Chapitre 1
Algorithmique et complexité

Programmation objet – Licence IMAE – 2005/2006

PARI54

K. Bertet

Définition d'un algorithme

- Algorithme: « spécification d'un schéma de calcul sous forme d'une suite d'opérations élémentaires » (*Encyclopédie universalis*)
- Algorithme en informatique: « séquence non ambigüe qui produit une solution à un problème donné dans un temps fini »
- Programme: description d'un algorithme dans un langage de programmation accepté par la machine

Algorithme	*Recette de cuisine*
Programme	*Plat cuisiné*
Programmeur	*Cuisinier*
Machine	*Cuisine*
Langage/Système	*Casseroles / Plaques de cuisson*

Métaphore de la recette de cuisine

Algorithme d'Euclide

•**Théorème d 'Euclide:**

$PGCD(A,B) = PGCD(B,R)$

avec $A<B$, $B \neq 0$ *et R reste de A/B*

$PGCD(A,0) = A$

•**Algorithme d'Euclide:** schéma de calcul du PGCD basé sur le Théorème d 'Euclide

Nom: **PGCD**
Données: deux entiers A et B avec A<B
Résultat: le PGCD de A et B

tant que $B \neq 0$ *faire*
  $R \leftarrow A$ *modulo* B
  $A \leftarrow B$
  $B \leftarrow R$
retourner A

Algorithmique

•1800 av J.C.: *notion d'algorithme*: formulation de règles pour résoudre certaines opérations. Ex: PGCD.

•9ème siècle: *mot algorithme* venant d 'un mathématicien perse (al-Khowa-rismi) qui généralise l'utilisation de règles

•1930: *concept d'algorithme* pour distinguer entre les problèmes calculables et les problèmes non calculables

•1945: première génération d'ordinateurs

Calculabilité

Problème calculable: il existe un algorithme pour le résoudre

⇒ on peut le programmer sur n'importe quelle machine, avec n'importe quel langage de programmation

Mais: un problème calculable ne peut pas toujours se calculer raisonnablement

Exemple du jeu d'échec:
- nombre fini et calculable de mouvements possibles après chaque coup
- nombre fini de coups dans une partie

⇒ il existe un algorithme pour calculer tous les coups possibles, et donc pour gagner (environ 10^{19} coups possibles)

Ecriture d 'un algorithme

données entrées, input paramètres → [nom / rôle] → résultat sortie output } Spécification, Signature

Contenu:
a- Instructions d'assignation : variables et opérateurs
b- Instructions conditionnelles ou itératives : si, tant que, *pour*
c- *Fonctions*

Ecriture d'un algorithme

•Variables: pour stocker des valeurs

•Opérateurs: permet de réaliser des opérations élémentaires sur des variables.
addition (+), multiplication (*), modulo (mod),
affectation (← ou =), comparaison (=, <, >, ...),

•Instruction simple: combinaison de variables et opérateurs.

⇒Tout algorithme peut s'écrire avec des instructions simples, et les instructions de contrôle *si* et *tant que* (machine de Turing)

Instructions de contrôle *pour*: meilleure lisibilité
Fonctions: meilleure modularité

Instructions conditionelles

Nom: **maximum1**
Données: deux entiers A et B
Résultat: le maximum de A et B

si A > B alors
max = A
sinon
max = B
retourner max

si <*condition*>
alors <*instructions*>
sinon <*instructions*>

Instructions de contrôle de flux

Nom: **multiplication**
Données: deux entiers A et B
Résultat: le produit de A et B
Rôle:calcule A*B par additions successives

mult=0
tant que A ≥ 1 faire
mult = mult + B
A = A - 1
retourner mult

tant que <*condition*>
faire <*instructions*>

Instructions de contrôle de flux

Nom: **maximum1**
Données: un ensemble S d'entiers
Résultat: le maximum des entiers de l 'ensemble

max = premier élément de S
pour x ∈ S faire
si x > max alors
max = x
retourner max

pour
<*var*>∈<*ensemble*>
faire <*instructions*>

Instructions de contrôle de flux

Nom: **maximum2**
Données: un tableau T de n entiers
Résultat: le maximum des n entiers du tableau

max = T[1]
pour i allant de 2 à n faire
si T[i] > max alors
max = T[i]
retourner max

pour <*var*>
allant de <*n*> à <*m*>
(par pas de <*entier*>)
faire <*instructions*>

Ecriture d'un algorithme

• Un algorithme *s'adresse à un programmeur*, et non à une machine:

⇒ Pas de règles syntaxiques, mais des règles de compréhension

⇒ Degré de description de l'algorithme dépendant du programmeur (débutant ou expert)

• On peut *« mesurer » l'efficacité* d'un algorithme en calculant sa complexité.

Complexité

•Complexité: fonction O estimant le temps de calcul:

→ en fonction de *n*, la valeur ou la taille de la donnée d'entrée

→ dans le *pire des cas*

→ indépendamment de la machine et du langage de programmation utilisée

•Elle permet de comparer des algorithmes, ou de vérifier si un algorithme est raisonnable

Complexité

•Complexité d'un algorithme: fonction d'estimation du temps de calcul indépendamment de son implémentation (machine + langage)

⇒ outil pour vérifier si un algorithme est « raisonnable »
⇒ outil pour comparer des algorithmes résolvant le même problème

•La complexité s 'obtient on considérant:
- la ou les données d 'entrée (taille)
- les boucles imbriquées
- la complexité des fonctions utilisées

•Définition: une fonction $f(n)$ du temps d'exécution d 'un algorithme est dite O$((n))$ s 'il existe des constantes c et no telles que, pour tout $n \geq no$:

$$f(n) \leq c\ g(n)$$

Complexité : exemple

Nom: **maximum**
Données: un tableau T de n entiers
Résultat: le maximum des n entiers du tableau

max = T[1]
pour i allant de 2 à n faire
| *si T[i] > max alors*
| | *max = T[i]*
retourner max

donnée d 'entrée de taille n

n exécutions de la boucle

O(n)

Complexité

O(1)	Constante	raisonnable
O(log n)	Logarithmique	raisonnable
O(n)	Linéaire	raisonnable
O(n^2)	Polynomiale	raisonnable
O(n^k)	Polynomiale	pas raisonnable pour k>4
O(2^n)	Exponentielle	pas raisonnable