

LIFAPC: Algorithmique, Programmation et Complexité

Chaîne Raphaëlle (responsable semestre automne)
E-mail : raphaelle.chaine@liris.cnrs.fr
<http://liris.cnrs.fr/membres?idn=rchaine>

1

1

Conception et analyse des algorithmes

Décomposition de l'activité de programmation :

1. Exposé d'un problème
2. Modélisation et formalisation de ce problème (spécification)
3. Construction d'une solution algorithmique
4. Vérification justesse de l'algorithme (preuve)
5. Analyse complexité
6. Construction du programme
7. Exécution du programme pour résoudre le problème

92

92

1. Exposé du problème

- Exemple : tri d'une séquence

2. Modélisation et formalisation d'un problème :

- Étant donné un ensemble de valeurs ou l'état d'un système (entrée),
 - comment produire un ensemble de valeurs ou modifier l'état du système (sortie)
- avec relation désirée entre l'entrée et la sortie (peut faire appel à des prédicats et des fonctions d'accès)

- Exemple

- entrée : une séquence de n nombres $\{a_1, a_2, \dots, a_n\}$
- sortie : une permutation de la séquence $\{a'_1, a'_2, \dots, a'_n\}$
- telle que

$$a'_1 \leq a'_2 \leq \dots \leq a'_n$$

93

93

3. Algorithme :

- composition d'un nombre fini d'étapes dont chacune **devra** pouvoir être
 - définie de façon rigoureuse et non ambiguë
 - effectivement réalisable sur une machine
- ... **lors de la mise en œuvre dans un langage de programmation**
- Remarque
 - L'algorithme peut être exprimé dans un modèle abstrait du monde
 - ... qui se précisera au fur et à mesure que l'on affinera son degré de précision

94

94

- Exemple :

- Algorithme informel de tri :

```
tant que il existe  $i < j$  t. que  $a_i > a_j$  alors
    échanger  $a_i$  et  $a_j$ 
fin tantque
```

- Opérations élémentaires : comparaison et échange
- On peut d'ores et déjà prouver la justesse de cet algorithme, cependant, les différents raffinements de cet algorithme pourront conduire à des algorithmes d'efficacité potentiellement différentes

- Tri à bulle : échange d'éléments consécutifs
- Tri par sélection du minimum : échange du premier élément de la partie non triée avec son minimum
- Le tri par insertion peut également être vu comme une variante de cet algorithme de tri informel

95

95

4. Preuve :

- Un algorithme est dit **correct** si pour n'importe quelle entrée satisfaisant les spécifications il **s'arrête** avec la sortie correcte
- Un algorithme correct **résout** un problème de calcul

Il existe une théorie de la **calculabilité** qui caractérise les problèmes qui peuvent être résolus à l'aide d'un algorithme (MIF15)

- *Exemple* : le problème de l'arrêt
 - Il ne peut pas exister d'algorithme qui pour tout programme P et toute donnée D répond oui ou non à la question :
 - « P se termine-t-il pour D ? »

96

96

Démonstration:

- Supposons que vous réussissez à construire un algorithme **TestTermine** qui pour tout programme P et toute donnée répond **oui** ou **non** à la question : « P se termine-t-il pour D? »
- Exécutez **TestTermine** sur la procédure récursive suivante (qui contient elle-même un appel à **TestTermine**)

```
– Procédure Maligne()
  Debut
    si TestTermine(Maligne)
    alors Maligne
    fin si
  Fin
```

L'exécution de Maligne termine si et seulement si TestTermine répond en un temps fini qu'elle ne termine pas! Ce qui est contradictoire!

97

97

- Preuve de l'algorithme informel de tri

– Définitions

- Inversion dans une séquence a : couple (i,j) t. que $i < j$ et $a_i > a_j$
- Soit $\text{inv}(a)$ le nombre d'inversions de a

– Soit (i,j) une inversion : $i < j$ et $a_i > a_j$

- On décompose a en plusieurs parties :

$$a = \{sa_1, \mathbf{a_i}, sa_2, \mathbf{a_j}, sa_3\}$$

- Après permutation de a_i et a_j on obtient a'

$$a' = \{sa_1, \mathbf{a_j}, sa_2, \mathbf{a_i}, sa_3\}$$

- Montrons que $\text{inv}(a') < \text{inv}(a)$

98

98

– En effet :

