Algorithmique

Notion de complexité

Florent Hivert

Mél : Florent.Hivert@lri.fr
Adresse universelle : http://www.lri.fr/~hivert

Outils mathématiques

Outils mathématiques : analyse élémentaire

$(U_k)_{k\in\mathbb{N}}$ suite de terme général U_k, $k \in \mathbb{N}$

$(U_k)_{k\in K}$ famille d'index $K \subset \mathbb{N}$; suite extraite de $(U_k)_{k\in\mathbb{N}}$

$\displaystyle\sum_{k=p}^{q} U_k$ somme des termes U_k où k vérifie $p \leq k \leq q$ (entiers) ;

Convention utile en informatique lorsque $p > q$:

- la somme est *vide* et vaut 0,

Outils mathématiques : arithmétique

opérateurs usuels :

$$+ \quad - \quad \times \quad / \quad < \quad \leq \quad \text{mod}$$

$\lfloor x \rfloor$ partie entière inférieure (ou *plancher*) du réel x : le plus grand entier $\leq x$

$\lceil x \rceil$ partie entière supérieure (ou *plafond*) du réel x : le plus petit entier $\geq x$

$n!$ la *factorielle* de n :

$$n! := \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n$$

Parties entières et égalités

Pour tout réel x, pour tout entier n :

- $\lfloor x \rfloor = n \iff n \leq x < n+1$;
- $\lceil x \rceil = n \iff n-1 < x \leq n$;
- $\lfloor x+n \rfloor = \lfloor x \rfloor + n$;
- $\lceil x+n \rceil = \lceil x \rceil + n$.

Pour tout entier n :

- $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$.

Parties entières et inégalités

Pour tout réel x, pour tout entier n :

- $\lfloor x \rfloor < n \iff x < n$;
- $\lceil x \rceil \leq n \iff x \leq n$;
- $n < \lceil x \rceil \iff n < x$;
- $n \leq \lfloor x \rfloor \iff n \leq x$.

Pour tous réels x et y :

- $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor \leq \lfloor x \rfloor + \lfloor y \rfloor + 1$;
- $\lceil x \rceil + \lceil y \rceil - 1 \leq \lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil$.

Fonction Exponentielle et Logarithme

$$\exp(a)\exp(b) = \exp(a+b) \qquad \exp(-a) = \frac{1}{\exp(a)}$$

$$\exp(x) = y \iff x = \ln(y) \text{ (pour } y > 0)$$

$$\exp(\ln(y)) = y \qquad \ln(\exp(x)) = x$$

$$\ln(uv) = \ln(u) + ln(v) \qquad \ln\left(\frac{1}{u}\right) = -\ln(u)$$

On en déduit (au moins pour n entier) :

$$a^n = \exp(\ln(a))^n = \exp(n\ln(a))$$

On défini donc, pour $x > 0$ et a quelconque

$$x^a := \exp(x\ln(a))\,.$$

Différents logarithmes

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

$$a^x = y \iff x = \log_a(y)$$

$$2^x = y \iff x = \log_2(y)$$

Évaluation des performances

Temps de calcul

Sur les machines actuelles, le temps pris par un calcul est **très difficile à prévoir**, avec une forte composante aléatoire :

- traduction (interprétation, compilation) code de haut niveau vers code de bas niveau, microcode.
- forte dépendance à l'environement (mémoire, système d'exploitation, multi-threading, hyper-threading...) ;
- nombreuses optimisations qui dépendent de l'historique (caches, prédictions de branches...).

Retenir

On travaille avec un modèle de machine simplifiée.

Complexités

Définitions (complexités temporelle et spatiale)

- ***complexité temporelle*** *: (ou* ***en temps****) : temps de calcul ;*
- ***complexité spatiale*** *: (ou* ***en espace****) : l'espace mémoire requis par le calcul.*

Définitions (complexités pratique et théorique)

- *La* ***complexité pratique*** *est une mesure précise des complexités temporelles et spatiales pour un* ***modèle de machine donné****.*
- *La* ***complexité (théorique)*** *est un* ***ordre de grandeur*** *de ces couts, exprimé de manière la plus* ***indépendante*** *possible des conditions pratiques d'exécution.*

Un exemple

Problème (plus grand diviseur)

Décrire une méthode de calcul du plus grand diviseur autre que lui-même d'un entier $n \geq 2$.

Notons $pgd(n)$ le plus grand diviseur en question. On a :

- $1 \leq pgd(n) \leq n - 1$;
- $pgd(n) = 1 \iff n$ est premier.

Un exemple

Problème (plus grand diviseur)

Décrire une méthode de calcul du plus grand diviseur autre que lui-même d'un entier $n \geq 2$.

