CM 4 : Invariant, Variant

Info1.Algo1

2022-2023 Semestre Impair

Plan

Introduction

Cadre du problème

Pour un problème dont on nous fournit les exigences, on souhaite répondre aux questions suivantes :

- Quel est le problème (**analyse**) ?
- **Comment** construire l'algorithme de la boucle ?
- Comment **vérifier** que le code développé est correct en répondant aux exigences?

La première étape consiste en la **spécification** de la fonction à implémenter. La méthodologie employée dans ce chapitre permet de répondre aux deux derniers points

Introduction

Plus précisément :

Méthode générale

On souhaite :

- **Implémenter** la **spécification** d'une fonction.
- En utilisant un **algorithme de répétition** (boucle).
- En contrôlant :
  - l'état des variables **à chaque itération** $\rightarrow$ **invariant**
  - la bonne **terminaison** de l'algorithme $\rightarrow$ **variant**

Plan

Exemple introductif

Exemple

On considère deux entiers naturels `a` et `b`. On souhaite écrire la fonction `division_euclidienne` qui calcule et retourne le quotient `q` (c'est-à-dire `a//b`) et le reste `r` (c'est-à-dire `a%b`) de la division entière **en n'utilisant que les opérations** `+` **et** `-`.

Exemples :

a	b	q	r
13	4	3	1
4	17	0	4
0	4	0	0

Exemple introductif

Spécification de la fonction `division_euclidienne`:

- **Types des paramètres** : `a` et `b` entiers
- **Pré-condition** : `a>=0` et `b>0`
- **Types des valeurs de retour** : `q` et `r` entiers
- **Post-condition** : `a==q*b+r` et `0<=r<b`

Rappel

Les paramètres de la fonction (ici `a>=0` et `b>0`) **ne doivent pas être modifiés!**

Exemple introductif

Modèle de solution : On s'inspire de l'exemple `a=13` et `b=4` :

- On effectue des soustractions successives de la valeur `4` à `13`
- On obtient la suite de valeurs : 13, 9, 5, 1
- On s'arrête lorsque la valeur obtenue est inférieure à 4.

Le modèle de solution consiste donc en des **soustractions successives de** `b` **à une copie de** `a`. L'algorithme s'arrête lorsque l'on ne peut plus effectuer de soustraction. Le nombre de soustractions effectuées est le quotient de la division euclidienne.

Exemple introductif

Objectif : obtenir la post-condition

On décompose la **post-condition** `a==q*b+r and 0<=r<b` en deux sous-objectifs maîtrisables :

- **propriété 1** : `a==q*b+r and 0<=r`.
    - Cette propriété doit pouvoir être vraie à l'**initialisation** ainsi qu'à chaque **itération**
    - Cette propriété est nommée **invariant**.
- **propriété 2** : `r<b`
    - Cette propriété conditionne l'arrêt de la boucle *(On doit sortir de la boucle quand* `r<b` *)*

Si le programme s'exécute en suivant un chemin qui satisfait la **propriété 1** et si la boucle termine selon la **propriété 2** alors la **post-condition** est atteinte.

Étape 1 : Tests des propiétés

(pré- et post-conditions, invariant)

```
def division_euclidienne(a,b):
  assert a>=0 and b>0, 'Pre-condition'
  q = ...
  r = ...
  assert a==q*b+r and 0<=r, 'Invariant (initialisation)'
  while ...:
    q = ...
    r = ...
    assert a==q*b+r and 0<=r, 'Invariant (iteration)'
  assert a==q*b+r and 0<=r<b, 'Post-condition'
  return q,r
```

Exemple introductif : mise en oeuvre

Étape 2 : Condition de boucle

```
def division_euclidienne(a,b):
  assert a>=0 and b>0, 'Pre-condition'
  q = ...
  r = ...
  assert a==q*b+r and 0<=r, 'Invariant (initialisation)'
  while r>=b:
    q = ...
    r = ...
    assert a==q*b+r and 0<=r, 'Invariant (iteration)'
  assert a==q*b+r and 0<=r<b, 'Post-condition'
  return q,r
```

