Complexité d'un algorithme

Alix Munier-Kordon et Maryse Pelletier

LIP6
Université P. et M. Curie
Paris

2I003 Initiation à l'algorithmique

Plan du cours

Questions au sujet de l'évaluation d'un algorithme

1. Est-ce que l'algorithme résout le problème ?
   - terminaison
   - validité
2. Quelle est la complexité de l'algorithme ?
   - en temps de calcul
   - en taille mémoire

Objet de ce cours : la complexité en temps de calcul.

Taille de codage des paramètres d'un algorithme

Definition

La *taille de codage* d'un paramètre est une évaluation, la plus "raisonnable" possible, de la place nécessaire en mémoire pour le stocker.

Quelle est la taille de stockage d'un entier ? d'un tableau d'entiers ?

Complexité d'un algorithme

Definition

La complexité d'un algorithme est une évaluation du nombre d'instructions élémentaires [1] dans une exécution de l'algorithme.

On l'exprime en fonction de la taille de codage des paramètres.

On en calcule un *ordre de grandeur* (notations de Landau).

[1] Parfois on se concentre sur une instruction élémentaire représentative.

Pire cas, meilleur cas

Complexité pire cas : on évalue le nombre d'instructions dans le pire des cas (borne supérieure).

Complexité meilleur cas : on évalue le nombre d'instructions dans le meilleur des cas (borne inférieure).

Exemple : recherche séquentielle d'un élément x dans un tableau T de taille n.
Instruction élémentaire représentative : comparaison.

- **Pire cas :** x en dernière place de T ou pas présent $\rightarrow$ n comparaisons
- **Meilleur cas :** x en première place de T $\rightarrow$ 1 comparaison

Par pessimisme, on identifie la complexité d'un algorithme avec sa complexité dans le pire des cas.

Notations de Landau : Θ, $\mathcal{O}$ et Ω

Definition

Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}_+$:

1. $f \in \mathcal{O}(g)$ si $\exists D > 0$ et $n_0 \geq 0$ tels que
$$\forall n > n_0 \quad f(n) \leq Dg(n).$$
2. $f \in \Omega(g)$ si $\exists C > 0$ et $n_0 \geq 0$ tels que,
$$\forall n > n_0 \quad Cg(n) \leq f(n).$$
3. $f \in \Theta(g)$ si $\exists C > 0$, $D > 0$ et $n_0 \geq 0$ tels que,
$$\forall n > n_0 \quad Cg(n) \leq f(n) \leq Dg(n).$$

Exemples

- $5n + 3 \in \mathcal{O}(n^2)$
- $n^5 \in \mathcal{O}(2^n)$
- $5n^2 + 3n + 4 \in \Theta(n^2)$

Classement des ordres de grandeur

Inclusions entre ordres de grandeur courants :

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n)$$

Un algorithme de complexité logarithmique est meilleur qu'un algorithme de complexité linéaire, meilleur qu'un algorithme de complexité quadratique, etc.

Un algorithme de complexité exponentielle est à bannir.

Comparaison de complexités

Avec une durée de 10^{-6} secondes par instruction, on obtient les durées suivantes pour 100 instructions[2] :

complexité	durée
$\log n$	$4,60 \times 10^{-6}$
n	10^{-4}
$n \log n$	$4,60 \times 10^{-4}$
n^2	10^{-2}
n^3	1
2^n	$\approx 1,27 \times 10^{24}$

Remarque : $1,27 \times 10^{24}$ secondes $\approx 4 \times 10^{16}$ ans !

[2] $\log n$: logarithme népérien

Somme des entiers

Pour $n \in \mathbb{N}$, on veut calculer la somme $Som(n)$ des entiers de 0 à n.
Autrement dit :

$$Som(n) = \sum_{0}^{n} i$$

Cette somme vaut 0 si $n = 0$.

Somme des entiers, en itératif

Algorithme itératif calculant la somme $Som(n)$:

```
def somIte(n):
    res = 0
    for i in range(1, n + 1):
        res = res + i
    return res
```

Complexité en nombre d'additions.

