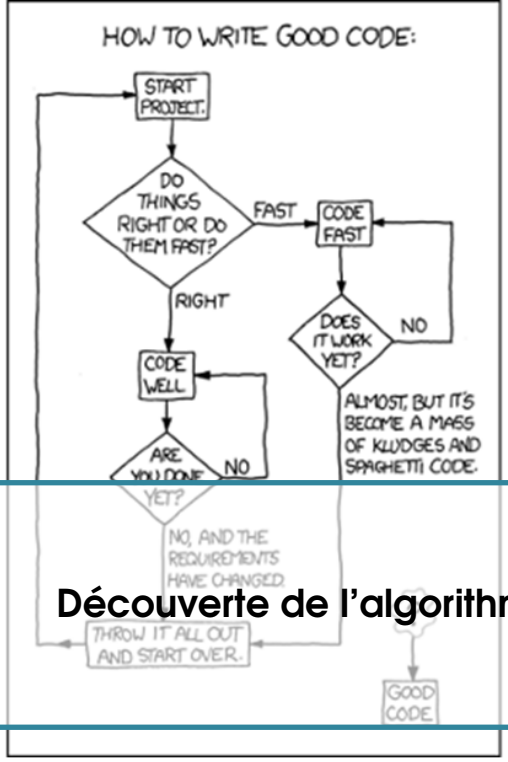
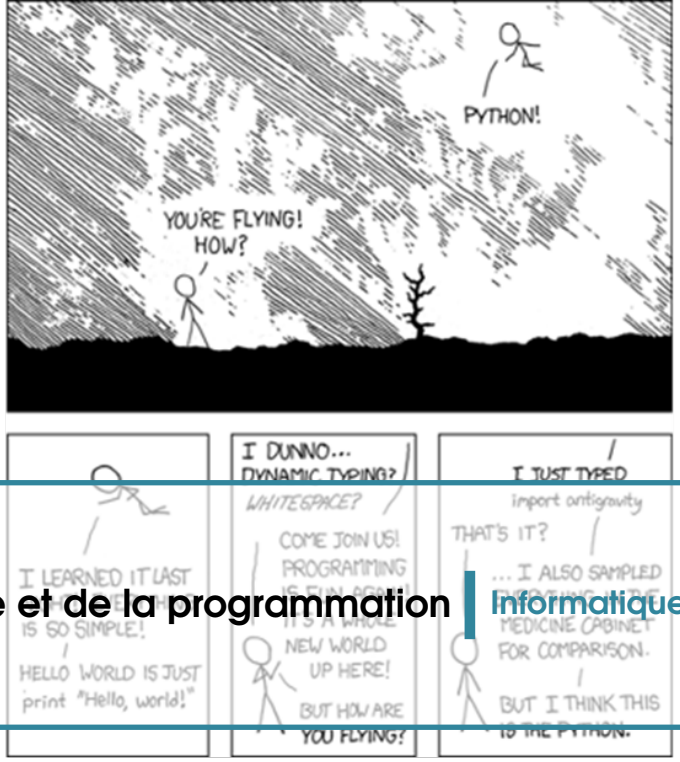




Découverte de l'algorithmique et de la programmation | Informatique

Chapitre 6

Méthode de programmation

Cours

Savoirs et compétences :

- *Spécifications des données attentdues en entrée et fournie en sortie/retour.*
- *Annotation d'un bloc d'instructions par une précondition, une postcondition, une propriété invariante*
- *Assertion.*
- *Jeu de tests associé à un programme.*

Xavier Pessoles
MPSI – PTSI

1 Contexte

Soit la fonction suivante :

```
def fonction(x,y = []) :
    if x == [] :
        return y
    else :
        z = x.pop(0)
        if z not in y :
            y.append(z)
        return fonction(x, y)
```

Problèmes :

- à quoi sert cette fonction ?
- quels sont les paramètres à donner ? leur type ?
- que renvoie la fonction ?

Lors de l'écriture d'une fonction, on peut être amener à l'utiliser, le jour même, le lendemain, une semaine plus tard, un mois plus tard... On peut être amener à la partager lorsqu'on travaille sur un projet. Il est donc essentiel de les documenter, de les commenter.

