BUT Informatique - R3.02
Conception d'algorithmes corrects et efficaces
Bases de la construction des algorithmes

J. Christian Attiogbé

Novembre 2023

Nantes Université

Contenu prévisionnel

Plan de la suite (adaptation selon contraintes)

Introduction

Cahier de charges
↓

Exigences → Spécification → Logiciel

Étapes de développement

Analyse - **Conception** - Développement+Test - Maintenance

Caractérisation d'un logiciel ou d'un système

Le fonctionnement d'un système : une succession d'états.
Le fonctionnement d'**un logiciel : une succession d'états**.

L'**état d'une entité** : perception à un instant précis dans le temps.
Une entité peut changer d'état d'un instant à un autre ; elle évolue (dans le temps).

Un algorithme, lors de son fonctionnement, évolue dans le temps en changeant d'états ; effet de ses instructions.
Une instruction peut donc modifier l'état d'un algorithme.

L'état d'un algorithme est perçu à travers les valeurs de ses variables.

Espace d'états d'un système

État

L'état d'un système est **sa perception à un instant donné**, à travers des valeurs ou des données qui le caractérisent.

Espace d'états

L'espace d'états d'un système est l'**ensemble des états** par lesquels il peut passer pendant son fonctionnement.

Propriétés (en Logique)

Une propriété : caractéristique que possède une entité (un objet, un humain, ...).
Une propriété : formellement exprimée avec le langage de la Logique (des prédicats, premier ordre, etc).

Une propriété est donc un **énoncé** qui peut être **vrai** ou **faux** : une proposition, **un prédicat**.

On a appris à **formaliser** des énoncés en **Logique** (du premier ordre) à l'aide de propositions et de prédicats.

Exemples de propriétés (en Logique)

- tout processus a un identifiant unique ;
- le processus est bloqué ;
- le tableau est blanc ;
- tout entier naturel a un successeur ;
- le ciel est jaune ;
- tous les oiseaux à bec plat ne sont pas des cigognes ;
- tout dossier doit avoir au moins deux référents ;
- les indices du tableau doivent être situés entre les valeurs i et s ;
- le canal est crypté ;
- les processus doivent s'exécuter en exclusion mutuelle ;
- le serveur doit supporter la charge de 1000 connexions/min.

Spécification : précondition, postcondition, invariant

Préconditions, postconditions

Une **précondition** est une propriété qui doit être vraie avant d'effectuer une action, afin qu'elle puisse se dérouler convenablement.

Exemples :

- **précondition** pour division d'un entier n par un autre entier m :
  m non nul
  car la division n'est pas définie sinon.
- **précondition** pour allumer une lampe, un appareil, une machine,
  qu'il y ait de l'énergie

Préconditions, postconditions

Une **postcondition** est une propriété vraie après le déroulement d'une action (ou d'une suite d'actions).

Les actions sont par exemple : des instructions d'un programme, le déroulement d'un algorithme, un traitement, ou leurs abstractions.

Exemples

- postcondition de la **suppression d'un élément d'une liste**
  la longueur de la liste diminue de 1
- postcondition de l'**affectation d'une expression à une variable**
  la variable a la valeur de l'expression

Des exemples préconditions - postconditions

$\{n \in \mathbb{N} \ \wedge \ m \in \mathbb{N} \ \wedge \ m \neq 0\}$ **q,r = n/m** $\{q \in \mathbb{N} \ \wedge \ r \in \mathbb{N} \ \wedge \ n = m * q + r\}$

$\{energy > 10\}$ **activate_wifi()** $\{ \ wifi = on \ \}$

$\{True\}$ **If (k > g) Then m ← k Else m ← g Fi** $\{m = max(\{k, g\})\}$

$\{False\}$ **pm ← ppcm(k,g)** $\{ \ \exists u.(u \in \mathbb{N}_1 \wedge pm = u \times k) \ \}$

Spécification : préconditions - postconditions

Une spécification d'un programme est la donnée de ses precondition et postcondition :

Spécification ≡ (precondition, postcondition)

On parle aussi de **contrat**, pour indiquer la spécification.

Propriétés invariantes

Une propriété invariante d'un système est **une propriété toujours vraie** pour le système pendant son fonctionnement.

Des exemples :
- Tout réel a toujours un réel supérieur.

$$\forall r.(r \in \mathbb{R} \Rightarrow (\exists s.(s \in \mathbb{R} \wedge s > r)))$$

- Tous les animaux respirent

$$\forall ani.(ani \in ANIMAL \quad \Rightarrow \quad respire(ani))$$

- Tout triangle a trois côtés

$$\forall f.(f \in FIGURE \wedge f \in TRIANGLE \quad \Rightarrow nbCotes(f) = 3)$$

Propriétés invariantes (exemples)

- Il n'y a pas de variables sans type

$$\forall va.(va \in VARIABLE \quad \Rightarrow \quad (\exists ty.(ty \in TYPE \wedge (va, ty) \in typeOf)))$$

- Tout processeur a une unité arithmétique et logique
- ...

Dans l'analyse ou la modélisation d'un énoncé pour **concevoir un algorithme** ou un logiciel, on a **besoin d'exprimer des invariants**.

Types

Les **types** sont des catégories de données/valeurs ayant les mêmes propriétés, ou **un ensemble de données de même nature**.

Les types en programmation : entier, réel, structure composée d'un entier, d'un autre entier et d'un réel, tableau de type, etc

Un **type élémentaire** est un ensemble d'éléments. On construit des **types structurés** en composant des types, élémentaires ou non.