Soit c le nombre total d'additions et c_i le nombre d'additions dans le tour de boucle i.
Alors $c_i = 1$ pour tout $i \in \{1, \ldots, n\}$ et

$$c = \sum_{i=1}^{n} c_i = n$$

La complexité est en $\Theta(n)$, elle est *linéaire*.

Somme des entiers, en récursif

Remarquons que $Som(n) = Som(n-1) + n$ si $n > 0$ et $Som(0) = 0$.
Algorithme récursif calculant la somme $Som(n)$:

```
def somRec(n):
    if n == 0:
        return 0
    else:
        return n + somRec(n - 1)
```

Complexité en nombre d'additions.

Soit u_n le nombre d'additions effectuées par l'appel `somRec(n)`.
Alors u_n est la suite récurrente définie par :

$$u_n = u_{n-1} + 1 \quad \text{si } n > 0 \quad \text{et} \quad u_0 = 0$$

Par substitution :

$$u_n = u_{n-1} + 1 = u_{n-2} + 2 = \ldots = u_0 + n = n$$

La complexité est en $\Theta(n)$.

Somme des produits

Pour $n \in \mathbb{N}$, on veut calculer la somme $SomProd(n)$ de tous les produits $i * j$ pour $1 \leq j \leq i \leq n$. Autrement dit :

$$SomProd(n) = \sum_{1 \leq j \leq i \leq n} i * j$$

Cette somme vaut 0 si $n = 0$.

Somme des produits, en itératif

Algorithme itératif calculant la somme des produits *SomProd*(n) :

```
def somProdIte(n):
    res = 0
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            res = res + i * j
    return res
```

Complexité en nombre d'opérations arithmétiques $(+, *)$.

Soit d le nombre total d'opérations et d_i le nombre d'opérations effectuées dans le tour de boucle i. Pour i fixé, soit $d_{i,j}$ le nombre d'opérations effectuées dans le tour de boucle j.

Alors $d_{i,j} = 2$ et $d_i = \sum_{j=1}^{i} d_{i,j} = 2 * i$.

D'où $d = \sum_{i=1}^{n} d_i = 2 * \sum_{i=1}^{n} i = n(n+1)$

La complexité est en $\Theta(n^2)$, elle est *quadratique*.

Somme des produits, en récursif

Pour écrire un algorithme récursif de calcul de la somme des produits, il faut donner une définition récursive de *SomProd*. Pour cela, remarquons que :

$$
\begin{aligned}
SomProd(n) &= \sum_{1\leq j\leq i\leq n} i * j \\
&= \sum_{1\leq j\leq i\leq n-1} i * j + \sum_{1\leq j\leq n} n * j \\
&= SomProd(n-1) + n * \sum_{1\leq j\leq n} j \\
&= SomProd(n-1) + n * Som(n)
\end{aligned}
$$

Somme des produits, en récursif

Algorithme récursif calculant la somme des produits *SomProd*(n) :

```
def somProdRec(n):
    if n == 0:
        return 0
    else:
        return somProdRec(n - 1) + n * somRec(n)
```

Complexité en nombre d'opérations ($+$, $*$).

Soit v_n le nombre d'opérations effectuées par l'appel `somProdRec(n)`.
Rappelons que `somRec(n)` effectue n additions.
v_n est donc la suite récurrente définie par :

$$v_n = v_{n-1} + n + 2 \quad \text{si } n > 0 \quad \text{et} \quad v_0 = 0$$

Somme des produits, en récursif

$$v_n = v_{n-1} + n + 2 \quad \text{si } n > 0 \quad \text{et} \quad v_0 = 0$$

Par substitution :

$$\begin{aligned}
v_n &= v_{n-1} + n + 2 \\
&= v_{n-2} + (n+1) + (n+2) \\
&= v_{n-3} + n + (n+1) + (n+2) \\
&= \ldots \\
&= v_0 + 3 + 4 + \ldots + n + (n+1) + (n+2) \\
&= \frac{(n+2)(n+3)}{2} - (1+2) \\
&= \frac{n(n+5)}{2}
\end{aligned}$$

La complexité est en $\Theta(n^2)$.

Fibonacci, en itératif

Algorithme itératif calculant F_n :

```
def fibIte(n):
    if (n == 0):
        return 0
    else:
        x = 0 ; y = 1
        for i in range(2, n + 1):
            z = x + y ; x = y ; y = z
        return y
```

Complexité en nombre d'additions.

