Informatique PCSI
Analyse d'un programme

Le contenu doit être analysé afin de vérifier que le résultat obtenu lors de son exécution est celui attendu d'après la spécification quelles que soient les données en entrées pour ne pas provoquer des erreurs en chaînes qui pourraient être catastrophiques.

1 Validité d'un algorithme

Lorsqu'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 dans le sens où il est conforme à la spécification précisée. Nous disons alors que l'algorithme est valide.

Un algorithme itératif est construit avec des boucles. Le nombre de passages dans une boucle doit être fini. Si l'algorithme est récursif, le nombre d'appels récursifs doit être fini.

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 les deux conditions sont satisfaites, nous disons que l'algorithme est *valide*.

Si lorsqu'il termine, l'algorithme donne la réponse attendue on parle de *correction partielle*. Si la terminaison est assurée dans tous les cas et que la réponse est correcte, on parle de *correction totale*.

Pour prouver la terminaison d'un algorithme itératif, nous disposons de la notion de variant de boucle. Pour prouver qu'un algorithme itératif est correct, nous disposons de la notion d'invariant de boucle.

1.1 Terminaison

Méthode

Dans le cas de boucles non conditionnelles, le nombre d'étapes est déterminé. Nous parlons donc de boucles conditionnelles. Pour prouver la terminaison, nous exhibons une expression, qui peut être une simple variable, 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. Cette expression se nomme *un variant de boucle*.

Considérons par exemple le code suivant où la valeur de la variable `a` est un nombre quelconque :

```
x = 0
while x ** 2 < a:
    x = x + 1
```

Si la valeur de `a` est négative ou nulle, il n'y a aucun passage dans la boucle. Sinon, la suite des valeurs de la variable `x` est $0, 1, 2, \ldots, n$, où n est la première valeur dont le carré est supérieur ou égal à la valeur de `a`. Le nombre de passages dans la boucle est donc fini.

Voici un deuxième exemple où les variables `a` et `b` sont des entiers naturels.

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

Dans cet exemple, nous choisissons comme variant la variable `m`. Cette variable prend pour valeurs successives $0, 1, \ldots, a$ et il y a donc exactement a passages dans la boucle, ce qui prouve la terminaison.

1.2 Correction

Définition

Un *invariant d'une boucle* est une propriété qui est vérifiée avant l'entrée dans une boucle et après chaque passage dans cette boucle.

Pour démontrer qu'une propriété est un invariant d'une boucle, on commence donc par vérifier que la propriété est vraie avant l'entrée dans la boucle. On prouve ensuite que si la propriété est vraie avant un passage dans la boucle, alors elle est vraie après ce passage. On peut alors conclure, si la terminaison est assurée, que la propriété est vraie à la sortie de la boucle

Ceci traduit le fait que la boucle réalise bien la tâche souhaitée.

Exemple

Reprenons l'algorithme de calcul avec une boucle conditionnelle présenté plus haut :

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

Notons m et p les valeurs des variables `m` et `p`.

Nous allons montrer que la propriété « $p = m \times b$ » est un invariant de la boucle.

Avant le premier passage dans la boucle, $m = 0$ et $p = 0$, donc l'égalité $p = m \times b$ est vraie.

Supposons que $p = m \times b$ avant un passage dans la boucle. Les nouvelles valeurs de `m` et `p` après le passage, notées m' et p', vérifient : $m' = m+1$ et $p' = p+b$. Alors $p' = m \times b + b = (m+1) \times b = m' \times b$. La propriété est donc vraie après ce passage dans la boucle.

La terminaison a été prouvée précédemment et nous avons en sortie de boucle $p = m \times b$. Or, à la sortie de la boucle, la variable `m` a pour valeur celle de `a`. Nous avons finalement obtenu à la fin du programme le produit $p = a \times b$.

2 Tests d'un programme

2.1 Bugs

Un circuit électronique, et plus généralement une machine, fait ce qu'on lui demande de faire. Si une erreur est commise lors de l'exécution d'un programme, alors cette erreur est d'une certaine manière écrite dans le programme. De plus, ce qui peut sembler être une « petite erreur » peut avoir des conséquences très importantes.