- le nombre d'inversions de a_i (resp. a_j) avec sa_2 dans a' ne peut être que plus petit que le nombre d'inversions de a_j (resp. a_i) avec sa_2 dans a
- L'inversion entre a_i et a_j a disparu
- Le nombre d'inversions faisant intervenir a_j ou a_i avec un élément de sa_1 ou sa_3 ne change pas
- Le nombre d'inversions ne faisant intervenir ni a_i ni a_j ne change pas

– A chaque passage dans la boucle, $\text{inv}(a)$ décroît strictement

– La séquence est triée quand $\text{inv}(a)$ s'annule

99

99

5. Analyse de complexité

– Analyser un algorithme, c'est prévoir les ressources nécessaires à son exécution

- mémoire utilisée
- temps d'exécution
- largeur de bande d'une communication
- ...

– Comment mesurer l'efficacité d'un algorithme ?

- Mesure du temps d'exécution d'une mise en œuvre de l'algorithme?

NON car dépendance

- à la machine
- au langage de programmation
- au compilateur
- aux données

100

100

- On aimerait pouvoir dire :

– Sur toute machine et quel que soit le langage de programmation, l'algorithme A1 est meilleur que l'algorithme A2 pour des données « de grande taille »

➡ On compte le nombre d'opérations élémentaires de l'algorithme

Grande taille?

- Fibon(n: entier) : ici grande taille de la valeur de l'entier
- Addition (n1, n2 : entiers codés sur n bits)
ici grande taille du nombre de bits n
- Tri(n : entier, tab[1..n] de entiers)
ici grande taille du nombre n d'éléments à trier

101

101

- « de grande taille »?

– Exemples :

- Un algo qui s'exécute sur un ensemble de données en nombre de plus en plus grand
- Un algo qui s'exécute sur un nombre n stocké en mémoire en utilisant un nombre de bits de plus en plus grand (par exemple n de plus en plus grand ou de plus en plus précis)
- Un algo qui s'exécute sur un nombre qui implique un nombre d'instructions de plus en plus grand
-

102

102

- Complexité en temps :
 - $T(\text{algo})$: temps d'exécution de l'algorithme algo
 - $T(\text{algo}(d))$: temps d'exécution de algo appliqué aux données d
- Règles utilisées (architecture RAM)
 - $T(\text{opération élémentaire}) = \text{Constante}$
 - Séquence d'instructions
 $T(\{i_1, i_2\}) = T(i_1) + T(i_2)$ (somme)
 - Branchements conditionnels
 $T(\text{si } C \text{ alors } I_1 \text{ sinon } I_2) \leq T(C) + \max(T(I_1), T(I_2))$
 - Boucles
 $T(\text{pour } i \text{ de } i_1 \text{ à } i_2 \text{ faire } P(i)) = \sum_i T(P(i))$
 - Temps de calcul de procédures récursives
 - Solution d'équations de récurrence

103

103

- $T(\text{algo}(d))$ dépend en général de d et notamment de sa taille n :

- nombre d'éléments constituant d
- nombre de bits pour représenter d
- nombre de sommets et d'arêtes d'un graphe, etc.

- Complexité au pire

$$T_{MAX}(\text{algo}, n) = \max\{T(\text{algo}, d) : d \text{ de taille } n\}$$

- Complexité au mieux

$$T_{MIN}(\text{algo}, n) = \min\{T(\text{algo}, d) : d \text{ de taille } n\}$$

- Complexité en moyenne

$$T_{MOY}(\text{algo}, n) = \sum_{d \text{ de taille } n} p(d)T(\text{algo}, d)$$

Où $p(d)$ probabilité d'avoir la donnée d de taille n

104

104

Comportement asymptotique

- Généralement, on compare l'efficacité des algorithmes sur des données de taille importante
 - La manière dont évolue le temps d'exécution d'un algorithme est alors pertinente
 - Etude de l'efficacité **asymptotique** des algorithmes
- Les outils utilisés doivent permettre de comparer les profils de cette évolution
- Idée :
 - Indiquer que le profil asymptotique d'un temps d'exécution est meilleur, moins bon ou similaire à un profil type

