TD Terminaison et correction des algorithmes

DIU EIL – UE 2

23 avril 2020

1 Multiplication

```
1  def mult(a, b):
2      assert b >= 0
3      r = 0
4      while b > 0:
5          b -= 1
6          r += a
7      return r
```

1. Montrer que `b` ≥ 0 est vrai en tout point du programme à partir de la ligne 3.
2. Montrer que le programme `mult` termine quels que soient les arguments `a` et `b`.
3. Proposer une spécification pour le programme `mult`.
4. Proposer un invariant pour la boucle `while`, et montrer que c'est bien un invariant.
5. Montrer que le programme est fortement correct vis-à-vis de la spécification de la question 3.

Correction:

1. On montre que `b` ≥ 0 est un invariant de la boucle `while` : lorsque l'on arrive dans la boucle la première fois c'est vrai grâce à la ligne 2, puis dans le corps de la boucle on a `b` > 0 jusqu'à la ligne 5 qui donne `b` ≥ 0 jusqu'à la sortie de la boucle.

   On en déduit donc qu'en tout point du programme (à partir de la ligne 3), on a `b` ≥ 0.
2. Si `b` < 0, le programme termine à la ligne 2. Sinon la valeur de `b` est un variant de la boucle `while` : c'est un entier positif ou nul qui décroît strictement (ligne 5) à chaque tour de boucle.
3. Pré-condition : $P(a, b) := b \geq 0$.

   Post-condition : $Q(a, b, r) := r = a \times b$.
4. On considère l'invariant $I(a, b, r)$: « $r = \texttt{a}_{\texttt{init}} \times (\texttt{b}_{\texttt{init}} - b)$ et $a = \texttt{a}_{\texttt{init}}$ ».
   - Lors du premier accès à la boucle `while`, on a `r` $= 0$, `a` $= \texttt{a}_{\texttt{init}}$ et `b` $= \texttt{b}_{\texttt{init}}$ donc `r` $= 0 = \texttt{a}_{\texttt{init}} \times (\texttt{b}_{\texttt{init}} - \texttt{b})$.
   - Lors de l'exécution du corps de la boucle on effectue les transformations suivantes :

	avant	après
`a`	a	a
`b`	b	$b - 1$
`r`	r	$r + a$

   Avant le corps de la boucle, la validité de l'invariant donne : $r = \texttt{a}_{\texttt{init}} \times (\texttt{b}_{\texttt{init}} - b)$. On en déduit que : $r + a = r + \texttt{a}_{\texttt{init}} = \texttt{a}_{\texttt{init}} \times (\texttt{b}_{\texttt{init}} - (b - 1))$. Et on a toujours `a` $= a = \texttt{a}_{\texttt{init}}$, donc l'invariant est préservé.
5. Lorsque l'on quitte le programme, on vient de sortir de la boucle donc `b` ≤ 0. Comme on sait que `b` ≥ 0 (question 1), on a `b` $= 0$. L'invariant de la boucle nous donne alors : `r` $= \texttt{a}_{\texttt{init}} \times (\texttt{b}_{\texttt{init}} - \texttt{b}) = \texttt{a}_{\texttt{init}} \times \texttt{b}_{\texttt{init}}$.

   C'est-à-dire que si on exécute le programme avec les arguments `a` et `b` tels que `b` ≥ 0 (pré-condition $P(\texttt{a}, \texttt{b})$), alors le programme termine (question 2) et le résultat `r` vérifie `r` $= \texttt{a} \times \texttt{b}$ (post-condition $Q(\texttt{a}, \texttt{b}, \texttt{r})$).

   On peut noter que si on omet la pré-condition, le programme peut terminer sur une exception plutôt que sur un résultat valide. Si on omet la pré-condition et la ligne 2, on a `mult(2,-1)` $= 0$: la post-condition n'est pas vérifiée. L'invariant `r` $= \texttt{a}_{\texttt{init}} \times (\texttt{b}_{\texttt{init}} - \texttt{b})$ reste valide mais on quitte le programme avec `b` $= \texttt{b}_{\texttt{init}}$ au lieu de `b` $= 0$ du fait que la boucle `while` n'a pas été exécutée.

2 Somme

```
1  def sommer(n):
2      s = 0
3      for i in range(n):
4          s += i
5      return s
```

1. Montrer que le programme `sommer` termine quel que soit l'argument `n`.
2. Que calcule ce programme ?
3. Prouvez-le !

Correction:

