Chapitre 2
Correction d'un algorithme
Notions d'invariant et de variant

Notations :

Dans ce chapitre, si T est un tableau de taille n, alors :

- on notera T[i..j] la partie du tableau T formé par les éléments d'index compris entre i et j (avec i ≤ j), c'est-à-dire T[i], T[i+1], …, T[j].
  Exemples : T[0..i] désigne les éléments T[0], T[1], …, T[i].
  T[0..n-1] est donc le tableau T tout entier.
- L'écriture T[i..j] ≤ T[k..l] signifiera que tous les éléments d'index compris entre i et j sont inférieurs ou égaux à tous les éléments d'index compris entre k et l.

I. Correction d'un algorithme

La correction (ou preuve) d'un algorithme consiste à démontrer que celui-ci fonctionne, c'est-à-dire :

- Répond-il correctement à la question ?
- Ne boucle-t-il pas indéfiniment ?

La correction d'un algorithme correspond donc à deux critères :

- **Correction partielle** : l'algorithme calcule le bon résultat
- **Terminaison** : l'algorithme répond en temps fini

Faire la preuve d'un algorithme consiste ainsi à prouver que pour toute entrée vérifiant sa précondition :

- Il produit une sortie vérifiant sa postcondition (correction partielle) ;
- Et ce en temps fini (terminaison).

A. Correction partielle d'un algorithme avec un *invariant* de boucle

Il s'agit de montrer que la sortie produite vérifie la postcondition de l'algorithme (quelle que soit l'entrée vérifiant la précondition). Pour cela, on utilise la notion d'*invariant de boucle.* Il s'agit d'une propriété attachée à une boucle qui :

- Est vraie initialement, avant de commencer la boucle ;
- Est maintenue vraie par toute itération de la boucle, d'où son nom d'*invariant*.

En particulier, elle restera vraie à la sortie de la boucle.

Méthode

En pratique, on procède en 3 étapes pour prouver la correction partielle d'un algorithme :

Etape 1 : INITIALISATION

On prouve que l'invariant est vrai avant l'entrée dans la boucle et donc avant d'exécuter la première itération.

Etape 2 : CONSERVATION

CC BY NC SA

On prouve que l'invariant est conservé par une itération de boucle. Pour cela : on suppose que l'invariant est vrai avant l'itération i de boucle puis on montre que l'invariant est toujours vrai après l'itération i (et donc vrai avant l'itération i+1).
Etape 3 : CONCLUSION
On utilise le fait que l'invariant soit vrai en sortie de boucle pour montrer la correction partielle de l'algorithme.

Prenons de suite un exemple pour mieux comprendre. Considérons la fonction suivante de recherche séquentielle du maximum d'un tableau vue à la séquence 3.

Spécification	En Python

Entrées : tableau T de n entiers
Sortie : entier `maxi`
Rôle : chercher le maximum de T
Précondition : T est non vide
Postcondition : `maxi` est l'élément maximal de T
	`def maximum(T):`
	`    maxi = T[0]`
	`    for i in range(1, len(T)):`
	`        if T[i] > maxi:`
	`            maxi = T[i]`
	`    return maxi`

Prouver la correction de cet algorithme consiste donc à prendre un tableau `T` non vide (précondition) quelconque et de montrer qu'à la fin de l'algorithme, la variable `maxi` est bien l'élément maximal de `T` (postcondition). Démontrons cela !

Il faut commencer par trouver un invariant : pour cela, on cherche une propriété qui est vraie *avant* d'exécuter le tour de boucle `i`.
Nous pouvons choisir l'invariant de boucle suivant : `maxi = maximum(T[0..i-1])`.

La démonstration se fait alors en trois étapes :

INITIALISATION (L'invariant est-il vrai avant la première itération ?)
Vérifions que l'invariant est vrai pour i = 1 (c'est-à-dire que maxi = maximum(T[0..1-1]) = maximum(T[0..0])). Avant l'entrée dans la boucle, maxi = T[0] donc on a bien maxi = maximum(T[0..0]).

CONSERVATION (L'invariant est-il maintenu vrai par une itération de la boucle ?)
Supposons que l'invariant soit vrai au début de l'itération d'indice i de boucle, c'est-à-dire que l'invariant de l'itération précédente est vrai : maxi = maximum(T[0..i-1]).

- Si T[i] <= maxi alors on sait que maxi ne change pas puisque T[i] est inférieur ou égal à maxi. On a donc bien maxi = maximum(T[0..i]) à la fin de l'itération d'indice i.
- Si T[i] > maxi, alors maxi = T[i]. Cela signifie que T[i] > T[0..i-1] et donc T[i] = maximum(T[0..i]), c'est-à-dire maxi = maximum(T[0..i]) à la fin de l'itération d'indice i.