105

105

Ordre de grandeurs asymptotiques

- Borne asymptotique **supérieure** : « grand O »
 $T(\text{algo}, n) = O(f(n))$ ssi
 il existe $C > 0$ et $N > 0$
 tels que pour tout $n > N$ $T(\text{algo}, n) \leq Cf(n)$
- Borne asymptotique **inférieure** : « grand Oméga »
 $T(\text{algo}, n) = \Omega(f(n))$ ssi
 il existe $C > 0$ et $N > 0$
 tels que pour tout $n > N$ $T(\text{algo}, n) \geq Cf(n)$
- Borne approchée asymptotique « grand théta »
 $T(\text{algo}, n) = \Theta(f(n))$ ssi
 il existe $C_1 > 0, C_2 > 0$ et $N > 0$
 tels que pour tout $n > N$ $C_1 f(n) \leq T(\text{algo}, n) \leq C_2 f(n)$
- Propriété

$$f = O(g) \text{ et } f = \Omega(g) \Rightarrow f = \Theta(g)$$

106

106

- Exemple :

A-t-on

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)?$$

cela revient à montrer que

$$\exists c_1, \exists c_2, \exists n_0 \text{ tels que } \forall n \geq n_0, \quad c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2, \text{ soit}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$$\text{pour } n \geq 1, \frac{1}{2} \geq \frac{1}{2} - \frac{3}{n}$$

$$\text{pour } n \geq 7, \frac{1}{2} - \frac{3}{n} \geq \frac{1}{14}$$

$$\text{donc } c_1 = \frac{1}{14}, c_2 = \frac{1}{2} \text{ et } n_0 = 7$$

107

107

- Exemple

$$f(n) = \begin{cases} n+1 & \text{si } n \text{ pair} \\ n^2 - n + 1 & \text{si } n \text{ impair} \end{cases}$$

$$f(n) = O(n^2), \text{ car pour } n \geq 2, n^2 \geq n^2 - n + 1 \geq n + 1$$

$$f(n) = \Omega(n), \text{ car pour } n \geq 1, n^2 - n + 1 \geq n \text{ et } n + 1 \geq n$$

108

108

- Un outil supplémentaire

$T(\text{algo}, n) = o(f(n))$ ssi

pour tout $\epsilon > 0$, il existe $N > 0$

tel que pour tout $n > N$ $T(\text{algo}, n) \leq \epsilon f(n)$

- Propriété

si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, alors $f = o(g)$

- Si $f = o(g)$ alors $f + g = \theta(g)$

109

109

- Utilisation des notations

$$\bullet 2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

$$\bullet 2n^2 + \Theta(n) = \Theta(n^2)$$

$$\bullet 2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$

110

110

Un peu de vocabulaire

Types de complexité

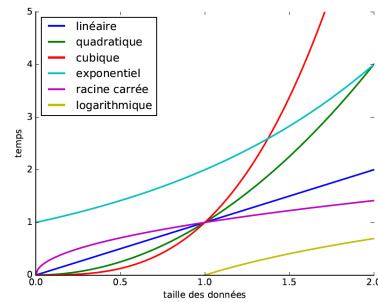
- $O(1)$ complexité constante (indépendante de la taille de la donnée)
- $O(\log(n))$ complexité logarithmique
- $O(n)$ complexité linéaire
- $O(n \log(n))$ complexité quasi-linéaire
- $O(n^2)$ complexité quadratique
- $O(n^3)$ complexité cubique
- $O(n^p)$ complexité polynomiale
- $O(n^{\log(n)})$ complexité quasi-polynomiale
- $O(2^n)$ complexité exponentielle
- $O(n!)$ complexité factorielle

111

111

Ordres de grandeur (exemples)

- $1, \log n, n, n \cdot \log n, n^2, n^3, 2^n$



112

112

Classification des algorithmes

- Il existe une classification grossière des algorithmes :

algorithmes $\begin{cases} \text{polynomiaux} & (\log n, n^{0.5}, n, n \log n, n^2, \dots) \\ \text{exponentiels} & (2^n, n^{\log n}, n!, n^n, \dots) \end{cases}$

- un bon algorithme est polynomial!

