Terminaison et correction

INFORMATIQUE COMMUNE - TP n° 2.1 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ programmer les algorithmes donnés en exemples.
- ☞ prouver la terminaison d'un algorithme simple.
- ☞ prouver la correction d'un algorithme simple.

A Terminaison

A1. Prouver la terminaison de l'algorithme 1 puis le traduire en Python.

Algorithme 1 Palindrome

1 : **Fonction** PALINDROME(w)
2 : $n \leftarrow$ la taille de la chaîne de caractères w
3 : $i \leftarrow 0$
4 : $j \leftarrow n-1$
5 : **tant que** $i < j$ **répéter**
6 : **si** $w[i] = w[j]$ **alors**
7 : $i \leftarrow i+1$
8 : $j \leftarrow j-1$
9 : **sinon**
10 : **renvoyer** Faux
11 : **renvoyer** Vrai

Solution : Si la condition en ligne 6 est invalidée, l'algorithme se termine. Si ce n'est pas le cas, on utilise le variant de boucle $v(i,j) = j - i$. On vérifie qu'il est bien initialement positif (n-1), à valeurs entières (i et j sont des entiers) et qu'il décroît strictement (de deux unités à chaque tour de boucle car i est incrémenté de 1 et j décrémenté de 1). Nécessairement, v va donc atteindre ou dépasser la valeur 0. Dans ce cas, la condition $i < j$ est invalidée et la boucle se termine. L'algorithme palindrome se termine.

```python
def palindrome(s):
    deb = 0
    fin = len(s) - 1
    v = fin -deb
    while v > 0 :
        v_prec = fin - deb
```

```python
        if s[deb] == s[fin]:
            deb += 1
            fin -= 1
        else:
            return False
        v = fin -deb
        assert v < v_prec  # loop variant
    return True
```

A2. Prouver la terminaison de l'algorithme 2 puis le traduire en Python.

Algorithme 2 Est une puissance de deux

1 : **Fonction** EST_PUISSANCE_DE_DEUX(n)
2 :   **si** $n < 2$ **alors**
3 :   **renvoyer** Faux
4 :   **sinon**
5 : $m \leftarrow n \mod 2$
6 :   **tant que** $m = 0$ **répéter**
7 : $n \leftarrow n//2$
8 : $m \leftarrow n \mod 2$
9 :   **renvoyer** n = 1

Solution : Si la condition en ligne 2 est validée, l'algorithme se termine. Si le nombre n est impair, l'algorithme se termine également trivialement. Si ce n'est pas le cas, on utilise le variant de boucle $v = n$. On vérifie qu'il est bien initialement positif, à valeurs entières et qu'il décroît strictement (car divisé par deux en division entière) à chaque tour de boucle. Nécessairement, v va donc atteindre la valeur 1 (car n est pair et `2//2` vaut 1). La condition $m = 0$ est donc invalidée car $1 \mod 2 = 1$ et la boucle se termine. L'algorithme est_puissance_de_deux se termine.

```python
def is_power_of_two(n):
    m = n % 2
    while m == 0:
        n = n // 2
        m = n % 2
    return n == 1
```

A3. Prouver la terminaison de l'algorithme récursif 3 puis le traduire en Python.

Algorithme 3 Somme des n premiers entiers

Fonction INT_SUM(n)
    si n=0 **alors**
        renvoyer 0
    sinon
        renvoyer n + INT_SUM(n-1)

Solution : On procède par récurrence sur n.

Initialisation : pour $n = 0$, l'algorithme se termine en renvoyant 0.

Hérédité : On suppose que l'algorithme se termine pour le paramètre $n-1$. L'opération $n +$ int_sum$(n-1)$ n'est qu'une addition et donc se termine.

Conclusion : l'algorithme se termine pour toute valeur de n.

```python
def int_sum(n):
    if n==0:
        return 0
    else:
        return n + int_sum(n-1)
```

B Correction

B1. Prouver la correction partielle de l'algorithme 4 puis le traduire en Python en matérialisant l'invariant utilisé par des assertions.