Soit c le nombre total d'additions et c_i le nombre d'additions dans le tour de boucle i.
Alors $c_i = 1$ et

$$c = \sum_{i=2}^{n} c_i = n - 1$$

La complexité en $\Theta(n)$.
Remarque : la complexité en nombre d'affectations est aussi linéaire.

Fibonacci, en récursif

Algorithme récursif calculant F_n :

```
def fibRec(n):
    if (n == 0) or (n == 1):
        return n
    else:
        return fibRec(n - 1) + fibRec(n - 2)
```

Complexité en nombre d'additions.

Soit u_n le nombre d'additions effectuées par l'appel `fibRec(n)`.
Alors u_n est la suite récurrente définie par :

$$u_n = u_{n-1} + u_{n-2} + 1 \quad \text{si } n > 1 \quad \text{et} \quad u_0 = u_1 = 0$$

Fibonacci, en récursif

$$u_n = u_{n-1} + u_{n-2} + 1 \quad \text{si } n > 1 \quad \text{et} \quad u_0 = u_1 = 0$$

Theorem

$u_n = F_{n+1} - 1.$

Preuve par récurrence.

Theorem

$F_n \geq \frac{1}{\sqrt{5}}(\varphi^n - 1)$ où φ est le nombre d'or : $\varphi = \frac{1+\sqrt{5}}{2} \approx 1,618.$

Preuve par récurrence.

La complexité est en $\Theta(\varphi^n)$. Comme $\varphi > 1$, elle est exponentielle.
Elle croît très vite, par exemple : $\varphi^{100} > 7,9 * 10^{20}$.

Calcul de la puissance : un premier algorithme

Algorithme basé sur la définition récursive "naturelle" de x^n :

$$x^n = x * x^{n-1} \quad \text{si} \quad n > 0 \quad \text{et} \quad x^0 = 1$$

```
def puissSeq(x, n):
    if (n == 0)):
        return 1
    else:
        return x * puissSeq(x, n - 1)
```

La complexité, en nombre de multiplications, est en $\Theta(n)$.

Calcul de la puissance, par dichotomie

Algorithme calculant x^n par dichotomie :

```
def puissDicho(x, n):
    if (n == 0):
        return 1
    elif n == 1:
        return x
    else:
        if n % 2 == 0:
            return carre(puissDicho(x, n // 2))
        else:
            return carre(puissDicho(x, n // 2)) * x
```

où la fonction `carre` est ainsi définie :

```
def carre(x):
    return x * x
```

Soit u_n le nombre de multiplications effectuées par l'appel `puissDicho(n)`.

Calcul de la puissance, par dichotomie

$$u_n = \begin{cases} 0 & \text{si } n = 0 \text{ ou } n = 1 \\ u_{n \div 2} + 1 & \text{si } n \text{ pair et } n > 1 \\ u_{n \div 2} + 2 & \text{si } n \text{ impair et } n > 1 \end{cases}$$

Dans tous les cas, pour $n > 1$, on a $u_n \leq u_{n_1} + 2$ où $n_1 = n \div 2$.

$$\begin{array}{rcll} u_n & \leq & u_{n_1} + 2 & \text{avec } n_1 = n \div 2 \\ & \leq & u_{n_2} + 4 & \text{avec } n_2 = n_1 \div 2 = n \div 2^2 \\ & \leq & u_{n_3} + 6 & \text{avec } n_3 = n_2 \div 2 = n \div 2^3 \\ & \leq & \ldots & \\ & \leq & u_{n_k} + 2 * k & \text{avec } n_k = n \div 2^k \end{array}$$

Les calculs s'arrêtent lorsque $n_k = 1$, c'est-à-dire lorsque $k = \lfloor log_2(n) \rfloor$.
Donc $u_n \leq 2 * \lfloor log_2(n) \rfloor$.
La complexité est en $\mathcal{O}(\log_2(n))$, elle est logarithmique.

Conclusion

Un même problème peut être résolu par différents algorithmes.

Il est important de connaître un ordre de grandeur de la complexité de chaque algorithme.

Il faut proscrire les algorithmes de complexité exponentielle.