Introduction à la complexité algorithmique

par

Éric TRICHET

Intégrateur TICE au

Centre de Ressources et d’Innovation Pédagogiques
de l’Université de Limoges

Remerciements

Je tiens à remercier messieurs Abdelkader NECER et Stéphane VINATIER pour leurs conseils et m'avoir donné l'occasion de publier ce travail.

Merci aussi leur collègue pour sa relecture et ses remarques.

Licence

Ce document est sous la licence Creative Commons.

This document is licensed under the Attribution-NonCommercial-ShareAlike 2.0 France license, available at http://creativecommons.org/licenses/by-nc-sa/2.0/fr/.

Abréviation(s)

RAM : Random Access Machine – machine à accès aléatoire

Prologue

Ce document est à la base un mémoire présenté en vue d'obtenir l'UE « Information et communication pour ingénieur » Spécialité : INFORMATIQUE du Conservatoire National des Arts et Métiers de Limoges.

Il a été soutenu le 19/06/2009 devant un jury composé de JURY Mme Élisabeth METAIS (présidente), M. Jean-Michel DURCE et M. Philippe JEULIN.

Je tiens à remercier M. Philippe JEULIN pour les conseils qu'il m'a prodigués pour la création de ce mémoire.

Préface

Glossaire des termes techniques

algorithme	procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie.
algorithme praticable, efficace	à l'inverse d'un algorithme naïf (complexité exponentielle) et par convention, un algorithme est dit praticable, efficace s'il est polynomial.
axiome	désigne une vérité indémontrable qui doit être admise.
classe L	L est l'ensemble des problèmes pour lesquels il existe un algorithme de résolution en temps logarithmique en fonction de la taille des entrées.
classe NP	la classe NP (problèmes **N**on-déterministes **P**olynomiaux) est l'ensemble des problèmes qui peuvent être résolus sur une machine non déterministe en temps polynomial en fonction de la taille des entrées.
classe P	P est l'ensemble des problèmes pour lesquels il existe un algorithme de résolution en temps polynomial en fonction de la taille des entrées.
complexité d'un algorithme	la complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée.
complexité d'un problème	la complexité d'un problème A est la complexité du meilleur algorithme qui résout A .
complexité dans le meilleur des cas	soit A un algorithme, n un entier, D_n l'ensemble des entrées de taille n et une entrée $d \in D_n$. Posons : $coût_A(d)$ le nombre d'opérations fondamentales effectuées par A avec l'entrée d . La complexité dans le meilleur des cas est obtenue par : $Min_A(n) = min\{coût_A(d) / d \in D_n\}$.
complexité dans le pire des cas	soit A un algorithme, n un entier, D_n l'ensemble des entrées de taille n et une entrée $d \in D_n$. Posons : $coût_A(d)$ le nombre d'opérations fondamentales effectuées par A avec l'entrée d . La complexité dans le pire des cas est donnée par : $Max_A(n) = max\{coût_A(d) / d \in D_n\}$.

…/…

complexité en moyenne	soit A un algorithme, n un entier, D_n l'ensemble des entrées de taille n et une entrée $d \in D_n$. Posons : $coût_A(d)$ le nombre d'opérations fondamentales effectuées par A avec l'entrée d. La complexité en moyenne est donnée par : $$Moy_A(n) = \sum_{d \in D_n} p(d).coût_A(d)$$ avec $p(d)$ une loi de probabilité sur les entrées.
corps de la boucle	ensemble des instructions comprises à l'intérieur de la boucle (par exemple entre le **Faire** et **Jusqu'à**).
dichotomie	la dichotomie (« couper en deux » en grec) est, en algorithmique, un processus itératif ou récursif de recherche où, à chaque étape, l'espace de recherche est restreint à l'une des deux parties.
efficacité d'un algorithme	on considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur.
file	structure de données basée sur le principe « Premier arrivé, premier servi ! » en anglais FIFO (First In, First Out)
fonctions équivalentes	Soient f et g deux fonctions définies sur une partie des nombres réels. f et g sont équivalentes si $f(x) = g(x)(1 + \varepsilon(x))$ avec $\lim_{x \to +\infty} \varepsilon(x) = 0$. C'est noté : $f \sim g$
instance	un exemple spécifique du problème : une configuration particulière des villes pour le problème du voyageur de commerce, un arbre particulier pour un problème de graphe. C'est une entrée particulière pour un algorithme.
limite	soit f une fonction. Nous disons que la limite de $f(x)$ quand x tend vers $+\infty$ est égale à $L \in \mathbb{R}$ si $\forall \varepsilon > 0, \exists S \in \mathbb{R}$ tels que $\forall x > S, \lvert f(x) - L \rvert < \varepsilon$ On note : $\lim_{x \to +\infty} f(x) = L$
machine non-déterministe	une machine non-déterministe est une variante purement théorique : on ne peut pas construire de telle machine. À chaque étape de son calcul, cette machine peut effectuer un choix non-déterministe : elle a le choix entre plusieurs actions, et elle en effectue une. Si l'un des choix l'amène à accepter l'entrée, on considère qu'elle a fait ce choix-là. En quelque sorte, elle devine toujours juste. Une autre manière de voir leur fonctionnement est de considérer qu'à chaque choix non-déterministe, elles se dédoublent, les clones poursuivent le calcul en parallèle suivant les branches du choix. Si l'un des clones accepte l'entrée, on dit que la machine accepte l'entrée.

…/…

machine déterministe	les machines déterministes font toujours un seul calcul à la fois. Ce calcul est constitué d'étapes élémentaires ; à chacune de ces étapes, pour un état donné de la mémoire de la machine, l'action élémentaire effectuée sera toujours la même.
o (petit o)	soit g une fonction définie de $\mathbb{R}^+$ dans $\mathbb{R}^+$. L'ensemble $o(g)$ est défini ainsi : $$o(g)=\{f \text{ fonction défine de } \mathbb{R}^+ \text{ dans } \mathbb{R}^+ / \forall c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, 0 \leqslant f(x) < c\,g(x)\}$$
O (grand O)	soit g une fonction définie sur une partie des nombres réels. L'ensemble $O(g)$ est défini ainsi : $$O(g)=\{f \text{ fonction définie sur une partie de } \mathbb{R} / \exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, \lvert f(x)\rvert \leqslant c\lvert g(x)\rvert\}$$
pile	structure de données fondée sur le principe « dernier arrivé, premier sorti » (ou LIFO pour Last In, First Out).
problème NP - complet	un problème est NP - complet s'il appartient à NP et qu'il est NP - difficile.
problème NP - difficile	un problème Q est dit C - difficile si ce problème est au moins aussi difficile que tous les problèmes dans C c.-à-d. que tout problème de C peut être réduit à Q par un algorithme polynomial.
profilage	le profiling d'un programme permet d'identifier les endroits où celui-ci passe le plus de temps, mais également quelles sont les fonctions qui sont exécutées et combien de fois. Un profiler est un programme capable d'analyser un exécutable. Le résultat de l'analyse est appelé un profile (ou profilage).
réduction de problèmes	la réduction d'un problème P_1 vers un problème P_2 est une transformation de toute instance de P_1 en une instance de P_2, de sorte que toute solution à P_1 puisse induire une solution à P_2. Cette transformation doit être polynomiale en temps et en mémoire selon la taille de P_1. Ainsi il y a une relation de difficulté de résolution entre ces deux problèmes.
Θ	soit g une fonction définie sur une partie des nombres réels. L'ensemble $\Theta(g)$ est défini ainsi : $$\Theta(g)=\{f \text{ fonction définie sur une partie de } \mathbb{R} / \exists c_1>0 \text{ et } c_2>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$$
ω	soit g une fonction définie sur une partie des nombres réels. L'ensemble $\omega(g)$ est défini ainsi : $$\omega(g)=\{f \text{ fonction définie sur une partie de } \mathbb{R} / \forall c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c\,g(n) < f(n)\}$$
Ω	soit g une fonction définie sur une partie des nombres réels. L'ensemble $\Omega(g)$ est défini ainsi : $$\Omega(g)=\{f \text{ fonction défine sur une partie de } \mathbb{R} / \exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c\,g(n) \leqslant f(n)\}$$

Sommaire

Glossaire des termes techniques....4
I – Introduction....3
II – Complexité....4
II.1 – Définitions....4
II.2 – La taille des données....5
II.3 – Le temps de calcul....5
II.4 – Opération fondamentale....6
II.5 – Coût des opérations....6
II.5.1 – Coût de base....6
II.5.2 – Coût en séquentiel....7
II.5.3 – Coût en récursif....7
II.6 – Les différentes mesures de complexité....8
II.6.1 – La complexité dans le meilleur des cas....8
II.6.2 – La complexité en moyenne....8
II.6.3 – La complexité dans le pire des cas....8
II.6.4 – Calcul des complexités des exemples....9
a) Multiplication de deux matrices....9
b) Factoriel....9
c) Recherche dichotomique....10
d) Tri à bulle....10
II.7 – Comparaison de deux algorithmes....12
II.7.1 – Méthode de comparaison....12
II.7.2 – Classification des algorithmes....12
II.8 – Complexité d’un problème....14
II.9 – Comportement des fonctions usuelles....15
III – Classes de problèmes....16
IV – Utilisation de la complexité algorithmique dans un programme....19
IV.1 – Optimisation, pas si sûr....19
IV.2 – Et si nous optimisions ?....20
IV.2.1 – Mesure de vitesse (profiling)....20
IV.2.2 – Optimisation à grande échelle....20
IV.2.3 – Optimisation à petite échelle....20
V – Notations mathématiques....21
V.1 – Rappel sur les limites en +∞....21
V.2 – Notation de Landau ou O....21
V.2.1 – Définition....21
V.2.2 – Exemple de classes de fonctions usuelles....22
V.2.3 – Opérations avec O....23
V.2.4 – Relation d’inclusion pour les classes de fonctions usuelles....27
V.3 – Notation o (petit o)....28
V.3.1 – Définition....28
V.3.2 – Propriétés....28
V.4 – Notation Ω (grand oméga)....30
V.5 – Notation ω (oméga)....30
V.6 – Notation Θ (thêta)....31
V.7 – Notation ~....35

V.8 – Récapitulatif des comparaisons de fonctions....35
VI – Conclusion....36
Table des illustrations et tableaux....37
Références....37
Ouvrages....37
Thèses....37
Articles....38
Internet....38

I – Introduction

"Si l’on veut résoudre un problème à l’aide d’un ordinateur, il est indispensable d’avoir recours à un algorithme car un ordinateur ne peut exécuter que des ordres précis et sans ambiguïté."

Stockmeyer et Chandra[1]

Ce document a été créé dans le cadre du module ENG 111 « Information et communication pour ingénieur » du CNAM de Limoges.

Il a pour but de prodiguer au lecteur des notions pour lui permettre de comprendre la complexité algorithmique et comment l’utiliser. Cette notion est importante car si vous voulez résoudre un problème avec un ordinateur, comme le disent Stockmeyer et Chandra, il vous faut un algorithme. Or pour pouvoir l’étudier et le comparer avec d’autres, il faut utiliser la notion de complexité : d’où ce document.

Nous donnerons les notions de base sur les différentes classes de problèmes et quelques conseils sur l’optimisation des programmes.

Enfin, nous ferons une présentation de notions mathématiques sur le comportement asymptotique des fonctions qui sont utilisées lors de l’étude de la notion de complexité algorithmique.

1 Les problèmes intrinsèquement difficiles, page 128

II – Complexité

Nous allons commencer par définir la complexité d'un algorithme, puis étudier les données qui lui sont liées, calculer certaines complexités pour pouvoir enfin obtenir un outil de comparaison.

II.1 – Définitions

Définition 1

Algorithme

Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein[2], ont donné cette définition :

"Procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie."

Exemple 1

Pour rechercher un nombre dans un tableau trié, le principe de dichotomie consiste à couper ce tableau en deux, à chercher dans une partie et si ce n'est dans celle-ci, alors nous cherchons dans l'autre et ainsi de suite jusqu'à la fin. Nous nous arrêtons quand la valeur a été trouvée ou bien lorsque nous sommes arrivé à la fin du tableau c.a.d. quand la valeur recherchée n'est pas dans le tableau.

