Terminaisons et corrections de boucles

Ivan Noyer

Lycée Thiers

1. Terminaison et variants

2. Correction et invariants

3. Compléments

Démontrer qu'une boucle termine

- Montrer la *terminaison* d'un algorithme, c'est prouver que l'algorithme termine quel que soit l'état initial.

Démontrer qu'une boucle termine

- Montrer la *terminaison* d'un algorithme, c'est prouver que l'algorithme termine quel que soit l'état initial.
- Le problème se pose principalement pour les boucles conditionnelles (**while**) et pour les fonctions récursives.

Démontrer qu'une boucle termine

- Montrer la *terminaison* d'un algorithme, c'est prouver que l'algorithme termine quel que soit l'état initial.
- Le problème se pose principalement pour les boucles conditionnelles (**while**) et pour les fonctions récursives.
- On identifie un $\boxed{variant}$, autrement dit une expression (c'est souvent le simple contenu d'une variable)

Démontrer qu'une boucle termine

- Montrer la *terminaison* d'un algorithme, c'est prouver que l'algorithme termine quel que soit l'état initial.
- Le problème se pose principalement pour les boucles conditionnelles (**while**) et pour les fonctions récursives.
- On identifie un $\boxed{variant}$, autrement dit une expression (c'est souvent le simple contenu d'une variable)
  - qui est un entier positif tout au long de la boucle/récursion,

Démontrer qu'une boucle termine

- Montrer la *terminaison* d'un algorithme, c'est prouver que l'algorithme termine quel que soit l'état initial.
- Le problème se pose principalement pour les boucles conditionnelles (**while**) et pour les fonctions récursives.
- On identifie un $\boxed{\mathtt{variant}}$, autrement dit une expression (c'est souvent le simple contenu d'une variable)
  - qui est un entier positif tout au long de la boucle/récursion,
  - qui diminue strictement après chaque itération/appel récursif interne,

Démontrer qu'une boucle termine

- Montrer la *terminaison* d'un algorithme, c'est prouver que l'algorithme termine quel que soit l'état initial.
- Le problème se pose principalement pour les boucles conditionnelles (**while**) et pour les fonctions récursives.
- On identifie un $\boxed{variant}$, autrement dit une expression (c'est souvent le simple contenu d'une variable)
  - qui est un entier positif tout au long de la boucle/récursion,
  - qui diminue strictement après chaque itération/appel récursif interne,
  - et qui, lorsqu'elle devient négative assure qu'on sort de la boucle.

Démontrer qu'une boucle termine

- Montrer la *terminaison* d'un algorithme, c'est prouver que l'algorithme termine quel que soit l'état initial.
- Le problème se pose principalement pour les boucles conditionnelles (**while**) et pour les fonctions récursives.
- On identifie un $\boxed{variant}$, autrement dit une expression (c'est souvent le simple contenu d'une variable)
  - qui est un entier positif tout au long de la boucle/récursion,
  - qui diminue strictement après chaque itération/appel récursif interne,
  - et qui, lorsqu'elle devient négative assure qu'on sort de la boucle.
- On peut alors en conclure que la boucle termine.

Le variant est un compteur

```python
p,c=1,3
while c>0:
    p=p*2
    c=c-1
```

- Détail des états :
- La variable `c` joue le rôle d'un compteur.

Le variant est un compteur

- Détail des états :

$$\begin{pmatrix} c \\ 3 \end{pmatrix} \quad \begin{pmatrix} p \\ 1 \end{pmatrix} \quad \text{état initial}$$
$$\begin{pmatrix} c \\ 2 \end{pmatrix} \quad \begin{pmatrix} p \\ 2 \end{pmatrix}$$
$$\begin{pmatrix} c \\ 1 \end{pmatrix} \quad \begin{pmatrix} p \\ 4 \end{pmatrix}$$
$$\begin{pmatrix} c \\ 0 \end{pmatrix} \quad \begin{pmatrix} p \\ 8 \end{pmatrix} \quad \text{état final}$$

- La variable `c` joue le rôle d'un compteur.

Le variant est un compteur

- Détail des états :
- La variable `c` joue le rôle d'un compteur. Au départ `c` contient le nombre d'itérations à effectuer. Après chaque itération, on enlève 1 à ce nombre. La condition d'arrêt de la boucle teste si toutes les itérations ont été faites.
  Il est garanti que l'on sorte de la boucle car :
  - la valeur de `c` est un entier,
  - elle décroît strictement après chaque itération.
  - si $c < 0$, on n'entre pas dans la boucle.

