

# 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 pose  $k = i - j$  et on utilise le variant de boucle  $v(k) = k$ . 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  $j$  décrétementé 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.

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

```

    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  $n = 0$  alors
3:     renvoyer Faux
4:   sinon
5:      $m \leftarrow n \bmod 2$ 
6:     tant que  $m = 0$  répéter
7:        $n \leftarrow n // 2$ 
8:        $m \leftarrow n \bmod 2$ 
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).

Si  $n = 2^k$ , alors  $v$  va décroître jusqu'à atteindre la valeur  $2/2 = 1$ . Pour cette valeur,  $m = 1\%2 = 1$  et la condition  $m = 0$  est donc invalidée. La boucle se termine.

Si  $n$  est pair mais n'est pas une puissance de deux.  $v$  décroît jusqu'à ce que les puissances de deux sont épuisées. Il ne reste alors plus que des facteurs premiers impairs et  $m$  n'est plus égal à zéro. La condition de boucle est donc invalidée.

L'algorithme est\_puissance\_de\_deux se termine.

```

def is_power_of_two(n):
    if n == 0:
        return False
    else:
        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

---

```

1: Fonction INT_SUM(n)
2:   si n=0 alors
3:     renvoyer 0
4:   sinon
5:     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  $n + \text{int\_sum}(n - 1)$  n'est qu'une addition et donc se termine.

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

```

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

```

---

A4. Prouver la terminaison de l'algorithme récursif 4.

**Algorithme 4** Exponentiation rapide  $a^n$ 


---

```

1: Fonction EXP_RAPIDE(a,n)
2:   si n = 0 alors                                     ▷ Condition d'arrêt
3:     renvoyer 1
4:   sinon si n est pair alors
5:      $p \leftarrow \text{EXP\_RAPIDE}(a, n//2)$                  ▷ Appel récursif
6:     renvoyer  $p \times p$ 
7:   sinon
8:      $p \leftarrow \text{EXP\_RAPIDE}(a, (n-1)//2)$              ▷ Appel récursif
9:     renvoyer  $p \times p \times a$ 

```

---

**Solution :** La suite des paramètres des appels récursif  $(u_n)_{n \in \mathbb{N}}$  définie par :

$$u_n = \begin{cases} n & \text{si } n = 0 \\ \frac{u_{n-1}}{2} & \text{sinon} \end{cases} \quad (1)$$

est une suite d'éléments positifs qui décroît strictement et qui est minorée par 0. Par conséquent, la condition d'arrêt est nécessairement atteinte et l'algorithme termine.

## B Algorithme d'Euclide du PGCD

---

### Algorithme 5 Algorithme d'Euclide (optimisé)

---

```

1: Fonction PGCD( $a, b$ )                                ▷ On suppose pour simplifier que  $a \in \mathbb{N}$ ,  $b \in \mathbb{N}^*$  et  $b \leq a$ .
2:    $r \leftarrow a \bmod b$ 
3:   tant que  $r > 0$  répéter                               ▷ On connaît la réponse si  $r$  est nul.
4:      $a \leftarrow b$ 
5:      $b \leftarrow r$ 
6:      $r \leftarrow a \bmod b$ 
7:   renvoyer  $b$                                              ▷ Le pgcd est  $b$ 

```

---

On cherche à prouver la terminaison et la correction de l'algorithme d'Euclide 5. Dans ce but, on rappelle quelques éléments mathématiques importants.

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

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

*Démonstration.* 1. Existence :  $a$  et  $b$  étant donné, on pose  $q = \lfloor \frac{a}{b} \rfloor$ . Par définition de partie entière, on a :  $0 \leq \frac{a}{b} - \lfloor \frac{a}{b} \rfloor < 1$ . En multipliant par  $b$ , on obtient :  $0 \leq a - b \times \lfloor \frac{a}{b} \rfloor < b$ . En choisissant donc  $q = \lfloor \frac{a}{b} \rfloor$  et  $r = a - b \times \lfloor \frac{a}{b} \rfloor$ , on a bien :

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

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  $0 \leq r < b$  et  $0 \leq r' < b$ . 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  $a \in \mathbb{N}^*$ . 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  $\text{PGCD}(a, b) = \text{PGCD}(b, a \bmod b)$ .

*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  $\gamma$  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 montré qu'un diviseur de  $a$  et de  $b$  est un diviseur de  $r$ .

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

$$r_0 = |a| \quad (2)$$

$$r_1 = |b| \quad (3)$$

$$q_k = \lfloor r_{k-1} / r_k \rfloor, 1 \leq k \leq n \quad (4)$$

Alors on a :

$$r_{k-1} = q_k r_k + r_{k+1} \quad (5)$$

$$r_{k+1} = r_{k-1} \bmod r_k \quad (6)$$

**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.

B1. Coder l'algorithme 5 en Python.

**Solution :**