Algorithme de recherche dichotomique

RECHERCHE_DICHOTOMIQUE
Entrée : un tableau de n *nombres* $A=[a_1, a_2, a_3, ..., a_n]$ *trié par ordre croissant et une valeur* v
Sortie : un indice i *tel que* $v=A[i]$*, ou bien la valeur spéciale NIL si* v *ne figure pas dans* A

```
1 début ← 1
2 fin ← n
3 trouvé ← FALSE
4 répéter
5       milieu ← partie entière(début + (fin – début) / 2)
6       si A[milieu]=v
7       alors trouvé ← TRUE
8       sinon
9               si v⩾A[milieu]
10              alors
11                      début ← milieu + 1
12                      si A[début]=v
13                      alors
14                              milieu ← début
15                              trouvé ← TRUE
16                      fin si
```

2 Introduction à l'algorithmique, page 3

```
17          sinon
18              fin ← milieu – 1
19              si A[fin]=v
20              alors
21                  milieu ← fin
22                  trouvé ← TRUE
23              fin si
24          fin si
25      fin si
26 jusqu'à trouvé ou début ≥ fin
```

F. Grimbert a expliqué que :

Définition 2

Complexité d'un algorithme

La complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée.

Pour Stockmeyer et Chandra[3], *"l'efficacité d'un algorithme est mesurée par l'augmentation du temps de calcul en fonction du nombre des données."*

Nous avons donc deux éléments à prendre en compte :

- la taille des données ;
- le temps de calcul.

II.2 – La taille des données

La taille des données (ou des entrées) va dépendre du codage de ces entrées.

On choisit comme taille la ou les dimensions les plus significatives.

Par exemple, en fonction du problème, les entrées et leur taille peuvent être :

- des éléments : le nombre d'éléments ;
- des nombres : nombre de bits nécessaires à la représentation de ceux-là ;
- des polynômes : le degré, le nombre de coefficients non nuls ;
- des matrices m × n : max(m,n), m.n, m + n ;
- des graphes : nombre de sommets, nombre d'arcs, produit des deux ;
- des listes, tableaux, fichiers : nombre de cases, d'éléments ;
- des mots : leur longueur.

3 Les problèmes intrinsèquement difficiles, page 130

II.3 – Le temps de calcul

Le temps de calcul d'un programme dépend de plusieurs éléments :

- la quantité de données bien sûr ;
- mais aussi de leur encodage ;
- de la qualité du code engendré par le compilateur ;
- de la nature et la rapidité des instructions du langage ;
- de la qualité de la programmation ;
- et de l'efficacité de l'algorithme.

Nous ne voulons pas mesurer le temps de calcul par rapport à toutes ces variables. Mais nous cherchons à calculer la complexité des algorithmes qui ne dépendra ni de l'ordinateur, ni du langage utilisé, ni du programmeur, ni de l'implémentation. Pour cela, nous allons nous mettre dans le cas où nous utilisons un ordinateur RAM (Random Access Machine) :

- ordinateur idéalisé ;
- mémoire infinie ;
- accès à la mémoire en temps constant ;
- généralement à processeur unique (pas d'opérations simultanées).

Pour connaître le temps de calcul, nous choisissons une opération fondamentale et nous calculons le nombre d'opérations fondamentales exécutées par l'algorithme.

II.4 – Opération fondamentale

C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme.

Par exemple :

Problème	Opération fondamentale
Recherche d'un élément dans une liste	Comparaison
Tri d'une liste, d'un fichier, …	Comparaisons, déplacements
Multiplication des matrices réelles	Multiplications et additions
Addition des entiers binaires	Opération binaire

Tableau 1 : opérations fondamentales en fonction des problèmes

II.5 – Coût des opérations

II.5.1 – Coût de base

Pour la complexité en temps, il existe plusieurs possibilités :

- première solution : calculer (en fonction de n) le nombre d'opérations élémentaires (addition, comparaison, affectation, …) requises par l'exécution puis le multiplier par le temps moyen de chacune d'elle ;
- pour un algorithme avec essentiellement des calculs numériques, compter les opérations coûteuses (multiplications, racine, exponentielle, …) ;
- sinon compter le nombre d'appels à l'opération la plus fréquente (cf. tableau 1).

Ici, nous adoptons le point de vue suivant : l'exécution de chaque ligne de pseudo code demande un temps constant. Nous noterons c_i le coût en temps de la ligne i.

II.5.2 – Coût en séquentiel

Séquence : $T_{séquence}(n)=\sum T_{éléments\ de\ la\ séquence}(n)$

Alternative : *si C alors J sinon K* $T(n)=T_C(n)+max\{T_J(n),T_K(n)\}$

Itération bornée : *pour i de j à k faire B* $T(n)=(k-j+1).(T_{entête}(n)+T_B(n))+T_{entête}(n)$. Dans l'en-tête est mis l'affectation de l'indice et le test de continuation.

Itération non bornée : *tant que C faire B* $T(n)=Nb_{boucles}.(T_B(n)+T_C(n))+T_C(n)$ avec $Nb_{boucles}$ le nombre de boucles qui s'évalue par méthode inductive.

Répéter B jusqu'à C $T(n)=Nb_{boucles}.(T_B(n)+T_C(n))$

Exemple 2

Fonction de multiplication de deux matrices[4]

```
MULTIPLICATIONMATRICES(A,B)
entrée : deux matrices A, B  n×n
sortie : matrice C  n×n
1  n ← ligne[A]
2  Soit C une matrice  n×n
3  pour i ← 1 à  n
4      faire pour j ← 1 à  n
5          faire c_ij ← 0
6              pour k ← 1 à  n
7                  faire c_ij ← c_ij + a_ik . b_kj
8              fin pour
9      fin pour
10 fin pour
11 retourner C
```

4 Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, Introduction à l'algorithmique, chapitre 25.1

II.5.3 – Coût en récursif

La méthode "diviser pour régner" crée des algorithmes de type récursif.

Diviser pour régner
Cette méthode utilise trois étapes à chaque niveau de la récursivité :

a) *Diviser : le problème en un certain nombre de sous-problèmes ;*
b) *Régner : sur les sous-problèmes en les résolvant de manière récursive. Si la taille du sous-problème est assez petite, il est résolu immédiatement ;*
c) *Combiner : les solutions des sous-problèmes pour produire la solution du problème originel.*

Il faut trouver :

- la relation de récurrence associée ;
- la base de la récurrence ;
- et les cas d'arrêt.

$T(n)$ est calculé grâce à des relations de récurrence.[5]

Exemple 3

Fonction factoriel calculée par récursivité

FACTORIEL (n)
entrée : un entier n
sortie : $n!$
1 ***Si*** $n \leqslant 1$
2 ***alors*** *retour ← 1*
3 ***sinon*** *retour ←* n ×*FACTORIEL(* $n-1$ *)*
4 ***fin si***
5 ***retourne*** *retour*

II.6 – Les différentes mesures de complexité

Il existe trois mesures différentes, la complexité dans le meilleur des cas, la complexité en moyenne et la complexité dans le pire des cas. Mais la troisième est beaucoup plus employée que les autres (cf. II.6.3).

Soit A un algorithme, n un entier, D_n l'ensemble des entrées de taille n et une entrée $d \in D_n$. Posons : $coût_A(d)$ le nombre d'opérations fondamentales effectuées par A avec l'entrée d.

II.6.1 – La complexité dans le meilleur des cas

Définition 3

Elle est obtenue par : $Min_A(n) = min\{coût_A(d) / d \in D_n\}$.

5 Pour calculer le coût des récurrences, cf. le chapitre 4 de Introduction à l'algorithmique.

II.6.2 – La complexité en moyenne

C'est le temps d'exécution moyen ou attendu d'un algorithme. La difficulté pour ce cas est de définir une entrée "moyenne" pour un problème particulier.

Définition 4

$Moy_A(n)=\sum_{d\in D_n} p(d).coût_A(d)$ avec $p(d)$ une loi de probabilité sur les entrées.

Il reste à définir $p(d)$. Une hypothèse est que toutes les entrées ayant une taille donnée sont équiprobables.

D'où : $Moyenne\,uniforme_A(n)=\frac{1}{card(D_n)}\sum_{d\in D_n} coût_A(d)$

D'autres hypothèses se basent sur l'analyse probabiliste[6].

II.6.3 – La complexité dans le pire des cas

Définition 5

Elle est donnée par : $Max_A(n)=max\{coût_A(d)/d\in D_n\}$.

C'est cette complexité qui est généralement calculée car c'est une borne supérieure du temps d'exécution associé à une entrée quelconque.

Exemple 4

Tri à bulle[7]

TRI-BULLE (A)
entrée : tableau A
sortie : tableau A trié par ordre croissant
1 **Pour** *i ← 1* ***à*** *longueur[A]*
2 ***faire pour*** *j ← longueur[A]* ***décroissant jusqu'à*** *i+1*
3 ***faire si*** *A[j] < A[j-1]*
4 ***alors*** *permuter A[j] ↔ A[j – 1]*
5 ***fin si***
6 ***fin pour***
7 ***fin pour***

II.6.4 – Calcul des complexités des exemples

a) Multiplication de deux matrices

L'algorithme est l'exemple 2 page 7.

Soient $T(n)$ *le temps d'exécution en fonction de l'argument* n *et* c_i *le coût en temps de la ligne* i.
Nous avons :

$$T(n)=c_1+c_2+n[c_3+n(c_4+c_5+n(c_6+c_7))]$$

6 Cf. Introduction à l'algorithmique, page 24 et chapitre 5 Analyse probabiliste et algorithmes randomisés.
7 Cf. Introduction à l'algorithmique, chapitre 2 problème 2.2 page 35

$$T(n)=c_1+c_2+c_3.n+(c_4+c_5).n^2+(c_6+c_7).n^3$$

Ainsi $T\in\Theta(n^3)$ [8]*pour les trois complexités.*

b) Factoriel

L'algorithme est l'exemple 3 page 8.

Soit $T(n)$ *le temps d'exécution en fonction de l'argument* n *.*
À la base $T(1)=c_1+c_2$
Sinon $T(n)=c_1+c_3+T(n-1)$ *pour* $n>1$
Démontrons que $T(n)=c_1+c_2+(n-1)(c_1+c_3)$ *pour* $n\geqslant 1$

Pour $n=1$ *: vrai.*
Nous le supposons vrai au degré n *.*
Démontrons-le au degré $n+1$ *.*

$$T(n+1)=c_1+c_3+T(n)$$

$$T(n+1)=c_1+c_3+c_1+c_2+(n-1)(c_1+c_3)=c_1+c_2+n(c_1+c_3)$$

Comme la proposition est vraie au degré 1, et si elle est vraie au degré n *, elle est vraie au degré* $n+1$ *, d'après le principe de récurrence, elle est vraie quel que soit* n *.*
Donc $T(n)=\Theta(n)$

c) Recherche dichotomique

L'algorithme est l'exemple 1 page 4.

Calcul de la complexité dans le meilleur des cas : l'élément $A[milieu]=v$.

$T(n)$ *vaut donc :*
$T(n)=c_1+c_2+c_3+c_4+c_5+c_6+c_7+c_{25}+c_{26}$
$T(n)=\Theta(1)$

Calcul de la complexité dans le pire des cas : v ne figure pas dans A .