Le variant est une variable

```python
#division euclidienne n/d; d>0 par hyp.
q,r=0,n
while r >= d:
    q = q + 1
    r = r - d
```

-
-
-

Le variant est une variable

```
#division euclidienne n/d, d>0 par hyp.
q,r=0,n
while r >= d:
    q = q + 1
    r = r - d
```

Chercher une quantité v qui vérifie bien : être un entier positif tout au long de l'algorithme ; décroître strictement après chaque itération. Si $v < 0$: on sort de la boucle

Le variant est une variable

```python
#division euclidienne n/d; d>0 par hyp.
q,r=0,n
while r >= d:
    q = q + 1
    r = r - d
```

- Etat initial :deux variables `n` et `d`. On veut qu'à la fin : `q` et `r` contiennent le quotient et le reste de la division euclidienne de la valeur de `n` par la valeur de `d`.
-
-

Le variant est une variable

```
#division euclidienne n/d; d>0 par hyp.
q,r=0,n
while r >= d:
    q = q + 1
    r = r - d
```

-
- Succession des états avec **n = 17** et **d = 4** hors **n** et **d** qui ne sont pas modifiés :
  $\begin{pmatrix} q \\ 0 \end{pmatrix}$ $\begin{pmatrix} r \\ 17 \end{pmatrix}$ (état initial) | $\begin{pmatrix} q \\ 1 \end{pmatrix}$ $\begin{pmatrix} r \\ 13 \end{pmatrix}$
  $\begin{pmatrix} q \\ 2 \end{pmatrix}$ $\begin{pmatrix} r \\ 9 \end{pmatrix}$ | $\begin{pmatrix} q \\ 4 \end{pmatrix}$ $\begin{pmatrix} r \\ 1 \end{pmatrix}$ (état final)
-

Le variant est une variable

```
#division euclidienne n/d; d>0 par hyp.
q,r=0,n
while r >= d:
    q = q + 1
    r = r - d
```

-
-
- Terminaison : Si $d > 0$ positif, le variant r (qui est entier) diminue à chaque étape. Et si $r < 0$, alors $r < d$ car $d > 0$: et on sort de la boucle. Terminaison OK

Le variant n'est pas une variable

```python
#p est entier
assert type(p) == int
c = 0
while p > 0:
  if c == 0:
      p = p - 2
      c = 1
  else:
      p = p + 1
      c = 0
```

.

Le variant n'est pas une variable

```
#p est entier
assert type(p) == int
c = 0
while p > 0:
    if c == 0:
        p = p - 2
        c = 1
    else:
        p = p + 1
        c = 0
```

.

Partant de l'état initial $\begin{pmatrix} \mathrm{p} \\ 5 \end{pmatrix}$, on obtient successivement les états

$$[\begin{pmatrix} \mathrm{c} \\ 0 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 5 \end{pmatrix}] \quad [\begin{pmatrix} \mathrm{c} \\ 1 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 3 \end{pmatrix}] \quad [\begin{pmatrix} \mathrm{c} \\ 0 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 4 \end{pmatrix}]$$

$$[\begin{pmatrix} \mathrm{c} \\ 1 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 2 \end{pmatrix}] \quad [\begin{pmatrix} \mathrm{c} \\ 0 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 3 \end{pmatrix} \quad [\begin{pmatrix} \mathrm{c} \\ 1 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 1 \end{pmatrix}]$$

$$[\begin{pmatrix} \mathrm{c} \\ 0 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 2 \end{pmatrix}] \quad [\begin{pmatrix} \mathrm{c} \\ 1 \end{pmatrix} ; \begin{pmatrix} \mathrm{p} \\ 0 \end{pmatrix}]$$

Le variant n'est pas une variable

```python
#p est entier
assert type(p) == int
c = 0
while p > 0:
  if c == 0:
      p = p - 2
      c = 1
  else:
      p = p + 1
      c = 0
```

ni **p** ni **c** ne sont des quantités entières décroissantes.

Le variant n'est pas une variable

Les passages dans la boucle sont numérotés à partir de 1.

- Notations : On note c_i le contenu de `c` à l'étape i et p_i le contenu de `p` à la fin du passage i dans la boucle. p_0 et c_0 sont les valeurs avant l'entrée dans la boucle pour la première fois.
  On cherche ICI le variant comme une combinaison linéaire de p_i et c_i.

Le variant n'est pas une variable

Les passages dans la boucle sont numérotés à partir de 1.

- Notations : On note c_i le contenu de `c` à l'étape i et p_i le contenu de `p` à la fin du passage i dans la boucle. p_0 et c_0 sont les valeurs avant l'entrée dans la boucle pour la première fois.
  On cherche ICI le variant comme une combinaison linéaire de p_i et c_i.
- Candidat variant : $2p_i + 3c_i$.
  On montre que c'est une quantité entière strictement décroissante qui, lorsqu'elle dévient négative, fait sortir de la boucle.

Le variant n'est pas une variable

Les passages dans la boucle sont numérotés à partir de 1.

- Notations : On note c_i le contenu de `c` à l'étape i et p_i le contenu de `p` à la fin du passage i dans la boucle. p_0 et c_0 sont les valeurs avant l'entrée dans la boucle pour la première fois.
  On cherche ICI le variant comme une combinaison linéaire de p_i et c_i.
- Candidat variant : $2p_i + 3c_i$.
  On montre que c'est une quantité entière strictement décroissante qui, lorsqu'elle dévient négative, fait sortir de la boucle.
- Sortie de boucle : Si au tour i, $2p_i + 3c_i < 0$ alors $2p_i < -3c_i <= -3 \times 0 = 0$. Donc sortie de boucle !