Présentons donc les choses ainsi.

```
def supprimer_doublons_liste(L,y = []) :
    """
    Fonction permettant de renvoyer, àpartir d'une liste d'entiers, une liste privée des ↙
        doublons.
    La liste initiale sera vidée.
    """
    if L == [] :
        return y
    else :
        z = L.pop(0)
        if z not in y :
            y.append(z)
        return fonction(L, y)
```

En modifiant le nom de la fonction et en ajoutant une court commentaire, il devient plus simple de partager la fonction ou de la réutiliser plus tard.

Problèmes (encore des problèmes) :

- on veut une fonction qui supprime les doublons d'une liste d'entiers... comment est-on sûrs que la fonction atteint bien cet objectif ?

Pour prouver qu'un algorithme on peut envisager deux méthodes :

- une méthode empirique (expérimentale) se basant sur des tests, sur le contrôle, des données d'entrées. Cette méthode à l'avantage d'être relativement simple à mettre en œuvre. Elle a l'inconvénient de valider le comportement d'une fonction sur des cas « ponctuels ». On ne montre donc pas qu'elle fonctionne dans le cas général.
- une méthode formelle, dans le cadre de laquelle on va prouver mathématiquement qu'une fonction atteint bien l'objectif fixé. On parle alors de de terminaison et de correction de l'algorithme.

La première méthode fait l'objet du présent chapitre. La seconde fera l'objet d'un chapitre ultérieur.

2 Spécification d'une fonction

2.1 Spécification à l'aide de commentaires

Définition Signature La **signature d'une fonction** définit les entrées et les sorties d'une fonction.

La signature peut comporter par exemple le type des paramètres d'entrées ou de sorties, des conditions sur ces paramètres.

La façon la plus simple de signer une fonction, est d'indiquer des commentaires juste après la déclaration de la fonction.

▪ **Exemple**

```
def supprimer_doublons_liste(L,y = []) :
    """
```

```
    Fonction permettant de supprimer les doublons d'une liste.
    Entrée :
     * L: list(int) : liste d'entiers.
    Sortie :
     * list(int) : liste d'entiers ne comportant aucun doublons.
    """
    if L == [] :
        return y
    else :
        z = L.pop(0)
        if z not in y :
            y.append(z)
        return fonction(L, y)
```

Lorsque les commentaires d'une fonction sont déclarés ainsi après la déclaration d'une fonction (triples "), `Python` génère automatiquement une documentation en utilisant l'instruction `help(supprimer_doublons_liste)`. On parle de `docstring`. ▪

2.2 Annotation d'une fonction– Limite du programme

Depuis la version 3.5 de `Python` il est possible d'annoter les fonctions.

Définition Les annotations de type permettent de donner une information sur le type attendu d'une fonction. Ce ne sont que des indications pour le programmeur. Rien ne l'empêche de ne pas les respecter.

▪ **Exemple**

```
def addition(a:int, b:int) -> int :
    return a+b
```

```
>>> addition(1,1)
       2
>>> addition('a','a')
       aa
```

▪

Comme précisé, les annotations ne sont que des indications. Mais il pourrait être intéressant de pouvoir s'assurer que l'utilisateur de la fonction respecte les conditions d'utilisation de la fonction que ce soit le type des arguments, ou d'autres conditions.

R En Python, il est possible d'utiliser `mypy` qui permet de vérifier que lors de l'appel d'une fonction, le typage précisé dans les annotations est bien respecté.

3 Assertion

3.1 Validation des entrées

Comme suite au paragraphe précédent, en phase de debuggage (ou développement), par opposition à la phase de production, `Python` prévoit la possibilité de vérifier que les arguments donnés à une fonction respectent bien les préconisations du concepteur.

En utilisant l'instruction `assert`, le programme s'interrompt quand une erreur est détectée.

▪ **Exemple** Dans la lignée des annotations de type, on peut tester le type d'argument donné à une fonction.

```
def addition(a:int, b:int) -> int :
    assert type(a) == int and type(b) ==↙
         int
    return a+b
```

```
>>> addition(1,1)
       2
>>> addition('a','a')
       (...)
       AssertionError
```

Si les arguments sont du mauvais type, `Python` déclenche une `AssertionError`. ▪

▪ **Exemple** Prenons un type d'exemple que nous rencontrerons plus souvent : l'algorithme fonctionne seulement si des préconditions sont remplies.