Supposons que v *est plus grand que tous les éléments de* A *.*
$T(n)$ *vaut donc :*

$$T(n)=c_1+c_2+c_3+x.(c_4+c_5+c_6+c_8+c_9+c_{10}+c_{11}+c_{12}+c_{16}+c_{24}+c_{25}+c_{26})$$

L'algorithme passe x *fois dans la boucle répéter jusqu'à ce que* $début\geq fin$

Calculons x*. Pour cela, nous allons émettre une hypothèse puis la démontrer par récurrence.*
$T(n)$ *est du type :* $T(n)=C_1+C_2x$ *avec* C_1,C_2 *des constantes.*
Si $n=2$ *ou* $n=3$ *,* $x=1$ *.*
Si $n=4$ *ou* $n=5$ *,* $x=2$ *.*
Si $n=8$ *ou* $n=9$ *,* $x=3$ *.*
En fait, C_1 *est le temps pour initialiser l'algorithme et* C_2x *le temps nécessaire à l'algorithme pour gérer le tableau de taille* n *. Posons* $T'(n)$ *le temps pour effectuer la boucle pour une taille* n *.*
Par la suite, pour simplifier, nous supposons que la taille du problème initial est une puissance de deux.
Démontrons que $T'(n)=C_2.\log_2(n)$ *.*
Nous le supposons vrai …
au niveau n *.*

8 Pour la notation Θ cf. V.6 Notation Θ (thêta) page 31

Démontrons-le au niveau $2n$ *.*

$T'(2n)=C_2+T'(n)=C_2+C_2.\log_2(n)=C_2(1+\log_2(n))=C_2(\log_2(2)+\log_2(n))=C_2\log_2(2n)$

Comme la proposition est vraie au degré 1, et si elle est vraie au degré n *, elle est vraie au degré* $2n$ *, d'après le principe de récurrence, elle est vraie quel que soit* n *puissance de 2.*

D'où

$$T(n)=C_1+T'(n)=C_1+C_2.\log_2(n)\in\Theta(1)+\Theta(\log_2(n))$$

D'où $T(n)\in\Theta(\log_2(n))$

Remarque

Question ouverte : Quelle est la complexité d'un tri dichotomique ?

Réponse : $n\log_2(n)$

d) Tri à bulle

L'algorithme est l'exemple 4 page 9.

Calcul de la complexité dans le meilleur des cas : le tableau est déjà trié par ordre croissant.

La quatrième ligne n'est jamais exécutée.

$$\begin{aligned}T(n)&=\sum_{i=1}^{n}(c_1+\sum_{j=i+1}^{n}(c_2+c_3))\\&=\sum_{i=1}^{n}c_1+\sum_{i=1}^{n}\sum_{j=i+1}^{n}(c_2+c_3)\\&=c_1.\sum_{i=1}^{n}1+(c_2+c_3).\sum_{i=1}^{n}\sum_{j=i+1}^{n}1\\&=c_1.n+(c_2+c_3).\sum_{i=1}^{n}(n-i)\end{aligned}$$

$$\begin{aligned}\sum_{i=1}^{n}(n-i)&=\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\\&=n^2-\frac{1}{2}n(n+1)^{*}\\&=n^2-\frac{1}{2}n^2-\frac{1}{2}n\\&=\frac{1}{2}n^2-\frac{1}{2}n\end{aligned}$$

$$\begin{aligned}T(n)&=c_1.n+(c_2+c_3).(\frac{1}{2}n^2-\frac{1}{2})\\&=\frac{(c_2+c_3)}{2}n^2+\frac{2c_1-c_2-c_3}{2}n\end{aligned}$$

D'où $T(n)\in\Theta(n^2)$

Calcul de la complexité dans le pire des cas : le tableau est trié par ordre décroissant

---

* Rappel : la sommation d'une série arithmétique a pour valeur $\sum_{k=1}^{n}k=\frac{1}{2}n(n+1)$

La quatrième ligne est toujours exécutée. Nous avons donc :

$$T(n)=\sum_{i=1}^{n}(c_1+\sum_{j=i+1}^{n}(c_2+c_3+c_4))$$
$$=c_1.n+(c_2+c_3+c_4).(\frac{1}{2}n^2-\frac{1}{2})$$
$$=\frac{(c_2+c_3+c_4)}{2}n^2+\frac{2c_1-c_2-c_3-c_4}{2}n$$

D'où $T(n)\in\Theta(n^2)$

L'algorithme tri à bulle est donc dans tous les cas de complexité $\Theta(n^2)$.

Remarque

Quand un algorithme contient une structure de contrôle itératif comme une boucle **tant que** ou **pour**, nous obtenons comme dans l'exemple ci-dessus des sommations qui nous ramènent à des séries.

II.7 – Comparaison de deux algorithmes

II.7.1 – Méthode de comparaison

Si le nombre d'entrées utilisées pour un algorithme est très faible, généralement le temps d'exécution de l'algorithme l'est aussi. Ce qui nous intéresse, c'est comment va réagir l'algorithme pour un nombre important de données.

Donc pour comparer deux algorithmes, nous allons comparer leur taux de croissance ou ordre de grandeur (cf. V.8 page 35).

Définition 6

Efficacité d'un algorithme par rapport à un autre

« On considère généralement qu'un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur. » (Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein[9]).

C'est pour ces raisons que la notation O [10] est souvent utilisée pour la mesure de la complexité car elle décrit une borne supérieure du cas le plus défavorable.

II.7.2 – Classification des algorithmes

M. T. FDIL[11] illustre la classification par des explications.

Les algorithmes habituellement rencontrés peuvent être classés dans les catégories suivantes :

9 Introduction à l'algorithmique chapitre 2.3 page 25
10 O cf. V.2 Notation de Landau ou O page 21
11 Algorithmique et langage C Chapitre 1 : Complexité des algorithmes

Complexité	Description
Complexité $O(1)$ Complexité **constante** ou temps constant	L’exécution ne dépend pas du nombre d’éléments en entrée mais s’effectue toujours en un nombre constant d’opérations
Complexité $O(\log(n))$ Complexité **logarithmique**.	La durée d’exécution croît légèrement avec n. Ce cas de figure se rencontre quand la taille du problème est divisée par une entité constante à chaque itération.
Complexité $O(n)$ Complexité **linéaire**	C’est typiquement le cas d’un programme avec une boucle de 1 à n et le corps de la boucle effectue un travail de durée constante et indépendante de n.
Complexité $O(n\log(n))$ Complexité **n-logarithmique**	Se rencontre dans les algorithmes où à chaque itération la taille du problème est divisée par une constante avec à chaque fois un parcours linéaire des données. Un exemple typique de ce genre de complexité est l’algorithme de tri "quick sort" qui, de manière récursive, divise le tableau à trier en deux morceaux par rapport à une valeur particulière appelée pivot, trie ensuite la moitié du tableau puis l’autre moitié … S’il n’y avait pas cette opération de "division" l’algorithme serait logarithmique puisqu’on divise par 2 la taille du problème à chaque étape. Mais, le fait de reconstituer à chaque fois le tableau en parcourant séquentiellement les données ajoute ce facteur n au $\log(n)$. Noter que la complexité n-logarithmique est tributaire du bon choix du pivot.
Complexité $O(n^2)$ Complexité **quadratique**	Typiquement c’est le cas d’algorithmes avec deux boucles imbriquées chacune allant de 1 à n et avec le corps de la boucle interne qui est constant.
Complexité $O(n^3)$ Complexité **cubique**	Idem quadratique mais avec ici par exemple trois boucles imbriquées.
Complexité $O(n^p)$ Complexité **polynomiale**	Algorithme dont la complexité est de la forme $O(n^p)$ pour un certain p. Toutes les complexités précédentes sont incluses dans celle-ci (cf. théorème 2 page 28).
Complexité $O(2^n)$ Complexité **exponentielle**	Les algorithmes de ce genre sont dits "naïfs" car ils sont inefficaces et inutilisables dès que n dépasse 50.

Tableau 2 : classification des algorithmes

Définition 7

À l’inverse d’un algorithme naïf (complexité exponentielle) et par convention, un algorithme est dit **praticable, efficace** s’il est polynomial.

II.8 – Complexité d'un problème

La complexité des algorithmes a abouti à une classification des problèmes en fonction des performances des meilleurs algorithmes connus qui les résolvent.

Définition 8

La **complexité d'un problème** A est la complexité du meilleur algorithme qui résout A.

Exemples donnés par J.-Ch. Arnulfo[12]

Structure	Ajout d'un nouvel élément	Retrait du premier élément	Retrait de l'élément médian	Recherche d'un élément
Tableau	$O(1)$	$O(n)$	$O(1)$ ou $O(n)$ [13]	$O(n)$ ou $O(\log_2 n)$ [14]
Liste	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Pile	$O(1)$	$O(1)$	-	-
File	$O(1)$	$O(1)$	-	-

Tableau 3 : comparaison de complexité des opérations élémentaires en fonction de la structure choisie

Complexité	Exemple
$O(1)$	Accès à un élément de tableau
$O(\log(n))$	Recherche dichotomique
$O(n)$	Recherche dans un tableau non trié
$O(n\log(n))$	Tri rapide[15]
$O(n^2)$	Tri à bulles (cf. exemple 4 page 9)
$O(n^3)$	Multiplication de matrices[16](cf. exemple 2 page 7)
$O(2^n)$	Algorithme du "voyageur de commerce"[17]

Tableau 4 : exemples de problèmes en fonction de la complexité

12 Métier Développeur kit de survie, page 151
13 Constant si on remplace un élément existant, linéaire sinon(nous devons décaler les éléments suivants).
14 Logarithmique si le tableau est ordonné, linéaire sinon.
15 Dans le cas moyen et favorable, en $O(n^2)$ dans le cas défavorable cf. Introduction à l'algorithmique chapitre 7
16 L'algorithme de Strassen (Introduction à l'algorithmique chapitre 28.2) est lui en $\Theta(n^{(\log(7))})=\Theta(n^{2,81})$
17 Cf. définition 16 page 17 et l'exemple associé

II.9 – Comportement des fonctions usuelles

Voici quelques données illustrant le comportement des fonctions vues dans la classification précédente.

n	10	100	1000	1E+6	1E+9
$\log_2(n)$	3,32	6,64	9,97	19,93	29,9
n	10	100	1000	1E+06	1E+09
$n.\log_2(n)$	33,22	664,39	1E+04	2E+07	3E+10
n^2	100	1E+04	1E+06	1E+12	1E+18
n^3	1000	1E+06	1E+09	1E+18	1E+27
n^5	1E+05	1E+10	1E+15	1E+30	1E+45
2^n	1024	1,27E+030	1,07E+301		

Tableau 5 : nombre d'opérations nécessaires pour exécuter un algorithme en fonction de sa complexité et de la taille des entrées du problème traité