Notons $pgd(n)$ le plus grand diviseur en question. On a :

- $1 \leq pgd(n) \leq n - 1$;
- $pgd(n) = 1 \iff n$ est premier.

Algorithme (0)

On parcours les nombres de 2 à $n-1$ et l'on note le dernier diviseur que l'on a trouvé :

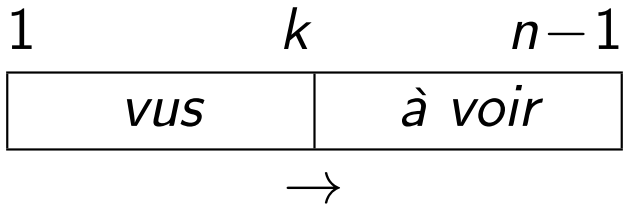
Algorithme (calcul du plus grand diviseur (solution 0))

- *Entrée : un entier n*
- *Sortie : pgd(n)*

```
res ← 1
Pour k de 2 à n − 1 :
    si k divise n alors res ← k
retourner res
```

Algorithme (0)

On parcours les nombres de 2 à $n-1$ et l'on note le dernier diviseur que l'on a trouvé :



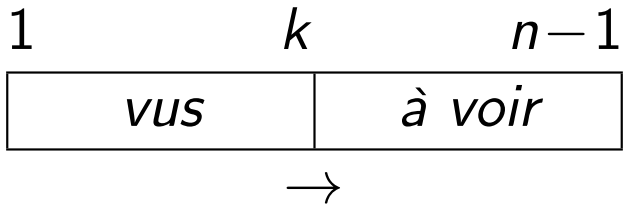
Algorithme (calcul du plus grand diviseur (solution 0))

- *Entrée : un entier n*
- *Sortie : pgd(n)*

```
res ← 1
Pour k de 2 à n − 1 :
    si k divise n alors res ← k
retourner res
```

Algorithme (0)

On parcours les nombres de 2 à $n-1$ et l'on note le dernier diviseur que l'on a trouvé :



Algorithme (calcul du plus grand diviseur (solution 0))

- *Entrée : un entier n*
- *Sortie : pgd(n)*

res $\leftarrow 1$
Pour k de 2 à $n-1$ *:*
    si k divise n alors res $\leftarrow k$
retourner res

Algorithme (1)

Puisqu'il s'agit de trouver le plus grand diviseur, on peut procéder en décroissant sur les diviseurs possibles :

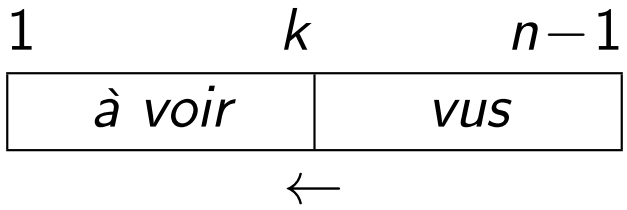
Algorithme (calcul du plus grand diviseur (solution 1))

- *Entrée : un entier n*
- *Sortie : $pgd(n)$*

$k \leftarrow n-1$
tant que $n \bmod k \neq 0 : k \leftarrow k-1$
retourner k

Algorithme (1)

Puisqu'il s'agit de trouver le plus grand diviseur, on peut procéder en décroissant sur les diviseurs possibles :



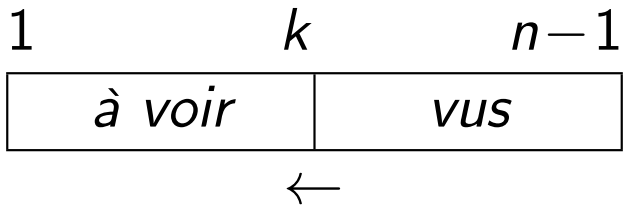
Algorithme (calcul du plus grand diviseur (solution 1))

- *Entrée : un entier n*
- *Sortie : $pgd(n)$*

$k \leftarrow n-1$
tant que $n \bmod k \neq 0 : k \leftarrow k-1$
retourner k

Algorithme (1)

Puisqu'il s'agit de trouver le plus grand diviseur, on peut procéder en décroissant sur les diviseurs possibles :



Algorithme (calcul du plus grand diviseur (solution 1))

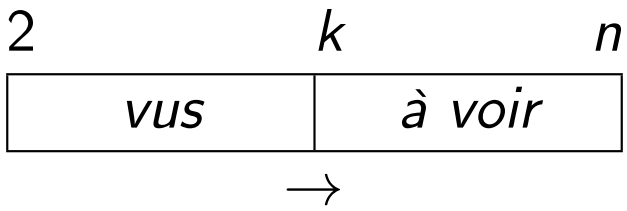
- *Entrée : un entier n*
- *Sortie : pgd(n)*

$k \leftarrow n-1$
tant que $n \bmod k \neq 0$ *:* $k \leftarrow k-1$
retourner k

Algorithme (2)

Remarque : le résultat cherché est $n \div p$, où p est le *plus petit* diviseur supérieur ou égal à 2 de n.