Exemple introductif : mise en oeuvre

Étape 3 : Initialisation et invariant

```python
def division_euclidienne(a,b):
  assert a>=0 and b>0, 'Pre-condition'
  q = 0
  r = a
  assert a==q*b+r and 0<=r, 'Invariant (initialisation)'
  while r>=b:
    q = ...
    r = ...
    assert a==q*b+r and 0<=r, 'Invariant (iteration)'
  assert a==q*b+r and 0<=r<b, 'Post-condition'
  return q,r
```

Exemple introductif : mise en oeuvre

Étape 4 : Corps de boucle

```python
def division_euclidienne(a,b):
  assert a>=0 and b>0, 'Pre-condition'
  q = 0
  r = a
  assert a==q*b+r and 0<=r, 'Invariant (initialisation)'
  while r>=b:
    q = q+1
    r = r-b
    assert a==q*b+r and 0<=r, 'Invariant (iteration)'
  assert a==q*b+r and 0<=r<b, 'Post-condition'
  return q,r
```

Exemple introductif : interprétation des erreurs

Erreur d'initialisation

```
def division_euclidienne(a,b):
  assert a>=0 and b>0, 'Pre-condition'
  q = 0
  r = 0
  assert a==q*b+r and 0<=r, 'Invariant (initialisation)'
  while r>=b:
    q = ...
    r = ...
    assert a==q*b+r and 0<=r, 'Invariant (iteration)'
  assert a==q*b+r and 0<=r<b, 'Post-condition'
  return q,r
```

```
-- AssertionError Traceback (most recent call last)
AssertionError: Invariant (initialisation)
```

Exemple introductif : interprétation des erreurs

Erreur dans l'itération

```python
def division_euclidienne(a,b):
  assert a>=0 and b>0, 'Pre-condition'
  q = 0
  r = a
  assert a==q*b+r and 0<=r, 'Invariant (initialisation)'
  while r>=b:
    q = q+2
    r = r-b
    assert a==q*b+r and 0<=r, 'Invariant (iteration)'
  assert a==q*b+r and 0<=r<b, 'Post-condition'
  return q,r
```

```
-- AssertionError Traceback (most recent call last)
AssertionError: Invariant (iteration)
```

Exemple introductif : interprétation des erreurs

Erreur dans l'arrêt de boucle

```
def division_euclidienne(a,b):
  assert a>=0 and b>0, 'Pre-condition'
  q = 0
  r = a
  assert a==q*b+r and 0<=r, 'Invariant (initialisation)'
  while r>=b:
    q = q+1
    r = r-b
    assert a==q*b+r and 0<=r, 'Invariant (iteration)'
  assert a==q*b+r and 0<=r<b, 'Post-condition'
  return q,r
```

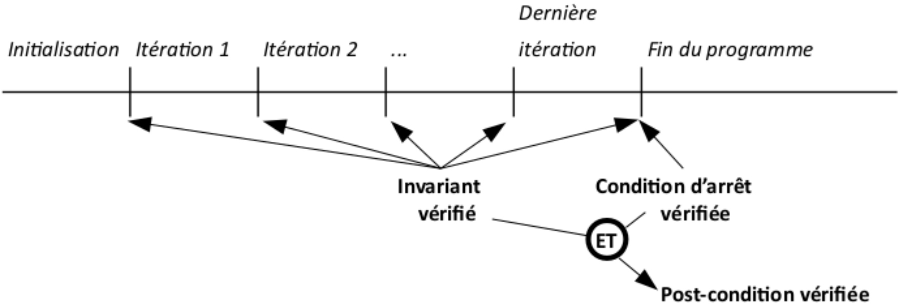
```
-- AssertionError Traceback (most recent call last)
AssertionError: Post-condition
```

Généralisation

Objectif : obtenir la post-condition

On décompose la **post-condition** en 2 propriétés :

- L'**Invariant** (Propriété 1), vrai à l'initialisation ainsi qu'à chaque itération.
- La **Condition d'arrêt** (Propriété 2) de la boucle.