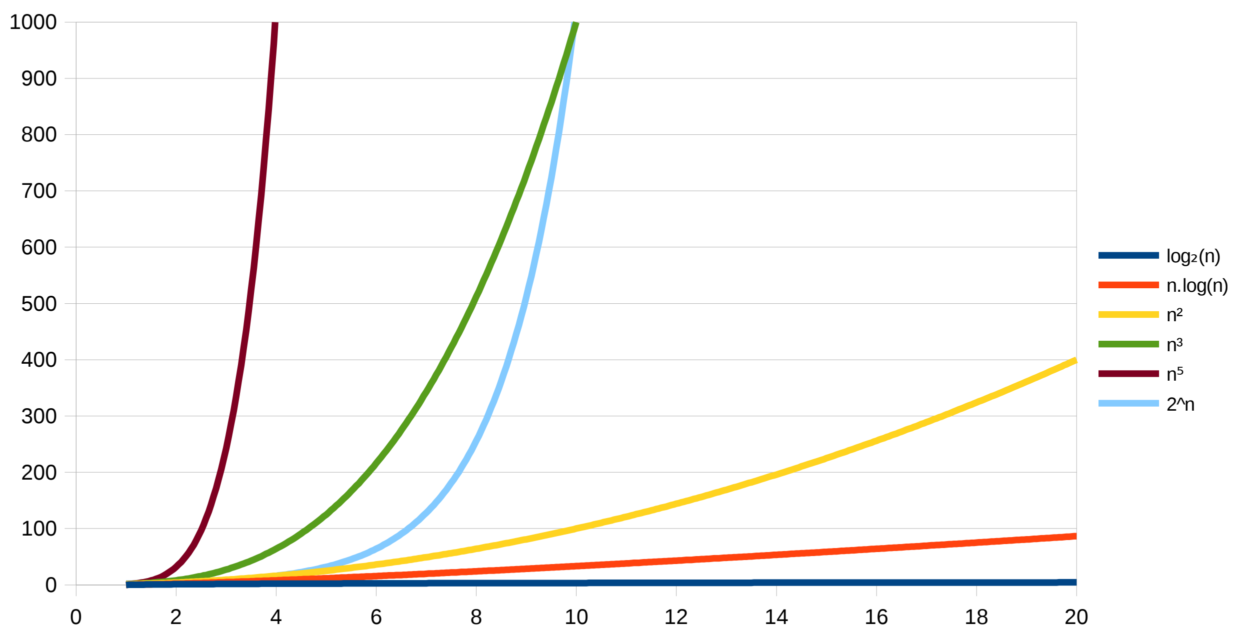

Illustration 1 : graphique représentant les différentes courbes des fonctions utilisées dans les complexités usuelles dans l'intervalle [0;20]

n	10		100		1 000		1E+06		1E+09	
$\text{Log}_2(n)$	3,32E-009	10^{-9} s	6,64E-009	10^{-9} s	9,97E-009	10^{-8} s	1,99E-008	10^{-8} s	2,99E-008	10^{-8} s
n	1,00E-008	10^{-8} s	1,00E-007	10^{-7} s	1,00E-006	10^{-6} s	1,00E-003	10^{-3} s	1,00E+000	1 s
n.log(n)	3,32E-008	10^{-8} s	6,64E-007	10^{-6} s	9,97E-006	10^{-5} s	1,99E-002	10^{-2} s	2,99E+001	30 s
n^2	1,00E-007	10^{-7} s	1,00E-005	10^{-5} s	1,00E-003	10^{-3} s	1,00E+003	17 min	1,00E+009	32 ans
n^3	1,00E-006	10^{-6} s	1,00E-003	10^{-3} s	1,00E+000	1 s	1,00E+009	32 ans	1,00E+018	3.10^{8} siècles
n^5	1,00E-004	10^{-4} s	1,00E+001	10 s	1,00E+006	11,5 jours	1,00E+021	3.10^{11} siècles	1,00E+036	
2^n	1,02E-006	10^{-6} s	1,27E+021	4.10^{11} siècles	1,07E+292					

Tableau 6 : temps nécessaire pour exécuter un algorithme en fonction de sa complexité et de la taille des entrées du problème traité en supposant que nous disposons d'un ordinateur capable de traiter 10^9 opérations par seconde

III – Classes de problèmes

Définitions 9

Machine déterministe

Les machines déterministes font toujours un seul calcul à la fois. Ce calcul est constitué d'étapes élémentaires ; à chacune de ces étapes, pour un état donné de la mémoire de la machine, l'action élémentaire effectuée sera toujours la même.

Nos ordinateurs sont des machines déterministes.

Définitions 10

Machine non-déterministe

Une machine non-déterministe est une variante purement théorique : on ne peut pas construire de telle machine. À chaque étape de son calcul, cette machine peut effectuer un choix non-déterministe : elle a le choix entre plusieurs actions, et elle en effectue une. Si l'un des choix l'amène à accepter l'entrée, on considère qu'elle a fait ce choix-là. En quelque sorte, elle devine toujours juste. Une autre manière de voir leur fonctionnement est de considérer qu'à chaque choix non-déterministe, elles se dédoublent, les clones poursuivent le calcul en parallèle suivant les branches du choix. Si l'un des clones accepte l'entrée, on dit que la machine accepte l'entrée. (définitions tirées de Techno-Science.net)

Les différents problèmes ont été classés dans des ensembles dont voici une liste non exhaustive.

Définition 11

L'ensemble L est l'ensemble des problèmes pour lesquels il existe un algorithme de résolution en temps logarithmique en fonction de la taille des entrées.

Définition 12

l'ensemble P est l'ensemble des problèmes pour lesquels il existe un algorithme de résolution en temps polynomial en fonction de la taille des entrées.

Les algorithmes des problèmes de P sont dits efficaces (cf. II.7.2 page 13).

Mais il existe des problèmes pour lesquels on ne connaît aucune solution efficace.

Définition 13

La classe NP (problèmes **N**on-déterministes **P**olynomiaux) est l'ensemble des problèmes qui peuvent être résolus sur une machine non déterministe en temps polynomial en fonction de la taille des entrées.

Propriété 1

La classe NP est équivalente à l'ensemble des problèmes pour lesquels il existe un algorithme de vérification de la solution en temps polynomial en fonction de la taille des entrées.

Démonstration
Propriété admise

Exemples

Factorisation d'entiers : il est facile de vérifier une décomposition en facteurs premiers mais très difficile de la trouver sur un produit de nombres premiers très grands. C'est sur cette différence qu'est basé le cryptosystème à clés publiques RSA[18].

Propriété 2

Nous avons l'inclusion $P \subset NP$.

Démonstration
Si le problème possède un algorithme de résolution en temps polynomial, cet algorithme peut être converti en algorithme de vérification.

Mais est-ce que $P = NP$? Ce n'est pas parce que nous n'avons pas trouvé d'algorithme efficace à un problème qu'il n'en existe pas.

Si vous démontrez que $P = NP$ ou $P \neq NP$, vous deviendrez riche. En effet, l'Institut Clay[19] l'a sélectionnée parmi les questions dont la résolution est mise à prix un million de dollars. Certaines personnes pensent que l'égalité ou l'inégalité sont indémontrables dans la théorie des ensembles actuels et que nous pourrions rajouter l'un ou l'autre comme axiome (cf. J.-P. Delahaye[20]).

Définition 14 : Réduction de problèmes

Benoît Darties[21] en donne une explication :

" La réduction d'un problème P_1 vers un problème P_2 est une transformation de toute instance de P_1 en une instance de P_2 , de sorte que toute solution à P_1 puisse induire une solution à P_2 . Cette transformation doit être polynomiale en temps et en mémoire selon la taille de P_1 . Ainsi il y a une relation de difficulté de résolution entre ces deux problèmes. "

Soit C une classe de complexité (comme P , NP , …).

Définition 15

Un problème Q est dit C - difficile si ce problème est au moins aussi difficile que tous les problèmes dans C c.-à-d. que tout problème de C peut être réduit à Q par un algorithme polynomial.

Remarque

Le problème Q n'appartient pas nécessairement à C .

Définition 16

Un problème est C -complet s'il appartient à C et qu'il est C -difficile.

---

18 RSA : du nom de leurs inventeurs en 1977 Rivest, Shamir et Adleman.
19 cf. http://www.claymath.org/millennium/
20 Un algorithme à un million de dollars
21 Problèmes algorithmiques et de complexité dans les réseaux sans fils

Exemples de problèmes NP-complet

a) problème du cycle hamiltonien (cycle passant par tous les sommets d'un graphe une et une seule fois.)
b) problème du "voyageur de commerce" qui consiste, étant donné un ensemble de villes séparées par des distances données, à trouver le plus court chemin qui relie toutes les villes. Ce problème est NP - complet comme l'expliquent Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein[22].

C'est Stephan A. Cook, de l'Université de Toronto, qui a démontré pour la première fois en 1971, l'existence d'un problème NP-complet. (Harry R. Lewis, Ch. H. Papadimitriou[23])

Voici une illustration sur la hiérarchie des classes de problèmes (cf. Illustration 2) faite à partir de celle de Nathalie Revol[24].

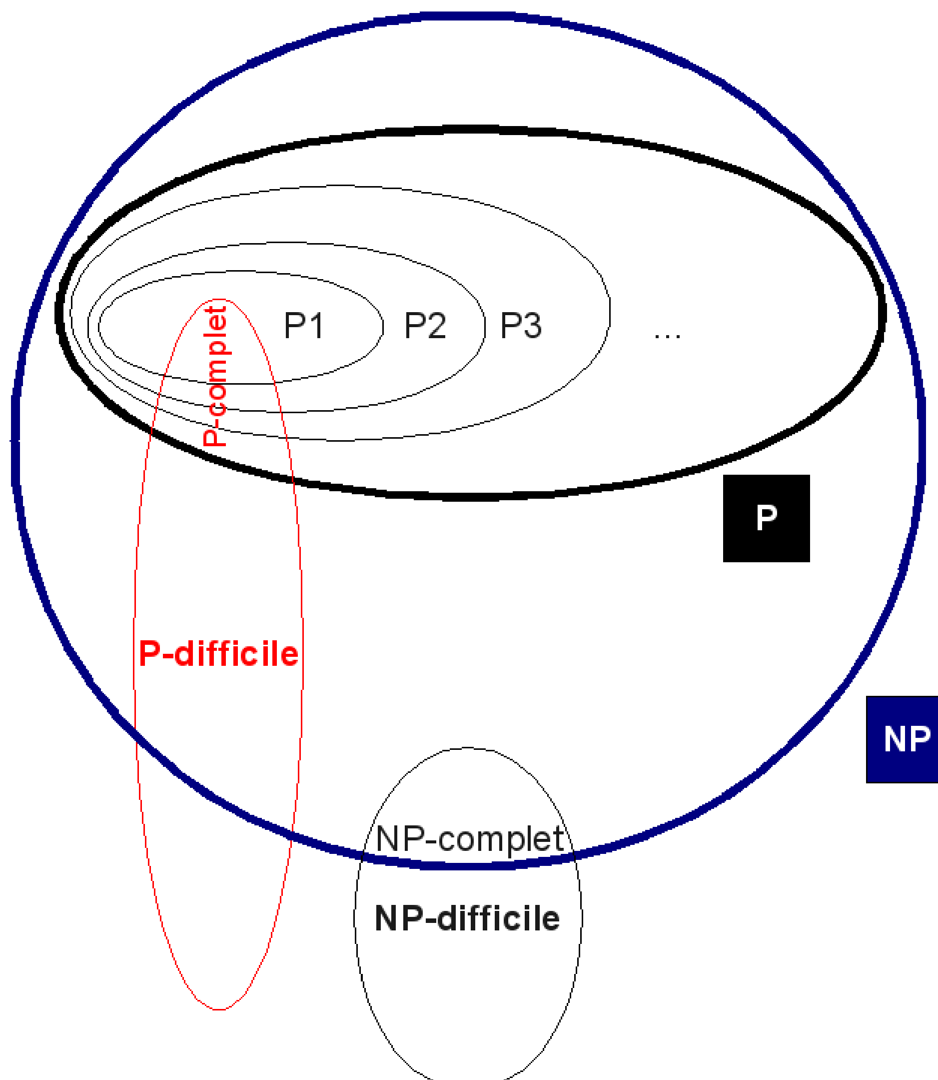

Illustration 2 : hiérarchie d'ensembles de classes de problèmes

22 Introduction à l'algorithmique, chapitre 34.5.4 page 978
23 L'efficacité des algorithmes, page 125
24 Algorithmes et complexité, illustration en arithmétique des ordinateurs page 54

IV – Utilisation de la complexité algorithmique dans un programme

" "Faites-le d'abord fonctionner. Puis, faites-le bien. Enfin, faites en sorte qu'il aille vite". Ce principe et ses variantes, est considéré comme la règle d'or de la programmation. Il est attribué à Kent Beck, qui lui-même l'attribue à son père."

Alex Martelli, Python en concentré, chapitre 17

IV.1 – Optimisation, pas si sûr

Avant d'optimiser un code, il faut absolument que le code fonctionne et qu'il soit bien conçu (que l'architecture et sa conception vous conviennent).

Remarque

J-Ch. Arnulfo[25] nous explique que : "L'optimisation d'un programme ne doit pas être l'étape ultime du développement. Les performances se jouent dès la phase de conception et tout au long de l'implémentation."