Par exemple dans le cas de la résolution de l'équation $f(x)=0$, il est nécessaire que f soit continue, monotone et que $f(a)$ et $f(b)$ soient de signes différents. Il faut aussi que ε soit positif. On peut donc procéder de la façon

suivante.

```
def dichotomie(f, a:float , b:float , epsilon:float) -> float :
    """Zéro de f sur [a,b] àepsilon près, par dichotomie
        Préconditions : f(a) * f(b) <= 0
        f continue sur [a,b]
        epsilon > 0"""
    assert (f(a) * f(b) <= 0) and (epsilon > 0)
    c, d = a, b
    fc, fd = f(c), f(d)
    while d - c > 2 * epsilon:
        m = (c + d) / 2.
        fm = f(m)
        if fc * fm <= 0:
            d, fd = m, fm
        else:
            c, fc = m, fm
    return (c + d) / 2.
```

■

3.2 Vers la gestion d'exceptions – hors programme

L'avantage des assertions et leur facilité à être mise en œuvre. L'inconvénient et qu'il n'est pas possible de savoir précisément ce qui a généré l'erreur d'assertion : un problème de précondition ? de type ? autre ?

Du fait de leur manque de précision, il peut être difficile de les gérer.

Un premier pas vers la gestion d'erreur est de donner à l'utilisateur la raison pour laquelle la fonction génère une erreur. Pour cela, on va *lever une exception*.

```
def dichotomie(f, a:float , b:float , epsilon:float) -> float :
    """Zéro de f sur [a,b] àepsilon près, par dichotomie
        Préconditions : f(a) * f(b) <= 0
        f continue sur [a,b]
        epsilon > 0"""
    if f(a) * f(b) > 0 :
        raise Exception("f(a) et f(b) sont de même signe")
    elif epsilon < 0 :
        raise Exception("epsilon est négatif")

    c, d = a, b
    fc, fd = f(c), f(d)
    while d - c > 2 * epsilon:
        m = (c + d) / 2.
        fm = f(m)
        if fc * fm <= 0:
            d, fd = m, fm
        else:
            c, fc = m, fm
    return (c + d) / 2.
```

```
>>> dichotomie(f, 0, 1, 0.00001)
   (...)
   Exception: f(a) et f(b) sont de même signe
```

4 Génération de tests

4.1 Tests unitaires

On a vu comment vérifier que les données d'entrées d'une fonction sont compatibles avec ce qu'attendait le concepteur de la fonction. On va maintenant chercher, grâce à des tests unitaires, si le résultat attendu est bien celui déterminé par une fonction.

▪ Exemple

```
def addition(a:int, b:int) -> int :
    assert type(a) == int and type(b) == int
    return a+b
```

```
>>> addition(1,1) == 2
       True
>>> addition(1,1) == 3
       False
```

▪

Ces tests ont l'avantage d'être simples à mettre en œuvre mais ne garantissent pas que la fonction « marche » dans tous les cas.

Il est alors possible de créer une fonction de tests.

```
def test_addition() :
    assert addition(1,1) == 2
    assert addition(1,1) !=3
```

Des jeux de tests pertinents peuvent être des tests sur les cas limites. Dans le cas de la recherche dichotomique, on peut se demander si la boucle `while` n'est pas réalisée une fois de trop par rapport à la valeur de `epsilon`.

```
>>> res = dichotomie(f, 0, 1, epsilon)
>>> f(res) < epsilon # MAUVAIS test : f(res) peut être négatif
    True
>>> abs(f(res)) < epsilon
    False
>>> # L'algorithme dichotomique doit itérer une fois de plus ?
```

En modifiant l'algorithme, et en itérant une fois de plus, le test reste faux. La condition d'arrêt de l'algorithme ne permet pas de trouver x tel que $f(x) < \varepsilon$ mais de trouver une valeur de x telle que l'écart entre deux solutions consécutives est inférieur à ε. Il faut donc clarifier l'objectif de la fonction pour proposer un test plus adapté.

4.2 Vers Pytest – hors programme

Lorsqu'on connaît dans un grand nombre de cas le résultat qui est sensé être trouvé par une fonction, il est possible de faire passer une batterie de tests à cette fonction en recherchant si le résultat calculé et bien le résultat attendu.

