926 Analyse des algorithmes : complexité. Exemples.

(926)

Ref : Beauquier _ Eléments d'algorithmique -> www-igm.univ-mlv.fr/berstel/Elements/Elements.p
Cormen & all _ Introduction to algorithms
Froidevaux _ Types de données et algorithmes -> www.lri.fr/~mcg/PDF/FroidevauxGaudelSoria.pdf
Flajolet-Sedgewick. Introduction à l'analyse des algorithmes <- partie 1/ les séries gen.

I_CONTEXTE

<- Froidevaux

A. Algorithmes

Def 1 : Un algorithme est la composition d'un ensemble fini d'étapes, chacune étant formée d'un nombre fini d'opérations dont chacune est :
- définie de façon rigoureuse et non ambigüe
- effective, c'est-à-dire pouvant être effectivement réalisée par une machine.

Rq 2 : Quelle que soit la donnée sur laquelle il travaille, l'algorithme doit toujours se terminer après un nombre fini d'opérations et renvoyer un résultat.

Def 3 : Pour un algorithme A, on définit, parmi les opérations possibles dans A, un ensemble d'opérations fondamentales auxquelles on attribue un coût.

Rq 4 : On considère qu'on a accès/on peut mettre un élément en mémoire en temps constant. De plus, on ne peut traiter qu'une seule opération à la fois.

Ex 5 : (i) Algorithme de tri de tableau : les opérations fondamentales sont (l'accès / l'écriture d'une donnée) les comparaisons entre éléments.
(ii) Algorithme de multiplications de matrices : les opérations fondamentales sont (l'accès/l'écriture d'une donnée) les additions et les multiplications.

B_ Complexité

Def 6 : Soit A un algorithme, et x une entrée possible pour l'algorithme A.
(i) complexité sur l'entrée x : $c(x)$ = nombre d'opérations fondamentales effectuées par l'algorithme A sur l'entrée x.
(ii) complexité spatiale sur l'entrée x : $c_s(x)$ = nombre de blocs mémoires utilisés par l'algorithme A sur l'entrée x.

Ex 7 : Recherche d'un élément x dans un tableau T de taille n. $c_s(x,T) = n+1$ et si x est dans le tableau à la position i : $c(x,T) = i$, et si x n'est pas dans le tableau : $c(x,T) = n$, avec pour opérations fondamentales l'accès aux éléments du tableau, en coût 1, et la comparaison entre éléments, en coût 1.

Prop 8 : Pour A un algorithme, x une entrée de taille de n blocs mémoires, et pour des blocs mémoires de taille k : $c_s(x) \le c(x)$ et $c(x) \le 2^{c_s(x) \times k}$

Def 9 : Soit A un algorithme et D_n l'ensemble des entrées possibles pour A, de taille n.
(i) Complexité dans le meilleur des cas :
$$C_{min}(n) = \min\{c(x), x \in D_n\}$$
(ii) Complexité dans le pire des cas :
$$C_{max}(n) = \max\{c(x), x \in D_n\}$$
(iii) Complexité moyenne suivant $p(x)$, la probabilité que x soit l'entrée de l'algorithme
$$C_{moy}(n) = \sum_{x \in D_n} p(x)c(x).$$

Rq 10 : le choix du modèle de probabilité doit être adapté au problème : pour la recherche d'un mot dans un texte, la probabilité d'avoir le mot "est" est bien plus grande que pour "xylophone".

Ex 11 : Soit T un tableau de taille n dont les éléments sont dans $\{1,..,k\}$. On cherche si i est à l'intérieur. L'algorithme en annexe A a pour complexité :
$$C_{max}(n) = n \text{ et } C_{moy}(n) = \sum_{i=1}^{n} \left(\frac{9}{10}\right)^{i-1} \times \frac{1}{10} + \left(\frac{9}{10}\right)^{n}.$$

Prop 12 : Pour A un algorithme et $n \in \mathbb{N}$,
$$C_{min}(n) \le C_{moy}(n) \le C_{max}(n)$$

C - Comparaison d'algorithmes.

Le calcul de complexité peut être relativement compliqué, et en général, l'ordre de grandeur de complexité est suffisant.

Def 13 (Notations de Landau) : Soient $f, g : \mathbb{N} \to \mathbb{R}^+$. On note :

(i) $f(n) = O(g(n)) \Leftrightarrow \exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq c g(n)$.

