableau de Elem, E n1,n2, S T : tableau de Elem, E n : entier)
// Tri fusion
Action TRIFUSION( E T : tableau de Elem, E m : entier ) ;        // Ici, on choisit n = m
Var : T1,T2 : Tableau de Elem ;
n1,n2 : entier ;
début
    si m > 1 alors
        DIVISE( T,m,T1,T2,n1,n2 ) ;                    // On a facilement : T(0) = Θ(1)
        TRIFUSION( T1,n1 ) ;                           // et T(n) = Θ(n) + T(⌈n/2⌉)
        TRIFUSION( T2,n2 ) ;                           //       +T(⌊n/2⌋) + Θ(n)
        FUSIONNE( T1,T2,n1,n2,T,m ) ;
```

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* $T(n) = \Theta(n^{\log_b a} \log n)$,
3. *si* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *pour une constante* $\epsilon > 0$*, et si* $af(n/b) \geq cf(n)$ *pour une certaine constante* $c < 1$ *et pour tout* n *assez grand, alors* $T(n) = \Theta(f(n))$.

Dit autrement, dans le cas 1, c'est le terme de gauche et l'initialisation qui dominent. Dans le cas 2, les deux ont une influence commune et cela induit un terme logarithmique. Dans le cas 3, c'est la fonction f qui domine largement. On peut voir une preuve de ce théorème dans (Corben *et al.*, 2004).

Exemple : Qu'est-ce que cela donne sur le TRIFUSION ?

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) \approx 2T(\frac{n}{2}) + \Theta(n)$$

Prenez $a = 2, b = 2$, le nombre de noeuds de l'arbre d'exploration est donc $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$. Or $f(n) = \Theta(n)$, c'est donc le cas 2, et on obtient la complexité connue en $T(n) = \Theta(n \log n)$.

1.4 Analyse asymptotique expérimentale

Il est dans certains cas difficile d'obtenir des formules explicites pour la complexité d'un algorithme. Dans d'autres cas, il peut être utile de deviner quelle est la complexité théorique pour pouvoir mieux la prouver. Une approche possible est l'expérimentation. On mesure alors le temps d'exécution d'un programme pour des valeurs n de plus en plus grandes, ce qui donnent m couples $(n_i, T(n_i))$ où T est la fonction mesurant le temps d'exécution, que l'on suppose triés dans l'ordre croissant des n_i.

Sauf peut-être dans le cas où T est linéaire, un simple tracé de ces points n'est souvent pas suffisant pour deviner la complexité asymptotique de T. Nous disposons néanmoins d'un moyen pour déterminer la complexité de T dans le cas où T a la forme $T(n) = an^b$. En effet, en prenant le logarithme de T,

nous obtenons :

$$\begin{aligned} T(n) &= an^b \\ \Leftrightarrow \log(T(n)) &= \log a + b \log n \\ \Leftrightarrow y &= cst + bx, \end{aligned}$$

en choisissant $x = \log n$ et $y = \log(T(n))$. Prendre le logarithme des valeurs sur chaque axe, c'est se placer en échelle logarithmique. Dans cette échelle, le graphe de T est donc une droite de pente b où b est la puissance de la complexité. Il ne reste plus qu'à estimer cette pente sur le graphe pour déterminer une borne possible de la complexité de l'algorithme.

NB : il est important de prendre des n suffisamment grands pour que ce soit bien le comportement asymptotique qui domine dans le temps mesuré. Evidemment, cette méthode ne *prouve* rien. Elle donne des éléments de réflexion pour approfondir l'analyse.

En résumé : Analyse des fonctions récursives

— Pour analyser la complexité d'une fonction récursive, on se ramène à une suite récurrente paramétrée par un seul entier n, qui est généralement lié à la taille du problème.

— On dispose ensuite d'outils standards pour expliciter les complexités asymptotiques de la plupart des suites, comme les suites récurrentes linéaires ou les suites "diviser pour régner".