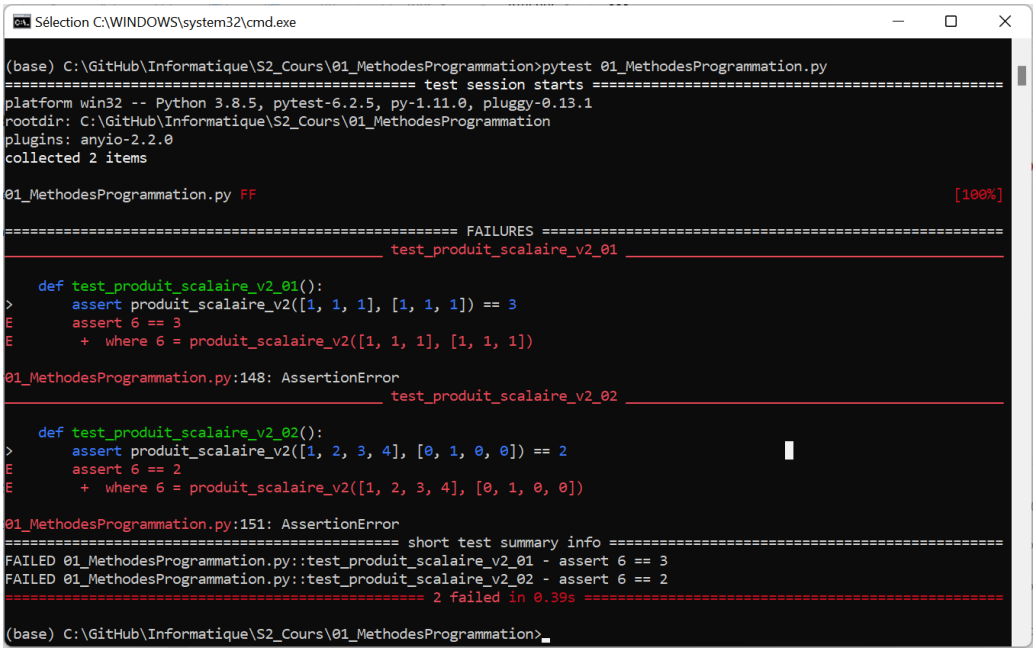
Soit la fonction suivante permettant de calculer le produit scalaire entre deux vecteurs.

```
def produit_scalaire_v2(vecteur1, vecteur2):
    somme = 0
    for i in range(len(vecteur1)):
        somme += vecteur1[i]*vecteur2[i]+1
    return somme
```

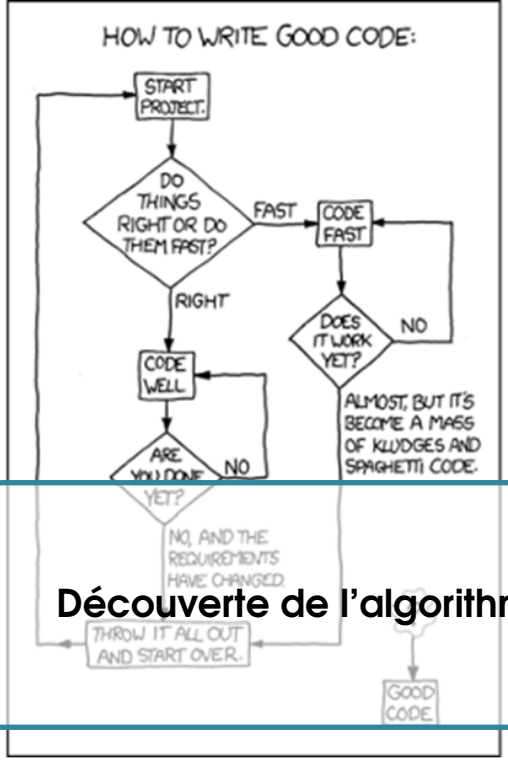
`Pytest` permet de faire passer une batterie de tests à une fonction. Ces tests sont des fonctions commençant obligatoirement par `test_`.