Si nous désirons optimiser le code, il faut en fait se poser quelques questions listées par J.-Ch. Arnulfo :

- Est-ce que quelqu'un s'est plaint de la lenteur du logiciel ?
- Est-ce ce bout de code qui pose problème et qui doit être optimisé ? Un profilage (cf. IV.2.1) sera intéressant à effectuer.
- Est-ce que le gain sera assez significatif pour que l'amélioration soit visible par l'utilisateur ? La fonction en question est-elle assez souvent utilisée ?
- Est-ce que l'optimisation est sans effet de bord ? Pensez à conserver l'ancien code en commentaire pour pouvoir y revenir si nécessaire.
- Est-ce que la lisibilité du code ne va pas être réduite ? En effet si nous écrivons du code avec par exemple des opérations sur bit pour gagner quelques microsecondes celui-ci se complexifie et pourrait devenir illisible. Et cela est source d'ennuis.

M. Luc Brun dans les qualités d'un algorithme nous parle de l'optimisation qui n'est pas le seul élément à prendre en compte mais fait partie d'un tout.

Qualités d'un algorithme : [26]

1. Maintenable (facile à comprendre, coder, déboguer) ;
2. Rapide.

Conseils :

1. Privilégier le point 2 sur le point 1 uniquement si l'algorithme diminue en complexité
2. « ce que fait » l'algorithme doit se lire lors d'une lecture rapide :
   a) Une idée par ligne ;
   b) Indenter[27] le programme.
3. Faire également attention à la précision, la stabilité et la sécurité.

La rapidité d'un algorithme est l'un des éléments définissant les qualités de celui-ci.

25 Métier Développeur kit de survie, page 130
26 M. Luc BRUN, Algorithmique … Complexité, diapositive 15
27 Indenter consister à ajouter des espaces ou des tabulations au niveau des lignes de code pour rendre celui-ci plus lisible.

IV.2 – Et si nous optimisions ?

IV.2.1 – Mesure de vitesse (profiling)

Définition 17

Profilage

" Le **profiling** d’un programme permet d’identifier les endroits où celui-ci passe le plus de temps, mais également quelles sont les fonctions qui sont exécutées et combien de fois. Un **profiler** est un programme capable d’analyser un exécutable. Le résultat de l’analyse est appelé un **profile** " (ou profilage). (par M. Idrissi Aouad et O. Zendra[28])

A. Martelli[29] explique :

"Un profilage met souvent en évidence le fait que les problèmes de performances se situent dans un petit sous-ensemble du code, soit 10 à 20 %, dans lequel votre programme passe 80 à 90 % de son temps. Ces régions cruciales sont également appelées *goulets d’étranglement* ou *points chauds*."

Pour mesurer le temps que prend une partie du code, une solution simple est de doubler (exécuter deux fois) ou de supprimer (ne pas exécuter) cette partie de code.

Sinon il est possible de mettre manuellement des chronomètres dans le code.

Enfin, il existe dans certains langages des bibliothèques/modules de profilage. Elles collectent des données lors de l’exécution de votre programme.

IV.2.2 – Optimisation à grande échelle

Il faut dans un premier temps, rechercher les goulets d’étranglement dus à l’architecture de l’application, le choix des algorithmes (en fonction de leur complexité) et le choix des structures de données.

Or, comme énoncé dans la remarque du IV.1, cette étude ne doit pas être faite à la fin mais tout au long de la programmation.

IV.2.3 – Optimisation à petite échelle

Cela peut être utile sur certains points chauds. Il faut bien connaître les particularités du langage utilisé (gestion de la mémoire en particulier). Sinon les goulets d’étranglement se situent généralement dans les boucles, notamment les boucles imbriquées. Regardez s’il n’y a pas d’invariant que vous pouvez extraire de la boucle.

---

28 Outils de caractérisation du comportement mémoire et d’estimation de la consommation énergétique, http://hal.archives-ouvertes.fr/docs/00/26/05/83/PDF/RT-0350.pdf (lien vérifié le 10/11/2013)

29 Python en concentré, chapitre 17 page 401

V – Notations mathématiques

La notion de complexité nécessite quelques notions de mathématiques qui vont être rappelées ci-dessous.

V.1 – Rappel sur les limites en +∞

Définition 18

Soit f une fonction. Nous disons que la **limite** de $f(x)$ quand x tend vers $+\infty$ est égale à $L \in \mathbb{R}$ si :

$$\forall \varepsilon > 0, \exists S \in \mathbb{R} \text{ tel que } \forall x > S, 0 < |f(x) - L| < \varepsilon$$

Nous notons : $\lim_{x \to +\infty} f(x) = L$.

Ceci signifie que $f(x)$ est une valeur approchée de L de plus en plus précise lorsque x est de plus en plus grand, et que n'importe quelle précision peut être choisie pourvu que les x sélectionnés soient assez grand.

Définition 19

De même, soit f une fonction. La limite de $f(x)$ est égale à $+\infty$ quand x tend vers $+\infty$ si :

$$\forall R > 0, \exists S > 0 \text{ tels que } \forall x > S, f(x) > R$$

Elle est notée : $\lim_{x \to +\infty} f(x) = +\infty$.

V.2 – Notation de Landau ou O

V.2.1 – Définition

Remarque

La notation O est utilisée pour indiquer la borne supérieure asymptotique.

Définition 20

Soit g une fonction définie sur une partie des nombres réels. L'ensemble $O(g)$ est défini ainsi :

$$O(g) = \{ f \text{ fonction définie sur une partie de } \mathbb{R} / \exists c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, |f(x)| \leqslant c|g(x)| \}$$

Remarque

Nous lisons « grand o » de g pour $O(g)$.

En informatique ne sont utilisées que des fonctions définies sur $\mathbb{N}$. La définition devient donc :

Définition 21

Soit g une fonction définie sur $\mathbb{N}$ à valeurs entières positives. L'ensemble $O(g)$ est défini par :

$$O(g)=\{f \text{ fonction définie sur } \mathbb{N} \text{ à valeurs entières positives} / \\ \exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant c\, g(n)\}$$

Remarques

a) Le symbole O est aussi appelé symbole de Landau (de son créateur Edmund Georg Hermann Landau[30] né le 14 février 1877 et décédé le 19 février 1938).

b) Pour indiquer qu'une fonction f est un élément de l'ensemble $O(g)$, une autre écriture est : $f(n)=O(g(n))$

c) La notation O sert donc à comparer des limites en majorant une fonction, à un facteur constant près.

V.2.2 – Exemple de classes de fonctions usuelles

Voici deux exemples classiques :

a) L'ensemble $O(1)$ est l'ensemble des fonctions majorées par une constante à partir d'un certain rang [pour les fonctions définies sur $\mathbb{N}$, elles sont majorées (tout court).

Par exemple $\forall x \in \mathbb{R}^{+}, f(x)=\frac{2}{x} \in O(1)$.

Démonstration

Posons $c=2$ *et* $n_0=1$ $\forall x \geqslant n_0$, $\frac{2}{x} \leqslant c$

b) Soit $p \in \mathbb{N}^*$. Soit $a_0, a_1, \ldots a_p$ des nombres réels tels que $a_p \neq 0$ et Pour tout polynôme $P(n)=a_0+a_1 n+a_2 n^2+\cdots+a_{p-1} n^{p-1}+a_p n^p$ un polynôme réel de degré p, on a alors $P \in O(n^p)$.

Démonstration

Soit $P(n)=a_0+a_1 n+a_2 n^2+\cdots+a_{p-1} n^{p-1}+a_p n^p$ *avec* $a_p \neq 0$

$$\lim_{n \to +\infty} \frac{P(n)}{n^p}=\lim_{n \to +\infty} \frac{a_0}{n^p}+\lim_{n \to +\infty} \frac{a_1}{n^{p-1}}+\cdots+\lim_{n \to +\infty} \frac{a_{p-1}}{n}+\lim_{n \to +\infty} \frac{a_p n^p}{n^p}=0+\cdots+0+a_p=a_p$$

Donc par définition de la limite :

$$\forall \varepsilon>0, \exists S \in \mathbb{N} \text{ tel que } \forall n>S, \left|\frac{P(n)}{n^p}-a_p\right|<\varepsilon$$

Nous avons donc

$$\forall \varepsilon>0, \exists S \in \mathbb{N} \text{ tel que } \forall n>S, |P(n)-a_p . n^p|<\varepsilon . n^p$$

D'où

$$\forall \varepsilon>0, \exists S \in \mathbb{N} \text{ tel que } \forall n>S, 0<P(n)-a_p . n^p<\varepsilon . n^p \; ou \; 0<a_p . n^p-P(n)<\varepsilon . n^p$$

Si $0<a_p . n^p-P(n)<\varepsilon . n^p$ *alors* $P(n)<a_p . n^p$

Si $P(n)-a_p . n^p<\varepsilon . n^p$

$$P(n)<a_p . n^p+\varepsilon . n^p$$

30 cf. http://fr.wikipedia.org/wiki/Edmund_Landau, http://www.numbertheory.org/obituaries/LMS/landau/index.html

$$P(n)<(a_p+\varepsilon).n^p$$

Posons $c=max(a_p, a_p+\varepsilon)=a_p+\varepsilon$,

$$\exists n_o\in\mathbb{N}\, n_0=S \text{ tels que } \forall n\geqslant n_o, P(n)\leqslant c.n^p$$

V.2.3 – Opérations avec O

À partir de maintenant, les fonctions considérées seront définies de $\mathbb{R}^+$ dans $\mathbb{R}^+$.

Dans ce paragraphe, nous désignons par f et g deux fonctions de $\mathbb{R}^+$ dans $\mathbb{R}^+$ et nous donnerons des conditions nécessaires pour que l'une soit un O de l'autre.

Théorème 1

Soient f et g deux fonctions de $\mathbb{R}^+$ dans $\mathbb{R}^+$ et $b\in\mathbb{R}^*$. Nous supposons que $g(x)$ est non nul sur un intervalle de type]*A*;+∞[avec A réel. 1 – Si $\lim\limits_{x\to+\infty}\frac{f(x)}{g(x)}=b$ et $b>0$ alors $f\in O(g)$ et $g\in O(f)$. 2 – Si $\lim\limits_{x\to+\infty}\frac{f(x)}{g(x)}=0$ alors $f\in O(g)$ et $g\notin O(f)$.

Démonstration
1 – Première partie

Soient deux fonctions f *et* g *telles que* $\lim\limits_{x\to+\infty}\frac{f(x)}{g(x)}=b$ *et* $b>0$ *.*

Donc

$$\forall\varepsilon>0,\exists S\in\mathbb{R} \text{ tel que } \forall x>S, \left|\frac{f(x)}{g(x)}-b\right|<\varepsilon$$

Ce qui se traduit par :

$$\forall\varepsilon>0,\exists S\in\mathbb{R} \text{ tel que } \forall x>S, -\varepsilon<\frac{f(x)}{g(x)}-b<\varepsilon$$

Ou encore par :

$$\forall\varepsilon>0,\exists S\in\mathbb{R} \text{ tel que } \forall x>S, -\varepsilon+b<\frac{f(x)}{g(x)}<\varepsilon+b$$

$$\forall\varepsilon>0,\exists S\in\mathbb{R} \text{ tel que } \forall x>S, (-\varepsilon+b)g(x)<f(x)<(\varepsilon+b)g(x)$$

D'où avec $\varepsilon=1$ *:*

$$\exists S\in\mathbb{R} \text{ tel que } \forall x>S, f(x)<(b+1)g(x)$$

Donc $f\in O(g)$ *.*

Et

$$\forall\varepsilon>0,\exists S\in\mathbb{R} \text{ tel que } \forall x>S, (-\varepsilon+b)g(x)<f(x)$$

$$\forall b>\varepsilon>0,\exists S\in\mathbb{R} \text{ tel que } \forall x>S, g(x)<\frac{1}{(-\varepsilon+b)}f(x)$$

Avec $\varepsilon=\frac{b}{2}$ *:*

$$\exists S\in\mathbb{R} \text{ tel que } \forall x>S, g(x)<\frac{1}{(\frac{-b}{2}+b)}f(x)$$

$$\exists S\in\mathbb{R} \text{ tel que } \forall x>S, g(x)<\frac{2}{b}f(x)$$

Donc $g \in O(f)$ *.*

2 – Seconde partie

Soient deux fonctions f *et* g *telles que* $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = 0$ *.*

