926

Analyse des algorithmes : Complexité. Exemples.

Motivations : Calculer l'efficacité d'un algorithme A

I Cadre et généralités

1) Qu'est-ce que la complexité ?

Def 1 : On appelle données d'entrées l'ensemble des variables externes à l'algorithme sur lequel on va exécuter l'algorithme.

Ex 2 : Un algorithme de tri d'un tableau doit prendre en entrée un tableau.

Def 3 : La taille de l'entrée x est un entier n qui est défini grâce à une fonction $f : \{\text{entrées}\} \mapsto \mathbb{N}$

Ex 4 : Pour un mot $w = w_1 \dots w_n$, on peut dire $f(w) = n$, i.e. nbr de caractères.
Pour un tableau T de taille n, on pourrait dire $f(T) = n$ mais aussi l'espace mémoire nécessaire pour mettre le tableau.

Def 5 : La complexité est une fonction φ qui a une donnée d'entrée x renvoie la consommation d'une certaine ressource.
La ressource peut être le temps, l'espace mémoire, la chaleur dissipée lors de l'exécution, le nombre d'accès au disque dur externe, ...

Rmq 6 : En pratique pour la complexité temporelle, on compte le nombre d'opérations, de comparaisons, d'affectations ou d'appels récursifs par exemple.

Ex 7 : Tri Bulle en comptant le nombre de comparaisons il y en a $\sum_{i=1}^{n-1} i = \frac{n(n+1)}{2}$ pour toutes les entrées possibles

2) Notation de Landau

Parfois, on ne peut pas calculer la complexité exacte mais on peut en donner un ordre de grandeur

Def 8 : Soient $f, g : \mathbb{N} \to \mathbb{R}^+$
On note $O(g) = \{f$ telle qu'il existe $c > 0, n_0 \in \mathbb{N}$ / $0 \leq f(n) \leq c\,g(n)$ $\forall n \geq n_0\}$
et $\Theta(g) = \{f$ telle qu'il existe $c_1, c_2 > 0, n_0 \in \mathbb{N}$ / $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall n \geq n_0\}$

Remarque 9 : On écrira $f = O(g)$ (resp $= \Theta(g)$) pour $f \in O(g)$ (resp $\in \Theta(g)$).

Proposition 10 : $[f = O(g)$ et $g = O(f)] \iff f = \Theta(g)$
$f = \Theta(g) \iff g = \Theta(f)$

Exemple 11 : $n^2 + n = O(n^3)$ $\quad n^2 + n = O(n^2)$
et $n^2 = O(n^2 + n)$ d'où $n^2 + n = \Theta(n^2)$
mais $n^3 \neq O(n^2 + n)$

Exemple 12 : Pour le tri bulle de l'exemple 7, on a une complexité en $O(n^2)$

Prop 13 : L'ordre de croissance des O est :
$O(1)$; $O(\log n)$; $O(n)$; $O(n \log n)$; $O(n^2)$; $O(n^k)$; $O(k^n)$; $O(n!)$

3) Différentes approches de complexité

Def 14 : Soit $\Omega_n = \{$entrées de taille n pour l'algorithme considéré$\}$
Soit φ la fonction de complexité défini à la définition 5
On a $C_{pire}(n) = \max_{x \in \Omega_n} \varphi(x)$ $\quad C_{mieux}(n) = \min_{x \in \Omega_n} \varphi(x)$

distribution uniforme : $C_{moy}(n) = \frac{1}{|\Omega_n|} \sum_{x \in \Omega_n} \varphi(x)$ $\quad$ si $\varphi(x) = cste$ $\forall x \in \Omega_n$ on notera $C(n) = Cste$

II Premières techniques pour calculer la complexité

1) Calcul direct

On peut parfois calculer directement la complexité en dénombrant les opérations que l'on doit compter

Ex 15 : Recherche du maximum dans un tableau en comptant le nombre de comparaison
Pour un tableau de taille n, il y aura n comparaisons car une boucle for de taille n, $C(n) = O(n)$

2) Complexité moyenne vue comme calcul d'espérance

Prop 16 : Soit p la distribution des données d'entrées
On a en fait $C_{moy} = \sum_{x \in \Omega} p(x)\,\varphi(x)$

Rmq 17: Souvent on prend une distribution uniforme d'où la définition de C_{moy} à la Def 14.