Le variant n'est pas une variable

- Hérédité. **A la fin de l'étape** i **:** Si $p_i \leq 0$, alors on sort de la boucle (pas de passage $i+1$) et l'algorithme termine.
  On suppose $p_i > 0$.
  On compare $2p_i + 3c_i$ avec $2p_{i+1} + 3c_{i+1}$.
  1. Si $c_i = 0$, alors $c_{i+1} = 1$, $p_{i+1} = p_i - 2$ et $2p_i + 3c_i = 2p_i$.
  $$0 \leq 1 \leq 2p_{i+1} + 3c_{i+1} = 2p_i - 4 + 3 = 2p_i - 1 < 2p_i = 2p_i + 3c_i.$$
  2. Si $c_i = 1$, alors $c_{i+1} = 0$, $p_{i+1} = p_i + 1$ et $2p_i + 3c_i = 2p_i + 3$.
  $$0 \leq 2p_i \leq 2p_{i+1} + 3c_{i+1} = 2p_i + 2 + 3 \times 0 = 2p_i + 2 < 2p_i + 3 = 2p_i + 3c_i.$$
- La quantité $2p_i + 3c_i$ est entière, strictement décroissante et, si elle négative, on sort de la boucle. L'algorithme termine.

Exercice

Exercice

Ecrire un programme pour déterminer le rang du dernier terme strictement positif de la suite récurrente définie par $u_{n+1} = \frac{1}{2}u_n - 3n$.

Puis, montrer que le programme termine.

Exercice

Exercice

Ecrire un programme pour déterminer le rang du dernier terme strictement positif de la suite récurrente définie par $u_{n+1} = \frac{1}{2}u_n - 3n$.
Puis, montrer que le programme termine.

```python
u = a
n = 0
while u > 0:
  u = 0.5*u-3*n
  n+=1
n-=1
```

Exercice

Exercice

Ecrire un programme pour déterminer le rang du dernier terme strictement positif de la suite récurrente définie par $u_{n+1} = \frac{1}{2}u_n - 3n$.
Puis, montrer que le programme termine.

```
u = a
n = 0
while u > 0:
    u = 0.5*u-3*n
    n+=1
n-=1
```

On va étudier la terminaison

Exercice

- `On étudie` u_i `et` n_i`.` Soit u_i le contenu de **u** à la fin du passage i dans la boucle, n_i celui de **n**. Avant l'entrée dans la boucle, les valeurs sont u_0, n_0

Exercice

- `On étudie` u_i `et` n_i`.` Soit u_i le contenu de **u** à la fin du passage i dans la boucle, n_i celui de **n**. Avant l'entrée dans la boucle, les valeurs sont u_0, n_0
- u_i n'est pas un entier (donc pas un variant). On montre que c'est une quantité décroissante tendant vers $-\infty$, donc qui sera négative à partir d'un certain rang (d'où l'arrêt).

Exercice

- $\boxed{\texttt{On étudie } u_i \texttt{ et } n_i\texttt{.}}$ Soit u_i le contenu de **u** à la fin du passage i dans la boucle, n_i celui de **n**. Avant l'entrée dans la boucle, les valeurs sont u_0, n_0
- u_i n'est pas un entier (donc pas un variant). On montre que c'est une quantité décroissante tendant vers $-\infty$, donc qui sera négative à partir d'un certain rang (d'où l'arrêt).
- Il est immédiat que $n_0 = 0$ et que $n_i = i$

Exercice

- Cas de base. Si $u_0 \leq 0$, on n'entre pas dans la boucle et le programme termine.
  On suppose donc $u_0 > 0$.

Exercice

- Cas de base. Si $u_0 \leq 0$, on n'entre pas dans la boucle et le programme termine.
  On suppose donc $u_0 > 0$.
- Hérédité. On suppppose qu'il y a un passage $i \geq 1$.

Exercice

- Cas de base. Si $u_0 \leq 0$, on n'entre pas dans la boucle et le programme termine.
  On suppose donc $u_0 > 0$.
- Hérédité. On supppose qu'il y a un passage $i \geq 1$.
  - Si $u_i \leq 0$ le programme termine.

Exercice

- Cas de base. Si $u_0 \leq 0$, on n'entre pas dans la boucle et le programme termine.
  On suppose donc $u_0 > 0$.
- Hérédité. On supppose qu'il y a un passage $i \geq 1$.
  - Si $u_i \leq 0$ le programme termine.
  - Supposons $u_i > 0$. Comme $i > 0$, on a $n_i \geq 1$.
    Alors

$$u_{i+1} = \frac{1}{2}u_i - 3n_i \leq \frac{1}{2}u_i - 3 < u_i - 1 \text{ car } u_i > 0 \text{ et } n_i > 0.$$

Exercice

- Cas de base. Si $u_0 \leq 0$, on n'entre pas dans la boucle et le programme termine.
  On suppose donc $u_0 > 0$.
- Hérédité. On supppose qu'il y a un passage $i \geq 1$.
  - Si $u_i \leq 0$ le programme termine.
  - Supposons $u_i > 0$. Comme $i > 0$, on a $n_i \geq 1$.
    Alors
    $$u_{i+1} = \frac{1}{2}u_i - 3n_i \leq \frac{1}{2}u_i - 3 < u_i - 1 \text{ car } u_i > 0 \text{ et } n_i > 0.$$
  - Le programme termine car
    $u_n < u_{n-1} - 1 < u_{n-2} - 2 < \cdots < u_1 - (n-1)$ qui tend vers $-\infty$ puisque u_1 est une constante. Donc u_i sera négatif a.p.c.r ce qui est la condition de sortie de boucle.