Généralisation

On a donc le modèle d'algorithme suivant :

```
def <nom de la fonction>(<parametre 1>,...):
  assert <pre-condition>, 'Pre-condition'
  # initialisation des variables
  assert <invariant>, 'Invariant (initialisation)'
  while not <condition d'arret>:
    # traitement de la boucle et itération
    assert <invariant>, 'Invariant (iteration)'
  assert <invariant> and <condition d'arret>, 'Post-condition'
  return <valeur de retour>
```

Généralisation

Méthode basée sur l'invariant : bilan

- Cette méthode a permis de guider l'élaboration de l'algorithme en répondant aux questions suivantes :
  - Quelle condition de boucle utiliser ?
  - Comment initialiser les variables ?
  - Comment modifier les variables dans la boucle ?
- Il reste à répondre à la question suivante :
  - Comment être sûr que cette boucle n'est pas infinie ?

Application à la recherche par dichotomie

Recherche par dichotomie

Si tab est trié par ordre croissant, il est possible de tirer partie de cet ordre pour rendre la recherche beaucoup plus rapide :

- On consulte l'élément situé au milieu du tableau. Si cet élément est inférieur à la valeur recherchée, la recherche n'a besoin d'être poursuivie que dans la partie droite du tableau, sinon on poursuit la recherche dans la partie gauche.
- À chacune des étapes suivantes on coupe en deux cet intervalle de recherche et selon la valeur de l'élément situé au milieu, on choisit l'intervalle de gauche ou de droite pour poursuivre l'algorithme, jusqu'à ce que cet intervalle soit de longueur minimale.
- Ce principe de division en deux parties donne son nom à la méthode de recherche par dichotomie.

Application à la recherche par dichotomie

Exemple

Si **tab=[-19,-17,-17,-6,3,9,14,14,19,21,26]** et **valeur=14**, on obtient le déroulé suivant (la zone de recherche est grisée) :

↓i=0					↓m=5						↓j=11
-19	**-17**	**-17**	**-6**	**3**	**9**	**14**	**14**	**19**	**21**	**26**	`tab[5]<=14`

					↓i=5			↓m=8			↓j=11
-19	**-17**	**-17**	**-6**	**3**	**9**	**14**	**14**	**19**	**21**	**26**	`tab[8]>14`

					↓i=5	↓m=6		↓j=8			
-19	**-17**	**-17**	**-6**	**3**	**9**	**14**	**14**	**19**	**21**	**26**	`tab[6]<=14`

						↓i=6	↓m=7	↓j=8			
-19	**-17**	**-17**	**-6**	**3**	**9**	**14**	**14**	**19**	**21**	**26**	`tab[7]<=14`

							↓i=7	↓j=8			
-19	**-17**	**-17**	**-6**	**3**	**9**	**14**	**14**	**19**	**21**	**26**	`tab[7]==14`

Implémentation de la dichotomie

```python
def recherche_par_dichotomie(tab,valeur):
  if tab==[] or tab[0]>valeur:
    return -1
  i,j = 0,len(tab)
  while i+1<j:
    m = (i+j)//2
    if tab[m]<=valeur:
      i = m
    else:
      j = m
  if tab[i]==valeur:
    return i
  else:
    return -1
```

Dichotomie et invariants...

```
def recherche_par_dichotomie(tab,valeur):
  # précondition : tab est trie par ordre croissant
  assert ...

  if tab==[] or tab[0]>valeur:
    return -1

  i,j = 0,len(tab)
  # invariant : valeur appartient à l' intervalle t[i] , t[j] (borne droite exclue /!\ si j ==len(tab))
  assert ...
  while i+1<j:
    m = (i+j)//2
    if tab[m]<=valeur:
      i = m # tab[m] <= valeur : tab[i] se rapproche
    else:
      j = m # tab[m] > valeur : tab[j] reste extérieur
    # invariant : valeur appartient à l' intervalle t[i] , t[j] (droite exclue /!\ si j ==len(tab))
    assert ...


  # postcondition : j==i+1 ET valeur appartient à l' intervalle (droite exclue /!\ si i+1 ==len(tab))
  assert ...

  if tab[i]==valeur:
    return i
  else:
    return -1
```

