1 Introduction

• Lorsque l'on écrit un algorithme, il est impératif de vérifier que cet algorithme produit un résultat après un nombre fini d'étapes et que ce résultat est correct, c'est-à-dire conforme à ce qui est attendu. En effet, les problèmes de sécurité (données personnelles, systèmes sensibles comme dans un hôpital) et de sûreté sont devenus des questions très importantes dans notre société de plus en plus connectée.

Définition : On dit qu'un algorithme est **valide** s'il se termine et produit un résultat correct.

• Deux conditions sont donc à vérifier :

- ▶ l'algorithme donne une réponse, c'est l'étude de la **terminaison**,
- ▶ la réponse donnée est celle attendue, c'est l'étude de la **correction**.

• Si lorsqu'il se termine, l'algorithme donne la réponse attendue, on parle de **correction partielle**. Dans le cas où la terminaison et la correction sont assurées, on parle de **correction totale.**

• L'objectif de ce chapitre, est de mettre en place des outils théoriques permettant de démontrer la validité d'un algorithme.

2 Terminaison d'un algorithme

2.1 Méthode de justification

• Il est clair qu'une boucle **for** va se terminer au bout d'un nombre fini d'itérations et même si le temps mis peut-être très long, d'un point de vue théorique la terminaison est assurée.

• Toute la problématique de ce paragraphe est de démontrer qu'une boucle *while* va se terminer. Pour cela, on va trouver une quantité, c, qui vérifie les deux conditions suivantes :

- ▶ c est un entier naturel
- ▶ c décroît strictement à chaque itération de la boucle *while*.

Comme il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il ne peut y avoir qu'un nombre fini d'itérations de la boucle *while*. Cette quantité c est appelée un **variant de boucle**.

• On peut étendre la notion de variant de boucle en trouvant une expression dont les valeurs prises au cours des itérations constituent une suite qui converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt de la boucle.

2.2 Deux exemples

• Voici un exemple simple :

```
1 x = 0
2 while x < 10:
3     x = x + 2
```

Un variant de boucle est $10 - x$. En effet, la suite des valeurs prises par $10 - x$ au cours de l'exécution de la boucle est une suite d'entiers naturels qui décroit strictement à chaque passage dans la boucle puisque x devient $x + 2$. Le nombre de passages dans la boucle est donc fini, ce qui démontre que le programme se termine.

• On considère la fonction pgcd qui prend en arguments deux entiers naturels a et b non nuls et qui renvoie la valeur du pgcd de a et b. Cette fonction se base sur l'algorithme d'Euclide qui permet d'obtenir le pgcd comme le dernier reste non nul dans la suite des divisions euclidiennes.

```
def pgcd(a, b):
    while b > 0: # tant que le reste est non nul, on poursuit les divisions
        a, b = b, a%b # a%b désigne le reste dans la division euclidienne de a par b
    return(a) # on renvoit le dernier reste non nul
```

Un variant de boucle est b. En effet, b est un entier naturel et d'après le théorème de la division euclidienne, si l'on effectue la division de a par b :

$$\exists (q,r) \in \mathbb{N}^2,\ a = bq + r \quad \text{avec } 0 \leq r \leq b-1$$

À chaque passage dans la boucle, b est remplacé par ce reste r ainsi la valeur de b décroit strictement comme voulu. On a trouvé une suite d'entiers naturels strictement décroissante, ce qui assure la terminaison de la boucle.

3 Correction d'un algorithme

3.1 Méthode de justification

• Lorsque l'on écrit un algorithme, on peut tester quelques cas significatifs pour voir s'il fonctionne mais il est plus satisfaisant de démontrer qu'il est correct dans tous les cas. Comme l'a dit le mathématicien et informaticien Edsger Dijkstra :

"Testing shows the presence, not the absence of bugs"

• Pour démontrer qu'un algorithme est correct, on doit trouver un **invariant de boucle**, c'est-à-dire une propriété qui :

- ▶ est vérifiée avant d'entrer dans la boucle,
- ▶ si elle est vérifiée avant une itération de la boucle, elle est aussi vérifiée après celle-ci
- ▶ et si la terminaison est assurée, la propriété sera vraie à la sortie de la boucle.

C'est analogue au principe de récurrence, la première étape correspond à l'initialisation, la deuxième étape à l'hérédité et la troisième étape à la conclusion.

3.2 Deux exemples

• Voici un premier exemple :

```
m = 0
p = 0
while m < a:
    m = m + 1
    p = p + b
```

En analysant ce programme, on comprend qu'il effectue le produit $a \times b$ puisqu'il ajoute a fois le nombre b à la variable p. Nous allons démontrer ceci en considérant l'invariant de boucle suivant :

$$p = m \times b$$

▶ Cette propriété est vraie initialement, c'est-à-dire avant de rentrer dans la boucle puisque les variables m et p sont initialisées à 0.

▶ On suppose que la propriété $p = m \times b$ est vraie avant un passage dans la boucle. Les nouvelles valeurs de m et p après le passage dans la boucle, notées m' et p', sont $p' = p + b$ et $m' = m + 1$, ainsi :

$$m' \times b = (m+1) \times b = m \times b + b = p + b = p'$$

On a bien $p' = m' \times b$, ce qui montre que la propriété reste vraie après ce passage dans la boucle.