Ex 18: Pour le tri rapide, en comptant le nombre de comparaisons on a $C_{moy}(n) = \Theta(n \log n)$

3) Calcul par la résolution de récurrences

On peut parfois exprimer la complexité pour une donnée de taille n par rapport à la complexité pour une donnée de taille strictement inférieure et donc de résoudre l'équation obtenu pour avoir la complexité.

Prop 19: Pour une suite récurrente linéaire d'ordre 1 $(u_n)_{n\in\mathbb{N}}$ avec $u_{n+1} = a u_n + b$ $\forall n \geq 0$ où $u_0 \in \mathbb{R}$, $a, b \in \mathbb{R}$

Alors $u_n = a^n(u_0 - l) + l$ où $l = \frac{b}{1-a}$ si $a \neq 1$

$u_n = bn + u_0$ si $a = 1$

Ex 20: Pour l'algorithme de calcul du factoriel où l'on compte le nombre de multiplication $C(n) = C(n-1) + 1$ donc $a = 1$, $b = 1$, $C(0) = 0$ d'où $C(n) = 1 \times n + 0 = n$

Prop 21: Master theorem

Soit $t : \mathbb{N} \to \mathbb{R}^+$ croissante à partir d'un certain rang

$n_0 \geq 0$, $b \geq 2$ entiers

$k \geq 0$, $a, c, d > 0$ réels positifs.

tels que $\begin{cases} t(n_0) = d \\ t(n) = a\, t(\frac{n}{b}) + c n^k \text{ pour } \frac{n}{n_0} \text{ puissance de } b \end{cases}$

Alors on a $t(n) = \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log_b(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$

DEV 1

Ex 22: • Tri Fusion $C(n) = \Theta(n \log n)$

• Algorithme de Strassen pour la multiplication matricielle en $\Theta(n^{\log 7})$

III Technique pour calculer la complexité amortie

Def 23: Soit une suite de k opérations $op_1, \ldots, op_k$ on appelle complexité amortie $C_{amo} = \max_{op_1 \ldots op_k} \frac{\sum_{i=1}^{k} C(op_i)}{k}$

Rmq 24: Le max veut simplement dire qu'on se place dans le pire cas

Exemple 25: que l'on va suivre dans toute la suite de cette partie

Le tableau dynamique où l'on compte "allocations + écriture".

Pour ajouter un élément dans le tableau T_n de taille n :

→ Soit il n'est pas rempli et on met l'élément dans la 1ère case vide

→ Soit il est rempli et on crée un tableau T_{2n}, on copie les n éléments de T_n dans T_{2n} et on met notre élément dans la case n+1.

1) Agrégat

C'est un calcul direct : on considère une suite de k ajouts à T_1

Il faut doubler $\lceil \log_2(k) \rceil$ fois la taille du tableau

$\text{nbr d'opérations} = \underbrace{k}_{\text{écriture}} + \underbrace{\sum_{i=0}^{\lceil \log_2(k) \rceil} 2^i}_{\text{copie}} \leq 3k$ ainsi $C_{amo} = 3$ (au pire)

2) Comptable

On définit un crédit et une dépense pour chaque type d'opérations, en mettant des crédits plus forts aux opérations peu coûteuses pour rattraper les lourdes dépenses des opérations coûteuses, il faut néanmoins avoir $\sum_{i=1}^{k} cred(op_i) - \sum_{i=1}^{k} dép(op_i) \geq 0$. On a $C_{amo}(op) = C_{réel}(op) + cred(op) - dép(op)$