1. Le programme ne contient pas d'appel récursif et une unique boucle `for` dont le corps ne modifie ni l'indice de boucle ni les bornes.
   `i` est un variant : entier (strictement) plus petit que `n` et strictement croissant à chaque nouveau passage dans la boucle (incrémenté de 1 par le `for`).
   Si on préfère les variants pris dans $(\mathbb{N}, \leq)$, `n - i` est un tel variant.
2. On a `sommer(n)` $= \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$.
3. On considère l'invariant $I(i, s) := s = \frac{i(i+1)}{2}$ en fin de boucle `for` (fin de ligne 4).
   Au premier passage à la fin de la ligne 4, on a : `s` $= 0$ et `i` $= 0$ donc l'invariant est vérifié.
   Si l'invariant est vérifié en sortie de boucle `for`, au passage suivant, on aura effectué :

	avant	après
`i`	i	$i+1$
`s`	s	$s+i+1$

   donc si $s = \frac{i(i+1)}{2}$, on a bien $s + i + 1 = \frac{i(i+1)}{2} + i + 1 = \frac{i(i-1)+2(i+1)}{2} = \frac{(i+2)(i+1)}{2}$ et l'invariant est préservé.
   Lorsque l'on quitte le programme, on a effectué un dernier passage dans la boucle `for` avec `i` $= n - 1$, l'invariant lors de cette sortie de boucle nous donne : `s` $= \frac{(n-1)n}{2}$, qui est la valeur retournée par le programme.

3 Division

```
1  def div(a, b):
2      r = a
3      q = 0
4      if b == 0:
5          raise ZeroDivisionError("division by zero")
6      else:
7          while r >= b:
8              r -= b
9              q += 1
10         return (q, r)
```

1. En utilisant un variant pour la boucle `while`, déterminer si ce programme termine pour toutes les valeurs de `a` et `b`.
2. Ce programme est-il faiblement correct pour la division euclidienne ? Expliquer.
3. Donner des pré-conditions pour assurer la correction.
4. Donner une version améliorée de ce programme qui se termine et est correcte pour toutes valeurs de `a` et `b`.

Correction:

Notons que `a` et `b` ne sont pas modifiés dans le programme. Leur valeur est donc toujours $\texttt{a}_{\texttt{init}}$ et $\texttt{b}_{\texttt{init}}$.

1. On retire `b` à `r` à chaque tour de boucle `while`. Si `b` > 0 cela fait décroître strictement `r`. Par hypothèse du `if`, `b` $\neq 0$. Par contre si `b` < 0, on ne peut rien dire grâce à ça.
   On peut montrer que :
   - si `b` $= 0$, le programme termine toujours par une exception ;
   - si `b` > 0, le programme termine toujours car `r` est un variant de la boucle : `r` $\geq \min(0, \texttt{a})$ et décroît strictement (de `b`) à chaque tour de boucle ;
   - si `a` $<$ `b` < 0, le programme termine (et répond `(0, a)`) ;

- si b < 0 et a $\geq$ b, le programme boucle : lorsque l'on arrive à la boucle `while` pour la première fois, r $=$ a $\geq$ b donc on rentre dans la boucle. Ensuite la ligne 8 fait augmenter r strictement donc la condition de boucle reste vérifiée indéfiniment, et le programme ne s'arrête pas.

2. On a vu que si a $<$ b < 0, le programme termine avec `div(a, b)` $=$ `(0, a)`.
   Quelques éléments concernant la division euclidienne pour les entiers relatifs :

   `http://python-history.blogspot.com/2010/08/why-pythons-integer-division-floors.html`

   En particulier, on voudrait donc (en suivant les conventions de PYTHON) `div(-7, -3)` $=$ `(-3, -2)`. On n'a pas la correction faible.

3. Si on cherche la correction faible, on pourrait considérer des valeurs b < 0 à condition que a $\geq$ b : la non-terminaison ne contredit jamais la correction faible.
   On va plutôt éviter ce comportement pathologique du programme et viser la correction forte avec comme pré-condition b > 0. Attention ce n'est pas suffisant : si a $<$ b, on ne rentre pas dans la boucle et on retourne `(0, a)` ce qui n'est une valeur correcte que si a ≥ 0 (sinon on n'a pas la condition de reste $0 \leq$ a $<$ b).
   On se fixe donc comme pré-conditions : b > 0 et a ≥ 0. On prouve l'invariant de boucle suivant : a $=$ b $\times$ q $+$ r. Il est vrai à la première entrée dans la boucle : a $=$ b $\times 0 +$ a. Et d'un passage de boucle au suivant, on a (a et b ne sont pas modifiés) :

	avant	après
q	q	$q+1$
r	r	$r-$ b

   donc si a $=$ b $\times q + r$, on a bien a $=$ b $\times (q+1) + r -$ b ensuite.
   Lorsque l'on quitte le programme, on a donc a $=$ b $\times$ q $+$ r, mais de plus (sortie de boucle) r $<$ b (et on a déjà vu que r ≥ 0 dans tout le programme si a ≥ 0). Ceci permet de conclure que le programme est correct pour la spécification de la division euclidienne.

4. En considérant différents cas selon que a et b sont positifs ou négatifs, on peut obtenir :

```python
def div(a, b):
    r = a
    q = 0
    if b == 0:
        raise ZeroDivisionError("division by zero")
    elif b > 0:
        while r >= b:
            r -= b
            q += 1
        while r < 0:
            r += b
            q -= 1
        return (q, r)
    else:
        while r <= b:
            r -= b
            q += 1
        while r > 0:
            r += b
            q -= 1
        return (q, r)
```

   On maintient l'invariant a $=$ b $\times$ q $+$ r et on vérifie que les conditions de fin de boucles garantissent que $0 \leq$ r $<$ b si b > 0 et que b $<$ r ≤ 0 si b < 0. On peut vérifier également que toutes les boucles terminent.