▶ Si la terminaison de l'algorithme a été démontrée au préalable, on en déduit qu'à la fin de la boucle, on a toujours $p = m \times b$. Or à la sortie de la boucle, la variable m a pour valeur celle de a. Ainsi $p = a \times b$ comme voulu. Cet algorithme effectue bien le produit de a par b.

• Voici un autre exemple où l'on considère l'algorithme suivant qui détermine le plus grand élément d'une liste non vide de nombres.

```
def maximum(L):
    maxi = L[0]
    n = len(L)
    for i in range(1, n):
        if L[i] > maxi:
            maxi = L[i]
    return(maxi)
```

La terminaison de l'algorithme est assurée puisqu'il met en jeu une boucle *for*, justifions la correction de l'algorithme. Pour $i \in [\![0, n-1]\!]$, on considère la propriété :

P_i : *"après i passages dans la boucle, maxi est le maximum des i + 1 premiers éléments de la liste L"*

▶ La propriété P_0 est vraie puisque si l'on a effectué 0 passage dans la boucle *maxi* est le premier élément de la liste.

▶ On suppose que P_i est vraie pour $i \in [\![0, n-2]\!]$, c'est-à-dire que *maxi* est le maximum des $i+1$ premiers éléments de la liste. Lors du passage dans la boucle numéro $i+1$, on compare *maxi* au $i+2$-ème élément de la liste (celui d'indice $i+1$), ainsi la fin de ce passage dans la boucle *maxi* sera bien le maximum des $i+2$ premiers éléments de la liste. Ceci démontre que P_{i+1} est vraie.

▶ À la fin de la boucle P_{n-1} est vraie (on fait $n-1$ passages dans la boucle), c'est-à-dire que *maxi* est le maximum des n premiers éléments de la liste, c'est bien le maximum de la liste.

4 L'exemple de la division euclidienne

4.1 L'algorithme

Voyons un dernier exemple important car la fonction mise en jeu est classique. Il s'agit d'un algorithme qui prend en paramètres deux entiers naturels a et b avec $b \neq 0$ et qui renvoie le quotient et le reste dans la division euclidienne de a par b. Voici cette fonction :

```
def euclide(a, b):
    """renvoie le quotient et le reste dans la division euclidienne de a par b"""
    q = 0 # initialisation du quotient et du reste
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return(q,r)
```

• Prenons un exemple afin d'appréhender le fonctionnement de l'algorithme : $a = 17$ et $b = 4$. Voici l'évolution des valeurs des variables q et r au cours de l'algorithme.

Étape	q	r
Avant la première itération de la boucle while	0	17
Après la première itération de la boucle while	1	13
Après la deuxième itération de la boucle while	2	9
Après la troisième itération de la boucle while	3	5
Après la quatrième itération de la boucle while	4	1

C'est la technique que l'on apprend à l'école primaire pour effectuer une division euclidienne, on regarde "combien de fois 4 va dans 17" en enlevant 4 à 17 autant de fois que possible. Ce nombre de fois est notre quotient et le nombre restant est notre reste. Démontrons à présent la terminaison et la correction de notre algorithme afin de justifier sa validité.

4.2 Terminaison

On peut choisir comme variant de boucle r. En effet, tout au long de l'algorithme, on a bien r qui est un entier et $r \geq 0$ puisqu'à chaque fois que l'on rentre dans la boucle **while**, on a $r \geq b$ et r est remplacé par $r - b$ dans la boucle. D'autre part, r décroît strictement à chaque itération de la boucle car $b > 0$. On a trouvé une suite d'entiers naturels qui décroit strictement, cela démontre que le programme se termine.

4.3 Correction

On va démontrer qu'un invariant de boucle est la propriété :

$$P \ : \ "r \geq 0 \text{ et } a = bq + r"$$

▶ Avant de rentrer dans la boucle, on a $q = 0$ et $r = a$ ainsi $r \geq 0$ et $a = bq + r$.

▶ On suppose qu'au début d'un passage dans la boucle **while**, on a $r \geq 0$ et $a = bq + r$. Il y a deux possibilités :

-soit la condition $r \geq b$ n'est pas vérifiée dans ce cas, on sort de la boucle et notre propriété reste inchangée.

-soit la condition est vérifiée et l'on rentre dans la boucle, dans ce cas q est transformé en $q' = q + 1$ et r devient $r' = r - b$. On a alors :

$$a = bq + r = b(q + 1) + r - b = bq' + r'$$

De plus $r' = r - b \geq 0$ car $r \geq b$ puisque l'on est entré dans la boucle. La propriété est donc bien inchangée après un passage supplémentaire dans la boucle.

▶ Lorsque l'on sort de la boucle, on a $a = bq + r$ et $0 \leq r < b$ avec q et r deux entiers naturels. Cette écriture correspond bien à la division euclidienne de a par b avec q le quotient et r le reste. Ceci démontre que l'algorithme renvoie bien le résultat attendu et termine la preuve de la correction.