Nous avons donc :

$$\forall \varepsilon > 0, \exists S \in \mathbb{R} \text{ tels que } \forall x > S, \left|\frac{f(x)}{g(x)}\right| < \varepsilon$$

Ce qui signifie :

$$\forall \varepsilon > 0, \exists S \in \mathbb{R} \text{ tel que } \forall x > S, -\varepsilon < \frac{f(x)}{g(x)} < \varepsilon$$

Ou encore :

$$\forall \varepsilon > 0, \exists S \in \mathbb{R} \text{ tel que } \forall x > S, -\varepsilon . g(x) < f(x) < \varepsilon . g(x)$$

D'où avec $\varepsilon = 1$ *:*

$$\exists S \in \mathbb{R} \text{ tel que } \forall x > S, f(x) < 1 . g(x)$$

Donc $f \in O(g)$ *.*

Et

$$\forall \varepsilon > 0, \exists S \in \mathbb{R} \text{ tel que } \forall x > S, f(x) < \varepsilon . g(x)$$

$$\forall \varepsilon > 0, \exists S \in \mathbb{R} \text{ tel que } \forall x > S, \frac{1}{\varepsilon} f(x) < g(x)$$

Posons $\varepsilon' = \frac{1}{\varepsilon}$ *, nous avons*

$$\forall \varepsilon' > 0, \exists S \in \mathbb{R} \text{ tel que } \forall x > S, \varepsilon' f(x) < g(x)$$

Or

$$O(f) = \{g / \exists c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, g(x) \leqslant c f(x)\}$$

Donc

$$g \notin O(f) \Leftrightarrow \forall c > 0, \forall n_0 \in \mathbb{N} \text{ tels que } \exists x \geqslant n_0, g(x) > c f(x)$$

D'où $g \notin O(f)$ *.*

Propriété 3

Soit g une fonction de $\mathbb{R}^+$ dans $\mathbb{R}^+$. Nous avons :

$$\forall \ d \in \mathbb{R}^{+*}, \ O(d.g) = O(g)$$

Démonstration

Soit d un réel positif non nul. Il existe c inverse de d tel que pour $n_0 = 1, \forall x \geqslant n_0 \ g(x) = c(d.g(x))$

Donc $O(d.g) = O(g)$

Exemple

$$O(10\,000\,n^2) = O(0{,}001\,n^2) = O(n^2) \text{ .}$$

Propriété 4

Soient f et g deux fonctions de $\mathbb{R}^+$ dans $\mathbb{R}^+$. L'addition est effectuée en prenant la valeur maximale :

$$O(g) + O(h) = O(g+h) = O(max(g, h))$$

Démonstration
Par définition

$$O(g)=\{f/\exists c>0, \exists n_0\in\mathbb{N} \text{ tels que } \forall n\geqslant n_0, f(n)\leqslant c\, g(n)\}$$

et

$$O(h)=\{r/\exists d>0, \exists m_0\in\mathbb{N} \text{ tels que } \forall m\geqslant m_0, r(m)\leqslant d\, h(m)\}$$

1) Montrons l'égalité $O(g)+O(h)=O(g+h)$
a) Montrons d'abord que $O(g)+O(h)\subseteq O(g+h)$
Soient $f\in O(g)$ *et* $r\in O(h)$ *. Nous avons par définition :*

$$\exists c>0, \exists n_0\in\mathbb{N} \text{ tels que } \forall n\geqslant n_0, f(n)\leqslant c.g(n)$$

et

$$\exists d>0, \exists m_0\in\mathbb{N} \text{ tels que } \forall n\geqslant m_0, r(n)\leqslant d.h(n)$$

Nous avons donc (par « addition ») :

$$\exists c>0, \exists d>0, \exists n_0 et\, m_0\in\mathbb{N} \text{ tels que } \forall n\geqslant max(n_0, m_0), f(n)+r(n)\leqslant c.g(n)+d.h(n)$$

Posons $N_0=max(n_0, m_0)$ *et vue la nature de* c *et* d *, nous pouvons supposer que* $c>d$ *.*

$$\exists c>0, \exists d>0, \exists N_0\in\mathbb{N} \text{ tels que } \forall n\geqslant N_0, f(n)+r(n)\leqslant c(g(n)+\frac{d}{c}.h(n))$$

Or $\frac{d}{c}<1$ *d'où*

$$\exists c>0, \exists d>0, \exists N_0\in\mathbb{N} \text{ tels que } \forall n\geqslant N_0, f(n)+r(n)\leqslant c(g(n)+\frac{d}{c}.h(n))\leqslant c(g(n)+h(n))$$

Comme

$$O(g+h)=\{f/\exists c>0, \exists n_0\in\mathbb{N} \text{ tels que } \forall n\geqslant n_0, f(n)\leqslant c(g(n)+h(n))\}$$

Nous avons bien

$$f+r\in O(g+h)$$

Ainsi $O(g)+O(h)\subset O(g+h)$

b) Montrons maintenant que $O(g+h)\subseteq O(g)+O(h)$
Soit $f\in O(g+h)$ *.*

$$\exists c>0, \exists n_0\in\mathbb{N} \text{ tels que } \forall n\geqslant n_0, f(n)\leqslant c(g(n)+h(n))$$

Ce qui signifie :

$$\exists c>0, \exists n_0\in\mathbb{N} \text{ tels que } \forall n\geqslant n_0, f(n)\leqslant c\, g(n)+c\, h(n)$$

$f\in O(g)+O(h)$ *donc* $O(g+h)\subset O(g)+O(h)$

D'où

$$O(g)+O(h)=O(g+h)$$

2) Montrons que $O(g+h)=O(max(g,h))$
a) Montrons d'abord que $O(g+h)\subset O(max(g,h))$
Soit $f\in O(g+h)$
Nous avons par définition :

$$\exists c>0, \exists n_0\in\mathbb{N} \text{ tels que } \forall n\geqslant n_0, f(n)\leqslant c(g(n)+h(n))$$

Supposons que $h\in O(g)$ *.*
Donc

$$\exists d>0, \exists m_0\in\mathbb{N} \text{ tels que } \forall n\geqslant m_0, h(n)\leqslant d\, g(n)$$

$$\exists d>0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, \frac{h(n)}{g(n)} \leqslant d$$

Posons $N_0 = max(n_0, m_0)$ *.*

$$\exists c>0, \exists N_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant c(g(n)+h(n)) = c.g(n)(1+\frac{h(n)}{g(n)}) \leqslant c.g(n)(1+d)$$

D'où en posant $C = c.(1+d)$ *,*

$$f \in O(g)$$

Donc

$$O(g+h) \subset O(max(g,h))$$

b) Montrons maintenant que $O(max(g,h)) \subset O(g+h)$
Supposons que sur un intervalle : $g = max(g,h)$
Donc

$$O(max(g,h)) = O(g) \subset O(g+h)$$

D'où

$$O(g+h) = O(max(g,h)) .$$

Propriété 5

Soient f et g deux fonctions de $\mathbb{R}^+$ dans $\mathbb{R}^+$. Nous notons $O(g).O(h)$ l'ensemble $\{f.r / f \in O(g), r \in O(h)\}$.

Nous avons alors :

$$O(g).O(h) = O(gh)$$

Démonstration
Par définition

$$O(g) = \{f / \exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant c\, g(n)\}$$

et

$$O(h) = \{r / \exists d>0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall m \geqslant m_0, r(m) \leqslant d\, h(m)\}$$

1) Montrons d'abord l'inclusion $O(g).O(h) \subseteq O(gh)$
Soient $f \in O(g)$ *et* $r \in O(h)$ *. Nous avons par définition :*

$$\exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant c.g(n)$$

et

$$\exists d>0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant m_0, r(n) \leqslant d.h(n)$$

Par la jonction $f.r$ *élément de* $O(g).O(h)$ *, nous avons :*

$$\exists c>0, \exists d>0, \exists n_0 \, et \, m_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant max(n_0, m_0), f(n).r(n) \leqslant c.d.g(n).h(n)$$

Posons : $C = c.d$ *et* $N_0 = max(n_0, m_0)$ *.*

$$\exists C>0, , \exists N_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant N_0, f(n).r(n) \leqslant C.g(n).h(n)$$

D'où

$$f.r \in O(gh)$$

Par conséquent

$$O(g).O(h) \subseteq O(gh)$$

2) Montrons maintenant que $O(gh) \subseteq O(g).O(h)$
Par définition

$$O(gh) = \{f / \exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant c.g(n).h(n)\}$$

Soit $f \in O(gh)$ *. Nous avons par définition :*

$$\exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant c.g(n).h(n)$$

Ce qui se traduit par :

$$\exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant (\sqrt{c}.g(n))(\sqrt{c}.h(n))$$

Ou encore par :

$$f \in O(g).O(h)$$

Ce qui montre l'inclusion cherchée :

$$O(gh) \subset O(g).O(h)$$

Conclusion

$$O(g).O(h) = O(gh)$$

Pour retenir : « Le produit de grands O est un grand O du produit ».

V.2.4 – Relation d'inclusion pour les classes de fonctions usuelles

Propriété 6

Soient trois fonctions f, g et h de $\mathbb{R}^+$ dans $\mathbb{R}^+$ telles que $f \in O(g)$ et $g \in O(h)$, alors $f \in O(h)$.

Démonstration

Soit $f \in O(g)$*. Nous avons donc :*

$$\exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, f(x) \leqslant c\, g(x)$$

Comme $g \in O(h)$*, il vient :*

$$\exists d>0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant m_0, g(x) \leqslant d\, h(x)$$

Posons : $N_0 = max(n_0, m_0)$*.*

$$\exists c>0 \; et \; d>0, \exists N_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant N_0, f(x) \leqslant c\, g(x) \leqslant c.d.h(x)$$

Posons : $e = c.d$

$$\exists e>0, \exists N_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant N_0, f(x) \leqslant e\, h(x)$$

D'où :

$$f \in O(h)$$

Propriété 7

Soient deux fonctions f et g de $\mathbb{R}^+$ dans $\mathbb{R}^+$ telles que $f \in O(g)$ alors $O(f) \subset O(g)$.

Démonstration

Soit $f \in O(g)$*. Nous avons donc :*

$$\exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, f(x) \leqslant c\, g(x)$$

Soit $h \in O(f)$*. Nous obtenons donc :*

$$\exists d>0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant m_0, h(x) \leqslant d\, f(x)$$

Posons $e = max(c, d)$ *et* $P = max(n_0, m_0)$*.*

En combinant les 2 inégalités, nous obtenons :

$$\exists e>0, \exists P \in \mathbb{N} \text{ tels que } \forall x \geqslant P, h(x) \leqslant d\, f(x) \leqslant d.(c.g(x))$$

D'où :

$$\exists e>0, \exists P \in \mathbb{N} \text{ tels que } \forall x \geqslant P, h(x) \leqslant (d.c).g(x)$$

Donc :

$$h \in O(g)$$

Et

$$O(f) \subset O(g)$$

Théorème 2

Soit $n \in \mathbb{N}$. Nous avons :
$O(1) \subset O(\log(n)) \subset O(\sqrt{n}) \subset O(n) \subset O(n\log(n)) \subset O(n^2) \subset O(n^3) \subset \cdots \subset O(n^{10}) \cdots \subset O(2^n) \subset O(n!)$

Démonstration
Pour cela nous nous basons sur la comparaison de fonctions[31] *:*
$\lim_{x \to +\infty} \frac{1}{\log(x)} = 0$, $\lim_{x \to +\infty} \frac{\log(x)}{\sqrt{x}} = 0$, $\lim_{x \to +\infty} \frac{\sqrt{x}}{x} = 0$, …
et le théorème 1 page 23.

Montrons que $O(1) \subset O(\log(n))$
Nous savons que :

$$\lim_{x \to +\infty} \frac{1}{\log(x)} = 0$$

D'où par le théorème 1 : la fonction constante égale à 1 appartient à $O(\log(n))$
Et par la propriété 7, nous obtenons :

$$O(1) \subset O(\log(n))$$

Même principe pour les autres inclusions.