```
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
```

B2. Grâce au théorème 3, coder une version récursive de l'algorithme du PGCD.

**Solution :**

```
def rec_pgcd(a, b):
    if b == 0:
```

```

    return a
else:
    return rec_pgcd(b, a % b)

```

B3. 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 :  $0 \leq r < b$ . 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.

B4. 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 5 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.

B5. 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  $r = a \bmod b$  et d'après le point 2 du théorème 3 on a  $\text{PGCD}(a, b) = \text{PGCD}(b, a \bmod b)$ .

**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 :**  $\mathcal{I}$  n'a pas été modifié par les instructions de la boucle. Comme il est vérifié à l'entrée de la boucle, il est donc vérifié également à la fin de celle-ci. 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.

## C Correction d'algorithmes classiques

C1. Prouver la correction partielle de l'algorithme 6 puis le traduire en Python en matérialisant l'invariant utilisé par des assertions.

**Solution :** On choisit l'invariant  $\mathcal{I}$  :  *$m$  est le plus grand élément de  $t[0 : i - 1]$ .*

**Initialisation :** à l'entrée de la boucle,  $i = 1$  et  $m = t[0]$ . L'invariant est trivialement vérifié puisque le tableau  $t[0 : i - 1] = t[0 : 0] = t[0] = m$ .

**Hérédité :** On suppose que l'invariant est vérifié pour l'itération  $k$ , 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

**Algorithme 6** Élément maximum d'un tableau

```

1: Fonction MAX( $t$ )
2:   si  $t$  est vide alors
3:     renvoyer  $\emptyset$ 
4:   sinon
5:      $n \leftarrow$  la taille du tableau
6:      $m = t[0]$ 
7:     pour  $i = 1$  à  $n - 1$  répéter
8:       si  $m < t[i]$  alors
9:          $m \leftarrow t[i]$ 
10:    renvoyer  $m$ 

```

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 l'entrée de la boucle et est invariant par les instructions de la 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.

```

def max_val(L):
    if len(L) > 0:
        maxi = L[0]
        for i in range(1, len(L)):
            assert maxi == max(L[0:i]) # invariant
            if L[i] > maxi:
                maxi = L[i]
        return maxi
    else:
        return None

```

## C2. Prouver la correction partielle de l'algorithme de tri par sélection 7

**Solution :** On choisit les invariants suivants :

1.  $\mathcal{I}_1 : t[0 : i - 1]$  est trié et  $t[i - 1]$  est plus petit que tous les éléments de  $t[i : n - 1]$ . pour la boucle de la fonction TRIER\_SELECTION.
2.  $\mathcal{I}_2 : t[\text{min\_index}]$  est le plus petit élément de  $t[i : j - 1]$ . pour la boucle de la fonction GET\_MIN.

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

**Initialisation :** à l'entrée de la boucle,  $\text{min\_index}$  vaut  $i$  et  $j = i + 1$ .  $t[\text{min\_index}]$  est bien le plus petit élément de  $t[i : i] = t[i]$ .

**Hérédité :** supposons que l'invariant est vérifié à l'entrée de l'itération  $k$ , c'est à dire que  $t[\text{min\_index}]$  est le plus petit élément de  $t[i : k - 1]$ . À la fin de l'itération, si  $t[k]$  est plus petit que  $t[\text{min\_index}]$ , alors  $k$  est affecté à  $\text{min\_index}$ . Alors  $t[\text{min\_index}]$  est nécessairement le plus petit élément de  $t[k - 1 : k]$  et donc de  $t[i : k]$ .

**Algorithme 7** Tri par sélection

---