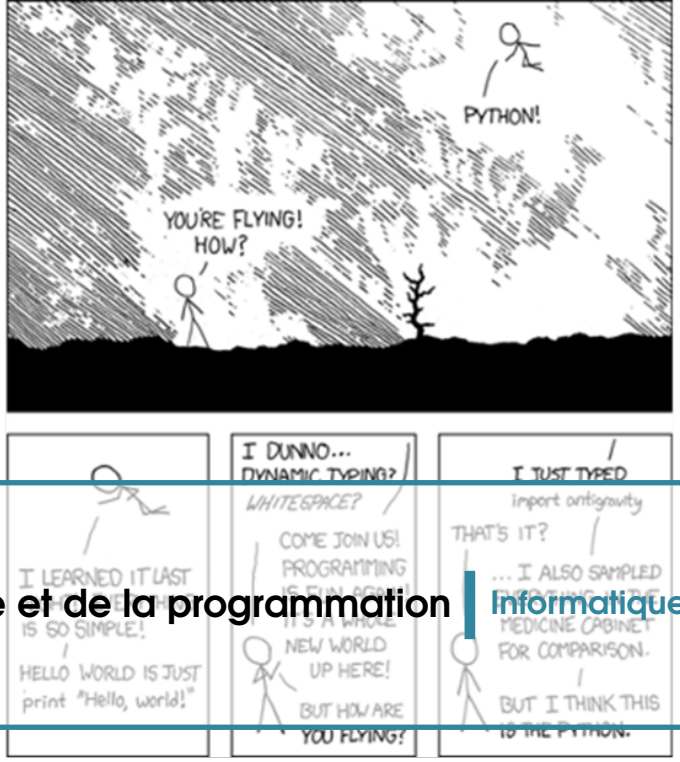
```
def test_produit_scalaire_v2_01():
    assert produit_scalaire_v2([1, 1, 1], [1, 1, 1]) == 3

def test_produit_scalaire_v2_02():
    assert produit_scalaire_v2([1, 2, 3, 4], [0, 1, 0, 0]) == 2
```

Le lancement de `Pytest` donne le résultat suivant.



On peut lire que les deux tests ont échoué. La fonction ne remplit donc vraisemblablement pas son objectif.





Découverte de l'algorithmique et de la programmation | Informatique

Chapitre 6

Méthode de programmation – Suite

Cours

Savoirs et compétences :

- *Terminaison.*
- *Correction partielle.*
- *Correction totale.*
- *Variant. Invariant.*

5 Anlayse des algorithmes

5.1 Définition

Définition Terminaison d'un algorithme
Prouver la terminaison d'un algorithme signifie montrer que cet algorithme se terminera en un temps fini. On utilise pour cela un **variant de boucle**.

Définition Correction d'un algorithme
Un algorithme est dit (partiellement) correct s'il est correct dès qu'il termine.

Prouver la correction d'un algorithme signifie montrer que cet algorithme fournit bien la solution au problème qu'il est sensé résoudre. On utilise pour cela un **invariant de boucle**.

Définition Invariant de boucle Un invariant de boucle est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.

5.2 Un exemple ...

Objectif L'objectif est ici de montrer la nécessité d'utiliser un invariant de boucle.
Pour cela, on propose la fonction suivante sensée déterminer le plus petit entier `n` strictement positif tel que $1+2+...+n$ dépasse strictement la valeur entière strictement positive `v`. Cette fonction renvoie-t-elle le bon résultat ? Desfois ? Toujours ?

```
def foo(v:int) -> int:
    r = 0
    n = 0
    while r < v :
        n = n+1
        r = r+n
    return n
```

Montrer intuitivement que `foo(v)` *se termine pour* $v \in \mathbb{N}^*$.

L'algorithme se terminera si on sort de la boucle `while`. Il faut pour cela que la condition `r<v` devienne fausse (cette condition est vraie initialement). Pour cela, il faut que `r` devienne supérieure ou égale à `v` dont la valeur ne change jamais.

`n` étant incrémenté de 1 à chaque itération, la valeur de `r` augmente donc à chaque itération. Il y aura donc un rang `n` au-delà duquel `r` sera supérieur à `v`. L'algorithme se termine donc.

Que renvoie `foo(9)` *? Cela répond-il au besoin ?*

Début de la i^e itération	r	n	r < v
Itération 1	0	0	0 < 9 ⇒ True
Itération 2	1	1	1 < 9 ⇒ True
Itération 3	3	2	3 < 9 ⇒ True
Itération 4	6	3	6 < 9 ⇒ True
Itération 5	10	4	10< 9 ⇒ False

La fonction renvoie 4. On a $1+2+3+4=10$. On dépasse strictement la valeur 9. La fonction répond au besoin dans ce cas.