113

113

taille	20	50	100	200	500	1000
complexité						
$10^3 n$	0.02 s	0.05 s	0.1 s	0.2 s	0.5 s	1 s
$10^3 n \log n$	0.09 s	0.3 s	0.6 s	1.5 s	4.5 s	10 s
$100 n^2$	0.04 s	0.25 s	1 s	4 s	25 s	2 mn
$10 n^3$	0.02 s	1 s	10 s	1 mn	21 mn	27 h
$n^{\log n}$	0.4 s	1.1 h	220 jours	2500 ans	5.10^{10} ans	—
$2^{n/3}$	0.0001 s	0.1 s	2.7 h	$3 \cdot 10^6$ ans	—	—
2^n	1 s	36 ans	—	—	—	—
3^n	58 mn	$2 \cdot 10^{11}$ ans	—	—	—	—
$n!$	$77 \cdot 100$ ans	—	—	—	—	—

Opération : 1ms

114

114

Problèmes faciles et difficiles

- On va considérer deux types de problèmes :
 - Problème d'extraction**
 - on cherche dans un ensemble fini **S** un élément ou une configuration **s** des données qui vérifie une propriété donnée **Pr**
 - Problème d'existence (PExist)**
 - Savoir si un problème d'extraction admet ou non une solution est un problème d'existence

Attention! La taille de **S** peut être grande par rapport à la taille **n** des données en entrée.

- Exemple :
 - n** nombres en entrée, et **S** l'ensemble des sous ensembles possibles que l'on peut faire avec ces **n** nombres
 - La taille de **S** est exponentielle 2^n par rapport à **n**

115

115

Problèmes faciles et difficiles

- La réponse à un problème d'existence est Oui ou Non
 - définition 1** : les **PExist** qui admettent des algorithmes polynomiaux forment la **classe P**
 - définition 2** : la **classe NP** est celle des **PExist**
 - pour lesquels on ne sait pas forcément dire directement s'il existe un élément ou une configuration solution (satisfaisant **Pr**),
 - MAIS** qui admettent un algorithme polynomial capable de tester si un élément ou une configuration **s** satisfait ou non **Pr**.

116

116

- Remarque :
 - Une manière de résoudre les problèmes dans **NP** consiste donc à énumérer l'ensemble des éléments/configurations **s** de **S** et de tester s'ils satisfont **Pr** à l'aide de l'algorithme polynomial
 - Si le problème est dans **NP**, cela signifie que la taille de **S** n'est donc pas polynomiale par rapport à la taille **n** des entrées!
- Exemple :
 - Etant donné un ensemble **E** de **n** entiers et un entier **b**, existe-t-il un sous-ensemble **T** de **E** tel que la somme des éléments de **T** soit égale à **b** ?
 - S** est l'ensemble des sous-ensembles **T** de **E**

117

117

- Il n'existe pas d'algorithme polynomial connu à ce problème. Il est donc dans **NP**.
- En effet :
 - une solution de problème d'extraction peut être proposée sous forme d'un sous-ensemble **T** d'éléments de **E**
 - un algorithme (linéaire) de vérification est le suivant
 - la somme des éléments de **T** est elle égale à **b**?
- définition 3** : un problème **NP-complet** est un problème de **NP** en lequel se transforme polynomialement tout autre problème de **NP**
- on notera **NPC** l'ensemble des problèmes **NP-complets**

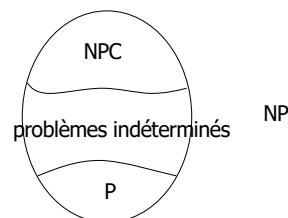
118

118

- Exemple : SAT (satisfaisabilité)
- Données :
 - n** variables booléennes x_i
 - m** clauses C_j (combinaisons de ET et de OU)
- Question :
 - Peut-on affecter à chaque variable x_i une valeur de façon à rendre vraies toutes les clauses ?
- Ce problème est **NP-complet** (Cook, 1970)

119

119



le problème de l'isomorphisme de 2 graphes est indéterminé

conjecture : $P \neq NP$

120

120

- Conclusion : il existe des problèmes qu'il ne faut pas s'acharner à résoudre de manière optimale ou de manière exacte
- Mais tout n'est pas perdu!
- Si on a un problème pour lequel on ne connaît pas d'algorithme polynomial, on peut essayer de résoudre le problème en tirant parti :
 - de la taille des données (petits cas, ...)
 - de la nature des données (cas particuliers, ...)
 - de la complexité moyenne (le pire cas est rare, ...)
 - de méthodes diminuant la combinatoire (méthodes arborescentes, heuristiques ...)
 - de méthodes approchées

121