```

1: Fonction GET_MIN_INDEX(t, i)
2:    $n \leftarrow \text{taille}(t)$ 
3:   min_index  $\leftarrow i$ 
4:   pour  $j$  de  $i + 1$  à  $n - 1$  répéter
5:     si  $t[j] < t[\text{min\_index}]$  alors
6:       min_index  $\leftarrow j$ 
7:   renvoyer min_index
8: Fonction TRIER_SELECTION(t)
9:    $n \leftarrow \text{taille}(t)$ 
10:  min_index  $\leftarrow$  GET_MIN_INDEX(t, 0)
11:  échanger(t, 0, min_index)
12:  pour  $i$  de 1 à  $n - 1$  répéter
13:    min_index  $\leftarrow$  GET_MIN_INDEX(t, i)
14:    échanger(t, i, min_index)
```

▷ Initialisation : le plus petit en tête  
 ▷ indice du prochain plus petit  
 ▷ c'est le plus grand des triés!

---

**Conclusion :**  $\mathcal{I}_2$  est vérifié à l'entrée de la boucle et est invariant par les instructions de la boucle. À la fin de la boucle,  $j = n - 1$  et donc  $t[\text{min\_index}]$  est le plus petit élément de  $t[i:n-1]$ .

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

**Initialisation :** à l'entrée de la boucle  $i = 1$ .  $t[0 : i - 1] = t[0]$  est trivialement trié et  $t[0]$  est plus petit que tous les éléments de  $t[1:n-1]$ , d'après l'initialisation et la correction de la fonction GET\_MIN.

**Hérédité :** supposons que  $\mathcal{I}_1$  soit vérifié à l'entrée de la  $k$ ème itération. Alors  $t[0 : k - 1]$  est correctement trié. Tous les éléments du restant du tableau (à droite de  $k - 1$ ) sont plus grands que  $t[k - 1]$  puisqu'on a pris le minimum à chaque fois. Le minimum du tableau de droite 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 à l'entrée de la boucle et est invariant par les instructions de la boucle. À la fin de la boucle,  $i$  vaut  $n - 1$ .  $t[n-1]$  est le plus grand éléments de  $t[n - 1]$  c'est aussi le plus grand élément du tableau d'après l'invariant. Donc  $t[0:n-1]$ , c'est à dire l'entièreté du tableau, est trié. L'algorithme de tri est correct.

```

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

def get_min_index(t, start):
    min_index = start
    for j in range(start + 1, len(t)):
        assert t[min_index] == min(t[start:j]) # invariant
        if t[j] < t[min_index]:
            min_index = j
    return min_index

def selection_sort(t):
    # Place min at first place
    min_index = get_min_index(t, 0)
```



```

swap(t, 0, min_index)
for i in range(1, len(t)):
    assert is_sorted(t[0:i]) and t[i - 1] <= min(t[i: len(t)]) #
        Invariant
    min_index = get_min_index(t, i)
    swap(t, i, min_index)

```

C3. Prouver la correction de l'algorithme du tri par insertion 8.

#### Algorithme 8 Tri par insertion

```

1: Fonction INSERTION(t, i)
2:   à_insérer ← t[i]
3:   j ← i
4:   tant que t[j-1] > à_insérer et j>0 répéter
5:     t[j] ← t[j-1]                                > faire monter les éléments
6:     j ← j-1
7:   t[j] ← à_insérer                                > insertion de l'élément
8: Fonction TRIER_INSERTION(t)
9:   n ← taille(t)
10:  pour i de 1 à n-1 répéter
11:    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 l'entrée de la boucle et est invariant par les instructions de la boucle. À la fin de la boucle, on a  $t[j-1] < \text{à\_insérer}$  et  $t[j] = t[j+1]$ . L'élément  $\text{à\_insérer}$  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 :  $\mathcal{J}$  : *à chaque itération, le tableau  $t[0:i-1]$  est trié.*

**Initialisation :** à l'entrée de la boucle,  $i-1$  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 :**  $\mathcal{J}$  est vérifié à l'entrée de la boucle et n'est pas modifié par les instructions de la boucle. 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.

```
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
```

---