Les types de bases sont des ensembles :

$\mathbb{N}$ (NAT), $\mathbb{Z}$ (INTEGER), $\mathbb{R}$ (REAL), etc

On utilise les types pour **structurer les données dans les algorithmes**

Types - Propriétés

En Logique **on définit un ensemble par une propriété** ;
par conséquent **les types** sont aussi vus comme **des propriétés**.

Exemple : Un entier paire, est un entier qui a la propriété d'être divisible par 2.

Les entiers **paires** constituent un **sous-ensemble des entiers** ayant la propriété d'être divisible par 2.

Soit $\mathbb{P} \subseteq \mathbb{N}$ l'ensemble des entiers paires.
Dire qu'un entier n donné est paire revient à exprimer :

$$n \in \mathbb{P}$$

Techniques de raisonnement :

Pour construire des algorithmes/programmes, on va raisonner :
Induction (ou récurrence)
Déduction

Récurrence

Le raisonnement par induction/récurrence permet de **généraliser une démonstration d'une propriété sur un ensemble d'objets** (de même nature) sans avoir à le faire sur tous les objets.

Par exemple, la démonstration de propriétés sur les entiers naturels.

Exemple : démontrer que

$$S_n = \sum_{k=0}^{n} k = 0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Pour cela une démonstration par récurrence vous évite de vérifier la propriété pour tout $n \in \mathbb{N}$.

Récurrence

Le schéma de la démonstration consiste à :

- **(dé)montrer/prouver** la propriété pour le premier objet (de **rang 0**, cas de base, initialisation), ou les premiers objets,
- **supposer** que la propriété est vraie pour le n^{ieme} objet (de **rang n**, c'est l'étape dite hérédité), puis sur cette base
- **(dé)montrer/prouver** que, elle est vraie **pour le** $n+1^{ieme}$ objet. En fait, on montre que la propriété vraie à l'ordre n, implique la propriété vraie à l'ordre (n+1).

Illustration

$$S_n = \sum_{k=0}^{n} k = 0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Cas de base

$0 = \frac{0(0+1)}{2}$

$1 = 0 + 1 = \frac{1(1+1)}{2}$

$\ldots$

Ordre n : supposons

$$S_n = \frac{n(n+1)}{2}$$

qu'en est-il de S_{n+1} **?**

$$S_{n+1} = 0 + 1 + 2 + \cdots + n + (n+1)$$

Illustration - récurrence

Cas de base: S_0 ...
Ordre n : supposons $S_n = \frac{n(n+1)}{2}$
qu'en est-il de S_{n+1} **?**

$$\begin{aligned} S_{n+1} &= 0 + 1 + 2 + \cdots + n + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \\ &= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

en conclusion on a démontré :

$$\forall n \in \mathbb{N} \; . \; S_n = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

Généralisation - récurrence

La méthode (**récurrence**) se généralise au delà des entiers (sur un ensemble d'éléments). On peut l'appliquer à **n'importe quel ensemble (bien fondé : muni d'un ordre entre les éléments) pour faire des démonstrations**.

Induction (du particulier au général)

L'induction est la méthode de raisonnement qui **consiste à généraliser des lois ou des propriétés** sur des objets à partir de leur observation sur des faits ou des objets particuliers.

L'induction (en Logique, Informatique), est souvent confondue avec la récurrence.

Preuve par induction ou preuve par récurrence.

Déduction (du général au particulier).

C'est la méthode de raisonnement qui consiste à **attribuer une propriété** à un (ou des objets) particuliers en **partant du fait que la propriété est générale à l'ensemble dont est issu l'objet** (ou les objets).

Exemple : sachant que *un nombre entier est divisible par 9 si la somme de ses chiffres est un multiple de 9*,
si on arrive à montrer que la somme des chiffres d'un entier N donné est multiple de 9, alors on déduit que N est divisible par 9.

✍ Comparer les deux algorithmes !

L'induction et la déduction sont **deux méthodes** ou outils efficaces d'**analyse de problème et de conception d'algorithmes**.

Structures de données

Une structure de données est un **moyen d'organiser** et de **représenter** des données en vue de **simplifier leur manipulation**.

Une structure de données permet de manipuler aisément les données. Il existe différentes structures de données telles que **tuple, tableau, table, liste, file, ensemble, sac, arbre, graphe, etc**.

Ces structures de données sont **employées en fonction de leur adéquation ou efficacité à la résolution du problème** qu'on a sous la main.

Structures de données

Le **choix d'une structure** de donnée relève de la **conception** et ne doit pas précéder l'analyse d'un problème.

Des **données déjà structurées** sont parfois en entrée d'un problème ; cela n'est **pas un carcan pour la conception** car les données peuvent toujours être pré-traitées ou post-traitées. L'analyse globale des coûts de toutes les étapes permet de décider des bons choix.

Structures de données

Il peut être **parfois plus intéressant (moins coûteux) de prétraiter** des données puis **utiliser des méthodes algorithmiques éprouvées**, plutôt que de garder les structures des données en entrée et de devoir concevoir des algorithmes très coûteux en ressource.

Le **choix d'une ou des structures de données** pour un algorithme ou pour un projet, doit être **basé sur des critères soigneusement établis**, et leurs coûts étudiés.

Conclusion : conception et complexité

La prise en compte de l'**étude de la complexité impacte la conception des algorithmes** en particulier et du logiciel de façon générale. Inversement, **une conception rigoureuse des algorithmes impacte leur correction et leur complexité**.

Fort heureusement, en algorithmique, on dispose de **méthodes et d'outils bien établis** pour aider le concepteur-programmeur dans son travail.