Algorithme 4 Élément maximum d'un tableau

Fonction MAX(t)
    si t est vide **alors**
        renvoyer $\emptyset$
    sinon
        $n \leftarrow$ la taille du tableau
        $m = t[0]$
        pour $i = 1$ à $n-1$ **répéter**
            si m < t[i] **alors**
                $m \leftarrow t[i]$
        renvoyer m

Solution : On choisit l'invariant $\mathcal{I}$:*m est le plus grand élément de* $t[0:i]$.

Initialisation : à l'entrée de la boucle, $i = 0$ et $m = t[0]$. L'invariant est trivialement vérifié puisque le tableau possède un seul élément.

Hérédité : On suppose que l'invariant est vérifié pour l'itération $k-1$, c'est à dire que m est le plus grand élément de $t[0:k-1]$. À la fin de l'itération k, si $t[k]$ est plus grand que m, alors celui-ci est affecté à m. Donc, m est le plus grand élément de $t[0:k]$ à la fin de l'itération k. La propriété $\mathcal{I}$ est invariante par les instructions de la boucle.

Conclusion : $\mathcal{I}$ est vérifié à chaque itération. C'est bien un invariant de boucle. À la sortie de la boucle, on a parcouru tout le tableau, i vaut $n-1$ et m est le plus grand élément du tableau. L'algorithme est correct.

```python
def max_val(L):
    if len(L) > 0:
        maxi = L[0]
```

```python
        assert maxi == L[0]  # before the loop
        for i in range(1, len(L)):
            if L[i] > maxi:
                maxi = L[i]
            assert maxi == max(L[0:i + 1])  # loop invariant
        return maxi
    else:
        return None
```

B2. Prouver la correction partielle de l'algorithme de tri par sélection 5

Algorithme 5 Tri par sélection

```
Fonction TRIER_SELECTION(t)
    n ← taille(t)
    pour i de 0 à n − 1 répéter
        min_index ← i                          ▷ indice du prochain plus petit
        pour j de i + 1 à n − 1 répéter        ▷ pour tous les éléments non triés
            si t[j] < t[min_index ] alors
                min_index ← j                  ▷ c'est l'indice du plus petit non trié!
        échanger(t[i], t[min_index])           ▷ c'est le plus grand des triés!
```

Solution : On choisit les invariants suivants :

1. $\mathcal{I}_1$: $t[0:i]$ *est trié et ses éléments sont plus petits que les autres éléments de* t. pour la boucle sur i
2. $\mathcal{I}_2$: $t[min\_index]$ *est le plus petit élément de* $t[i:j]$. pour la boucle sur j.

On commence par prouver la correction de la boucle sur j, avec l'invariant $\mathcal{I}_2$.

Initialisation : à l'entrée de la boucle, min_index vaut i et $j = i$. t[min_index] est bien le plus petit élément de t[i].

Hérédité : supposons que l'invariant est vérifié à l'entrée de l'itération k, c'est à dire que t[min_index] est le plus petit élément de $t[i:k-1]$. À la fin de l'itération, t[min_index] est nécessairement le plus petit élément de $t[k-1:k]$ et donc de $t[i:k]$.

Conclusion : $\mathcal{I}_2$ est vérifié à chaque itération. C'est bien un invariant de boucle. À la fin de la boucle, t[min_index] est le plus petit élément de t[i:n-1].

Pour l'invariant $\mathcal{I}_1$:

Initialisation : avant la boucle, on considère le tableau pour lequel on ne prend aucun éléments. Il est vide et donc trivialement trié. Comme il ne possède pas d'éléments, l'invariant $\mathcal{I}_1$ est vérifié avant la boucle.

Hérédité : supposons que $\mathcal{I}_1$ soit vérifié à l'entrée de la kième itération. Alors $t[0:k-1]$ est correctement trié et tous les éléments du restant du tableau (à droite de $k-1$) sont plus grands que $t[k-1]$. Le minimum du restant du tableau de droite restant est alors placé à l'indice k. Comme il est plus grand que $t[k-1]$, le tableau $t[0:k]$ est correctement trié. L'invariant $\mathcal{I}_1$ n'est pas modifié par les instructions de la boucle.