(ii) $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$

(iii) $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ et $g(n) = O(f(n))$.

DVT 1 Ex 14 : Comparaisons des complexités des tris fusion et rapide (DEV)

II. METHODES DE CALCUL

A. "A la main"

Principe 15 : On compte pour chaque opération de l'algorithme le nombre de fois où elle sera exécutée.

DVT 2 Ex 16 : Complexité de l'algorithme de Hopcroft (DEV) (Carton + beauquier)

Principe 17 : Les algorithmes récursifs permettent en général d'obtenir des relations de récurrence que l'on peut soit résoudre directement, ou pour lesquelles il faut des méthodes plus poussées.

B. Récursion avec partitions.

Th 18 : Soient $a \geq 1$, $b > 1$, $f : \mathbb{N} \to \mathbb{R}$ et $c : \mathbb{N} \to \mathbb{R}^+$ avec $c(n) = ac(n/b) + f(n)$ avec $n/b = \lfloor n/b \rfloor$ ou $\lceil n/b \rceil$, alors :

(i) $f(n) = O(n^{\log_b a - \varepsilon})$ avec $\varepsilon > 0 \Rightarrow c(n) = \Theta(n^{\log_b a})$.

(ii) $f(n) = \Theta(n^{\log_b a}) \Rightarrow c(n) = \Theta(n^{\log_b a} \log n)$

(iii) $f(n) = \Omega(n^{\log_b a + \varepsilon})$ avec $\varepsilon > 0$ et $af(n/b) \leq \alpha f(n)$ pour $\alpha < 1$ et n suffisamment grand $\Rightarrow c(n) = \Theta(f(n))$

Ex 19 : Algorithme de Strassen : $c(n) = 7c(n/2) + \Theta(n^2)$

$\Rightarrow$ Cas 1 : $\Theta(n^2) = O(n^{\log_2 7 - \varepsilon})$ avec $\varepsilon = \log_2 7 - 2 \Rightarrow c(n) = \Theta(n^{\log_2 7})$

C. Récurrences linéaires

Principe 20 : Soit $c : \mathbb{N} \to \mathbb{R}$, $k \in \mathbb{N}^*$, $a_0, \dots, a_{k-1} \in \mathbb{R}$. Si c vérifie :

$c(n+k) = a_{k-1} c(n+k-1) + \dots + a_0 c(n)$ (E).

-> on pose $C(x) = \sum_{n \geq 0} c(n) x^n$

-> on introduit $C(x)$ dans E

-> on résoud l'équation en C.

-> on développe $C(x)$ en série entière.

-> on récupère la valeur de $c(n)$.

Ex 21 : L'algorithme naïf de calcul de la suite de Fibonacci vérifie :

$c(n+2) = c(n+1) + c(n) \Rightarrow c(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{-2}{1+\sqrt{5}}\right)^{n+1} \right)$. (en annexe)

III. ANALYSE AMORTIE ← Cormen

A. Principe

Principe 22 : Soit un algorithme A dans lequel on effectue au plus n fois une opération de coût k. Naïvement, on peut majorer par nk.

Ex 23 : On considère un tableau T de taille k rempli de 0 et de 1. On utilise l'algorithme en annexe qui incrémente A de 1, et on veut le faire n fois. On obtient $O(nk)$ car n passages de complexité $O(k)$.

Principe 24 : On note c_i le coût de la i-ème opération. On cherche a_i le coût amorti tel que $n c_{max} \geq \sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} c_i$.

B. Méthode de l'agrégation

Principe 25 : On commence par calculer c_{max} et on pose $a_i = \frac{c_{max}}{n}$, pour tout $i \in \{1, \dots, n\}$.

Ex 26 : On reprend l'algorithme de l'exemple 23. Sur une suite d'incrémentation, en partant de 0, on a $c_{max}(n) = \sum_{i=0}^{\log_2(n)} \left\lfloor \frac{n}{2^i} \right\rfloor \leq 2n$.

$\Rightarrow$ on pose pour $i \in \{1, \dots, n\}$, $a_i = \frac{2n}{n} = 2 \Rightarrow n$ passages à un coût de 2 en amorti $\Rightarrow$ complexité de $2n$.

C. Méthode comptable

Principe 27 : Certaines opérations ne peuvent pas avoir lieu avant d'autres. On surévalue le coût des dernières et sous-évalue le coût des premières.