Notons $ppd(n)$ le plus petit diviseur en question.



Algorithme (calcul du plus grand diviseur (solution 2))

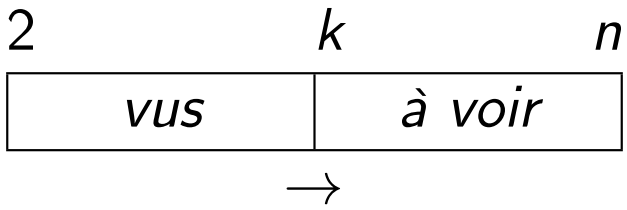
- *Entrée : un entier n*
- *Sortie : pgd(n)*

$k \leftarrow 2$
tant que $n \bmod k \neq 0 : k \leftarrow k + 1$
retourner n/k

Algorithme (2)

Remarque : le résultat cherché est $n \div p$, où p est le *plus petit* diviseur supérieur ou égal à 2 de n.

Notons $ppd(n)$ le plus petit diviseur en question.



Algorithme (calcul du plus grand diviseur (solution 2))

- *Entrée : un entier n*
- *Sortie : pgd(n)*

  $k \leftarrow 2$
  tant que $n \bmod k \neq 0$ *:* $k \leftarrow k + 1$
  retourner n/k

Algorithme (3)

On peut maintenant tenir compte de ce que :

$$n \text{ non premier } \Longrightarrow 2 \leq ppd(n) \leq pgd(n) \leq n-1.$$

D'où il vient que :

$$n \text{ non premier } \Longrightarrow (ppd(n))^2 \leq n.$$

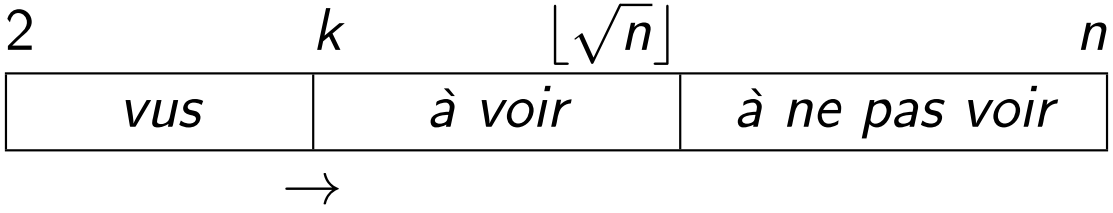
Proposition

Si n ne possède pas de diviseur compris entre 2 et $\lfloor\sqrt{n}\rfloor$, c'est qu'il est premier ;

Cece permet d'*améliorer* le temps de calcul pour les nombres premiers : il est donc inutile de chercher en croissant entre $\lfloor\sqrt{n}\rfloor + 1$ et n.

Algorithme (3)

En procédant en croissant sur les diviseurs possibles :



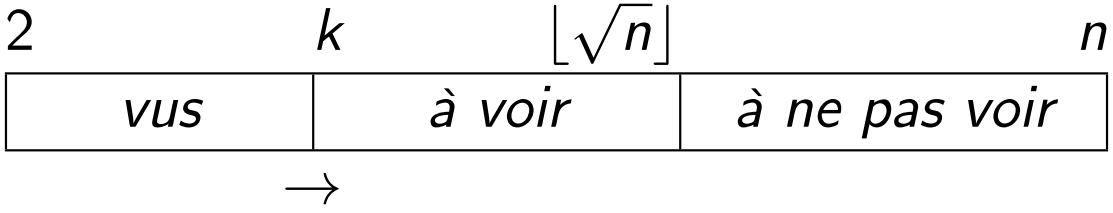
Algorithme (calcul du plus grand diviseur (solution 3))

- *Entrée : un entier n*
- *Sortie : $pgd(n)$*

$k \leftarrow 2$
tant que $n \bmod k \neq 0$ *et* $k \leq n/k : k \leftarrow k+1$
si $k > n/k$ *retourner* 1 *sinon retourner* n/k

Algorithme (3)

En procédant en croissant sur les diviseurs possibles :



Algorithme (calcul du plus grand diviseur (solution 3))

- *Entrée : un entier n*
- *Sortie : pgd(n)*

  $k \leftarrow 2$
  tant que $n \bmod k \neq 0$ *et* $k \leq n/k$: $k \leftarrow k+1$
  si $k > n/k$ *retourner* 1 *sinon retourner* n/k

Analyse des algorithmes

Calcul des complexités temporelles pratiques des solutions (0), (1), (2) et (3) :

Les algorithmes ont la même structures :

A_1
tant que C : A_2
A_3

Hypothèse : boucle compilée par un branchement conditionnel et un branchement inconditionnel.