Considérons le programme qui suit.

```
>>> def f(x, y):
        return (x * y) ** 0.5

>>> def g(x, y):
        return (x ** 0.5) * (y ** 0.5)
```

```
>>> g(1e152, 1e152) / f(1e152, 1e152)
1.0
>>> g(1e155, 1e155) / f(1e155, 1e155)
0.0
```

En mathématiques, si x et y sont positifs : $\sqrt{xy} = \sqrt{x}\sqrt{y}$.

Le premier résultat est donc celui attendu, pas le second. Ce simple exemple montre la nécessité de tester un programme. Il prouve que l'un des deux programmes n'est pas correct (mais ne prouve pas que l'autre est correct).

Le même type de bug peut être présent avec un algorithme bien connu, celui d'une recherche dichotomique, lorsqu'on écrit `m = (g + d) // 2` pour obtenir l'indice du milieu des indices gauche et droite. La solution pour éviter ce type de bug est d'écrire `m = g + (d - g) // 2`.

2.2 Jeu de tests

Les programmes que nous écrivons ne fonctionnent pas toujours au premier essai comme nous le prévoyons.

- Des tests permettent d'observer le comportement d'un programme. Est-ce qu'il produit le résultat attendu dans un cas précis ?
- Le débogage consiste à corriger un programme lorsque nous savons qu'il ne fonctionne pas correctement.

Afin de rendre les tests et le débogage plus simples, il est important d'y penser lors de l'écriture d'un programme. Une bonne attitude est de décomposer le programme en éléments qui peuvent chacun être testés et débogués indépendamment des autres. L'utilisation de fonctions est une bonne habitude. La documentation des programmes, l'écriture de spécifications et l'utilisation d'assertions participent à l'élaboration de tests pertinents.

Des tests peuvent permettre de faire apparaître différents types d'erreurs.

- une erreur survient lors de certains test et pas avec d'autres, c'est un bug intermittent ;
- une erreur survient pour chaque test, c'est un bug permanent.
- le programme s'arrête de manière prématurée ou ne s'arrête pas, le bug est visible ;

Quand le programme s'exécute entièrement, il est possible de vérifier que le résultat obtenu est celui qui est attendu en ajoutant au programme des assertions. Lorsqu'une assertion n'est pas vérifiée, par exemple `assert f(3) == 0`, le message `AssertionError` est affiché.

Lorsque des tests, aussi nombreux soient-ils, ont été effectués avec succès, il reste encore la possibilité d'avoir un bug caché avec un programme qui produit un résultat faux dans des cas qui n'ont pas été testés.

2.3 Construire un jeu de tests

Construire un jeu de tests consiste à définir un ensemble de données qui vont être utilisées pour vérifier que le programme produit bien les résultats attendus avec ces données. Ces vérifications peuvent s'effectuer de plusieurs manières. On peut utiliser la fonction `print` pour afficher quelques réponses et vérifier visuellement qu'elles sont correctes. Pour un nombre conséquent de tests, il est plus pratique d'utiliser des assertions. Un message d'erreur est affiché seulement pour les cas qui posent problème.

Dans la construction d'un jeu de tests, on distingue plusieurs types de tests.

- Tester quelques cas simples typiques (pour une utilisation basique du programme).
- Tester des valeurs extrêmes, des cas limites, des cas interdits.

- Tester un nombre important de données (choisies de manière aléatoire par exemple).
- Tester des cas qui pourraient nécessiter un temps d'exécution important afin de pouvoir évaluer l'efficacité du programme.

Pour effectuer ces tests, on partitionne le domaine d'entrée. On utilise ensuite une donnée, représentante de chaque partie, pour effectuer un test en vérifiant la sortie.

Par exemple avec une fonction qui renvoie le pgcd de deux entiers naturels m et n, l'ensemble des données acceptables en entrées est l'ensemble des couples d'entiers naturels. On peut distinguer les cas où $m > n$, $m < n$, $m = n$, mais aussi les cas où m est divisible par n ou n divisible par m, les cas où m et n sont premiers entre eux. Il y a aussi les cas limites, soit l'un des deux entiers est nul ou les deux entiers sont nuls. Les tests peuvent alors être effectués avec par exemple les couples $(16, 12)$, $(9, 15)$, $(8, 8)$, $(12, 6)$, $(10, 30)$, $(15, 14)$, $(4, 0)$, $(0, 4)$, $(0, 0)$.

On écrit donc des instructions comme `assert pgcd(16, 12) == 4`.