Dichotomie et invariants...

```python
def recherche_par_dichotomie(tab,valeur):
  # précondition : tab est trie par ordre croissant
  assert est_croissant(tab), "Précondition : est_croissant à coder ... "

  if tab==[] or tab[0]>valeur:
    return -1

  i,j = 0,len(tab)
  # invariant : valeur appartient à l'intervalle t[i] , t[j] (borne droite exclue /!\ si j ==len(tab))
  assert 0 <= i < j <= len(tab) and tab[i] <= valeur and (j==len(tab) or valeur < tab[j]), "Invariant initialisation "
  while i+1<j:
    m = (i+j)//2
    if tab[m]<=valeur:
      i = m
    else:
      j = m
    # invariant : valeur appartient à l'intervalle t[i] , t[j] (droite exclue /!\ si j ==len(tab))
    assert 0 <= i < j <= len(tab) and tab[i] <= valeur and (j==len(tab) or valeur < tab[j]), "Invariant boucle"


  # postcondition : j==i+1 ET valeur appartient à l'intervalle (droite exclue /!\ si i+1 ==len(tab))
  assert 0<=i<len(tab) and j==i+1 and tab[i] <= valeur and (i+1==len(tab) or valeur<tab[i+1]), "Postcondition"

  if tab[i]==valeur:
    return i
  else:
    return -1
```

Compléments

Pourquoi décomposer la post-condition?

On pourrait envisager le schéma suivant :

```
# Initialisation des variables
...
while not <post-condition>:
  # Modification des variables
  ...
# Ici la post-condition est atteinte
```

Compléments

Exemple

```
q,r = ...,...
while not (a==q*b+r and 0<=r<b):
  q,r = ...,...
# Ici la post-condition a==q*b+r and 0<=r<b est atteinte
```

- fonctionne dans certains cas simples, MAIS...
- n'est pas efficace : on re-évalue inutilement `a==q*b+r` à chaque itération.
- ne guide pas l'écriture du code vers la partie de la post-condition qui n'est pas vérifiée.

Compléments

Finalisation du code

Une fois le code mis au point, **désactiver les asserts** :

- Soit par un commentaire.
- Soit par les options `-O` ou `-OO` de l'interpréteur.

On évite ainsi le coût d'exécution associé.

Plan

Le baladeur Zune

Le baladeur mp3 *Zune 30* de chez Microsoft a été mis sur le marché en novembre 2006. Pendant les premiers mois de son utilisation, aucun problème notable n'est survenu. Soudain, le 31 décembre 2008, tous les baladeurs Zune 30 ont été paralysés pendant 24 heures. **Que s'est-il donc passé???**

Le baladeur Zune

Le bout de code en langage C ci-contre est extrait du pilote de ces baladeurs :
Il accepte en paramètre un entier `days` qui représente le nombre de jours (absolus) écoulés depuis début 1980.
Une fois exécuté, l'entier `year` contient l'année courante, et `days` contient le nombre de jours (relatifs) écoulés depuis le début de cette année.

```c
year = 1980;
while (days>365) {
   if (IsLeapYear(year)) {
      if (days>366) {
         days -= 366;
         year += 1;
      }
   }
   else {
      days -= 365;
      year += 1;
   }
}
```

Le baladeur Zune

Traduction en Python :

```
def calculer_annee_jour(jours):
    annee = 1980
    while jours>365:
        if est_bissextile (annee):
            if jours>366:
                jours -= 366
                annee += 1
        else :
            jours -= 365
            annee += 1
    return annee, jours
```

Terminaison d'une boucle

Vocabulaire

Vérifier la **terminaison d'un algorithme**, c'est s'assurer que celui-ci va **s'arrêter quelles que soient les données initiales** qu'on lui fournit.

Méthode

En pratique, il suffit de montrer que chacune des boucles de l'algorithme se termine :

- Boucle `for` : vite résolu (nombre d'itérations spécifié par un entier).
- Boucle `while` : nécessaire d'introduire la notion de **variant**.

