Analyse d'algorithmes

Nous allons nous intéresser à l'analyse de programmes en étudiant 2 questions concernant la résolution d'un problème à l'aide d'un programme :

1. Le programme donne-t-il un résultat ? Autrement dit, est-ce que son exécution termine et est-ce qu'il renvoie un résultat ?

2. Le programme est-il correct ? Le résultat renvoyé est-il le bon ?

On peut aussi se poser la question de l'efficacité d'un programme, mais c'est une autre question, celle de la *complexité*, que nous l'aborderons pas ici.

I. Terminaison

Il peut arriver que l'exécution d'un programme ne termine pas à cause d'une boucle infinie. Cela peut arriver dans deux cas importants :

- ⋆ une boucle `while` où la condition de boucle est toujours vraie (c'est en particulier le cas si l'on ne modifie pas dans le corps de la boucle des variables servant à évaluer la condition),
- ⋆ une ou des fonctions récursives qui s'appellent sans jamais arriver à un cas de base.

Un outil praqieu pour prouver qu'une boucle de va pas être répétée indéfiniment est d'identifier un *variant de boucle*.

Définition Un **variant de boucle** est une quantité entière dépendant des variables du programme qui :

- ⋆ est à valeurs positives dès lors que la condition de la boucle est vérifiée ;
- ⋆ décroit **strictement** d'une itération à la suivante.

On peut adapter la notion de variant à une fonction récursive, auquel cas un appel récursif implique que le variant est positif, et la valeur du variant décroit strictement d'un appel récursif au suivant.

Proposition 1

Si une boucle admet un variant, alors elle termine.

Preuve

En formant la liste constituée des valeurs successives du variant, on obtient une suite à valeur dans **N** strictement décroissante. Elle est nécessairement finie.

Exemple Considérons la fonction suivante :

```
def appartient(val, tab):
    n = len(tab)
    p = 0
    while p < n and tab[p] != val:
        p = p + 1
    return p < n
```

Il est clair que l'on peut utiliser comme variant de boucle la quantité « `n - p` ». En effet,

- ⋆ Si la condition de boucle est vérifiée, on a `p < n` et donc `n - p` ≥ 0 ;
- ⋆ d'une itération à la suivante, `p` a été incrémenté de 1 et donc `n - p` a diminué strictement.

L'existence d'un variant assure que la boucle termine.

Exercice 1 Prouvez que la fonction suivante termine.

```
def compte(n):
    assert(n >= 0)
    cnt = 0
    while n > 0:
        cnt = cnt + (n % 2)
        n = n // 2
    return cnt
```

Exercice 2 Prouvez que la fonction suivante termine.

```
def euclide(a, b):
    while b > 0:
        r = a % b
        a = b
        b = r
    return a
```

Remarque La notion de *variant de boucle* est un outil assez simple pour déterminer si une boucle termine, mais il n'est pas toujours possible d'en trouver.

Voici un exemple classique de fonction dont on ne sait pas si elle termine ou non :

```
def siracuse(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
```

D'après le mathématicien hongrois Paul Erdős, « les mathématiques ne sont peut-être pas encore prêtes pour de tels problèmes. »

Question 3 Concrètement, que signifie l'affirmation que l'on ne sait pas si cette fonction termine ?

Remarque, suite Le mathématicien Alan Turing a prouvé qu'il n'existe pas d'algorithme permettant de décider si un programme termine ou non.

II. Correction

Maintenant que l'on a (espérons-le) réussi à prouver que notre fonction termine, il reste à s'assurer que le résultat obtenu est correct : on veut prouver la **correction** de l'algorithme.

Remarque On parle de **correction partielle** si l'on peut prouver que le résultat obtenu est correct, mais sans assurer que l'algorithme termine.

Par exemple, considérons la fonction suivante :

```
def somme(t):
    s = 0
    for v in t:
        s = s + v
    return s
```

Il semblerait qu'elle renvoie la somme des éléments du tableau passé en argument.

Nous allons prouver cela en utilisant une méthode basée sur l'utilisation de boucles `while` sans sortie anticipée. Commençons par réecrire la fonction précédente.

```
def somme(t):
    n = len(t)
    s = 0
    i = 0
    while i < n:
        s = s + t[i]
        i = i + 1
    return s
```

Définition Un **invariant de boucle** est un propriété mathématique, dont la valeur de vérité dépend des variables du programme, qui :

- ⋆ est vraie en entrée de boucle, lors de la première évaluation de la condition ;
- ⋆ si elle est vraie en entrant dans le corps de la boucle, est encore vraie en sortie du corps de la boucle.

Proposition 2

Un invariant de boucle est encore vérifié en sortie de boucle.

Intuitivement, un invariant de boucle est une propriété qui va être vrai tout au long de l'exécution de la boucle, indépendemment du nombre d'itérations effectuées. Cela permet de raisonner formellement sur le comportement des variables du programme durant l'exécution d'une boucle. Il peut bien sûr y avoir plusieurs invariants.

Exemple Dans la fonction `somme` précédente, on peut utiliser comme invariant :

$$s = \sum_{k=0}^{i-1} \texttt{t}[k]$$

Cette propriété est vrai en entrée de boucle, quand $s = 0$ et $i = 0$. De plus, si en entrée de boucle, on a

$$s_i = \sum_{k=0}^{i_i-1} \texttt{t}[k]$$

alors en fin de boucle on a $s_f = s_i + \texttt{t}[i_i]$ et $i_f = i_i + 1$ et donc

$$\begin{aligned} s_f &= s_i + \texttt{t}[i_i] \\ &= \sum_{k=0}^{i_i-1} \texttt{t}[k] + \texttt{t}[i_i] \\ &= \sum_{k=0}^{i_i} \texttt{t}[k] \\ &= \sum_{k=0}^{i_f-1} \texttt{t}[k] \end{aligned}$$

On a prouvé qu'il s'agit d'un invariant de la boucle. En sortie de boucle, celui-ci

est encore vérifié. Et puisqu'alors on a $i = n$, on en déduit que :

$$s = \sum_{k=0}^{n-1} \texttt{t}[k]$$

Exercice 4 Dans l'exemple précédent, on a affirmé qu'en sortie de boucle, on avait bien $i = n$. Pouvez-vous le prouver, à l'aide d'un invariant bien choisi ?

III. Étude de l'algorithme de recherche dichotomique

Nous allons analyser la fonction suivante. Elle prends en entrée un tableau **trié par ordre croissant** et un élément, et renvoie `True` si l'élément apparaît dans la tableau, et `False` sinon.

```
def recherche_dichotomique(tab, elt):
    deb = 0
    fin = len(tab) - 1
    while deb <= fin:
        mil = (deb + fin) // 2
        if tab[mil] == elt:
            return True
        if tab[mil] < elt:
            deb = mil + 1
        else:
            fin = mil - 1
    return False
```

Exercice 5 Prouver que l'algorithme termine.

On ne s'occupe pas ici de la complexité (qui est logarithmique en la taille du tableau), mais seulement de sa terminaison.

Exercice 6

1. Prouver que l'on a l'invariant suivant :

$$\forall k \in [\![0, \texttt{len(tab)} - 1]\!], \quad k < \texttt{deb} \implies \texttt{tab}[k] < \texttt{elt}$$
$$\text{et } k > \texttt{fin} \implies \texttt{tab}[k] > \texttt{elt}$$

2. En déduire la correction du programme.