Exercice

- Cas de base. Si $u_0 \leq 0$, on n'entre pas dans la boucle et le programme termine.
  On suppose donc $u_0 > 0$.
- Hérédité. On supppose qu'il y a un passage $i \geq 1$.
  - Si $u_i \leq 0$ le programme termine.
  - Supposons $u_i > 0$. Comme $i > 0$, on a $n_i \geq 1$.
    Alors
    $$u_{i+1} = \frac{1}{2}u_i - 3n_i \leq \frac{1}{2}u_i - 3 < u_i - 1 \text{ car } u_i > 0 \text{ et } n_i > 0.$$
  - Le programme termine car
    $u_n < u_{n-1} - 1 < u_{n-2} - 2 < \cdots < u_1 - (n-1)$ qui tend vers $-\infty$ puisque u_1 est une constante. Donc u_i sera négatif a.p.c.r ce qui est la condition de sortie de boucle.
- Dans la preuve ci-dessus, on n'a pas vraiment identifié un *variant* au sens de la définition (u_n n'est pas une quantité entière).

Boucle infinie

- Le programme suivant termine-t-il toujours ?

Boucle infinie

- Le programme suivant termine-t-il toujours ?

Boucle infinie

- Le programme suivant termine-t-il toujours ?

```
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

Boucle infinie

- Le programme suivant termine-t-il toujours ?

```
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

Boucle infinie

- Le programme suivant termine-t-il toujours ?

  - **On met -1 dans** `c`

```
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

Boucle infinie

- Le programme suivant termine-t-il toujours ?

```
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

- On met -1 dans c
- Après le premier passage c contient -2, après le troisième -3 etc...

Boucle infinie

- Le programme suivant termine-t-il toujours ?

```
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

- On met -1 dans $\boxed{c}$
- Après le premier passage $\boxed{c}$ contient -2, après le troisième -3 etc...
- En notant c_i le contenu de $\boxed{c}$ après la i-ème itération on montre $c_i = -(i+1)$.

Boucle infinie

- Le programme suivant termine-t-il toujours ?

```python
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

- On met -1 dans $\boxed{\texttt{c}}$
- Après le premier passage $\boxed{\texttt{c}}$ contient -2, après le troisième -3 etc...
- En notant c_i le contenu de $\boxed{\texttt{c}}$ après la i-ème itération on montre $c_i = -(i+1)$.
- Cas de base $c_0 = -1 = -(0+1)$

Boucle infinie

- Le programme suivant termine-t-il toujours ?

```
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

- On met -1 dans `c`
- Après le premier passage `c` contient -2, après le troisième -3 etc...
- En notant c_i le contenu de `c` après la i-ème itération on montre $c_i = -(i+1)$.
- Cas de base $c_0 = -1 = -(0+1)$
- Hérédité. Si pour $i \in \mathbb{N}$, $c_i = -(i+1)$, on entre dans la boucle ($c_i \neq 0$). Alors $c_{i+1} = c_i - 1 = -(i+1) - 1 = -((i+1) - 1)$. OK.

Boucle infinie

- Le programme suivant termine-t-il toujours ?

```
# pour calculer 2^c
p,c = 1,n
while c != 0:
  p = p * 2
  c = c - 1
```

- On met -1 dans `c`
- Après le premier passage `c` contient -2, après le troisième -3 etc...
- En notant c_i le contenu de `c` après la i-ème itération on montre $c_i = -(i+1)$.
- Cas de base $c_0 = -1 = -(0+1)$
- Hérédité. Si pour $i \in \mathbb{N}$, $c_i = -(i+1)$, on entre dans la boucle ($c_i \neq 0$). Alors $c_{i+1} = c_i - 1 = -(i+1) - 1 = -((i+1)-1)$. OK.
- Donc c_i n'est jamais nul. Boucle

Boucles **for**

- Une boucle **for** termine toujours si ses instructions internes terminent *et* si la liste sur laquelle on boucle n'est pas modifiée en cours d'exécution.
- Ceci termine :

```
res, t = 0, [1,2,3,4]
for e in t:
    res+=e
```

- Mais pas cela (cas ou un appel interne ne termine pas) :

```
def f(x):
    while x > 0:
          x = x + 1
    return x
res, t = 0, [1,2,3]
for e in t:
    res += f(e)
```

Boucle for

Exercice

Imaginer une situation de boucle **for** pour laquelle la liste de référence est modifiée dynamiquement, entrainant une boucle infinie.

Boucle for

Exercice

Imaginer une situation de boucle **for** pour laquelle la liste de référence est modifiée dynamiquement, entrainant une boucle infinie.