Dans le deux cas, l'invariant est conservé par une itération de la boucle.

CONCLUSION (On utilise l'invariant en sortie de boucle pour prouver que l'algorithme donne le bon résultat)
En particulier, l'invariant sera toujours vrai après la dernière itération de la boucle, celle d'indice i = n – 1. Au début de cette dernière itération on a maxi = maximum(T[0..n-2]) et après les instructions de la dernière itération on a maxi = maximum(T[0..n-1]). Autrement dit, après l'exécution de l'algorithme, la variable maxi est bien le maximum de tous les éléments de T. Cela prouve la correction partielle de l'algorithme.

CC BY NC SA

B. Terminaison d'un algorithme avec un *variant* de boucle

Pour prouver qu'un algorithme termine, il suffit de montrer qu'il ne boucle pas à l'infini.

- Toute algorithme sans appel de fonction ni répétitive termine ;
- Toute répétitive `pour` itère un nombre fini de fois (en Python) ;
- Une répétitive `tant que` (et les appels récursifs) peut boucler à l'infini si sa condition reste toujours vraie.

Ainsi, pour prouver la terminaison d'un algorithme il faudra montrer que les répétitives `tant que` ne bouclent pas à l'infini, c'est-à-dire que leurs conditions deviennent fausses au bout d'un certain nombre d'itération.

Méthode

Pour établir la terminaison d'une répétitive `tant que`, on doit identifier un **variant** (ne pas confondre avec l'invariant !) de la répétitive.
Un **variant** est une expression à valeur entière dépendant des variables impliquées dans la répétitive dont on peut démontrer que la valeur :

- Décroit strictement au cours des itérations ;
- Est positive ou nulle (du fait des initialisations et de la condition de la répétitive).

Ainsi, le variant est un entier qui diminue strictement à chaque itération et qui reste positif. Il ne peut donc prendre qu'un nombre fini de valeurs, ce qui prouve que le nombre d'itérations de la répétitive `tant que` est fini, donc celle-ci termine.

Premier exemple
Prouver la terminaison de l'algorithme de recherche du maximum précédent est facile. En effet, cet algorithme ne contient qu'une boucle `pour` qui termine nécessairement.

Deuxième exemple
Considérons l'algorithme de recherche séquentielle d'une occurrence (voir Séquence n°3) suivant et prouvons sa terminaison.

```
def appartient(v, T):
    i = 0
    trouvee = False
    while i < len(T) and trouvee == False:
        if T[i] == v:
            trouvee = True
        i = i + 1
    return trouvee
```

Il s'agit de trouver un variant de la boucle `while`, c'est-à-dire une quantité entière qui diminue strictement à chaque itération tout en restant positive.

Un **variant possible** est : `len(T) - i` En effet :

- `len(T) - i` est un entier puisque c'est la différence de deux entiers.
- Avant la première itération : `len(T) - i` = `len(T)` ≥ 0.
- A chaque itération : `i` augmente de 1 (par l'instruction `i = i + 1`) et donc `len(T) - i` diminue de 1.

On vient de prouver que la boucle `while` termine et donc cet algorithme termine.

cc BY NC SA

II. Correction des algorithmes de tri

A. Tri par sélection

Rappelons cet algorithme.

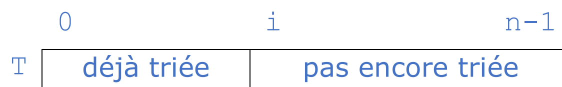
```
def echange(T, i, j):
    """échange T[i] et T[j]"""
    temp = T[i]
    T[i] = T[j]
    T[j] = temp

def tri_par_selection(T):
    """trie le tableau T dans l'ordre croissant"""
    for i in range(len(T)):
        ind_min = i
        for j in range(i+1, len(T)):
            if T[j] < T[ind_min]:
                ind_min = j
        echange(T, i, ind_min)
```

Correction partielle

Il s'agit de montrer que l'algorithme donne le bon résultat, autrement dit qu'il trie bien par ordre croissant le tableau T donné en entrée.

La boucle `for` interne (`for j`) consiste en une recherche du minimum dont la correction partielle est similaire à celle de la recherche du maximum qui a déjà été faite dans le paragraphe précédent (un invariant possible est `ind_min = indice_minimum(T[i..j-1])`). On ne s'intéresse donc ici qu'à la correction partielle de la boucle `for` externe (`for i`).