Analyse des algorithmes

Calcul des complexités temporelles pratiques des solutions (0), (1), (2) et (3) :

Les algorithmes ont la même structures :

A_1
tant que C : A_2
A_3

Hypothèse : boucle compilée par un branchement conditionnel et un branchement inconditionnel.

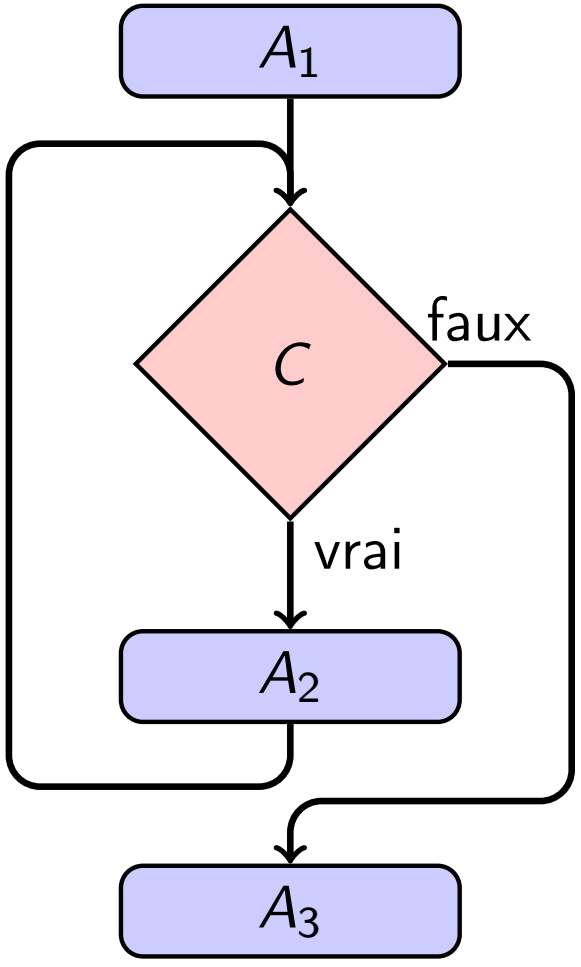
Analyse des algorithmes

Calcul des complexités temporelles pratiques des solutions (0), (1), (2) et (3) :

Les algorithmes ont la même structures :

A_1
tant que C : A_2
A_3

Hypothèse : boucle compilée par un branchement conditionnel et un branchement inconditionnel.



Analyse des trois algorithmes

Calcul des complexités temporelles pratiques des différentes solutions :

Les quatres algorithmes ont la même structures (for = tant que) :

A_1
tant que C : A_2
A_3

Pour un algorithme donné, soient t_1, t_C, t_2 et t_3 les temps d'exécution respectifs des actions A_1, C, A_2 et A_3.

Temps branchement conditionnel et inconditionnel : t_{BC} et t_{BI}.

Le temps d'exécution est :

$$t_1 + (t_C + t_{\mathrm{BC}} + t_2 + t_{\mathrm{BI}})B(n) + t_C + t_{\mathrm{BC}} + t_3,$$

où $B(n)$ est le nombre de boucles exécutées.

Analyse des trois algorithmes

Calcul des complexités temporelles pratiques des différentes solutions :

Les quatres algorithmes ont la même structures (for = tant que) :

A_1
tant que C : A_2
A_3

Pour un algorithme donné, soient t_1, t_C, t_2 et t_3 les temps d'exécution respectifs des actions A_1, C, A_2 et A_3.

Temps branchement conditionnel et inconditionnel : t_{BC} et t_{BI}.

Le temps d'exécution est :

$$t_1 + (t_C + t_{\mathrm{BC}} + t_2 + t_{\mathrm{BI}})B(n) + t_C + t_{\mathrm{BC}} + t_3,$$

où $B(n)$ est le nombre de boucles exécutées.

Analyse des trois algorithmes

Calcul des complexités temporelles pratiques des différentes solutions :

Les quatres algorithmes ont la même structures (for = tant que) :

A_1
tant que C : A_2
A_3

Pour un algorithme donné, soient t_1, t_C, t_2 et t_3 les temps d'exécution respectifs des actions A_1, C, A_2 et A_3.

Temps branchement conditionnel et inconditionnel : t_{BC} et t_{BI}.