```
t = [1,2,3]
for i in t:
  t.append(i)
```

Position du problème

- Objectif : Déterminer si un programme est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).

Position du problème

- Objectif : Déterminer si un programme est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :

Position du problème

- Objectif : Déterminer si un programme est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :
  1. qui est vérifiée avant d'entrer dans la boucle,

Position du problème

- Objectif : Déterminer si un programme est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :
  1. qui est vérifiée avant d'entrer dans la boucle,
  2. qui si elle est vérifiée avant une itération est vérifiée après celle-ci,

Position du problème

- Objectif : Déterminer si un programme est correct vis à vis de sa *spécification* (c.a.d s'il fait bien ce qu'on attend de lui).
- Moyen : On utilise un *invariant* de boucle, c'est-à-dire une propriété :
  1. qui est vérifiée avant d'entrer dans la boucle,
  2. qui si elle est vérifiée avant une itération est vérifiée après celle-ci,
  3. qui lorsqu'elle est vérifiée en sortie de boucle permet d'en déduire que le programme est correct.

Exercice : calcul de 2^n

Montrer que ce programme de calcul de 2^n est correct pour tout entier $n \geq 0$, c.a.d qu'en sortie de boucle `p` contient 2^n.

```
p,c = 1,n
while c > 0:
  p = p * 2
  c = c - 1
```

On note c_i, p_i les contenus des variables `c` et `p` après l'itération i.

- .
-

Exercice : calcul de 2^n

```
p,c = 1,n
while c > 0:
  p = p * 2
  c = c - 1
```

On note c_i, p_i les contenus des variables `c` et `p` après l'itération i.

- Avant l'entrée dans la boucle : $c_0 = n$, $p_0 = 1$.
- .
-

Exercice : calcul de 2^n

```
p,c = 1,n
while c > 0:
  p = p * 2
  c = c - 1
```

On note c_i, p_i les contenus des variables `c` et `p` après l'itération i.

- Avant l'entrée dans la boucle : $c_0 = n$, $p_0 = 1$.
- Après l'itération i : $c_{i+1} = c_i - 1$ et $p_{i+1} = 2p_i$.
- .
-

Exercice : calcul de 2^n

```python
p,c = 1,n
while c > 0:
  p = p * 2
  c = c - 1
```

On note c_i, p_i les contenus des variables `c` et `p` après l'itération i.

- Avant l'entrée dans la boucle : $c_0 = n$, $p_0 = 1$.
- Après l'itération i : $c_{i+1} = c_i - 1$ et $p_{i+1} = 2p_i$.
- Invariant potentiel $\boxed{I(i)\ :\ c_i \geq 0 \ \texttt{et}\ p_i = 2^{n-c_i}}$. Astuce : faire figurer le signe de `c` dans l'invariant ! .
-

Exercice : calcul de 2^n

```python
p,c = 1,n
while c > 0:
  p = p * 2
  c = c - 1
```

On note c_i, p_i les contenus des variables `c` et `p` après l'itération i.

- Invariant potentiel $\boxed{I(i)\ :\ c_i \geq 0 \texttt{ et } p_i = 2^{n-c_i}}$. Astuce : faire figurer le signe de `c` dans l'invariant !
- Cas de base, itération 0 (avant l'entrée dans la boucle) : $c_0 = n \geq 0$, $p_0 = 1 = 2^0 = 2^{n-c_0}$: OK.
-

Exercice : calcul de 2^n

```
p,c = 1,n
while c > 0:
  p = p * 2
  c = c - 1
```

On note c_i, p_i les contenus des variables `c` et `p` après l'itération i.

- Invariant potentiel $\boxed{I(i) \ : \ c_i \geq 0 \ \texttt{et} \ p_i = 2^{n-c_i}}$. Astuce : faire figurer le signe de `c` dans l'invariant ! .
- Hérédité. On suppose $I(i)$ vérifié.

Exercice : calcul de 2^n

```
p,c = 1,n
while c > 0:
  p = p * 2
  c = c - 1
```

On note c_i, p_i les contenus des variables `c` et `p` après l'itération i.

- Invariant potentiel $\boxed{I(i) \; : \; c_i \geq 0 \text{ et } p_i = 2^{n-c_i}}$. Astuce : faire figurer le signe de `c` dans l'invariant ! .
- Hérédité. On suppose $I(i)$ vérifié.
  1. Si on entre dans la boucle, alors $c_i > 0$ et $c_{i+1} = c_i - 1 \geq 0$. De plus $p_{i+1} = 2p_i = 2^{n-c_i+1} = 2^{n-(c_i-1)} = 2^{n-c_{i+1}}$. Donc $I(i+1)$ vérifié.
  2. Si on sort de la boucle, alors $c_i \leq 0$ (condition de sortie) et $c_i \geq 0$ (par $I(i)$) donc $c_i = 0$, donc $p_i = 2^{n-c_i} = 2^n$.

Exercice 2 : division euclidienne

Montrer que ce programme de division euclidienne d'un nombre entier naturel $n \geq 0$ par un entier $d > 0$ est correct.

