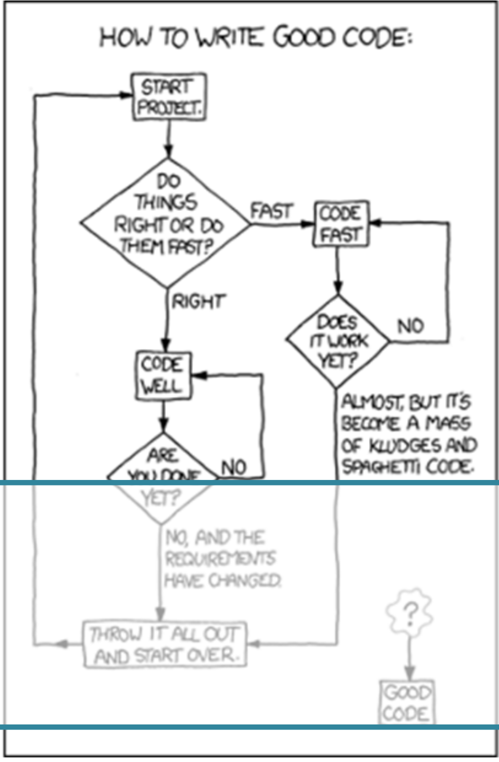

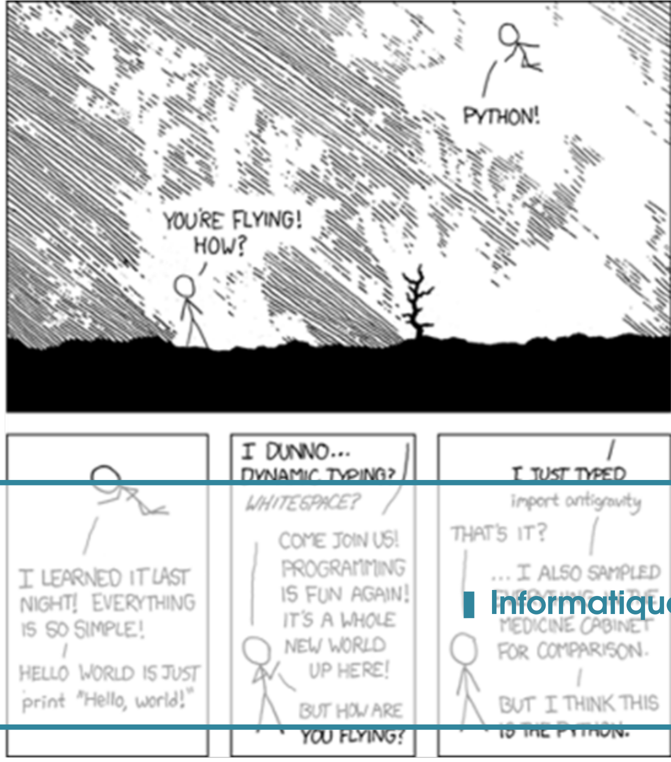

Semestre 2

MPSI

Informatique

Fiche

Introduction à la programmation en Python

Informatique

E. Durif – X. Pessoles
J.-P. Berne
MPSI

1 Anlayse des algorithmes

1.1 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
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 Analyser
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**.

1.2 Un exemple ...

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` .

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

Montrer intuitivement que `foo()` *se termine.* 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 incrémenter 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 donc se termine.

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 dépassement strictement la valeur 10. 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 dépassement pas strictement la valeur 10. La fonction ne répond pas au besoin dans ce cas.

Bilan : la fonction proposée ne remplit pas le cahier des charges. Aurait-on pu le prouver formellement ?

2 Terminaison d'un algorithme

2.1 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.

R Un algorithme qui n'utilise ni boucles inconditionnelles 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 $u_n < u_{n-1}$.
  - cas 1 : $r \geq v$. Dans ce cas, n et r n'évoluent pas l'hypothèse de récurrence reste vraie.
  - 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. La boucle se termine donc.

2.2 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.

3 Correction d'un algorithme

3.1 Invariant de boucle

Cycle 01

Définition **Invariant de boucle** Soit une boucle. Une propriété est appelée un invariant de boucle lorsque :
- cette propriété est vérifiée avant d'entrer dans la boucle;
- si cette propriété est vérifiée en entrée d'itération, alors elle est vérifiée en sortie de l'itération.

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 bouclen 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, $p \to p*2$ et $k \to k+1$. On a donc $p \leftarrow 2^k * 2 = 2^{k+1}$.

La propriété citée est donc un invariant de boucle.

3.2 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:
    r = 0
    n = 0
    while r < v :
        n = n+1
        r = r+n
    return n
```

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

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 en répond pas au cahier des charges.

Modifions donc l'algorithme ainsi.

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

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

- **Initialisation :** à l'entrée dans la boucle $r = 1$ et $n = 1$, $n \in \mathbb{N}^*$
  - d'une part on a bien $i = \sum_{i=0}^{1} i$;
  - d'autre part $\sum_{n=0}^{0} i = 0 < v$.
- On considère que la propriété est vraie au n^{e}tour de boucle c'est à dire $r = \sum_{i=0}^{n} i$ et $\sum_{n=0}^{n-1} i < v$.
- 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, $p \leftarrow p * 2$ et $k \leftarrow k + 1$. On a donc $p \leftarrow 2^k * 2 = 2^{k+1}$.

La propriété citée est donc un invariant de boucle.

4 A TRIER

Définition **Preuve d'algorithme**
Une preuve d'algorithme est une démonstration montrant qu'un algorithme réalise la tâche pour laquelle il a été conçu.
Il faut alors montrer sa **terminaison** c'est-à-dire montrer que l'algorithme se termine. On utilise pour cela un **variant de boucle**.
Il faut ensuite montrer sa **correction** c'est-à-dire montrer que l'algorithme réalise la tâche attendue. On utilise pour cela un **invariant de boucle**.

4.1 Un premier exemple

Donner l'algorithme permettant de déterminer le plus petit entier n tel que $1+2+...+n$ dépasse strictement 1000.
Proposons cet algorithme.