V.3 – Notation o (petit o)

V.3.1 – Définition

Remarque

La notation o est utilisée pour indiquer que la borne supérieure n'est pas asymptotiquement approchée.

Définition 22

Soit g une fonction définie de $\mathbb{R}^+$ dans $\mathbb{R}^+$. L'ensemble $o(g)$ est défini ainsi :

$$o(g) = \{ f \text{ fonction défine de } \mathbb{R}^+ \text{ dans } \mathbb{R}^+ / \forall c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, 0 \leqslant f(x) < c\, g(x) \}$$

Remarque

Nous disons pour $f \in o(g)$ que f est un « petit o » de g et que f est négligeable devant g.

V.3.2 – Propriétés

Propriété 8

Soient f et g des fonctions définies de $\mathbb{R}^+$ dans $\mathbb{R}^+$. Nous supposerons que g ne s'annule pas.

La relation $f \in o(g)$ implique que $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = 0$ si la limite existe.

---

31 cf. par exemple http://www.bibmath.net/formulaire/index.php?action=affiche&quoi=limite2 (lien valide le 10/11/2013)

Démonstration
Soit $f \in o(g)$.
Donc

$$\forall c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, 0 \leqslant f(x) < c\, g(x)$$

$$\forall c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, 0 \leqslant \frac{f(x)}{g(x)} < c$$

$$\forall c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, 0 \leqslant \frac{f(x)}{g(x)} - 0 < c$$

D'où $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = 0$

Propriété 9

Soit f une fonction définie de $\mathbb{R}^+$ dans $\mathbb{R}^+$. Nous avons :

$$o(f) + o(f) = o(f)$$

Démonstration
Soient $g \in o(f)$ et $h \in o(f)$.
Nous avons par définition :

$$\forall c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant n_0, 0 \leqslant g(x) < c\, f(x)$$

$$\forall c > 0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall x \geqslant m_0, 0 \leqslant h(x) < c\, f(x)$$

Posons

$$p_0 = Max(n_{0,} m_0) .$$

$$\forall d > 0, \text{ prenons } c = \frac{1}{2} d .$$

$$\forall d > 0, \exists p_0 \text{ tels que } \forall x \geqslant p_{0,} 0 \leqslant g(x) + h(x) < c\, f(x) + c\, f(x)$$

$$\forall d > 0, \exists p_0 \text{ tels que } \forall x \geqslant p_{0,} 0 \leqslant g(x) + h(x) < \frac{1}{2} d\, f(x) + \frac{1}{2} d\, f(x)$$

D'où

$$\forall d > 0, \exists p_0 \text{ tels que } \forall x \geqslant p_{0,} 0 \leqslant g(x) + h(x) < d\, f(x)$$

Exemple

$\forall n \in \mathbb{N}, x^n \in o(x^{n+1})$

Démonstration
Soit $n \in \mathbb{N}$ *et* $c > 0$.
$\forall x \geqslant 0, \quad x^n \geqslant 0$
Démontrons qu'il existe une constante $m_0 \in \mathbb{N}$ *telle que* $\forall x \geqslant m_0, x^n < c\, x^{n+1}$

$$x^n < c.x^{n+1} \Leftrightarrow 1 < c.\frac{x^{n+1}}{x^n} \Leftrightarrow 1 < c.x \Leftrightarrow x > \frac{1}{c}$$

Il suffit de prendre pour m_0 *l'entier juste supérieur à* $\frac{1}{c}$.

V.4 – Notation Ω (grand oméga)

La notation Ω est utilisée pour indiquer la borne inférieure asymptotique.

Définition 23

Soit g une fonction définie sur une partie des nombres réels. L'ensemble $\Omega(g)$ est défini ainsi :

$$\Omega(g)=\{f \text{ fonction défine sur une partie de } \mathbb{R} / \exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c\, g(n) \leqslant f(n)\}$$

Propriété 10

Soient f et g des fonctions définies sur une partie des nombres réels. Une autre manière de définir $\Omega(g)$ est :

$f \in \Omega(g)$ si et seulement si $g \in O(f)$.

Démonstration

Soit $f \in \Omega(g)$.

Donc

$$\exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c\, g(n) \leqslant f(n)$$

$$\exists c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant g(n) \leqslant \frac{1}{c} f(n)$$

Posons $C=\frac{1}{c}$.

$$\exists C>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant g(n) \leqslant C\, f(n)$$

Donc $g \in O(f)$

V.5 – Notation ω (oméga)

La notation ω est utilisée pour indiquer que la borne inférieure n'est pas asymptotiquement approchée.

Définition 24

Soit g une fonction définie sur une partie des nombres réels. L'ensemble $\omega(g)$ est défini ainsi :

$$\omega(g)=\{f \text{ fonction définie sur une partie de } \mathbb{R} / \forall c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c\, g(n) < f(n)\}$$

Propriété 11

Soient f et g des fonctions définies sur une partie des nombres réels. Une autre manière de définir $\omega(g)$ est :

$f(n) \in \omega(g(n))$ si et seulement si $g(n) \in o(f(n))$

Démonstration

Soit $f(n) \in \omega(g(n))$

$$\forall c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c\, g(n) < f(n)$$

$$\forall c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant g(n) < \frac{1}{c} f(n)$$

Posons $C=\frac{1}{c}$.

$$\forall C>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant g(n) < C\, f(n)$$

D'où $g(n) \in o(f(n))$.

Propriété 12

Soient f et g des fonctions définies sur une partie des nombres réels. La relation $f \in \omega(g)$ implique que $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = +\infty$ si la limite existe.

Démonstration

Soit $f \in \omega(g)$.

Donc

$$\forall c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c\, g(n) < f(n)$$

$$\forall c>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, c < \frac{f(n)}{g(n)}$$

D'où $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = +\infty$.

V.6 – Notation Θ (thêta)

Définition 25

Soit g une fonction définie sur une partie des nombres réels. L'ensemble $\Theta(g)$ est défini ainsi :

$$\Theta(g) = \{ f \text{ fonction définie sur une partie de } \mathbb{R} / \exists c_1 > 0 \text{ et } c_2 > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, \\ 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \}$$

Propriété 13

Soient f et g deux fonctions définies sur une partie des nombres réels.

$f \in \Theta(g)$ si et seulement si $g \in \Theta(f)$.

Démonstration

Soit $f(n) \in \Theta(g(n))$

Donc

$$\exists c_1>0 \text{ et } c_2>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

D'où $g(n) \leqslant \frac{1}{c_1} f(n)$ *et* $0 \leqslant \frac{1}{c_2} f(n) \leqslant g(n)$

Posons $C_1 = \frac{1}{c_1}$ *et* $C_2 = \frac{1}{c_2}$

$$\exists C_1>0 \text{ et } C_2>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant C_2 f(n) \leqslant g(n) \leqslant C_1 f(n)$$

Théorème 3

Pour deux fonctions quelconques f et g définies sur une partie des nombres réels :

$f \in \Theta(g)$ si et seulement si $f \in O(g)$ et $f \in \Omega(g)$.

Démonstration

1) Montrons d'abord que $f \in \Theta(g)$ *entraîne que* $f \in O(g)$ *et* $f \in \Omega(g)$.

Posons $f \in \Theta(g)$

$$\exists c_1>0 \text{ et } c_2>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

$$\exists c_1>0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c_1 g(n) \leqslant f(n)$$

Donc $f \in \Omega(g)$

$$\exists c_2 > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant f(n) \leqslant c_2 g(n)$$

Donc $f \in O(g)$

2) Montrons maintenant que $f \in O(g)$ et $f \in \Omega(g)$ entraîne que $f \in \Theta(g)$.
Soit $f \in O(g)$ *et* $f \in \Omega(g)$

$$\exists c_2 > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant f(n) \leqslant c_2 g(n)$$

$$\exists c_1 > 0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant m_0, 0 \leqslant c_1 g(n) \leqslant f(n)$$

Posons : $N_0 = max(n_0, m_0)$

$$\exists c_1 > 0 \, et \, c_2 > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant N_0, 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

Donc $f \in \Theta(g)$.

Propriété 14

Soient f et g deux fonctions définies sur une partie des nombres réels et $b \in \mathbb{R}^*$.

Si $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = b$ et $b > 0$ alors $f \in \Theta(g)$.

Démonstration

Nous savons que : si $\lim_{x \to +\infty} \frac{f(x)}{g(x)} = b$ *et* $b > 0$ *alors* $f \in O(g)$ et $g \in O(f)$.
Or $g \in O(f)$ *est équivalent à* $f \in \Omega(g)$
Donc $f \in O(g)$ *et* $f \in \Omega(g)$.
D'où par le théorème 3 : $f \in \Theta(g)$.

Propriété 15

La relation Θ est une relation d'équivalence.

Démonstration

1) Réflexive

$$\exists c_1 = 1 > 0 \, et \, c_2 = 1 > 0, \exists n_0 = 1 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c_1 f(n) \leqslant f(n) \leqslant c_2 f(n)$$

Donc $f \in \Theta(f)$.

2) Symétrique
Démontrée dans la propriété 13.

3) Transitive
Soient $f \in \Theta(g)$ *et* $g \in \Theta(h)$.
Donc

$$\exists c_1 > 0 \, et \, c_2 > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

et

$$\exists d_1 > 0 \, et \, d_2 > 0, \exists m_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant m_0, 0 \leqslant d_1 h(n) \leqslant g(n) \leqslant d_2 h(n)$$

Posons : $N_0 = max(n_0, m_0)$.

D'où : $\exists c_1 > 0, c_2 > 0, d_1 > 0, d_2 > 0,$

$$\exists n_0 \in \mathbb{N} \, tels \, que \, \forall n \geqslant N_0, 0 \leqslant d_1 c_1 h(n) \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \leqslant c_2 d_2 h(n)$$

En posant, $C_1 = c_1 . d_1, C_2 = c_2 . d_2$
nous obtenons :

$$\exists n_0 \in \mathbb{N} \text{ tel que } \forall n \geqslant N_0, 0 \leqslant C_1 h(n) \leqslant f(n) \leqslant C_2 h(n)$$

Donc $f \in \Theta(h)$.

Définition 26

Soient f et g deux fonctions définies sur une partie des nombres réels. Nous définissons la relation ≤ sur l'ensemble des ordres de grandeur des fonctions à l'infini par :

$$\Theta(f) \leqslant \Theta(g) \text{ si } f \in O(g)$$

Propriété 16

Cette relation ≤ est une relation d'ordre non totale.

Rappel

Une relation d'ordre est :

- réflexive,
- antisymétrique,
- transitive.

Une relation d'ordre R sur E est totale si tout élément de E est comparable avec tout autre élément de E.

Démonstration

1) Réflexive

Par définition

$$O(f) = \{h / \exists c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, h(n) \leqslant c\, f(n)\}$$

Posons $c=1$ *et* $n_0=1$ *.*

$$\exists c=1>0, \exists n_0=1 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, f(n) \leqslant 1 \times f(n)$$

D'où

$$f \in O(f)$$

Donc $\Theta(f) \leqslant \Theta(f)$

2) Antisymétrique

Soient f *et* g *telles que* $\Theta(f) \leqslant \Theta(g)$ *et* $\Theta(g) \leqslant \Theta(f)$

$$\Theta(f) \leqslant \Theta(g)$$

Donc

$$f \in O(g)$$

Et

$$\Theta(g) \leqslant \Theta(f)$$

Donc

$$g \in O(f)$$

qui équivaut à

$$f \in \Omega(g)$$

Par le théorème ci-dessus $f \in \Theta(g)$ *.*

ce qui est la définition de : $\Theta(f) = \Theta(g)$

3) Transitive

Soient f*,* g *et* h *telles que* $\Theta(f) \leqslant \Theta(g)$ *et* $\Theta(g) \leqslant \Theta(h)$

$$\Theta(f) \leqslant \Theta(g)$$

Donc

$$f \in O(g)$$

Et

$$\Theta(g) \leqslant \Theta(h)$$

Donc

$$g \in O(h)$$

Par la propriété 6 du V.2.4, $f \in O(h)$

L'ordre est partiel à cause de la remarque du V.8.