```python
#division euclidienne n/d; d>0 par hyp.
q,r=0,n
while r >= d:
  q = q + 1
  r = r - d
```

Exercice 2 : division euclidienne

- ```python
  #division euclidienne n/d; d>0 par hyp.
  q,r=0,n
  while r >= d:
    q = q + 1
    r = r - d
  ```

- On note r_i, q_i les contenus des variables `r,q` après l'itération i.
  ```

Exercice 2 : division euclidienne

- ```python
  #division euclidienne n/d; d>0 par hyp.
  q,r=0,n
  while r >= d:
    q = q + 1
    r = r - d
  ```
- On note r_i, q_i les contenus des variables `r,q` après l'itération i.
- **Invariant de boucle :** $I(i) = q_i \geq 0 \wedge r_i \geq 0 \wedge n = q_i d + r_i$.
  ```

Exercice 2 : division euclidienne

- ```python
  #division euclidienne n/d; d>0 par hyp.
  q,r=0,n
  while r >= d:
    q = q + 1
    r = r - d
  ```
- On note r_i, q_i les contenus des variables `r,q` après l'itération i.
- Invariant de boucle : $I(i) = q_i \geq 0 \wedge r_i \geq 0 \wedge n = q_i d + r_i$.
- Entrée dans la boucle $q_0 = 0 \geq 0$; $r_0 = n \geq 0$ et $r_0 + q_0 d = n$: OK.
  ```

Exercice 2 : division euclidienne

- ```
  #division euclidienne n/d; d>0 par hyp.
  q,r=0,n
  while r >= d:
    q = q + 1
    r = r - d
  ```
- On note r_i, q_i les contenus des variables `r,q` après l'itération i.
- Invariant de boucle : $I(i) = q_i \geq 0 \wedge r_i \geq 0 \wedge n = q_i d + r_i$.
- Entrée dans la boucle $q_0 = 0 \geq 0$; $r_0 = n \geq 0$ et $r_0 + q_0 d = n$: OK.
- Hérédité. On suppose $I(i)$ vérifié $(i \geq 0)$.
  ```

Exercice 2 : division euclidienne

-
```python
#division euclidienne n/d; d>0 par hyp.
q,r=0,n
while r >= d:
  q = q + 1
  r = r - d
```

- On note r_i, q_i les contenus des variables `r,q` après l'itération i.
- Invariant de boucle : $I(i) = q_i \geq 0 \land r_i \geq 0 \land n = q_i d + r_i$.
- Entrée dans la boucle $q_0 = 0 \geq 0$; $r_0 = n \geq 0$ et $r_0 + q_0 d = n$: OK.
- Hérédité. On suppose $I(i)$ vérifié ($i \geq 0$).
  1. Si on entre dans la boucle. $r_i \geq d$. $r_{i+1} = r_i - d \geq 0$, $q_{i+1} = q_i + 1 \geq q_i \geq 0$. Et $r_{i+1} = r_i - d = n - q_i d - d = n - (q_i + 1)d = n - q_{i+1} d$: OK.

Exercice 2 : division euclidienne

```
#division euclidienne n/d; d>0 par hyp.
q,r=0,n
while r >= d:
  q = q + 1
  r = r - d
```

- On note r_i, q_i les contenus des variables `r,q` après l'itération i.
- Invariant de boucle : $I(i) = q_i \geq 0 \wedge r_i \geq 0 \wedge n = q_i d + r_i$.
- Entrée dans la boucle $q_0 = 0 \geq 0$; $r_0 = n \geq 0$ et $r_0 + q_0 d = n$: OK.
- Hérédité. On suppose $I(i)$ vérifié ($i \geq 0$).
  1. Si on entre dans la boucle. $r_i \geq d$. $r_{i+1} = r_i - d \geq 0$, $q_{i+1} = q_i + 1 \geq q_i \geq 0$. Et $r_{i+1} = r_i - d = n - q_i d - d = n - (q_i + 1)d = n - q_{i+1} d$: OK.
  2. Sortie. $r_i \geq 0$ et $r_i < d$ et $q_i \geq 0$ et $n = q_i d + r_i$.

Terminaison d'un calcul de produit

```
x = a
y = b
r = 0
while  y > 0 :
    r = r+x
    y = y-1
#on veut que r contienne la valeur de a * b
```

Démonstration.

□

Terminaison d'un calcul de produit

```
while  y > 0 :# on suppose a,b entiers, a>=0, b>0
    r,y = r+x, y-1
```

Démonstration.

- Variant : y_i.
- y_0 entier positif

Terminaison d'un calcul de produit

```
while   y > 0 :# on suppose a,b entiers, a>=0, b>0
    r,y = r+x, y-1
```

Démonstration.

- Variant : y_i.
- y_0 entier positif
- Si $y_i > 0$ (sinon on sort de la boucle), alors $y_{i+1} = y_i - 1 < y_i$; $y_{i+1} \geq 1 - 1 = 0$ et $y_{i+1} \in \mathbb{N}$.

$\square$

Terminaison d'un calcul de produit

```
while   y > 0 :# on suppose a,b entiers, a>=0, b>0
    r,y = r+x, y-1
```

Démonstration.

- Variant : y_i.
- y_0 entier positif
- Si $y_i > 0$ (sinon on sort de la boucle), alors $y_{i+1} = y_i - 1 < y_i$; $y_{i+1} \geq 1 - 1 = 0$ et $y_{i+1} \in \mathbb{N}$.
- Terminaison prouvée car (y_i) est une suite d'entiers positifs strictement décroissante.

□

Correction d'un calcul de produit