Le temps d'exécution est :

$$t_1 + (t_C + t_{\mathrm{BC}} + t_2 + t_{\mathrm{BI}})B(n) + t_C + t_{\mathrm{BC}} + t_3,$$

où $B(n)$ est le nombre de boucles exécutées.

Analyse des trois algorithmes

Calcul des complexités temporelles pratiques des différentes solutions :

Les quatres algorithmes ont la même structures (for = tant que) :

A_1
tant que C : A_2
A_3

Pour un algorithme donné, soient t_1, t_C, t_2 et t_3 les temps d'exécution respectifs des actions A_1, C, A_2 et A_3.

Temps branchement conditionnel et inconditionnel : t_{BC} et t_{BI}.

Le temps d'exécution est :

$$t_1 + (t_C + t_{\mathrm{BC}} + t_2 + t_{\mathrm{BI}})B(n) + t_C + t_{\mathrm{BC}} + t_3,$$

où $B(n)$ est le nombre de boucles exécutées.

Analyse des algorithmes

Retenir

Sur une machine où les opérations sur les entiers s'effectuent en temps constant, le temps d'exécution est donc de la forme :

$$a\,B(n) + b$$

où a et b sont des constantes.

Borne maximale :

Pour les solution (0), (1) et (2)	$B(n) \leq n - 2$
Pour la solution (3)	$B(n) \leq \lfloor \sqrt{n} \rfloor - 1$

Complexité temporelle maximale :

Pour les solution (0), (1) et (2)	$a'n + b'$
Pour la solution (3)	$a'\lfloor \sqrt{n} \rfloor + b'$

Analyse des algorithmes

Retenir

Sur une machine où les opérations sur les entiers s'effectuent en temps constant, le temps d'exécution est donc de la forme :

$$a\,B(n) + b$$

où a et b sont des constantes.

Borne maximale :

Pour les solution (0), (1) et (2)	$B(n) \leq n - 2$
Pour la solution (3)	$B(n) \leq \lfloor \sqrt{n} \rfloor - 1$

Complexité temporelle maximale :

Pour les solution (0), (1) et (2)	$a'n + b'$
Pour la solution (3)	$a'\lfloor \sqrt{n} \rfloor + b'$

En pratique

Voici les temps d'exécution mesurés pour quelques nombres à la fois premiers et proches de puissances de 10 :

n	solution 0	solution 1	solution 2	solution 3
11	$1.26 \cdot 10^{-7}$	$1.12 \cdot 10^{-7}$	$1.01 \cdot 10^{-7}$	$8.93 \cdot 10^{-8}$
1 009	$2.14 \cdot 10^{-6}$	$2.26 \cdot 10^{-6}$	$2.41 \cdot 10^{-6}$	$1.55 \cdot 10^{-7}$
100 003	$1.86 \cdot 10^{-4}$	$1.96 \cdot 10^{-4}$	$2.15 \cdot 10^{-4}$	$1.05 \cdot 10^{-6}$
10 000 019	$1.88 \cdot 10^{-2}$	$1.95 \cdot 10^{-2}$	$2.15 \cdot 10^{-2}$	$9.26 \cdot 10^{-6}$
1 000 000 007	$1.85 \cdot 10^{0}$	$1.92 \cdot 10^{0}$	$2.08 \cdot 10^{0}$	$8.88 \cdot 10^{-5}$
$\approx \times 100$	$\approx \times 100$	$\approx \times 100$	$\approx \times 100$	$\approx \times 10$
Théorique	$an + b$	$an + b$	$an + b$	$a\sqrt{n} + b$

Bilan

Retenir

- *Il est très difficile de prévoir le temps de calcul d'un programme.*
- *En revanche, on peut très bien prévoir comment ce temps de calcul augmente quand la donnée augmente.*

Notion de complexité

Problématique :

On cherche à définir une notion de complexité **robuste**, indépendante

- de l'ordinateur,
- du langage de programmation,
- du compilateur ou de l'interpréteur,
- *etc.*

Exprimée en fonction de la **Taille** de la donnée à traiter.

Problématique :

On cherche à définir une notion de complexité **robuste**, indépendante

- de l'ordinateur,
- du langage de programmation,
- du compilateur ou de l'interpréteur,
- *etc.*

Exprimée en fonction de la **Taille** de la donnée à traiter.

Problématique :

On cherche à définir une notion de complexité **robuste**, indépendante

- de l'ordinateur,
- du langage de programmation,
- du compilateur ou de l'interpréteur,
- *etc.*

Exprimée en fonction de la **Taille** de la donnée à traiter.

Opérations élémentaires

Pour ceci on utilisera un modèle de machine *simplifié*.

Retenir (opération élémentaire)

Opération qui prend un temps constant (ou presque).