Dans notre exemple :

T non plein : $cred(op) = 2$; $dép(op) = 0$

T plein : $cred(op) = 2$; $dép(op) = |T|$

D'où $C_{amo}(op) = \begin{cases} 1 + 2 - 0 \\ (|T|+1) + 2 - |T| \end{cases} = 3$

3) Potentiel

Au lieu d'assigner des crédits à des opérations on va associer une énergie potentielle Φ à la structure donnée elle-même. (il faut que $\Phi(\text{structure vide}) = 0$ et $\Phi(T) \geq 0$ $\forall T$) on a alors

$C_{amo}(op) = C_{réel}(op) + \Phi(T') - \Phi(T)$ pour $T \xrightarrow{op} T'$

Dans notre exemple : $\Phi(T) = 2 \times \text{cases remplies} - \#\text{cases totales}$

$C_{amo}(op) = \left.\begin{cases} 1 + 2 & \text{si pas de réallocation} \\ (|T|+1) + 2 - |T| & \text{sinon} \end{cases}\right\} = 3$

IV. Amélioration de la complexité d'un algorithme

On peut parfois améliorer la complexité d'un algorithme en choisissant une structure de données plus adaptée. Cependant pour certain problème, il existe une complexité minimale. On ne pourra pas aller en dessous, donc on voudrait atteindre cette borne

1) Complexité minimale pour une classe d'algorithme

Prop 26 : Pour un algorithme de tri par comparaison la complexité est au moins $O(n \log n)$

Ex 27 : Tri par insertion en $O(n^2) \geq O(n \log n)$

On peut trouver des algo de tri qui atteignent cette borne
- Tri fusion en $O(n \log n)$
- Tri par tas en $O(n \log n)$

2) Amélioration par des structures de données adaptées

Ex 28 : Pour le tri par tas on utilise une structure de tas pour obtenir un tri efficace.

Rmq 29 : La complexité peut dépendre de la façon d'implémenter la structure que l'on utilise

Ex 30 : Pour les algorithmes sur les graphes, on peut implémenter les graphes de différentes manières :
- liste de tous les couples (u,v) si il y a une arête de u à v dans le graphe
- liste d'adjacence
- matrice d'adjacence

Rmq 31 : En général, s'il y a beaucoup d'arêtes qui partent de chaque sommet, il est mieux d'utiliser des matrices d'adjacence, dans le cas contraire, les listes d'adjacence sont souvent plus performantes.

Ex 32 : Algorithme de Prim → liste d'adjacence
Algorithme de Floyd Warshall → matrice d'adjacence

On peut aussi ajouter une structure ou changer la structure pour améliorer la complexité.

Ex 33 : Amélioration de l'algo de Morris Pratt en Knuth Morris Pratt, en remplaçant le graphe des occurrences (qui prend une taille $m|\Sigma|$ où m est la taille du motif à rechercher et Σ l'alphabet) par un tableau des tailles des plus longs préfixes (de taille m)

Algorithme de Knuth Morris Pratt : Exemple et complexité — DEV 2

3) Compromis temps/espace

Souvent on peut améliorer la complexité spatiale au détriment de la complexité temporelle et réciproquement

Prop 34 : On peut parfois utiliser la programmation dynamique pour faire des économies de temps et d'espace

Ex 35 : Calculer les nombres de Fibonacci
- → Naïvement (récursif) en $O(2^n)$ pour le temps et en $O(1)$ pour l'espace si on compte le nombre de variables nécessaires
- → en utilisant 2 variables (pour calculer un seul nombre) en $O(n)$ pour le temps en $O(1)$ pour l'espace
- → programmation dynamique pour calculer les n premiers nombres en $O(n)$ pour le temps en $O(n)$ pour l'espace

Rmq 36 : Si on augmente trop la complexité spatiale, cela peut ralentir l'algorithme en pratique car l'accès à la mémoire sera plus long.