Algorithmique :

Terminaison et correction.

A.Mazza
MPSI 2-3
CPGE Oujda

Introduction

Il y a 3 grandes notions importantes qui permettent d'étudier un algorithme :

- Combien de **temps** met-il pour s'exécuter et combien d'**espace mémoire** prend-il ? C'est la **complexité** de l'algorithme.
- Est-ce qu'il donne un résultat, est-ce qu'il ne s'arrête jamais, est-ce qu'il génère une erreur ? C'est la **terminaison** de l'algorithme.
- Est-ce qu'il donne le résultat attendu ? C'est la **correction** (partielle) de l'algorithme.

Remarque:

Faire la preuve d'un algorithme consiste à prouver sa `terminaison` et sa `correction partielle`, on dit qu'on a prouvé la **correction totale de l'algorithme**.

Introduction

Parfois c'est simple de prouver la correction totale d'un algorithme (mais rarement), on est sûr qu'un algorithme :

- termine si le nombre d'instructions a effectuer est fini.
- est correct (partiellement) s'il reproduit directement une formule.

Exemple 1 :Calculer la moyenne de 2 nombres *a* et *b* :

```
1 def moyenne(a, b):
2     return (a + b)/2
```

- **Terminaison**: c'est un « algorithme-calcul », il a un nombre d'instructions fini : il se termine quand tous les calculs sont effectués, et ici il n'y a qu'un seul calcul.
- **Correction**: l'algorithme correspond à la formule,donc il donnera bien le résultat attendu.

Dans ce cas particulier, la terminaison et la correction de l'algorithme sont très simples à justifier. Mais dans la majorité des cas, c'est bien plus compliqué.

Introduction

Exemple 2 :Calculer la somme $0 + 1 + 2 + \cdots + n$:

```
1  def somme(n):
2      s = 0
3      for i in range(n + 1):
4          s = s + i
5      return s
```

Cette fois l'algorithme n'est pas réduit à la simple application d'une formule.

- **Terminaison**: Il faut être sûr qu'il y a un nombre fini d'étapes. Cet algorithme a une boucle mais c'est une boucle for donc il y a un nombre fini d'itérations : n + 1 itérations, puisque i varie de 0 à n. Le nombre d'instructions étant fini, l'algorithme termine.
- **Correction**: Il faut vérifier que s contient à la fin de l'exécution le résultat attendu.

Introduction

Correction de l'exemple 2

Il faut vérifier les étapes :

- L'initialisation de s est correcte ($s = 0$ au début, la somme de 0 à 0 est bien égale à 0).
- Les modifications de s à chaque itération sont correctes : on ajoute i à chaque itération et i varie de 0 à n en ajoutant 1 à i à chaque étape, donc à toute étape k (k étant inférieur à n), on aura calculé $0 + 1 + 2 + ... + (k - 1)$.
- Enfin, le nombre d'itérations est le bon $(n + 1)$ itérations, ce qui correspond bien au nombre de nombres de 0 à n), donc le résultat sera bien celui attendu, et donc la **correction est vérifiée**.

Remarque: "Normalement" les boucles for en Python itèrent un nombre fini de fois,donc, dans ce cas il est facile de prouver la terminaison puisqu'il y aura automatiquement un nombre fini d'itérations. **Les principales sources de non-terminaison sont les boucles while (tantque) et les appels récursifs .**

Notion de variant de boucle.

Pour prouver la terminaison d'un algorithme qui contient une boucle "tant que", on doit identifier un "variant de boucle".

Définition : Un variant de boucle

Un variant de boucle est une expression (c'est souvent le simple contenu d'une variable) dont la valeur varie à chacune des itérations de la boucle et dont on peut démontrer que la valeur :

- est un entier positif tout au long de boucle,
- décroit strictement à chaque itération,
- et qui, si elle devient négative (ou nulle), assure qu'on sort de la boucle.

Si on arrive à trouver un variant de boucle, alors on peut conclure que la boucle termine.

Notion de variant de boucle

Identifier les variants de boucles dans les exemples suivants:

Codes Python:

```
1  #Exemple 1
2  i = 2
3  while i > 0:
4      p = p * 2
5      i = i - 1
6  #Exemple 2
7  #table des 2
8  a = 0
9  while a <= 10 :
10     print('2 *',a,'=',2 * a)
11     a = a + 1
12 #Exemple 3
13 t = [0]
14 for i in t:
15     t.append(i)
```

Notion d'invariant de boucle.