```
res = 0
n = 0
while res < 1000 :
    n = n+1
    res = res+n

print(n,res)
```

4.2 Un deuxième exemple : $n!$

```
for i in range(1,n+1):
    # en entrant dans le ième tour de boucle, p = (i-1)!
    p=p*i
    # en sortant du ième tour de boucle, p = i!

print(p) #p = n!
```

Ici, l'invariant de boucle est « p contient $(i-1)!$ » :
1. c'est bien une propriété qui est vraie pour $i=1$;
2. supposons qu'au rang i, $p=(i-1)!$ à l'entrée de la boucle. Au cours de la boucle, p va prendre la valeur $p=(i-1)!\times i=i!=((i+1)-1)!$ donc la propriété est vérifiée en sortie de boucle ;
3. enfin, au dernier tour de boucle, i vaut n donc $p=n!$ ce qui répond à la question.

4.3 Un troisième exemple : algorithme d'Euclide

`https://lgarcin.github.io/CoursPythonCPGE/preuve.html`

```
def pgcd(a, b):
  while b!= 0:
      a, b = b, a % b
  return a
```

On suppose que l'argument b est un entier naturel. En notant b_k la valeur de b à la fin de la $k^{\text{ème}}$ itération (b_0 désigne la valeur de b avant d'entrer dans la boucle), on a $0\leq b_{k+1}<b_k$ si $b_k>0$. La suite (b_k) est donc une suite strictement décroissante d'entiers naturels : elle est finie et la boucle se termine.

On note a_k et b_k les valeurs de a et b à la fin de la $k^{\text{ème}}$ itération (a_0 et b_0 désignent les valeurs de a et b avant d'entrer dans la boucle). Or, si $a=bq+r$, il est clair que tout diviseur commun de a et b est un diviseur commun de b et r et réciproquement. Notamment, $a\wedge b=b\wedge r$. Ceci prouve que $a_k\wedge b_k=a_{k+1}\wedge b_{k+1}$. La quantité $a_k\wedge b_k$ est donc bien un invariant de boucle. En particulier, à la fin de la dernière itération (numérotée N), $b_N=0$ de sorte que $a_0\wedge b_0=a_N\wedge b_N=a_N\wedge 0=a_N$. La fonction `pgcd` renvoie donc bien le `pgcd` de a et b.

4.4 Un troisième exemple (bis) : algorithme d'Euclide

`https://mathematice.fr/fichiers/cpge/infoprepaC8.pdf`
On effectue la division euclidienne de a par b où a et b sont deux entiers strictement positifs. Il s'agit donc de déterminer deux entiers q et r tels que $a=bq+r$ avec $0\leq r<b$. Voici un algorithme déterminant q et r :

```
q = 0
r = a
while r >= b :
    q = q + 1
    r = r -b
```

On choisit comme invariant de boucle la propriété $a = bq + r$.

- Initialisation : q est initialisé à 0 et r à a, donc la propriété $a = bq + r = b \cdot 0 + a$ est vérifiée avant le premier passage dans la boucle.
- Hérédité : avant une itération arbitraire, supposons que l'on ait $a = bq + r$ et montrons que cette propriété est encore vraie après cette itération. Soient q' la valeur de q à la fin de l'itération et r' la valeur de r à la fin de l'itération. Nous devons montrer que $a = bq' + r'$. On a $q' = q+1$ et $r' = r-b$, alors $bq'+r' = b(q+1)+(r-b) = bq + r = a$. La propriété est bien conservée.

Terminaison Nous reprenons l'exemple précédent.

- Commençons par montrer que le programme s'arrête : la suite formée par les valeurs de r au cours des itérations est une suite d'entiers strictement décroissante : r étant initialisé à a, si $a \geq b$ alors la valeur de r sera strictement inférieure à celle de b en un maximum de $a - b$ étapes.
- Ensuite, si le programme s'arrête, c'est que la condition du "tant que" n'est plus satisfaite, donc que $r < b$. Il reste à montrer que $r \geq 0$. Comme r est diminué de b à chaque itération, si $r < 0$, alors à l'itération précédente la valeur de r était $r' = r + b$; or $r' < b$ puisque $r < 0$. Et donc la boucle se serait arrêtée à l'itération précédente, ce qui est absurde; on on déduit que $r \geq 0$.

En conclusion, le programme se termine avec $0 \leq r < b$ et la propriété $a = bq + r$ est vérifiée à chaque itération; ceci prouve que l'algorithme effectue bien la division euclidienne de a par b.

4.5 Quatrième exemple

L'objectif est de calculer le produit de deux nombres entiers positifs a et b sans utiliser de multiplication.

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

Comme dans l'exemple précédent, le programme se termine car la suite des valeurs de m est une suite d'entiers consécutifs strictement croissante, et atteint la valeur a en a étapes.

Un invariant de boucle est ici : $p = m.b$.

- Initialisation : avant le premier passage dans la boucle, $p = 0$ et $m = 0$, donc $p = mb$.
- Hérédité : supposons que $p = mb$ avant une itération; les valeurs de p et m après l'itération sont $p' = p + b$ et $m' = m + 1$. Or $p' = (p + b) = m.b + b = (m+1)b = m'b$. Donc la propriété reste vraie.
- Conclusion : à la sortie de la boucle $p = m.b$.

Puisqu'à la sortie de la boucle $m = a$, on a bien $p = ab$.