Chapitre 3

CALCUL DE COMPLEXITÉ DES ALGORITHMES ITÉRATIFS ET RÉCURSIFS

1. Coût uniforme et coût logarithmique

1.1. A propos de la notation O

Cette notation fait abstraction des détails liés à la machine mais aussi de certaines opérations de programmation.

Exemple 1 : comment vérifier qu'un élément est dans un tableau.

```
Fonction test(T : tableau [1..n]d'élément entier ;x,n,i : entier) :booléen
Début
   i←1 ;
   Tant que  (i ≤ n) et (T[i] ≠ x) alors
      i←i+1;
   fintant que;
   si (i ≤ n) alors
     retourner(vrai) ;
   sinon
     retourner(faux) ;
   finsi ;
Fin.
```

→ La complexité est en O(n).

Si on ajoute x à la fin du tableau,

```
Fonction test(T : tableau [1..n]d'élément entier ;x,n,i : entier) :booléen
Début
   i←1 ;
  si (T[n] = x) alors retourner(vrai) ;
  sinon
       T[n] ← x;
```

```
  finsi;
   Tant que  (i ≤ n) et (T[i] ≠ x) alors
      i←i+1;
   fintant que;
   si (i ≤ n) alors
     retourner(vrai) ;
   sinon
     retourner(faux) ;
finsi ;
Fin.
```

→ La complexité est en O(n).

Les opérations de programmation qui peuvent avoir une certaine importance disparaissent avec cette notation asymptotique.

2. Calcul de complexité des algorithmes itératifs

Les étapes de calcul de complexité sont les suivantes:

Etape 1 : Sous quel aspect doit-on considérer la donnée d'entrée ?

- Soit comme le nombre d'objets donnés → analyse uniforme (classique).
- Soit comme la taille des bits nécessaires pour la représentation → analyse logarithmique.

Pour un nombre x, il faut $\lceil \log_2(x+1) \rceil$ bits, ou $\lfloor \log_2(x) \rfloor + 1$

Etape 2 : Quelles sont les opérations élémentaires, i.e. celles qui se font en temps constant ?

- En analyse uniforme → Opérations arithmétiques +, -, *, /, %.
- En analyse logarithmique → Manipulation de bits : décalage, test à zéro, additions de bits.

Etape 3 : On se donne une base de données de complexité élémentaires et des règles sur les opérations.

- w(i, J) = w(i) + w(I). Le coût d'instructions séquentielles est le coût de l'une plus celui de l'autre.
- w(x ← e) = cost(e). Le coût d'une affectation est le coût du calcul/manipulation de la donnée.
- w(if(c) S_1 else S_2) = w(c) + max(w(S_1), w(S_2)). On paye la condition puis le pire des cas.
- w(while (c), S) = $\Sigma_{\text{itérations}}$(w(c) + w(S)). On paye le test et les opérations à chaque passage.

- w(for(S_1 ; S_2 ; S_3), S_4) = w(S_1) + $\Sigma_{\text{itérations}}$(w($S_2$) + w($S_3$) + w($S_4$)). ↔ S_1 ; while(S_3) { S_2,S_4}.
- w(récursivité) : coût de la récursivité.

Formules utiles pour le calcul

- ln fonction logarithme népérien (ou naturel), de base e
- $\log_a$ fonction logarithme de base a : $\log_a(x) = \ln x / \ln a$
- log fonction logarithme sans base précise, à une constante multiplicative près
- $\log_2$ fonction logarithme binaire, de base 2 : $\log_2(x) = \ln x / \ln 2$

Logarithme : $\log_b n = x \leftrightarrow b^x = n$, pour n > 0 et b > 1.

$$\sum_{i=1}^{n}(C.Fi + Oi) = \sum_{i=1}^{n} CFi + \sum_{i=1}^{n} Oi$$

$\sum_{i=1}^{n} O[Fi] = O[\sum_{i=1}^{n} Fi]$

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

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

$$\sum_{i=0}^{n} ix^i = \frac{1}{(1-x)^2} \quad \text{si } |x|<1$$

Problème :

- ⇨ O est un "pire des cas" : il n'indique donc pas que l'algorithme va prendre exactement ce temps, mais qu'il ne peut pas prendre davantage.
- ⇨ La notation O fait disparaître quelques constantes liées à la programmation dont les tests dans les boucles et de petites optimisations.

3. Calcul de complexité des algorithmes récursifs

- La complexité de l'algorithme récursif est donnée par son équation,
- La résoudre en utilisant un formulaire pour les cas standard.
- Le coût de la récursivité est donné par T(m) tel que :

  T(m) = α cas de base

  T(m) = c.T(f(m)) + g(m) équation de récurrence

Les paramètres sont :

- α : temps d'exécution dans le cas de base. En général O(1).
- c : nombre de fois qu'on fait un appel récursif.