Propriété 17

Soient f et g deux fonctions définies sur une partie des nombres réels. Nous avons alors :

$$\Theta(f)=\Theta(g) \text{ si } f \in \Theta(g) \text{ et } \Theta(f) < \Theta(g) \text{ si } f \in O(g) \text{ et } g \notin O(f)$$

Démonstration

Égalité

Soit f *tel que* $f \in \Theta(g)$

$f \in \Theta(g)$ *donc* $f \in O(g)$ *donc* $\Theta(f) \leqslant \Theta(g)$

$f \in \Theta(g)$ *donc* $g \in \Theta(f)$ *donc* $g \in O(f)$ *donc* $\Theta(g) \leqslant \Theta(f)$

D'où : $\Theta(f)=\Theta(g)$

Inégalité

Soient f *et* g *telles que* $f \in O(g)$ *et* $g \notin O(f)$

$$f \in O(g)$$

Donc

$$\Theta(f) \leqslant \Theta(g)$$

Nous savons que

si $g \in \Theta(f)$ *donc* $g \in O(f)$ *.*

est équivalent à

$g \notin O(f)$ *donc* $g \notin \Theta(f)$

D'où

$\Theta(f) \leqslant \Theta(g)$ *et* $g \notin \Theta(f)$

Donc $\Theta(f) < \Theta(g)$ *.*

Propriété 18

Nous avons :

$$\Theta(1) < \Theta(\log(n)) < \Theta(\sqrt{n}) < \Theta(n) < \Theta(n\log(n)) < \Theta(n^2) < \Theta(n^3) < \cdots < \Theta(n^{10}) \cdots < \Theta(2^n) < \Theta(n!)$$

Démonstration

Par le théorème du V.2.4 et la définition de l'inégalité stricte.

Propriété 19

Soient f et g deux fonctions définies sur une partie des nombres réels.

$$\text{Si } \lim_{x \to +\infty} \frac{f(x)}{g(x)} = 0 \text{ alors } \Theta(f) < \Theta(g) .$$

Démonstration

Nous savons que si

$$\lim_{x \to +\infty} \frac{f(x)}{g(x)} = 0$$

Alors

$$f(x) \in O(g(x)) \text{ et } g(x) \notin O(f(x)) .$$

D'où par définition de l'inégalité $\Theta(f) < \Theta(g)$

V.7 – Notation ~

Définition 27

Soient f et g deux fonctions définies sur une partie des nombres réels.
f et g sont **équivalentes** si $f(x)=g(x)(1+\varepsilon(x))$ avec $\lim\limits_{x \to +\infty} \varepsilon(x)=0$.

La notation est : $f \sim g$

V.8 – Récapitulatif des comparaisons de fonctions

Grâce à ces différentes notions, nous pouvons effectuer des comparaisons de fonctions à l'infini, autrement dit avoir le comportement asymptotique des fonctions, comme le liste B. Mariou :

a) f ne croît pas plus vite que g : $f(n)=O(g(n))$;
b) f croît plus lentement que g : $f(n)=o(g(n))$;
c) f ne croît pas plus lentement que g : $f(n)=\Omega(g(n))$;
d) f croît plus vite que g : $f(n)=\omega(g(n))$;
e) f et g ont des croissances comparables : $f(n)=\Theta(g(n))$;
f) f et g sont asymptotiquement identiques : $f \sim g$.

Remarque

Mais attention, deux fonctions ne sont pas nécessairement comparables asymptotiquement (c'est-à-dire que nous n'avons pas toujours $f=O(g)$ ou $f \in \Omega(g)$).

Par exemple n et $n^{1+\sin n}$ ne peuvent pas être comparées asymptotiquement car $1+\sin n$ varie entre 0 et 2.

Propriété 20

Soient f et g deux fonctions. Voici une synthèse du calcul de $\lim\limits_{x \to +\infty} \frac{f(x)}{g(x)}$.

$\lim\limits_{x \to +\infty} \frac{f(x)}{g(x)}=0$ implique $f \in O(g) et g \notin O(f)$ et $\Theta(f)<\Theta(g)$.

$\lim\limits_{x \to +\infty} \frac{f(x)}{g(x)}=1$ implique $f \sim g$.

$\lim\limits_{x \to +\infty} \frac{f(x)}{g(x)}=c \neq 0$ implique $f \in O(g) et g \in O(f)$ et $f \in \Theta(g)$, $g \in \Theta(f)$.

$\lim\limits_{x \to +\infty} \frac{f(x)}{g(x)}=\infty$ implique $f \notin O(g) et g \in O(f)$ et $\Theta(f)>\Theta(g)$.

VI – Conclusion

Ce document a permis, je l'espère, au lecteur d'accéder aux concepts de complexité, à la majorité des démonstrations (s'il le désirait) et aux différentes notions mathématiques nécessaires.

Il a ainsi pu voir que cette notion de complexité allait bien au-delà de l'étude d'un petit algorithme. En effet, elle entraîne des questions $P=NP$? qui si elles sont un jour résolues, pourraient modifier quelque peu notre quotidien.

Table des illustrations et tableaux

Index des illustrations

Illustration 1 : graphique représentant les différentes courbes des fonctions utilisées dans les complexités usuelles dans l'intervalle [0;20]....................15
Illustration 2 : hiérarchie d'ensembles de classes de problèmes....................18

Index des tableaux

Tableau 1 : opérations fondamentales en fonction des problèmes....................6
Tableau 2 : classification des algorithmes....................13
Tableau 3 : comparaison de complexité des opérations élémentaires en fonction de la structure choisie....................14
Tableau 4 : exemples de problèmes en fonction de la complexité....................14
Tableau 5 : nombre d'opérations nécessaires pour exécuter un algorithme en fonction de sa complexité et de la taille des entrées du problème traité....................15
Tableau 6 : temps nécessaire pour exécuter un algorithme en fonction de sa complexité et de la taille des entrées du problème traité en supposant que nous disposons d'un ordinateur capable de traiter 109 opérations par seconde....................15

Références

Ouvrages

J.-Ch. Arnulfo, Les optimisations et Kit de survie dans *Métier Développeur kit de survie*, chapitres 5.10 et 5.11, série 01 Informatique, Édition DUNOD, 2003

Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Introduction à l'algorithmique 2ème édition*, chapitres 2, 3, 34, série Sciences Sup, Édition DUNOD, 2002

R. Faure, B. Lemaire, Ch. Picouleau, Notions de complexité dans *Précis de recherche opérationnelle 5ème édition*, chapitre 2, série Sciences Sup, Édition DUNOD, 2004

A. Martelli, *Python en concentré Manuel de référence,* O'REILLY, 2004

Thèses

C. Bentz, partie 1.1.1 dans *Résolution exacte et approchée de problèmes de multiflot entier et de multicoupe : algorithmes et complexité*, Thèse de Doctorat, Conservatoire National des Arts et Métiers, 20 novembre 2006, http://www.lri.fr/~bentz/Publis.php lien « Ma thèse » (lien valide le 10/11/2013)

B. Darties, partie Complexité algorithmique dans *Problèmes algorithmiques et de complexité dans les réseaux sans fils*, chapitre 1.3, Thèse de Doctorat, Université Montpellier II, 14 décembre 2007, http://tel.archives-ouvertes.fr/index.php?

halsid=7erdbnlbi0v9766s0vaa9ifqs2&view_this_doc=tel-00270118&version=1 (lien valide le 10/11/2013)

Articles

J.-P. Delahaye, Un algorithme à un million de dollars, *Pour la Science* n° 334, pages 90-95, août 2005

L. Stockmeyer, A. Chandra, Les problèmes intrinsèquement difficiles, dans *Le Progrès des Mathématiques*, Bibliothèque Pour la Science, Pour la Science, Belin, pages 128-137, 1978

Harry R. Lewis, Ch. H. Papadimitriou, L'efficacité des algorithmes, dans *Le Progrès des Mathématiques*, Bibliothèque Pour la Science, Pour la Science, Belin, pages 114-127, 1978

Internet

a) R. Cori, J.-J. Lévy, Complexité des algorithmes dans *Algorithmes et Programmation*, École Polytechnique http://pauillac.inria.fr/~levy//x/tc/polycopie-1.6/index.html (lien valide le 04/01/2015)

b) F. Grimbert, *Algorithmes et Complexité*, http://www-sop.inria.fr/members/Francois.Grimbert/docs/Maplepdf/TDmaple06-03-02.pdf (lien valide le 04/01/2015)

c) L. Brun, *Algorithmique … Complexité*, http://www.greyc.ensicaen.fr/ensicaen/CoursEnLigne/AlgoComplexite.pdf (lien valide le 9/04/2012)

d) T. Fdil, *Algorithmique et langage C Chapitre 1 : Complexité des algorithmes*, http://www.sagma.ma/algo/al_complexite.html (lien valide en mai 2009)

e) Julia, *Complexité des algorithmes*, http://deptinfo.unice.fr/~julia/IT/0405/09it.pdf (lien valide le 04/01/2015)

f) B. Mariou, *Comparaison de fonctions, ordre de grandeur*, http://ufr6.univ-paris8.fr/lit-math/maths/BM/DOX/LCA/compfct.pdf, automne 2006, (lien valide le 9/04/2012)

g) S. Peyronnet, Algorithmique, cours 1, partie introduction, http://sylvain.berbiqui.org/ALGO-T1.pdf (lien valide le 04/01/2015)

h) N. Revol, *Algorithmes et complexité, illustration en arithmétique des ordinateurs*, http://perso.ens-lyon.fr/nathalie.revol/talks/IREM-APMEP.pdf (lien valide le 04/01/2015)

i) G. Savard, *Complexité des algorithmes et notation grand O*, *https://cours.etsmtl.ca/SEG/GSavard/mat210/Documents/grandO.pdf,* (lien valide le 04/01/2015)

j) Tchno-Science.net, *Théorie de la complexité*, http://www.techno-science.net/?onglet=glossaire&definition=6231 (lien valide le 04/01/2015)

k) Wikipédia, *Asymptote*, http://fr.wikipedia.org/wiki/Asymptote (lien valide le 04/01/2015)

l) Wikipédia, *Comparaison asymptotique*, http://fr.wikipedia.org/wiki/Comparaison_asymptotique (lien valide le 04/01/2015)

m) Wikipédia, Machine de Turing non déterministe, http://fr.wikipedia.org/wiki/Machine_de_Turing_non_déterministe (lien valide le 04/01/2015)
n) Wikipédia, *Théorie de la complexité*, http://fr.wikipedia.org/wiki/Théorie_de_la_complexité_(informatique_théorique) (lien valide le 04/01/2015)
o) *Complexité algorithmique*, http://www.info.univ-angers.fr/pub/bd/StrDonnees/complexite.pdf (lien valide le 9/04/2012)
p) Duret-Lutz Alexandre, *Algorithmique*, 2009 http://www.montefiore.ulg.ac.be/~dumont/pdf/ac8.pdf (lien valide le 04/01/2015)

Résumé

Ce mémoire a pour but d’expliquer la notion de complexité algorithmique et son utilisation dans la programmation. Il débute par des notions mathématiques nécessaires à l’étude et poursuit par des concepts de complexité, leurs propriétés démontrées et des exemples sur les coûts de différents types d’algorithmes. Il finit par un bref aperçu des différentes classes de problèmes et la difficulté pour en résoudre certains.

Mots clés

complexité, algorithme, coût, classe de problèmes

Abstract

The purpose of this manuscript is to explain the notion of algorithmic complexity and its use for the programming theory. It starts with the necessary mathematic basics useful for the understanding of the complexity concepts, and further develops their proved properties and examples on the cost of different algorithm types. Finally, a short insight in the various problem classes and the difficulty to solve some of them will be presented.

Keywords

complexity, algorithm, cost, class of problems