Cours 1 : Complexité

Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@lifl.fr

Rappels

Complexité en temps

C'est le temps mis par un algorithme pour traiter une donnée.
C'est une fonction de la taille de la donnée.

Complexité en espace

C'est l'espace mémoire utilisé par un algorithme pour traiter une donnée en plus de la taille de la donnée.
C'est une fonction de la taille de la donnée.

Calcul de la somme des n premiers entiers

```
function somme (n : CARDINAL) : CARDINAL;
  var s,i : CARDINAL;
begin
  s := 0;
  for i := 0 to n do
    s := s + i;
  somme := s;
end {somme};
```

Quelles sont les opérations effectuées ?

$$\begin{array}{cl} + & (n+1) \\ := & 1 + (n+1) + 1 \end{array}$$

Quelle est la quantité de mémoire utilisée ?

i, s 2

Calcul de la somme des n premiers entiers

On sait que $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$,

```
function somme (n : CARDINAL) : CARDINAL;
var
  s : CARDINAL;
begin
  s := (n * (n + 1)) / 2;
  somme := s;
end {somme};
```

Quelles sont les opérations effectuées ?

$$\begin{array}{cc} * & 1 \\ + & 1 \\ / & 1 \end{array}$$

Quelle est la quantité de mémoire utilisée ?

$$\begin{array}{cc} \mathrm{s} & 1 \end{array}$$

Calcul de la somme des *n* premiers entiers

```
function somme (n : CARDINAL) : CARDINAL;
var
  i : CARDINAL;
  t : TABLEAU;
begin
  t(0) := 0;
  for i := 1 to n do
    t(i) := t(i-1) + i;
  somme := t(n);
end {somme};
```

Quelles sont les opérations effectuées ?

+	n
:=	1 + n + 1

Quelle est la quantité de mémoire utilisée ?

t	n+1

Calcul de la somme des n premiers entiers

On sait que $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-2} i$,

```
function somme (n : CARDINAL) : CARDINAL;
var
  s : CARDINAL;
begin
  if n = 0 then
    s := 0
  else
    s := n + somme(n-1);
  somme := s;
end {somme};
```

Quelles sont les opérations effectuées ?

$$
\begin{array}{ll}
+ & 0 \text{ si } n = 0 \\
 & 1 + \text{nombre de + pour } n-1 \text{ sinon} \\
:= & 2 \text{ si } n = 0 \\
 & 2 + \text{nombre de := pour } n-1 \text{ sinon}
\end{array}
$$

Quelle est la quantité de mémoire utilisée ?

On en déduit que le nombre d'additions est n par récurrence.
On note $c(n)$ le nombre d'additions. On pose comme hypothèse que $c(n) = n$

- pour $n = 0$, c'est vrai.
- supposons que $c(n) = n$ pour tout $n \leq m$, m fixé.
- que se passe-t-il au rang $m + 1$

$$\begin{aligned} c(m+1) \quad &= 1 + c(m) \\ &= 1 + n \text{ par hypothèse} \end{aligned}$$

Calcul de la somme des *n* premiers entiers

Différentes complexités en temps

version	1	2	3	4
temps	2n+4	3	2n+2	3n+2
espace	2	2	n+2	n+1

Suivant l'algorithme, on arrive au même résultat mais pas nécessairement dans le même temps ni en utilisant la même quantité de mémoire.

Exemples 1/3

```
procedure max (const t : TABLEAU; out max : ELEMENT);
var
  m : ELEMENT;
  i : CARDINAL;
begin
  m := t[low(t)];
  for i := low(t) + 1 to high(t) do
    if m < t[i] then m := t[i];
  max := m;
end {max};
```

- Dans cet exemple, c'est la comparaison d'ELEMENT qui nous intéresse.
- On note $c(n)$ le nombre de comparaisons d'éléments du tableau `t`, de taille n,

$$c(n) = \mathtt{high}(\mathtt{t}) - (\mathtt{low}(\mathtt{t}) + 1) + 1 = n - 1$$

Exemples 2/3

```
function copie (t : TABLEAU) : TABLEAU;
var
  i : CARDINAL;
  u : TABLEAU;
begin
  for i := low(t) to hight(t) do
    u(i) := t(i);
  copie := u;
end {copie};
```

- Dans cet exemple, c'est l'espace mémoire occupé par des ELEMENT qui nous intéresse.
- On note $e(n)$ l'espace mémmoire occupé pour un tableau de taille n :

$$e(n) = n$$

Exemple 3/3

```
function estPresent (t : TABLEAU; e : ELEMENT) : BOOLEAN;
var
  i : CARDINAL;
begin
  i := low(t);
  while i < high(t) and t(i) /= e do
    inc(i);
  estPresent := (t(i) = e);
end {estPresent};
```

- e(n) = 0;
- c(n) ? dépend de la position de e dans t
  - si `t(low(t)) = e` $\mapsto c(n) = 1$
  - si e n'est pas présent $\mapsto c(n) = n$