Conclusion : L'invariant $\mathcal{I}_1$ est vérifié pour toutes les itérations de la boucle. À la fin de la boucle, i vaut $n-1$ et donc t[0:n-1], c'est à dire l'entièreté du tableau, est trié. L'algorithme de tri est correct.

```python
def swap(t, i, j):
    t[i], t[j] = t[j], t[i]


def selection_sort(t):
    assert is_sorted(t[0:0])  # Invariant Ii -> t[0:0] is empty and sorted !
    for i in range(len(t)):
        min_index = i
        for j in range(i + 1, len(t)):
            if t[j] < t[min_index]:
                min_index = j
            for k in range(i, j+1):  # invariant Ij
                assert t[min_index] <= t[k]  # Invariant Ij
        swap(t, i, min_index)
        assert is_sorted(t[0:i + 1])  # Invariant Ii
        for k in range(i + 1, len(t)): # Invariant Ii
            assert t[i] <= t[k]  # Invariant Ii
```

B3. Prouver la correction de l'algorithme du tri par insertion 6.

Algorithme 6 Tri par insertion

Fonction INSERTION(t, i)
    à_insérer ← t[i]
    j ← i
    tant que t[j-1] > à_insérer et j>0 **répéter**
        t[j] ← t[j-1] ▷ faire monter les éléments
        j ← j-1
    t[j] ← à_insérer ▷ insertion de l'élément
Fonction TRIER_INSERTION(t)
    n ← taille(t)
    pour i de 1 à n-1 **répéter**
        INSERTION(t,i)

Solution : On choisit d'abord de prouver la correction de l'algorithme d'insertion.

La terminaison de la fonction est garantie par j qui est un variant de la boucle tant que.

On utilise l'invariant suivant pour la correction de la boucle tant que $\mathcal{I}$: *le tableau t[0:i-1] est correctement trié.*

Initialisation : avant la boucle, j vaut i. Le tableau t[0:i-1] est supposé trié.

Hérédité : supposons que l'invariant est vérifié à l'entrée d'une certaine itération : t[0,i-1] est trié et on a $t[j] = t[j+1]$. À la fin de l'itération, on a fait monter (recopie) l'élément j-1 en j. Le tableau t[0,i-1] est toujours trié et $t[j-1] = t[j]$. L'invariant n'est pas modifié par ces instructions.

Conclusion : $\mathcal{I}$ est donc vérifié à chaque itération. C'est bien un invariant de boucle. À la fin de la boucle, on a t[j-1] < à_insérer et $t[j] = t[j+1]$. L'élément à_insérer se voit attribuer la place j : il n'écrase aucune valeur du tableau puisqu'on les a décalées. Cet élément est à sa place. Le tableau t[0:i] est donc correctement trié, l'insertion est correcte.

Pour la correction de la fonction *trier_insertion*, on choisit l'invariant de boucle suivant : $\mathcal{I}$: *à chaque itération, le tableau t[0:i] est trié.*

Initialisation : avant la boucle, i vaut 0 et t[0] est un tableau trivialement trié.

Hérédité : supposons que l'invariant est vérifié pour l'itération $k-1$: t[0,k-1] est trié. À la fin de l'itération k, comme la fonction d'insertion est correcte, $t[0:k]$ est correctement trié.