Les opérations suivantes sont généralement considérée comme élémentaires :

- Opérations arithmétiques (additions, multiplications, comparaisons) ;
- Accès aux données en mémoire ;
- Sauts conditionnels et inconditionnels ;

Note : tout ceci est très discutable en réalité.

Opérations élémentaires

Pour ceci on utilisera un modèle de machine *simplifié*.

Retenir (opération élémentaire)

Opération qui prend un temps constant (ou presque).

Les opérations suivantes sont généralement considérée comme élémentaires :

- Opérations arithmétiques (additions, multiplications, comparaisons) ;
- Accès aux données en mémoire ;
- Sauts conditionnels et inconditionnels ;

Note : tout ceci est très discutable en réalité.

Opérations élémentaires

Pour ceci on utilisera un modèle de machine *simplifié*.

Retenir (opération élémentaire)

Opération qui prend un temps constant (ou presque).

Les opérations suivantes sont généralement considérée comme élémentaires :

- Opérations arithmétiques (additions, multiplications, comparaisons) ;
- Accès aux données en mémoire ;
- Sauts conditionnels et inconditionnels ;

Note : tout ceci est très discutable en réalité.

Opérations élémentaires

Pour ceci on utilisera un modèle de machine *simplifié*.

Retenir (opération élémentaire)

Opération qui prend un temps constant (ou presque).

Les opérations suivantes sont généralement considérée comme élémentaires :

- Opérations arithmétiques (additions, multiplications, comparaisons) ;
- Accès aux données en mémoire ;
- Sauts conditionnels et inconditionnels ;

Note : tout ceci est très discutable en réalité.

Complexité d'un algorithme

Cout de $\mathcal{A}$ sur x : l'exécution de l'algorithme $\mathcal{A}$ sur la donné x requiert $C_{\mathcal{A}}(x)$ opérations élémentaires.

Définitions (Cas le pire, cas moyen)

n désigne la taille de la donnée à traité.

- *Dans le pire des cas :* $C_{\mathcal{A}}(n) := \max_{x \ |x|=n} C_{\mathcal{A}}(x)$
- *En moyenne :* $C_{\mathcal{A}}^{Moy}(n) := \sum_{x \ |x|=n} p_n(x) C_{\mathcal{A}}(x)$

  p_n : distribution de probabilité sur les données de taille n.

Idée

Retenir

- *On ne va pas comparer les nombres d'opérations.*
- *On va comparer aux constantes près la vitesse de croissance des fonctions qui comptent les nombres d'opérations.*

Notations asymptotiques

Les constantes n'importent pas !

Définition (notations asymptotiques)

Soit $g : \mathbb{N} \mapsto \mathbb{R}^{>0}$ *une fonction positive.*

- $O(g)$ *est l'ensemble des fonctions positives* f *pour lesquelles il existe une constante strictement positive* α *et un entier* n_0 *tels que :*

$$f(n) \leq \alpha g(n), \quad \text{pour tout } n \geq n_0.$$

- $\Omega(g)$ *est l'ensemble des fonctions positives* f *pour lesquelles il existe une constante strictement positive* α *et un entier* n_0 *tels que :*

$$f(n) \geq \alpha g(n), \quad \text{pour tout } n \geq n_0.$$

- $\Theta(g) = O(g) \cap \Omega(g)$.

Par commodité, les expressions « image » des fonctions sont souvent utilisées dans les notations plutôt que leurs symboles. On écrit ainsi « $f(n) \in X(g(n))$ » plutôt que « $f \in X(g)$ », où X signifie O, Ω ou Θ.

Par commodité toujours, on écrit souvent « est » plutôt que « $\in$ » et on dit souvent « est » plutôt que « appartient ».

Exemple

notations asymptotiques (1/2)

$n \in O(n)$	$n \in \Omega(n)$	$n \in \Theta(n)$
$7 + 1/n \in O(1)$	$7 + 1/n \in \Omega(1)$	$7 + 1/n \in \Theta(1)$
$\log_2 n \in O(\log n)$	$\log_2 n \in \Omega(\log n)$	$\log_2 n \in \Theta(\log n)$
$n + \ln n \in O(n)$	$n + \ln n \in \Omega(n)$	$n + \ln n \in \Theta(n)$
$n^2 + 3n \in O(n^3)$	$n^2 + 3n \notin \Omega(n^3)$	$n^2 + 3n \notin \Theta(n^3)$

Par commodité, les expressions « image » des fonctions sont souvent utilisées dans les notations plutôt que leurs symboles. On écrit ainsi « $f(n) \in X(g(n))$ » plutôt que « $f \in X(g)$ », où X signifie O, Ω ou Θ.