```
x = a
y = b
r = 0
while  y > 0 :
    r = r+x
    y = y-1
#on veut que r contienne la valeur de a * b
```

Démonstration.

- Invariant : $\mathrm{Inv}(i)$: $(r_i + x_i \times y_i = a \times b) \wedge (y_i \geq 0)$. Remarque : x_i constant.

Correction d'un calcul de produit

```python
while  y > 0 :# on suppose a,b entiers, a>=0, b>0
    r,y = r+x, y-1
```

Démonstration.

- Invariant : $\mathrm{Inv}(i)$: $(r_i + x_i \times y_i = a \times b) \wedge (y_i \geq 0)$. Remarque : x_i constant.
- $r_0 + x_0 \times y_0 = 0 + a \times b$. Et $y_0 = b > 0$. Cas de base : OK

Correction d'un calcul de produit

```
while  y > 0 :# on suppose a,b entiers, a>=0, b>0
    r,y = r+x, y-1
```

Démonstration.

- Invariant : $\mathrm{Inv}(i)$: $(r_i + x_i \times y_i = a \times b) \wedge (y_i \geq 0)$. Remarque : x_i constant.
- $r_0 + x_0 \times y_0 = 0 + a \times b$. Et $y_0 = b > 0$. Cas de base : OK
- Si $\mathrm{Inv}(i)$ après le passage i :

Correction d'un calcul de produit

```
while  y > 0 :# on suppose a,b entiers, a>=0, b>0
    r,y = r+x, y-1
```

Démonstration.

- Invariant : $\mathrm{Inv}(i)$: $(r_i + x_i \times y_i = a \times b) \wedge (y_i \geq 0)$. Remarque : x_i constant.
- $r_0 + x_0 \times y_0 = 0 + a \times b$. Et $y_0 = b > 0$. Cas de base : OK
- Si $\mathrm{Inv}(i)$ après le passage i :
    - Si $y_i > 0$, il y a un passage $i+1$. $y_{i+1} = y_i - 1$ et $r_{i+1} = r_i + x_i = r_i + a$.
      Comme $y_i > 0$, on a $y_{i+1} = y_i - 1 \geq 0$. OK
      Alors $r_{i+1} + a \times y_{i+1} = r_i + a + a(y_i - 1) = r_i + ay_i = a \times b$: hérédité OK

Correction d'un calcul de produit

```
while  y > 0 :# on suppose a,b entiers, a>=0, b>0
    r,y = r+x, y-1
```

Démonstration.

- Invariant : $\mathrm{Inv}(i)$: $(r_i + x_i \times y_i = a \times b) \wedge (y_i \geq 0)$. Remarque : x_i constant.
- $r_0 + x_0 \times y_0 = 0 + a \times b$. Et $y_0 = b > 0$. Cas de base : OK
- Si $\mathrm{Inv}(i)$ après le passage i :
    - Si $y_i > 0$, il y a un passage $i+1$. $y_{i+1} = y_i - 1$ et $r_{i+1} = r_i + x_i = r_i + a$.
      Comme $y_i > 0$, on a $y_{i+1} = y_i - 1 \geq 0$. OK
      Alors $r_{i+1} + a \times y_{i+1} = r_i + a + a(y_i - 1) = r_i + ay_i = a \times b$: hérédité OK
    - si $y_i \leq 0$ alors comme $y_i \geq 0$, $y_i = 0$. Et on a $r_i = r_i + 0 = r_i + a \times y_i = a \times b$. On a bien ce qu'on veut !

Conclusion

On retient :

- Pour montrer la *Terminaison* on cherche un *variant*, c.a.d une quantité entière (le plus souvent) positive strictement décroissante. Lorsque le variant devient négatif, c'est le signe qu'on sort de la boucle.
- Pour montrer la *Correction* on cherche un *invariant*, c.a.d une propriété vérifiée à chaque passage dans la boucle.
  Lorsqu'on sort de la boucle, la condition de sortie + l'invariant attestent que le programme fait bien ce qu'on en attend.
- Le variant est une suite de nombres indicée par les passages dans la boucle, l'invariant est suite de formules logiques indicées par les passages dans la boucle.

Syracuse

Exercice

Etablir la terminaison de