Commençons par chercher un invariant de boucle (du `for i`). Au début de l'itération d'indice i, on sait que l'élément d'indice i n'est pas encore examiné et ne fait donc pas partie des éléments déjà triés. Voici un schéma de la situation au début de l'itération d'indice i :



Voici l'invariant de boucle (du `for i`) que l'on va utiliser pour prouver la correction :

T[0..i-1] est trié et T[0..i-1] ≤ T[i..n-1]

Autrement dit : « la partie de gauche est triée et tous les éléments de la partie de gauche déjà triés sont inférieurs à tous ceux de la partie de droite par encore triés ».

INITIALISATION : Montrons que l'invariant est vrai avant l'entrée dans la boucle (for i), donc qu'il est vrai lorsque i = 0. Dans ce cas la partie de gauche est vide et est donc triée (T[0..0-1] est trié) et tous les éléments de la partie de droite sont supérieurs à ceux de la partie de gauche puisque cette dernière est vide (T[0..0-1] ≤ T[0..n-1]).

CONSERVATION : Montrons que l'invariant est conservé au cours d'une itération. Supposons donc qu'au début de l'itération i, on a T[0..i-1] est trié et T[0..i-1] ≤ T[i..n-1].

 Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS CC BY NC SA

Au cours de l'itération, T[i] va être remplacé par l'élément minimum de T[i..n-1]. Donc T[i] sera supérieur à T[0..i-1] (d'après l'hypothèse) donc T[0..i] est trié. De plus, T[i] ≤ T[i+1..n-1] donc on a T[0..i-1] ≤ T[i] ≤ T[i+1..n-1] et donc T[0..i] ≤ T[i+1..n-1]
L'invariant reste donc vrai après l'itération.

CORRECTION : L'invariant reste en particulier vrai après la dernière itération, lorsque i = n-1. On a donc en sortie de la boucle : T[0..n-1] est trié (et T[0..n-1] ≤ T[n..n-1] qui est vraie puisque la partie de droite est vide). CQFD.

Terminaison

L'algorithme termine puisqu'il est composé de deux répétitives `pour` qui terminent nécessairement.

B. Tri par insertion

Rappelons cet algorithme.

```
def tri_par_insertion(T):
    """trie le tableau T dans l'ordre croissant"""
    for i in range(1,len(T)):
        x = T[i]
        j = i
        while j>0 and x < T[j-1]:
            T[j] = T[j-1]
            j = j - 1
        T[j] = x
```

Correction partielle

Montrons que cet algorithme trie bien le tableau T par ordre croissant.

Au début de l'itération d'indice i, on est dans la situation suivante

	0	i	n-1
T	éléments triés	?	...

et on cherche à savoir où insérer l'élément d'indice i parmi T[0..i].

On sait donc que T[0], T[1], …, T[i-1] sont triés et cela va nous donner notre invariant.

Considérons l'invariant suivant : T[0..i-1] est trié (c'est-à-dire « la partie de gauche est triée »)

INITIALISATION : Montrons que l'invariant est vrai avant l'entrée dans la boucle (for i), donc qu'il est vrai lorsque i = 1. Dans ce cas, la partie de gauche est T[0..0] = T[0] qui est forcément triée puisqu'elle ne contient qu'un seul élément.

CONSERVATION : Montrons que l'invariant est conservé au cours d'une itération.
Supposons donc qu'au début de l'itération i, on a T[0..i-1] trié.
Au cours de l'itération, on vient insérer T[i] à sa place dans T[0..i] donc T[0..i] reste trié.
L'invariant reste donc vrai après l'itération i.

CORRECTION : L'invariant reste en particulier vrai après la dernière itération, lorsque i = n-1. On a donc en sortie de la boucle : T[0..n-1] est trié. CQFD.

cc BY NC SA

Terminaison

Il s'agit de montrer que la répétitive `tant que` termine en trouvant un variant.

Un variant possible est : `j`.

En effet :

- j est un entier puisque j = i qui est un entier
- Avant la première itération : j = i avec i compris entre 1 et len(T) − 1 donc i est positif et donc j aussi.
- A chaque itération : j diminue de 1 (instruction j = j − 1)

Ainsi, j est bien un variant puisqu'il s'agit d'une quantité entière positive qui diminue strictement à chaque itération. La condition j > 0 deviendra donc fausse au bout d'un certain nombre d'itérations et donc la boucle while termine.

Comme la répétitive `pour` externe termine, alors l'algorithme termine.

Ressources :

- Document ressource du DIU-EIL, Université de Nantes, C. JERMANN.
- *Numérique et Sciences Informatiques*, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, Ellipses.

cc BY NC SA