4 Aléa

```python
def fonction(l):
    debut = 0
    fin = len(l) - 1
    resultat = []
    while debut <= fin:
        if random.randint(0, 2) == 1:
            resultat.append(l[debut])
            debut += 1
        else:
            resultat.append(l[fin])
            fin -= 1
    return resultat
```

Montrer que cette fonction termine toujours et décrire ce qu'elle fait.

Correction:

On montre que `fin` $-$ `debut` est un variant de la boucle `while` : il est toujours supérieur à -1 et décroît strictement à chaque passage dans la boucle, soit à la ligne 8 soit à la ligne 11. La particularité de ce programme est qu'on ne sait pas laquelle de ces deux lignes va être exécutée à chaque passage, mais on est sûrs malgré tout que l'une des deux le sera.

Le programme ajoute les éléments de `l` dans une nouvelle liste, en les sélectionnant aléatoirement soit au début de `l` soit à la fin de `l`.

5 Tri par sélection

Le principe du tri par sélection d'un tableau `tab` de longueur n est le suivant. Pour chaque élément `i` de 0 à $n-2$, on échange `tab[i]` avec l'élément minimum de `tab[i:]`. Nous devons donc rechercher, pour chaque itération, le minimum d'un sous-tableau de plus en plus petit. Les éléments à gauche de `i` sont à leur place définitive et donc le tableau est complètement trié lorsque `i` arrive sur l'avant dernier élément (le dernier élément étant forcément le plus grand).

Voici un code PYTHON possible :

```
1  def tri_selection(tab):
2      for i in range(0, len(tab) - 1):
3          indmin = i
4          for j in range(i + 1, len(tab)):
5              if tab[j] < tab[indmin]:
6                  indmin = j
7          tab[indmin], tab[i] = tab[i], tab[indmin]
```

1. Justifier brièvement la complexité de cet algorithme de tri.
2. Donner un invariant pour chacune des deux boucles et montrer qu'ils sont corrects.
3. En conclure que l'algorithme est bien un algorithme de tri correct.

Correction:

1. On voit que l'on effectue une boucle sur tous les éléments de `tab` (moins le dernier). Et pour chaque itération la recherche du minimum demande, dans le pire des cas, de parcourir tous les éléments du sous-tableau. Nous avons donc une boucle sur `i` en $O(n)$ et une boucle sur `j` en $O(n)$ pour chaque. La complexité totale est donc $O(n^2)$.
2. En sortie de la boucle `for` sur `j`, on a : `indmin` contient l'indice d'un minimum de `tab[i:j+1]`.

   En sortie de la boucle `for` sur `i`, on a :
   - les éléments de `tab` sont les mêmes que ceux de $\texttt{tab}_{\texttt{init}}$;
   - les éléments de `tab[:i+1]` sont triés ;
   - les éléments de `tab[i+1:]` sont tous plus grands que `tab[i]`.

   Lors d'un premier passage dans la boucle `for j`, `j` $=$ `i` $+1$ et `indmin` $=$ `i`. Lorsque l'on quitte cette boucle, si `tab[i+1]` $<$ `tab[i]` alors `indmin` $=$ `i` $+1$, et sinon `indmin` $=$ `i`. `indmin` contient donc bien l'indice d'un minimum de `tab[i:i+2]`.

   Supposons maintenant que l'invariant est vérifié en fin de boucle `for j`. Au passage suivant en fin de boucle, soit `tab[j]` contient un plus petit élément que `tab[indmin]` et `indmin` est mis à jour, soit `indmin` n'est pas modifié. L'invariant est donc préservé.

   Concernant la boucle `for i`, en fin de premier passage, on a `i` $=0$, `tab[0]` contient l'élément minimum de $\texttt{tab}_{\texttt{init}}$ (contenu dans $\texttt{tab}_{\texttt{init}}$`[indmin]`), et l'élément contenu dans $\texttt{tab}_{\texttt{init}}$`[0]` a été mis dans `tab[indmin]`. L'invariant est vérifié.

   Si l'invariant est vérifié à une fin de boucle `for i`, à l'étape suivante : on n'a fait que permuter des éléments donc l'ensemble des éléments n'a pas changé. Le plus petit élément de `tab[i:]` (dont tous les éléments étaient plus grands que `tab[i-1]`) a été échangé avec `tab[i]`. On en déduit que `tab[:i+1]` est trié et que les éléments de `tab[i+1:]` sont tous plus grands que `tab[i]`.
3. Lorsque l'on quitte le programme, on a `i` $=$ `len(tab)` -2, et l'invariant de la boucle `for i` est valide. On a donc : les éléments n'ont pas été modifiés, les éléments de `tab[:len(tab)-1]` sont triés, et l'élément `tab[len(tab)-1]` est plus grands que `tab[len(tab)-2]`. Ceci signifie que `tab` est trié.