Différents cas

- pire des cas : correspond à une donnée ou un ensemble de données pour lesquelles l'algorithme s'exécute en temps maximal (ou avec un espace maximal)
  dans l'exemple, quand e n'est pas présent
- meilleur des cas : correspond à une donnée ou un ensemble de données pour lesquelles l'algorithme s'exécute en temps minimal (ou avec un espace minimal)
  dans l'exemple, quand e est en première position
- moyenne : correspond à la complexité moyenne sur tous les types de données possible
  dans l'exemple . . . il faudrait savoir dénombrer tous les cas et réaliser la moyenne

Comportement asymptotique

Supposons avoir deux algorithmes A et B, de complexité en temps respective $c_A(n) = n$ et $c_B(n) = 3 \times n + 100$, ont-ils une complexité réellement différente ?

Donnée de petite taille				Donnée de grande taille			
n	A	B	A/B	n	A	B	A/B
10	10	130	0.07	10^6	1000000	3000100	0.33
100	100	400	0.25	10^9	1000000000	3000000100	0.33

Lorsque n devient grand, la complexité est du même ordre : de l'ordre de n

Comportement asymptotique : comportement d'une fonction $f(n)$ quand n devient grand.
But : regrouper des fonctions ayant le même comportement pour des grandes valeurs de n

Comportement asymptotique

3 notations

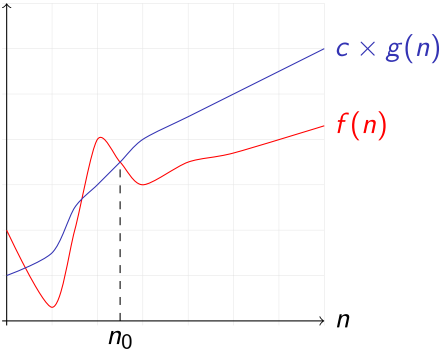
- $\mathcal{O}$: borne supérieure asymptotique d'une fonction
- Ω : borne inférieure asymptotique d'une fonction
- Θ : borne asymptotiquement approchée d'une fonction : à la fois une borne inférieure et supérieure

Ces notations vont être utilisées pour décrire le comportement asymtotique des algorithmes.

La notation $\mathcal{O}$

Definition ($\mathcal{O}$)

Pour une fonction $g(n)$ donnée, on note $\mathcal{O}(g(n))$ l'ensemble de fonctions suivant : $\mathcal{O}(g(n)) = \{f(n)$: il existe des constantes positives c et n_0 telles que $0 \leq f(n) \leq c \times g(n)$ pour tout $n \geq n_0\}$



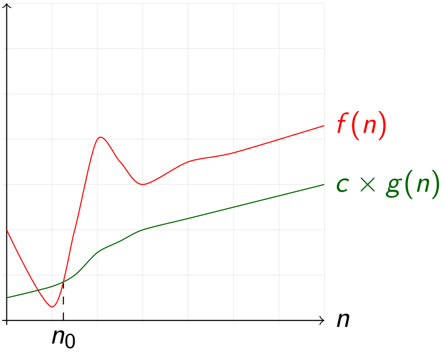
On note $f(n) = \mathcal{O}(g(n))$.
On dit "en grand o de g(n)".

Exemples :
$n = \mathcal{O}(n)$,
$3n + 10^{3700} = \mathcal{O}(n)$,
$3n^3 + 2n + 1 = \mathcal{O}(n^3)$

La notation Ω

Definition (Ω)

Pour une fonction $g(n)$ donnée, on note $\Omega(g(n))$ l'ensemble des fonctions suivant : $\Omega(g(n)) = \{f(n)$: il existe des constantes positives c et n_0 telles que $0 \leq c \times g(n) \leq f(n)$ pour tout $n \geq n_0\}$



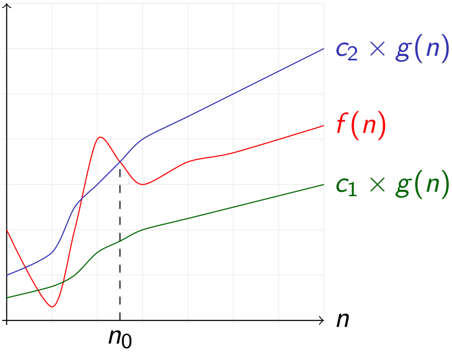
On note $f(n) = \Omega(g(n))$.
On dit "en grand omega de g(n)".

Exemples :
$3n^3 + 2n + 1 = \Omega(n)$

La notation Θ

Definition (Θ)

Pour une fonction donnée $g(n)$, on note $\Theta(g(n))$ l'ensemble des fonctions $\Theta(g(n)) = \{f(n)$, il existe des constantes positives c_1, c_2 et n_0 telles que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ pour tout $n \geq n_0\}$



On note $f(n) = \Theta(g(n))$.
On dit "en theta de g(n)".