Pour prouver la correction (partielle) d'un algorithme (c.a.d s'il fait bien ce qu'on attend de lui), on utilise un **invariant de boucle**.

Définition:invariant de boucle

On appelle **invariant de boucle** une propriété P telle que :

- P est vérifiée avant l'entrée dans la boucle,
- si P est vérifiée avant une itération, alors elle reste vérifiée après cette itération, c'est-à-dire que P est préservée lors d'un passage dans la boucle au suivant et
- en particulier P reste vérifiée en sortie de boucle, quand on la quitte.

La mise en évidence d'un invariant de boucle adapté permet de prouver la correction d'un algorithme.

Notion d'invariant de boucle

Dans la pratique, on procède alors en 3 étapes pour prouver la correction (partielle) d'un algorithme :

- **1ère étape : Initialisation.** on prouve (en général, c'est une simple vérification) que l'invariant est vrai avant le premier passage ;
- **2ème étape : Hérédité (ou conservation)**. On montre qu'il est préservé d'une itération à la suivante ;
- **3ème étape : conclusion.** On conclut qu'il est vrai lorsque l'on quitte la boucle.

Remarque: Ce type de démonstration est très proche du raisonnement par récurrence vu en mathématiques et l'invariant de boucle qu'on utilise en informatique correspond à "l'hypothèse de récurrence" mathématique.

Invariant de boucle

Prouver la correction partielle de la fonction suivante:

```
1     def maximum(L):   #L est une liste de n nombres.
2         maxi = L[0]
3         for i in range(1, len(L)):
4             if L[i] > maxi :
5                 maxi = L[i]
6         return maxi
```

Preuve:

On note la liste $L = [l_0, \ldots, l_{n-1}]$.
Pour prouver la correction de cette fonction, il faut trouver un invariant de boucle, donc il faut trouver une propriété qui est vraie avant d'exécuter le 1er tour de la boucle for et qui le reste.
Soit i un entier strictement positif, on note $P(i)$ la propriété :`maxi=maximum([`$l_0, \ldots, l_{i-1}$`])`.
On va démontrer que cette propriété est un **invariant de boucle**.

- **Initialisation:** On vérifie que P est vraie pour $i = 1$, c'est à dire que `maxi=maximum([`l_0, l_{1-1}`])`.
Avant la 1ère itération (en entrant dans la boucle) `maxi=L[0]` et donc on a bien `maxi=L[0]=maximum([`l_0`])`.

Preuve (suite):

- **Hérédité :** Supposons que P soit vraie au début de l'itération d'indice i , c'est-à-dire que `maxi=maximum(`$[l_0, \ldots, l_{i-1}]$`)` et montrons alors qu'elle sera conservée par itération de la boucle, c'est-à-dire que `maxi=maximum(`$[l_0, \ldots, l_{i-1}, l_i]$`)`
  On a : `maxi=maximum(`$[l_0, \ldots, l_{i-1}]$`)`
- Si $l_i \leq$ *maxi* alors *maxi* ne change pas et donc *maxi*`=maximum(`$[l_0, \ldots, l_{i-1}]$`)=maximum(`$[l_0, \ldots, l_{i-1}, l_i]$`)` à la fin de l'itération d'indice i.
- Si $l_i >$ *maxi* alors $maxi = l_i$ et le maximum de $[l_0, \ldots, l_{i-1}, l_i]$ est bien l_i puisque c'est la plus grande valeur, donc *maxi*`=maximum(`$[l_0, \ldots, l_{i-1}, l_i]$`)`

Donc, dans les 2 cas,P est conservée par itération de la boucle.

Conclusion

- **Conclusion:**
  En particulier, l'invariant sera toujours vrai après la dernière itération de la boucle, celle d'indice $n-1$. Avant cette itération, *maxi*=`maximum(`$[l_0, \ldots, l_{n-2}]$`)` et donc après cette itération, *maxi*=`maximum(`$[l_0, \ldots, l_{n-1}]$`)`,*maxi* est donc bien le maximum de tous les éléments de L.

  On a donc bien prouvé **la correction partielle** de cette fonction.
- **Remarque :** La terminaison étant évidente avec cette boucle for, on peut dire qu'on a prouvé **la correction totale** de l'algorithme ou que **l'algorithme est prouvé**.