Exemple 2 : diviser pour régner
Deux appels récursifs (sur les éléments de la première moitié, et de la seconde).
- g(m), le coût d'un appel récursif
- f(m), la taille de la donnée m lors de l'appel récursif, (f(m) = m – 1 ou m – 2).

Formulaire des solutions d'équations de récurrence où T(1) = O(1) :
1. $T(n) = T(n/2) + O(1)$ $\rightarrow T(n) = O(\log n)$
2. $T(n) = T(n-1) + O(\log n)$ $\rightarrow T(n) = O(n.\log n)$
3. $T(n) = c.T(n-1) + O(n^k)$ $\rightarrow T(n) = O(c^k)$
4. $T(n) = c.T(n/d) + O(n^k)$ $\rightarrow$ si $c = d^k$, alors $T(n) = O(n^k.\log n)$
   $\rightarrow$ si $c < d^k$, alors $T(n) = O(n^k)$
   $\rightarrow$ si $c > d^k$, alors $T(n) = O(n^{\log \text{ base d de c}})$

3.1. Master théorème, Équations de récurrences

Théorème 1. Les équations de récurrence :

$$\begin{cases} T(1) = c, \\ T(n) = aT(n/b) + cn, \quad n \geq 2 \end{cases}$$

avec a,b,c > 0 et où n/b représente soit [n/b] soit [n/b], ont pour solution :
- $a < b \Rightarrow T(n) \in \Theta(n)$
- $a = b \Rightarrow T(n) \in \Theta(n\log(n))$
- $a > b \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$

Théorème 2 . Les équations de récurrence :

$$\begin{cases} T(1) = c \\ T(n) = aT(n/b) + c, \quad n \geq 2 \end{cases}$$

avec $a \geq 1, b > 1, c > 0$ et où n/b représente soit [n/b] soit [n/b] ont pour solution :
- $a < b$ si $a = 1 \Rightarrow T(n) \in \Theta(\log(n))$
  si $a > 1 \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$
- $a = b \Rightarrow T(n) \in \Theta(n)$
- $a > b \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$

Remarque 1. Le cas où a=1 et b=2 correspond au processus dichotomique classique. Pour démontrer les deux théorèmes précédents, on peut commencer par considérer les entiers $n = b^t$.

Théorème 3. Les équations de récurrence :

$$\begin{cases} T(1) = c \\ T(n) = aT(n/b) + f(n) \text{ , } n \geq 2 \end{cases}$$

avec $a \geq 1, b > 1$ et où n/b représente soit $[n/b]$ soit $[n/b]$ ont pour solution :

- si $f(n) \in O(n^{\log_b(a)-\varepsilon})$ pour $\epsilon > 0 \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$
- si $f(n) \in \Theta(n^{\log_b(a)}) \Rightarrow T(n) \in \Theta(n^{\log_b(a)}\log(n)) = \Theta(f(n)\log(n))$
- si $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$ pour $\epsilon > 0$ et si $af(n/b) \leq df(n)$ avec $d < 1, \Rightarrow T(n) \in \Theta(f(n))$

Remarque 2. Ce théorème ne couvre pas tous les cas possibles de la fonction f. Enfin terminons par les équations d'autres schémas récursifs.

Théorème 4. Les équations de récurrence :

$$\begin{cases} T(1) = a \\ T(n) = bT(n-1) + a, \ n \geq 2 \end{cases}$$

avec $a > 0, b \geq 1$ ont pour solution :

- $b = 1 \Rightarrow T(n) = an \in \Theta(n)$
- $b \geq 1 \Rightarrow T(n) \in \Theta(b^{n-1})$

Remarque 3. Le problème des tours de Hanoï correspond au cas où $a = 1, b = 2$, et $T(n) \in \Theta(2^{n-1})$, donc un algorithme récursif exponentiel.

Théorème 5. Les équations de récurrence :

$$\begin{cases} T(1) = a \\ T(n) = bT(n-1) + an \text{ , } n \geq 2 \end{cases}$$

avec $a > 0$, $b \geq 1$ ont pour solution :

- $b = 1 \Rightarrow T(n) \in \Theta(n^2)$
- $b \geq 1 \Rightarrow T(n) \in \Theta(b^{n-1})$

Remarque 4. Le plus mauvais cas de l'algorithme Quicksort correspond au cas où $b = 1, a = 1$, cet algorithme a donc un comportement quadratique dans le pire des cas.

Propriétés.

1. $\begin{cases} T(1) = a \\ T(n) = T(n/5) + T(3n/10) + an, \ n \geq 2 \end{cases} \Rightarrow T(n) \in O(n)$
2. $\begin{cases} T(1) = a \\ T(n) = \sum_{i=1}^{i=k} ai \ T(\frac{n}{bi}) + an, \ n \geq 2 \end{cases} \Rightarrow$ si $\sum_{i=1}^{i=k} \frac{ai}{bi} < 1$ alors $T(n) \in O(n)$