Par commodité toujours, on écrit souvent « est » plutôt que « $\in$ » et on dit souvent « est » plutôt que « appartient ».

Exemple

notations asymptotiques (1/2)

$n \in O(n)$	$n \in \Omega(n)$	$n \in \Theta(n)$
$7 + 1/n \in O(1)$	$7 + 1/n \in \Omega(1)$	$7 + 1/n \in \Theta(1)$
$\log_2 n \in O(\log n)$	$\log_2 n \in \Omega(\log n)$	$\log_2 n \in \Theta(\log n)$
$n + \ln n \in O(n)$	$n + \ln n \in \Omega(n)$	$n + \ln n \in \Theta(n)$
$n^2 + 3n \in O(n^3)$	$n^2 + 3n \notin \Omega(n^3)$	$n^2 + 3n \notin \Theta(n^3)$

Fonctions de référence

constante, somme, produit, puissance, logarithme, minimum, maximum...

Définitions (désignations des complexités courantes)

notation	*désignation*	*notation*	*désignation*
$\Theta(1)$	*constante*	$\Theta(n^2)$	*quadratique*
$\Theta(\log n)$	*logarithmique*	$\Theta(n^3)$	*cubique*
$\Theta(\sqrt{n})$	*racinaire*	$\Theta(n^k),\ k \in \mathbb{N}, k \geq 2$	*polynomiale*
$\Theta(n)$	*linéaire*	$\Theta(a^n),\ a > 1$	*exponentielle*
$\Theta(n \log n)$	*quasi-linéaire*	$\Theta(n!)$	*factorielle*

Exemple

Suite aux résultats précédents, on peut énoncer que le problème du calcul du plus grand diviseur peut se résoudre en temps au plus racinaire avec un espace constant.

Fonctions de référence

constante, somme, produit, puissance, logarithme, minimum, maximum...

Définitions (désignations des complexités courantes)

notation	*désignation*	*notation*	*désignation*
$\Theta(1)$	*constante*	$\Theta(n^2)$	*quadratique*
$\Theta(\log n)$	*logarithmique*	$\Theta(n^3)$	*cubique*
$\Theta(\sqrt{n})$	*racinaire*	$\Theta(n^k)$, $k \in \mathbb{N}, k \geq 2$	*polynomiale*
$\Theta(n)$	*linéaire*	$\Theta(a^n)$, $a > 1$	*exponentielle*
$\Theta(n \log n)$	*quasi-linéaire*	$\Theta(n!)$	*factorielle*

Exemple

Suite aux résultats précédents, on peut énoncer que le problème du calcul du plus grand diviseur peut se résoudre en temps au plus racinaire avec un espace constant.

Fonctions de référence

constante, somme, produit, puissance, logarithme, minimum, maximum...

Définitions (désignations des complexités courantes)

notation	*désignation*	*notation*	*désignation*
$\Theta(1)$	*constante*	$\Theta(n^2)$	*quadratique*
$\Theta(\log n)$	*logarithmique*	$\Theta(n^3)$	*cubique*
$\Theta(\sqrt{n})$	*racinaire*	$\Theta(n^k)$, $k \in \mathbb{N}, k \geq 2$	*polynomiale*
$\Theta(n)$	*linéaire*	$\Theta(a^n)$, $a > 1$	*exponentielle*
$\Theta(n \log n)$	*quasi-linéaire*	$\Theta(n!)$	*factorielle*

Exemple

Suite aux résultats précédents, on peut énoncer que le problème du calcul du plus grand diviseur peut se résoudre en temps au plus racinaire avec un espace constant.

Aucun progrès technologique (modèle de machine standard) ne permet à un algorithme de changer de classe de complexité.

Exemple (tyranie de la complexité)

Effets de la multiplication de la puissance d'une machine par 10, 100 et 1 000 sur la taille maximale N des problèmes que peuvent traiter des algorithmes de complexité donnée :

Aucun progrès technologique (modèle de machine standard) ne permet à un algorithme de changer de classe de complexité.

Exemple (tyranie de la complexité)

Effets de la multiplication de la puissance d'une machine par 10, 100 et 1 000 sur la taille maximale N des problèmes que peuvent traiter des algorithmes de complexité donnée :

complexité	$\times 10$	$\times 100$	$\times 1\,000$
$\Theta(\log n)$	N^{10}	N^{100}	$N^{1\,000}$
$\Theta(\sqrt{n})$	$10^2\,N$	$10^4\,N$	$10^6\,N$
$\Theta(n)$	$10\,N$	$100\,N$	$1\,000\,N$
$\Theta(n \log n)$	$< 10\,N$	$< 100\,N$	$< 1\,000\,N$
$\Theta(n^2)$	$\simeq 3\,N$	$10\,N$	$\simeq 32\,N$
$\Theta(n^3)$	$\simeq 2\,N$	$\simeq 5\,N$	$10\,N$
$\Theta(2^n)$	$\simeq N+3$	$\simeq N+7$	$\simeq N+10$

Une erreur très courante

Retenir (Attention ! ! !)