Avec une fonction qui prend une liste en entrée, on teste naturellement le cas d'une liste vide et le cas d'une liste à un élément. En général, lorsqu'on commence à écrire des tests, des cas auxquels on ne pense pas immédiatement viennent peu à peu à l'esprit.

2.4 Exemples

Considérons le fonction `permute` définie ci-dessous avec sa spécification et des commentaires :

```
def permute(liste):
    """ liste est de type list
    la fonction permute le premier et le dernier élément
    et renvoie une nouvelle liste
    permute([1, 2, 3, 4]) renvoie [4, 2, 3, 1] """
    copie = liste[:] # une copie superficielle de la liste
    n = len(copie)
    copie[0], copie[n-1] = copie[n-1], copie[0] # permutation des valeurs
    return copie
```

Pour écrire un jeu tests, nous considérons quelques cas typiques ; une simple liste de nombres, une liste dont les éléments sont des listes, une liste qui contient deux éléments ou un unique élément, et une liste vide.

```
assert permute([1, 2, 3, 4]) == [4, 2, 3, 1]
assert permute([[1, 2], [3, 4], [5, 6]]) == [[5, 6], [3, 4], [1, 2]]
assert permute([1, 2]) == [2, 1]
assert permute([1]) == [1]
assert permute([]) == []
```

En exécutant le programme, nous remarquons que la fonction effectue bien ce qui est prévu même pour une liste ne contenant qu'un seul élément. Par contre, le cas d'une liste vide n'est pas traité et une erreur interrompt le programme.

Nous complétons donc le code de la fonction en ajoutant `if liste == []: return []` au début du code de la fonction. Avec ce code, tous les tests sont passés sans problème.

Les tests ont donc permis d'améliorer le comportement de la fonction en ajoutant la gestion d'un cas d'utilisation qui n'avait pas été prévu au départ.

Il peut être intéressant d'écrire une fonction `test` contenant une batterie de tests.
Pour un exemple nous considérons la division euclidienne.

```
def division(a, b):
    """a est un entier naturel
    b est un entier naturel non nul"""
    r = a
    q = 0
    while r >= b:
        r = r - b
        q = q + 1
    return q, r
```

Tous les types de tests envisageables ne sont pas explorés ici. La fonction écrite ci-dessous permet simplement de vérifier que le résultat renvoyé par la fonction `division` est correct dans une série de cas.

```
def test_division():
    """ la fonction division renvoie le quotient q et le reste r
    dans la division euclidienne de a par b
    un invariant est : a == b * q + r """
    for a in range(13):
        for b in range(1, 13):
            q, r = division(a, b)
            assert a == b * q + r and 0 <= r < b
```

Lors de l'exécution de la fonction `test_division`, aucun problème n'est signalé. Si par erreur il était écrit `while r > b` à la place de `while r >= b` dans la fonction `division`, la fonction de test révèlerait cette erreur avec une `AssertionError`.

De manière générale, si un test n'est pas réussi, la fonction ne précise pas quelle erreur a été commise mais elle informe que le test a échoué avec une `AssertionError`.

Les test présentés sont très simples. Le plus important est d'avoir toujours en tête que même avec un grand nombre de tests variés, **l'objectif n'est pas de prouver qu'il n'y a aucune erreur mais d'en trouver**.

Un algorithme classique, la recherche dichotomique dans une liste triée, pose souvent quelques difficultés de programmation. Pour écrire un jeu de tests, il faut envisager différents types de cas :

- un élément qui appartient à la liste : `2` et `[1, 2, 3, 4]` ;
- un élément qui n'appartient pas à la liste : `0` et `[1, 2, 3, 4]`, `5` et `[1, 2, 3, 4]`, `2.5` et `[1, 2, 3, 4]`.

On regarde ensuite des cas extrêmes, par exemple en lien avec la taille de la liste :

- `5` et `[]`, `5` et `[5]`, `0` et `[5]`.

On teste des cas en fonction de la place de l'élément cherché, en premier, en dernier, au milieu :

- `1` et `[1, 2, 3, 4]`, `4` et `[1, 2, 3, 4]`, `2` et `[1, 2, 3]`.

On peut ensuite effectuer des tests avec de très grandes listes.
Le calcul du milieu ne doit pas être négligé. Le résultat doit être correct même avec de très grands nombres. On peut envisager de séparer le calcul du milieu en créant une nouvelle fonction, ce qui produit un code encore plus lisible et plus facile à tester.