Terminaison d'une boucle

Définition

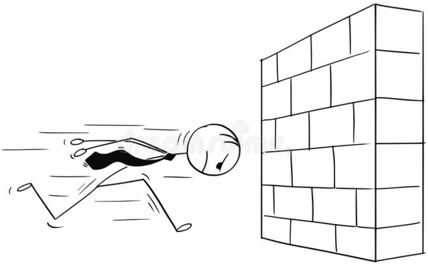
On appelle **variant d'une boucle** toute variable ou expression :

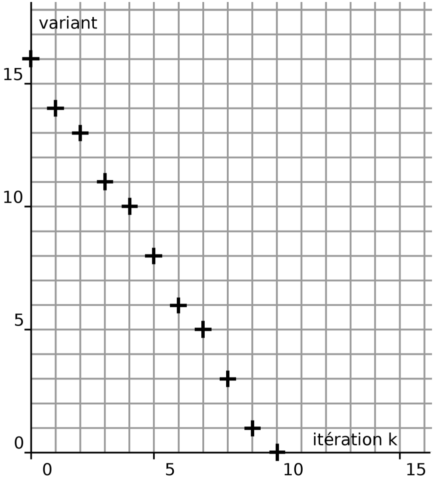
- entière.
- positive.
- strictement décroissante à chaque itération.

La suite formée par les valeurs successives d'un variant est nécessairement finie.

Propriété

Si une boucle possède un variant, alors cette boucle se termine.




variant
15
10
5
0
itération k
0
5
10
15

Terminaison d'une boucle

Exemples

Identifier un variant dans les deux exemples précédents :

- Fonction `division_euclidienne`.
- Fonction de recherche par dichotomie.

Vérification expérimentale

Le variant doit être :

- **entier** : par construction.
- **positif** : assertion en début de boucle.
- **strictement décroissant** : assertion en fin de boucle (comparaison valeurs de début et de fin).

Modèle de code

```
while ...:
  v_debut = <expression du variant>
  assert v_debut>=0, 'Variant (positif)'
  # Corps de la boucle
  ...
  v_fin = <expression du variant>
  assert v_fin<v_debut, 'Variant (decroissant)'
```

Vérification expérimentale

Exemples

Écrire la vérification expérimentale du variant dans les deux exemples précédents :

- Fonction `division_euclidienne`.
- Fonction de recherche par dichotomie.

Compléments : problème de l'arrêt

Le **problème de l'arrêt** consiste, étant donnée la description d'un programme informatique, à décider s'il s'arrête ou non. Nous avons vu qu'on peut effectuer cette décision sur certains programmes pour lesquels on parvient à déterminer un variant.

Problème de l'arrêt

En 1936, Alan Turing montre que dans le cas général ce problème est indécidable : il n'existe aucun programme informatique qui prenne comme entrée une description d'un programme informatique quelconque et puisse, grâce à la seule analyse de ce code, décider si ce programme s'arrête ou non.

https://fr.wikipedia.org/wiki/Problème_de_l'arrêt

Compléments : conjecture de Syracuse

Le fonction ci-dessous :

- Calcule les termes de la suite de Syracuse d'un entier `n`.
- Retourne le nombre d'itérations pour arriver à la valeur `1`.

```python
def temps_de_vol_syracuse(n):
  t = 0
  while n>1:
    if n%2==0:
      n = n//2
    else:
      n = 3*n+1
    t += 1
  return t
```

Compléments : conjecture de Syracuse

Exemple : Si `n=13`, les valeurs successives de `n` sont :
$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
Le temps de vol est alors égal à `9`.

Compléments : conjecture de Syracuse

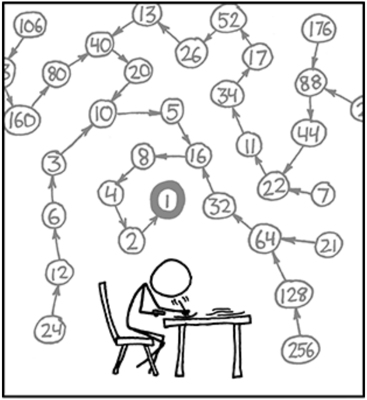
Cet algorithme a été testé **pour tous les entiers `n` inférieurs à** `2**62`, et pour toutes ces valeurs, **l'algorithme termine**.
Cependant, à ce jour (2021), **aucun variant** n'a été trouvé pour cet algorithme, et **il n'a pas été prouvé que cet algorithme termine** pour tous les entiers `n` strictement positifs.

Conjecture de Syracuse

L'ensemble des mathématiciens croit très fortement que cet algorithme termine pour tous les entiers `n` strictement positifs : il s'agit de la **conjecture de Syracuse**.

https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

Compléments : conjecture de Syracuse



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.