Que renvoie `foo(10)` *? Cela répond-il au besoin ?*

Début de la i^e itération	r	n	r < v
Itération 1	0	0	0 < 10 ⇒ True
Itération 2	1	1	1 < 10 ⇒ True
Itération 3	3	2	3 < 10 ⇒ True
Itération 4	6	3	6 < 10 ⇒ True
Itération 5	10	4	10 < 10 ⇒ False

La fonction renvoie 4. On a $1+2+3+4=10$. On ne dépasse pas strictement la valeur 10. La fonction ne répond pas au besoin dans ce cas.

Résultat La fonction proposée ne remplit pas le cahier des charges. Aurait-on pu le prouver formellement ?

6 Terminaison d'un algorithme

6.1 Variant de boucle

Définition Variant de boucle
Un variant de boucle permet de prouver la terminaison d'une boucle conditionnelle. Un variant de boucle est une **quantité entière positive** à l'entrée de chaque itération de la boucle et qui **diminue strictement à chaque itération**.

Théorème Si une boucle admet un variant de boucle, elle termine.

Propriété Un algorithme qui n'utilise ni boucles inconditionnelles (boucle `for`) ni récursivité termine toujours. Ainsi, la question de la terminaison n'est à considérer que dans ces deux cas.

Reprenons l'exemple précédent.

```
def foo(v:int) -> int:
    r = 0
    n = 0
    while r < v :
        n = n+1
        r = r+n
    return n
```

Dans cet exemple montrons que la quantité $u_n = v - r$ est un variant de boucle :
- initialement, $r = 0$ et $v > 0$; donc $u_0 > 0$;
- à la fin de l'itération n, on suppose que $u_n = v - r > 0$ et que $u_n < u_{n-1}$;
- à l'itération $n+1$:
  - cas 1 : $r \geq v$. Dans ce cas, n et r n'évoluent pas l'hypothèse de récurrence reste vraie. On sort de la boucle `while`. L'algorithme termine,
  - cas 2 : $r < v$. Dans ce cas, à la fin de l'itération $n+1$, montrons que $u_{n+1} < u_n$: $u_{n+1} = v - (r+n+1) = u_n - n - 1$ soit $u_{n+1} = u_n - n - 1$ et donc $u_{n+1} < u_n$. L'hypothèse de récurrence est donc vraie au rang $n+1$.

Au final, $u_n = v - r$ est donc un variant de boucle et la boucle se termine.

6.2 Un second exemple ressemblant...

[https://marcdefalco.github.io/pdf/complet_python.pdf]

Considérons l'algorithme suivant qui, étant donné un entier naturel n strictement positif (inférieur à 2^{30}), détermine le plus petit entier k tel que $n \leq 2^k$.

```
def plus_grande_puissance2(n):
    k = 0
    p = 1
    while p < n:
        k = k+1
        p = p*2
    return k
```

Démonstration [1] Dans l'exemple précédent, la quantité $n - p$ est un variant de boucle :
- au départ, $n > 0$ et $p = 1$ donc $n - p \geq 0$;
- comme il s'agit d'une différence de deux entiers, c'est un entier. Et tant que la condition de boucle est vérifiée $p < n$ donc $n - p > 0$.
- lorsqu'on passe d'une itération à la suivante, la quantité passe de $n - p$ à $n - 2p$ or $2p - p > 0$ car $p \geq 1$. Il y a bien une stricte diminution.

Démonstration [2] Montrons que, la quantité $u_j = n - p$ est un variant de boucle :
- intialement, $n > 0$ et $p = 1$ donc $n - p \geq 0$;
- à la fin de l'itération j, on suppose que $u_j = n - p > 0$ et $u_j < u_{j-1}$;
- à la fin de l'itération suivante, $u_{j+1} = n - 2p = u_j - p$. p est positif donc u_{j+1} est un entier et $u_{j+1} < u_j$. Par suite, ou bien $u_{j+1} < 0$ c'est à dire que $n - p < 0$ soit $p > n$. On sort donc de la boucle. Ou bien, $u_{j+1} > 0$, et la boucle continue.