*La complexité d'un algorithme ne parle pas du **temps de calcul absolu** des implémentations de cet algorithme mais de **la vitesse avec laquelle ce temps de calcul augmente** quand la taille des entrées augmente.*

On a effacé les constantes : $O(n) = O(2n)$. La complexité asymptotique ne nous permet pas de comparer deux algorithmes différents de même complexité. Une telle comparaison ne serait d'ailleurs pas forcément utile, à cause des différences entre les ordinateurs.

Une erreur très courante

Retenir (Attention ! ! !)

La complexité d'un algorithme ne parle pas du ***temps de calcul absolu*** *des implémentations de cet algorithme mais de* ***la vitesse avec laquelle ce temps de calcul augmente*** *quand la taille des entrées augmente.*

On a effacé les constantes : $O(n) = O(2n)$. La complexité asymptotique ne nous permet pas de comparer deux algorithmes différents de même complexité. Une telle comparaison ne serait d'ailleurs pas forcément utile, à cause des différences entre les ordinateurs.

Propriétés des notations asymptotiques

Proposition

Soient f, g, h, l des fonctions positives et $a, b > 0$.
X désigne n'importe lequel des opérateur O, Ω ou Θ

- *Si $f \in X(g)$ et $g \in X(h)$ alors $f \in X(h)$;*
- *Si $f, g \in X(h)$ alors $af + bg \in X(h)$;*
- *Si $f \in X(h)$ et $g \in X(l)$ alors $fg \in X(hl)$;*
- *Si $f \in \Omega(h)$ alors pour tout g on a $af + bg \in \Omega(h)$;*

Cas des polynômes

Proposition

Un polynôme est de l'ordre de son degré. Plus précisément si

$$P = \sum_{i=0}^{d} c_i x^i$$

avec $c_d \neq 0$ *(c'est-à-dire que* d *est le degré de* P*) alors*

$$P \in \Theta(x^d)$$

.

Par exemple, $5x^3 + 3x^2 + 100x + 12 \in \Theta(x^3)$.

Récapitulatif

Complexité	Vitesse	Temps	Formulation	Exemple
Factorielle	très lent	proportionnel à N^N	$N!$	Résolution par recherche exhaustive du problème du voyageur de commerce.
Exponentielle	lent	proportionnel à une constante à la puissance N	K^N	Résolution par recherche exhaustive du Rubik's Cube.
Polynomiale	moyen	proportionnel à N à une puissance donnée	N^K	Tris par comparaison, comme le tri à bulle (N^2).
Quasi-linéaire	assez rapide	intermédiaire entre linéaire et polynomial	$N\log(N)$	Tris quasi-linéaires, comme le Quicksort.
Linéaire	rapide	proportionnel à N	N	Itération sur un tableau.
Logarithmique	très rapide	proportionnel au logarithme de N	$\log(N)$	Recherche dans un arbre binaire.
Constante	le plus rapide	indépendant de la donnée	1	recherche par index dans un tableau.

Calcul de complexité dans les structures de contrôle

- Les instructions élémentaires (arithmétique affectations, comparaisons) sont en temps constant, soit en $\Theta(1)$.

- Tests : si $a \in O(A)$, $b \in O(B)$ et $c \in O(C)$ alors

$$(\texttt{if } a \texttt{ then } b \texttt{ else } c) \in O(A + \max(B, C))$$

- Tests : si $a \in \Omega(A)$, $b \in \Omega(B)$ et $c \in \Omega(C)$ alors

$$(\texttt{if } a \texttt{ then } b \texttt{ else } c) \in \Omega(A + \min(B, C))$$

Cas des boucles imbriquées

- Boucles si $a_i \in O(A_i)$ (idem Ω, Θ) alors

$$(\texttt{for } i \texttt{ from } 1 \texttt{ to } n \texttt{ do } a_i) \in O\left(\sum_{i=1}^{n}(A_i)\right)$$

- Lorsque A_i est constant égal à A, on a

$$(\texttt{for } i \texttt{ from } 1 \texttt{ to } n \texttt{ do } a_i) \in nO(A)$$

Retenir (Boucles imbriqués)

Cas particulier important : si $A_i \in O(i^k)$ (idem Ω, Θ) alors

$$(\textit{for } i \textit{ from } 1 \textit{ to } n \textit{ do } a_i) \in O(n^{k+1})$$