Ex 28 : On reprend l'algorithme de l'exemple 23. Toujours en partant de 0, on ne peut avoir $T[i]$ qui vaut 1, et donc l'opération $T[i] \leftarrow 0$, tant qu'on n'a pas fait $T[i] \leftarrow 1$. On ne peut plus refaire $T[i] \leftarrow 0$ ensuite sans un autre $T[i] \leftarrow 1$.

$\Rightarrow a(T[i] \leftarrow 1) = c(T[i] \leftarrow 1) + c(T[i] \leftarrow 0) = 2$ et $a(T[i] \leftarrow 0) = 0$.

$\Rightarrow$ A chaque incrémentation, on a au plus une fois l'opération $T[i] \leftarrow 1 \Rightarrow$ complexité de $2n$.

D_ Méthode du potentiel

Principe 29 : On associe une fonction de potentiel φ à la structure des données après chaque opération. Si D_{i-1} est la structure avant l'opération i de coût c_i et D_i est sa structure après, on pose : $a_i = c_i + \varphi(D_i) - \varphi(D_{i-1})$

Ex 30 : On reprend l'algorithme de l'exemple 23. On pose φ le nombre de 1, et on note T_i le tableau T après i incrémentations. Si la i-ème incrémentation remet t_i bits à 0, cette incrémentation avait un coût de $t_i + 1$ au plus. Si $\varphi(T_i) = 0$ $\Rightarrow \varphi(T_{i-1}) = t_i = k$. Si $\varphi(T_i) > 0 \Rightarrow \varphi(T_i) = \varphi(T_{i-1}) - t_i + 1$. Alors, $\Rightarrow \varphi(T_i) - \varphi(T_{i-1}) \leq 1 - t_i \Rightarrow a_i = c_i + \varphi(T_i) - \varphi(T_{i-1}) \leq t_i + 1 + (1 - t_i) = 2$.
On obtient encore une complexité de $2n$.

IV AMELIORATION DE LA COMPLEXITE

A_ Compromis temps/espace

Principe 31 : On peut parfois diminuer la complexité en temps d'un algorithme au prix d'une plus grande complexité en espace, et inversement.

Ex 32 : Pour le calcul du n-ème terme de la suite de Fibonacci, on peut utiliser différents algorithmes :
(i) l'algorithme naïf de l'exemple 21 est en $O(\varphi^n)$ en temps et $O(1)$ en espace.
(ii) l'algorithme de programmation dynamique en annexe D est en $O(n)$ en temps et en $O(n)$ en espace.

Rq 33 : Si on augmente trop la complexité en espace, cela peut aussi ralentir, en pratique, l'algorithme, car l'accès en mémoire secondaire est beaucoup plus lent que l'accès en mémoire principale.

B_ Structures de données

Principe 34 : Selon la structure de donnée utilisée dans un algorithme, et la façon dont elle est implémentée, on peut changer le coût des opérations fondamentales, et donc la complexité de l'algorithme.

DVT3

Ex 35 : Comparaison de l'algorithme de Dijkstra entre l'utilisation de tableaux ou de tas binaires (DEV)

C_ MODELE DE CALCUL

Principe 36 : En utilisant un autre modèle de calcul, on peut obtenir de meilleures complexités. Pour rester réaliste, on peut se placer dans le cas où des calculs peuvent être faits en parallèle, comme dans un ordinateur avec plusieurs processeurs.

Ex 37 : On peut améliorer la complexité du tri fusion à $\Theta(n)$ facilement, et pour un nombre infini d'unités de calcul, on peut atteindre $\Theta(\log^3 n)$.

cplxité en tps pr all

A_ Exemple 11

```
Cherche(T, i) :
| Pour j = 0 à T.taille
| | Si T[j] = i
| L L Renvoyer Vrai
L Renvoyer Faux
```

B_ Exemple 21

```
Fibo(n)
| Si n ≤ 1
| L Renvoyer n
| Sinon
L L Renvoyer Fibo(n-1) + Fibo(n-2)
```

C_ Exemple 23

```
Incrementer(T) :
| i ← 0
| Tant que T[i] = 1 et i < k
| | T[i] ← 0
| L i ← i+1
| Si i < k
L L T[i] ← 1
```

D_ Exemple 32

```
Fibo_ProgD(n)
| T tableau de taille n
| T[0] ← 0
| T[1] ← 1
| Pour i = 2 à n
| L T[i] ← T[i-1] + T[i-2]
L Renvoyer T[n].
```