$n - p$ est donc un variant de boucle.

7 Correction d'un algorithme

7.1 Invariant de boucle

Méthode

Pour montrer qu'une propriété est un invariant de boucle dans une boucle `while` :

- le propriété doit être vérifiée avant d'entrer dans la boucle ;
- la propriété doit être vraie en entrée de boucle ;
- la propriété doit être vraie en fin de boucle.

Reprenons un des exemples précédents. Reconsidérons l'algorithme suivant qui, étant donné un entier naturel n strictement positif (inférieur à 2^{30}), détermine le plus petit entier k tel que $n \leq 2^k$.

```
def plus_grande_puissance2(n):
    k = 0
    p = 1
    while p < n:
        k = k+1
        p = p*2
    return k
```

Démonstration Montrons que la propriété suivante est un invariant de boucle : $p = 2^k$ et $2^{k-1} < n$.

- **Initialisation :** à l'entrée dans la boucle $k = 0$ et $p = 1$, $n \in \mathbb{N}^*$
  - d'une part on a bien $1 = 2^0$;
  - d'autre part $2^{-1} < n$.
- On considère que la propriété est vraie au n^e tour de boucle c'est à dire $p = 2^k$ et $2^{k-1} < n$.
- Au tour de boucle suivant :
  - **ou bien** $p >= n$. Dans ce cas, on sort de la boucle et on a toujours $p = 2^k$ et $2^{k-1} < n$ (propriété d'invariance). La propriété est donc vraie au tour $n+1$.
  - **ou bien** $p < n$. Dans ce cas, il faut montrer que $p = 2^{k+1}$ et $2^k < n$. Etant entrés dans la boucle, $p < n \Rightarrow 2^k < n$. De plus, en fin de boucle, $p \rightarrow p*2$ et $k \rightarrow k+1$. On a donc $p \leftarrow 2^k * 2 = 2^{k+1}$.

La propriété citée est donc un invariant de boucle.

7.2 Un « contre exemple »

Reprenons le tout premier exemple où on cherche le plus petit entier `n` strictement positif tel que $1+2+...+n$ dépasse strictement la valeur entière strictement positive `v` .

```
def foo(v:int) -> int:
    assert v>0
    r = 0
    n = 0
    while r <= v :
        n = n+1
        r = r+n
    return n
```

La propriété suivante est-elle un invariant de boucle : $r = \sum_{i=0}^{n} i$ et $\sum_{i=0}^{n-1} i \leq v$, $n \in \mathbb{N}^*$?

La réponse est directement NON, car la phase d'initialisation n'est pas vérifiée car $n = 0$ et $n \notin \mathbb{N}^*$. Cela signifie donc que l'algorithme proposé ne répond pas au cahier des charges.

Modifions alors l'algorithme ainsi.

```
def foo2(v:int) -> int:
    assert v>0
    r = 1
    n = 1
    while r <= v :
        n = n+1
        r = r+n
    return n
```

Montrons que la propriété suivante est un invariant de boucle : $r = \sum_{i=0}^{n} i$ et $\sum_{i=0}^{n-1} i \leq v$, $n \in \mathbb{N}^*$.

- **Initialisation :** à l'entrée dans la boucle $r = 1$ et $n = 1$, $n \in \mathbb{N}^*$
  - d'une part on a bien $r = \sum_{i=0}^{1} i = 1$;

  - d'autre part $\sum_{i=0}^{0} i = 0 < v$ et $v > 0$ (spécification de la fonction).

- On considère que la propriété est vraie au début du n^{e} tour de boucle c'est-à-dire $r = \sum_{i=0}^{n} i$ et $\sum_{i=0}^{n-1} i \leq v$.
- Á la fin du n^{e} tour de boucle, $n_{n+1} = n_n + 1$ et $r_{n+1} = r_n + n_{n+1} = r_n + n_n + 1 = \sum_{i=0}^{n}(i) + n_n + 1 = \sum_{i=0}^{n+1} i$ (car $n_n = n$).

  On a alors,
  - ou bien $r_{n+1} > v$ et on sort de la boucle ; on peut renvoyer n.
  - ou bien $r_{n+1} \leq v$ et donc $\sum_{i=0}^{n} i \leq v$.

La propriété citée est donc un invariant de boucle.