```python
def syraccuse(n):
    assert n>=0 and type(n)==int
    x=n
    while x!=1:
        if x%2==0:
            x=x//2
        else:
            x=3*x+1
    return x
```

Syracuse

- En 1928, Lothar Collatz s'intéresse aux itérations dans les nombres entiers, qu'il représente au moyen de graphes et d'hypergraphes. Il invente alors le problème $3x + 1$, et le présentera souvent ensuite dans ses séminaires.

Syracuse

- En 1928, Lothar Collatz s'intéresse aux itérations dans les nombres entiers, qu'il représente au moyen de graphes et d'hypergraphes. Il invente alors le problème $3x + 1$, et le présentera souvent ensuite dans ses séminaires.
- En 1952, lors d'une visite à Hambourg, Collatz explique son problème à Helmut Hasse.

Syracuse

- En 1928, Lothar Collatz s'intéresse aux itérations dans les nombres entiers, qu'il représente au moyen de graphes et d'hypergraphes. Il invente alors le problème $3x+1$, et le présentera souvent ensuite dans ses séminaires.
- En 1952, lors d'une visite à Hambourg, Collatz explique son problème à Helmut Hasse.
- Ce dernier le diffuse en Amérique à l'université de Syracuse : la suite de Collatz prend alors le nom de « suite de Syracuse » .

Syracuse

- En 1928, Lothar Collatz s'intéresse aux itérations dans les nombres entiers, qu'il représente au moyen de graphes et d'hypergraphes. Il invente alors le problème $3x + 1$, et le présentera souvent ensuite dans ses séminaires.
- En 1952, lors d'une visite à Hambourg, Collatz explique son problème à Helmut Hasse.
- Ce dernier le diffuse en Amérique à l'université de Syracuse : la suite de Collatz prend alors le nom de « suite de Syracuse » .
- Entre temps, le mathématicien polonais Stanislas Ulam le répand dans le Laboratoire national de Los Alamos.

Syracuse

- En 1928, Lothar Collatz s'intéresse aux itérations dans les nombres entiers, qu'il représente au moyen de graphes et d'hypergraphes. Il invente alors le problème $3x + 1$, et le présentera souvent ensuite dans ses séminaires.
- En 1952, lors d'une visite à Hambourg, Collatz explique son problème à Helmut Hasse.
- Ce dernier le diffuse en Amérique à l'université de Syracuse : la suite de Collatz prend alors le nom de « suite de Syracuse » .
- Entre temps, le mathématicien polonais Stanislas Ulam le répand dans le Laboratoire national de Los Alamos.
- Dans les années 1960, le problème est repris par le mathématicien Shizuo Kakutani qui le diffuse dans les universités Yale et Chicago.

Syracuse

- En 1928, Lothar Collatz s'intéresse aux itérations dans les nombres entiers, qu'il représente au moyen de graphes et d'hypergraphes. Il invente alors le problème $3x + 1$, et le présentera souvent ensuite dans ses séminaires.
- En 1952, lors d'une visite à Hambourg, Collatz explique son problème à Helmut Hasse.
- Ce dernier le diffuse en Amérique à l'université de Syracuse : la suite de Collatz prend alors le nom de « suite de Syracuse » .
- Entre temps, le mathématicien polonais Stanislas Ulam le répand dans le Laboratoire national de Los Alamos.
- Dans les années 1960, le problème est repris par le mathématicien Shizuo Kakutani qui le diffuse dans les universités Yale et Chicago.
- Cette conjecture mobilisa tant les mathématiciens durant les années 1960, en pleine guerre froide, qu'une plaisanterie courut selon laquelle ce problème faisait partie d'un complot soviétique visant à ralentir la recherche américaine.

Problème de l'arrêt.

- Supposons qu'il existe une fonction **termine** qui prend en paramètres un nom de fonction et retourne un bouléen indiquant si la fonction termine dans tous les cas (**True**) ou non **False**.

Problème de l'arrêt.

- Supposons qu'il existe une fonction **termine** qui prend en paramètres un nom de fonction et retourne un bouléen indiquant si la fonction termine dans tous les cas (**True**) ou non **False**.
- On considère la fonction

```python
def absurde():
  while termine('absurde'):
     pass
  return 1
```

Problème de l'arrêt.

- Supposons qu'il existe une fonction **termine** qui prend en paramètres un nom de fonction et retourne un bouléen indiquant si la fonction termine dans tous les cas (**True**) ou non **False**.
- On considère la fonction

```python
def absurde():
  while termine('absurde'):
     pass
  return 1
```

- Si **termine('absurde')** retourne **True**, c'est à dire si l'appel **absurde()** termine, alors l'appel **absurde()** rentre dans la boucle infinie sans plus en sortir : ABSURDE

Problème de l'arrêt.

- Supposons qu'il existe une fonction **termine** qui prend en paramètres un nom de fonction et retourne un bouléen indiquant si la fonction termine dans tous les cas (**True**) ou non **False**.
- On considère la fonction

```python
def absurde():
  while termine('absurde'):
     pass
  return 1
```

- Si **termine('absurde')** retourne **True**, c'est à dire si l'appel **absurde()** termine, alors l'appel **absurde()** rentre dans la boucle infinie sans plus en sortir : ABSURDE
- Si **termine('absurde')** retourne **False**, c'est à dire si l'appel **absurde()** ne termine pas, alors l'appel **absurde()** retourne 1 donc termine : ABSURDE

Problème de l'arrêt.

- Supposons qu'il existe une fonction **termine** qui prend en paramètres un nom de fonction et retourne un bouléen indiquant si la fonction termine dans tous les cas (**True**) ou non **False**.
- On considère la fonction

```
def absurde():
  while termine('absurde'):
     pass
  return 1
```

- Si **termine('absurde')** retourne **True**, c'est à dire si l'appel **absurde()** termine, alors l'appel **absurde()** rentre dans la boucle infinie sans plus en sortir : ABSURDE
- Si **termine('absurde')** retourne **False**, c'est à dire si l'appel **absurde()** ne termine pas, alors l'appel **absurde()** retourne 1 donc termine : ABSURDE
- Bref, la recherche de variant est un art non automatisable !