Conclusion : $\mathcal{I}$ est vérifié à chaque itération. C'est bien un invariant de boucle. À la fin de la boucle, on a parcouru tous les éléments du tableau et i vaut $n-1$. t est donc complètement trié. L'algorithme est donc correct.

```python
def is_sorted(t):
    if len(t) == 0:
        return True
    else:
        for i in range(1, len(t)):
            if t[i - 1] > t[i]:
                return False
        return True


def insert(t, i):
    to_insert = t[i]
    j = i
    assert is_sorted(t[0:i])  # before loop
    while t[j - 1] > to_insert and j > 0:
        t[j] = t[j - 1]
        j -= 1
        assert is_sorted(t[0:i + 1])  # invariant
        assert t[j] == t[j + 1]  # invariant
    t[j] = to_insert


def insertion_sort(t):
    assert is_sorted(t[0:0])  # before loop
    for i in range(1, len(t)):
        insert(t, i)
        assert is_sorted(t[0:i + 1])  # loop invariant
```

C Algorithme d'Euclide du PGCD

Algorithme 7 Algorithme d'Euclide (optimisé)

```
Fonction PGCD(a, b)                  ▷ On suppose pour simplifier que a ∈ ℕ, b ∈ ℕ* et b ⩽ a.
    r ← a mod b
    tant que r > 0 répéter           ▷ On connaît la réponse si r est nul.
        a ← b
        b ← r
        r ← a mod b
    renvoyer b                       ▷ Le pgcd est b
```

On cherche à prouver la terminaison et la correction de l'algorithme d'Euclide 7. Dans ce but, on rappelle quelques éléments mathématiques importants.

Théorème 1 — Division euclidienne . Soient $a \in \mathbb{Z}$ et $b \in \mathbb{N}^\star$. Alors il existe un unique couple $(q,r) \in \mathbb{Z} \times \mathbb{N}$ tel que les deux critères suivants sont vérifiés :

$$\begin{cases} a = bq + r \\ 0 \leqslant r < b \end{cases}$$

Démonstration.

1. Existence : a et b étant donné, on pose $q = \lfloor \frac{a}{b} \rfloor$. Par définition de partie entière, on a : $0 \leqslant \frac{a}{b} - \lfloor \frac{a}{b} \rfloor < 1$. En multipliant par b, on obtient : $0 \leqslant a - b \times \lfloor \frac{a}{b} \rfloor < b$. En choisissant donc $q = \lfloor \frac{a}{b} \rfloor$ et $r = a - b \times \lfloor \frac{a}{b} \rfloor$, on a bien :
$$\begin{cases} a = bq + r \\ 0 \leqslant r < b \end{cases}$$
2. Unicité : supposons que l'on ait deux couples (q,r) et (q',r') appartenant à $\mathbb{Z} \times \mathbb{N}$: $a = bq + r = bq' + r'$ avec $0 \leqslant r < b$ et $0 \leqslant r' < b$. Cela peut également s'écrire : $b(q'-q) = r - r'$. Or, on a l'encadrement $-b < r - r' < b$. On en conclut que $-b < b(q'-q) < b$ et donc que $-1 < q' - q < 1$. Mais q et q' sont des entiers d'après nos hypothèses de départ. Donc, on en déduit de $q' - q = 0$. Il s'en suit que $q = q'$ et que $r = r'$. Il s'agit donc bien du même couple.

■

Théorème 2 — Existence du PGCD. Parmi tous les diviseurs communs de deux entiers a et b non nuls, il y en a **un** qui est le plus grand. Ce dernier est nommé plus grand commun diviseur de a et de b. On le note $\text{PGCD}(a,b)$.

Démonstration. Soit $a \in \mathbb{N}^\star$. Tous les diviseurs de a sont bornés par $|\, a \,|$. On peut tenir le même raisonnement pour ceux de b. Donc, parmi les diviseurs de a et de b, il y en a donc un plus grand. ■

Théorème 3 — Propriété du PGCD. Soit a et b deux entiers.

1. Si $b = 0$, alors $\text{PGCD}(a,b) = a$.
2. Si $b \neq 0$, alors $\text{PGCD}(a,b) = \text{PGCD}(b, a \mod b)$.

Démonstration. Démonstration de l'égalité de l'ensemble $\mathcal{D}_{ab}$ des diviseurs de a et de b et de l'ensemble $\mathcal{D}_{br}$ des diviseurs de b et de r par double inclusion.

