Preuves d'algorithmes

1 Méthodes

1.1 Position du problème

Lorsqu'on écrit un programme il est parfois très important de vérifier qu'il est correct, c'est à dire qu'il calcule bien ce qu'on attend. C'est même vital lorsque dans certaines applications industrielles une erreur aurait un impact sur des vies humaines (on pensera par exemple aux programmes qui contrôlent la conduite des lignes de métro automatiques, ou à des programmes touchant au secteur militaire).

Si la preuve - automatique ou non - de programmes est un vaste sujet où la recherche est active aujourd'hui, elle dépasse en général le cadre et le formalisme de ce cours. Néanmoins, un cas particulier nous est accessible : il s'agit de la preuve de la correction de boucles.

De manière plus détaillée, nous allons :

- Montrer qu'une boucle se termine bien. On appelle ce problème la **terminaison**.
- Montrer que si la boucle s'arrête, elle calcule bien ce qu'elle est supposée calculer. On appelle ce problème la **correction partielle**.
- La **correction totale** est la terminaison et la correction partielle.

1.2 Invariant de boucle

Les invariants de boucle sont un outil qui va permettre de montrer la correction partielle d'algorithmes.

Définition 1 (Invariant de boucle)**.** Un invariant de boucle est une propriété qui :

1. est vérifiée avant l'entrée d'une boucle
2. si elle est vérifiée avant une itération est vérifiée après celle-ci
3. lorsqu'elle est vérifiée à la sortie d'une boucle permet d'en déduire que le programme est correct.

À titre d'exemple, nous allons démontrer à l'aide d'un invariant de boucle la correction du programme suivant (qui calcule la somme des puissances k-ièmes des n premiers entiers naturels) :

```
def S(p):
   valeur_somme=0
   for i in range(1,p+1) :
         valeur_somme=valeur_somme+i**k
   return valeur_somme
```

Nous étudions la propriété suivante (qui est notre invariant de boucle) : après n itérations de cette boucle, on a :

$$p_n : \text{valeur\_somme(n)} = \sum_{i=0}^{i=n} i^k$$

1. La propriété p_0 est bien vérifiée avant l'entrée de la boucle : valeur_somme=0 par initialisation et on a bien :

$$\sum_{i=0}^{i=0} i^k = 0$$

2. Supposons que la propriété p_n soit vérifiée pour un certain n. On a alors avant la (n+1)-ème itération de cette boucle :

$$p_n : \text{valeur\_somme(n)} = \sum_{i=0}^{i=n} i^k$$

Après la (n+1)-ème itération de cette boucle on a :

$$\text{valeur\_somme(n+1)} = \text{valeur\_somme(n)} + i^{k+1} = \sum_{i=0}^{i=n} i^k + i^{n+1} = \sum_{i=0}^{i=n+1} i^k$$

donc p_{n+1} est bien vérifiée

3. En sortie de boucle, si p_p est bien vérifiée, le calcul est exact

1.3 Variant de boucle

Un algorithme est un programme de calcul qui s'exécute en un nombre fini d'étapes. Si dans le cadre des boucles inconditionnelles on a la certitude que le calcul se finira effectivement, lors de l'exécution d'une boucle conditionnelle, nous n'avons pas cette certitude.

Considérons par exemple le programme suivant :

```
c=int(input())
p=1
while c>0:
   p=2*p
   c=c-1
print(p)
```

Ce programme calcul 2^c au moyen d'une boucle. La variable c joue le rôle d'un compteur lors de l'exécution de la boucle. Comme c est strictement décroissante lors de l'exécution de celle-ci, ce programme s'arrêtera après c étapes.

Regardons maintenant une variante de ce programme :

```
c=int(input())
p=1
while c!=0:
   p=2*p
   c=c-1
print(p)
```

Ce programme comporte un problème de taille : si l'utilisateur rentre un nombre négatif lors de la saisie de c, la condition c!=0 ne sera jamais vérifiée (ce qui explique que le premier programme soit correct là où le second ne l'est pas) et ce programme ne s'interrompra jamais. Ceci s'appelle une boucle infinie. Ce type d'erreurs de conception est indétectable par l'ordinateur et il est en pratique assez courant car les conditions de sortie de boucles sont souvent plus complexes que dans cet exemple. Le seul moyen de sortir d'une boucle infinie est d'interrompre à la main l'exécution du programme en pressant les touches Ctrl et C simultanément dans l'interpréteur.

Il est également possible d'interrompre depuis le code source une boucle en entrant la commande "break". Cette méthode est occasionnellement efficace, par exemple lorsqu'on cherche si un nombre est premier en testant ses diviseurs. Néanmoins cette méthode est en général à proscrire car dans des programmes complexes elle rend complexe la compréhension des cas des sorties de boucles et est de ce fait génératrice de nombreuses erreurs de programmation.

Dans l'exemple précédent, la variable c joue le rôle d'un compteur. Cette variable diminue de 1 à chaque itération et la boucle s'arrête lorsqu'elle n'est plus strictement positive. En généralisant, une quantité qui vérifie certaines propriétés sera appelé un variant de boucle et permettra de montrer la terminaison d'un programme.

Définition 2 (variant de boucle)**.** Un variant de boucle est une quantité entière :

1. qui est un entier strictement positif avant l'exécution de la boucle
2. qui décroît strictement à chaque itération
3. qui lorsqu'elle est inférieure à un certain nombre (en particulier lorsqu'elle arrête d'être strictement positive) rend la condition d'exécution de la boucle conditionnelle fausse

Les variants de boucles sont des outils puissants pour étudier la terminaison d'une boucle ou d'un algorithme. Néanmoins il est parfois difficile d'en exprimer, et à l'heure actuelle on ne sait pas si certains programmes de calcul d'apparence simple (voir par exemple la suite de Syraccuse) s'arrêtent effectivement.

2 Exemples

2.1 Algorithme d'Euclide

Exercice 1. Montrer la terminaison et la correction de l'algorithme par soustraction de la division euclidienne de a par b. On utilisera la conservation de la quantité B*Q+R pour la correction

```
B=b
R=a
Q=0
while R>=B :
   R=R-B
   Q=Q+1
print ("Le quotient de la division euclidienne de a par b est : ",Q)
print ("Le reste de la division euclidienne de a par b est : ",R)
```

2.2 Exponentiation rapide

Exercice 2. On considère l'algorithme suivant :

```
def Puissance(a,n):
   A=a
   N=n
   R=1
   while N>0 :
         if N%2==0 :
               A=A**2
               N=N/2
         else :
               R=R*A
               N=N-1
   return R
```

1. Que calcule-t-il ?
2. Prouver sa terminaison
3. Prouver sa correction. On pourra vérifier que $a^n = R \times A^N$
4. Quelle est la complexité de cet algorithme ?