Exemples :
$n = \Theta(n)$,
$3n + 10^{3700} = \Theta(n)$,
$3n^3 + 2n + 1 = \Theta(n^3)$

Calcul sur les relations de comparaison

- R0 $g = \mathcal{O}(g)$
- R1 $f = \Theta(g) \Rightarrow g = \Theta(f)$
- R2 $f = \mathcal{O}(g)$ et $g = \mathcal{O}(h) \Rightarrow f = \mathcal{O}(h)$
- R3 $f = \mathcal{O}(g) \Rightarrow \lambda f = \mathcal{O}(g), \lambda \in \mathrm{IR}^{+*}$
- R4 $f_1 = \mathcal{O}(g_1)$ et $f_2 = \mathcal{O}(g_2) \Rightarrow f_1 + f_2 = \mathcal{O}(\max(g_1, g_2))$
- R5 soient f_1 et f_2 telles que $f_1 - f_2 \geq 0$,
  $f_1 = \mathcal{O}(g_1)$ et $f_2 = \mathcal{O}(g_2) \Rightarrow f_1 - f_2 = \mathcal{O}(g_1)$
- R6 soient f_1 et f_2 telles que $f_1 - f_2 \geq 0$,
  $f_1 = \Theta(g_1)$ et $f_2 = \Theta(g_2)$ et si $g_2 = \mathcal{O}(g1)$ et g_1 n'appartient pas à $\mathcal{O}(g_2)$, alors $f_1 - f_2 = \mathcal{O}(g_1)$
- R7 $f_1 = \mathcal{O}(g_1)$ et $f_2 = \mathcal{O}(g_2) \Rightarrow f_1 \times f_2 = \mathcal{O}(g_1 \times g_2)$

Sauf R5, les règles énoncées pour $\mathcal{O}$ sont aussi valables pour Θ.

Rapport avec la complexité des algorithmes

- Les notations $\mathcal{O}$ et Θ servent à décrire le comportement asymptotique des algorithmes.
- Si deux algorithmes ont la même borne asymptotique supérieure, alors pour des grandes données ils mettront le même temps (à une constante multiplicative près) au maximum.
- Si deux algorithmes ont la même borne asymptotique inférieure, alors pour des grandes données ils mettront le même temps (à une constante multiplicative près) au minimum.

Rapport avec la complexité des algorithmes

- Si on dispose d'un algorithme A dont on connaît la fonction de complexité en temps f dans le pire des cas, on dira qu'il s'exécute en $\mathcal{O}(f)$
- Si on dispose d'un algorithme A dont on connaît la fonction de complexité en temps f dans le meilleur des cas, on dira qu'il s'exécute en $\Omega(f)$
- Si on dispose d'un algorithme A dont la fonction de complexité en temps f est la même dans le meilleur des cas et dans le pire des cas, on dira qu'il s'exécute en $\Theta(f)$
- Si on dispose d'un algorithme A, pour lequel on ne sait pas calculer exactement la complexité, mais seulement un encadrement tel que $3n^2 \leq c(n) \leq 4n^2 - 10$, alors on pourra dire que la complexité est en $\Theta(n^2)$.

Retour sur les exemples

- exemple 1, recherche du max
  $c(n) = n - 1$ dans tous les cas, donc $c(n) = \Theta(n)$
  en effet, fixons par exemple $n_0 = 2$ et $c_1 = 0.5$, $c_2 = 2$, alors pour tout $n \geq n_0$ on a :

$$0 \leq 0.5 \times n \leq c(n) \leq 2 \times n$$

- exemple 2, copie d'un tableau
  $e(n) = n$ dans tous les cas, donc $e(n) = \Theta(n)$
- exemple 3, prédicat de présence d'un élément
  $c(n) = 1$ dans le meilleur des cas et $c(n) = n$ dans le pire des cas, on a donc $c(n) = \Omega(1)$
  en effet, fixons par exemple $n_0 = 1$ et $c_1 = 1$, pour tout $n \geq n_0$ on a :

$$0 \leq 1 \times 1 \leq c(n)$$

  et $c(n) = \mathcal{O}(n)$
  en effet, fixons $n'_0 = 1$ et $c'_1 = 2$, pour tout $n \geq n'_0$ on a :

$$0 \leq c(n) \leq 2 \times n$$

Classes d'équivalence

Nom de la classe	Comportement asymptotique
constant	$\Theta(1)$
logarithmique	$\Theta(\log n)$
linéaire	$\Theta(n)$
	$\Theta(n \log n)$
quadratique	$\Theta(n^2)$
polynomiale	$\Theta(n^k)$, $k > 0$ fixé
exponentielle	$\Theta(k^n)$, $k > 0$ fixé

Dans tout le cours :

- complexité = comportement asymptotique
- nombre d'opérations = une fonction
- log = logarithme en base 2