$\mathcal{D}_{ab} \subset \mathcal{D}_{br}$: La division euclidienne étant unique comme nous l'avons montré au théorème 1, il existe un entier q tel que $a = qb + r$. Ce qui peut s'écrire : $a - qb = r$. Si γ est un diviseur de a et de b, alors on peut écrire : $a - bq = \gamma a' + \gamma b' q = \gamma(a' - b'q) = r$. On a donc montrer qu'un diviseur de a et de b est un diviseur de r.

$\mathcal{D}_{br} \subset \mathcal{D}_{ab}$: De même, si η est un diviseur de b et de r, alors on a : $a = bq + r = \eta(b'q + r')$, ce qui signifie que η est un diviseur de a.

Donc, $\mathcal{D}_{ab} = \mathcal{D}_{br}$. Ceci est vrai, y compris pour le plus grand des diviseurs de a et de b. ■

■ **Définition 1 — Suite des restes de la division euclidienne.** Soient a et b des entiers. On définit la suite des restes de la division euclidienne comme suit :

$$r_0 = |\,a\,| \tag{1}$$
$$r_1 = |\,b\,| \tag{2}$$
$$q_k = \lfloor r_{k-1}/r_k \rfloor, 1 \leqslant k \leqslant n \tag{3}$$

Alors on a :

$$r_{k-1} = q_k r_k + r_{k+1} \tag{4}$$
$$r_{k+1} = r_{k-1} \bmod r_k \tag{5}$$

Théorème 4 — Stricte décroissance de $(r_n)_{n\in\mathbb{N}}$. La suite des restes de la division euclidienne est positive, strictement décroissante et minorée par zéro.

C1. Coder l'algorithme 7 en Python.

Solution :

```python
def pgcd(a, b):
    r = a % b
    assert rec_pgcd(a, b) == rec_pgcd(b, r) #  invariant (before loop)
    while r > 0:
        a = b
        b = r
        r = a % b
        assert 0 <= r < b # loop variant
        assert rec_pgcd(a,b) == rec_pgcd(b,r)  #  invariant inside loop
    return b
```

C2. Grâce au théorème 3, coder une version récursive de l'algorithme du PGCD.

```python
def rec_pgcd(a, b):
    if b == 0:
        return a
    else:
        return rec_pgcd(b, a % b)
```

C3. Donner une preuve du théorème 4.

> **Solution :** D'après le théorème 1, le reste r de la division euclidienne de a et de b est tel que : $0 \leqslant r < b$. Donc, la suite est minorée par zéro. Cette borne est atteinte lorsque r_k est un multiple de r_{k-1}. C'est une suite positive car elle est initialisée à des valeurs positives. Elle est strictement décroissante car $r_{k-1} < r_k$ d'après la définition de la division euclidienne 1.

C4. Montrer que r est un variant de boucle pour l'algorithme d'Euclide.

> **Solution :** On observe qu'un élément de la suite des restes est calculé à chaque tour de boucle (cf. algorithme 7 ligne 6). D'après la question précédente, r est positif, **strictement** décroissant et minoré par zéro. r est donc un variant de boucle. La condition d'arrêt, $r > 0$, est donc invalidée au bout d'un certain nombre d'itérations. Le programme se termine.

C5. Prouver la correction de l'algorithme d'Euclide.

> **Solution :** On choisit l'invariant $\mathcal{I}$: *le PGCD de b et r est le PGCD de a et de b*.
>
> **Initialisation :** L'invariant est vérifié à l'entrée de la boucle car $r = a \mod b$ et d'après le point 2 du théorème 3 on a $\text{PGCD}(a, b) = \text{PGCD}(b, a \mod b)$.
>
> **Hérédité :** Si l'invariant est vérifié à l'entrée de la boucle, la propriété du PGCD fait qu'il est vérifié à la fin de la boucle.
>
> **Conclusion :** À la sortie de la boucle, le reste est nul (d'après la démonstration de la terminaison) et la propriété du PGCD nous indique que le PGCD de b et r est le PGCD recherché. Or $\text{PGCD}(b, 0) = b$. Le PGCD vaut donc b, ce que renvoie la fonction. L'algorithme est correct.