TERMINAISON ET CORRECTION

À la fin de ce chapitre, je sais :

- ☞ expliquer le problème de la terminaison d'un algorithme
- ☞ utiliser un variant de boucle pour prouver la terminaison d'un algorithme
- ☞ expliquer la notion de correction d'un algorithme
- ☞ utiliser un invariant de boucle pour prouver la correction d'un algorithme

A Un programme se termine-t-il ?

Si un programme ne s'arrête pas de lui même, c'est soit qu'il :

- converge très lentement vers une solution,
- il ne s'arrêtera jamais, tournant en rond, répétant **indéfiniment** les mêmes séquences.

Cette dernière possibilité est à la fois utile, par exemple pour une système d'exploitation et des serveurs qui doivent servir en permanence, et en même temps guère admissible lorsqu'on souhaite avoir la réponse à un problème donné. C'est pourquoi les informaticiens s'intéressent tout particulièrement à la question de l'arrêt de leurs programmes.

Parmi les éléments de notre pseudo-langage algorithmique, les seuls qui pourraient engendrer une exécution infinie d'un programme sont les instructions de boucle, **pour** et **tant que**. C'est pourquoi les sections qui suivent y accorde un intérêt tout particulier.

■ **Exemple 1 — S'arrêtera ? S'arrêtera pas ?.** Imaginons un instant que nous disposions d'un algorithme capable de se prononcer sur la terminaison d'un autre algorithme et nommons le `termine?`. Cet algorithme renvoie vrai si l'algorithme passé en paramètre se termine et faux sinon.

Imaginons que l'on souhaite tester la terminaison d'un autre algorithme nommé SOUS_TEST comme dans l'algorithme 1. Que penser alors de cet algorithme ? À quelle condition se termine-t-il ? Peut-on le prouver ? Naturellement, ce paradoxe souligne l'impossibilité de répondre à la question de la terminaison d'un algorithme.

On dit que le problème de l'arrêt est **indécidable**, tout comme toute propriété non triviale sur un programme (Théorème de Rice [**rice_classes_1953**]).

Algorithme 1 Test de la terminaison de l'algorithme SOUS_TEST

```
1 : Fonction TEST_TERMAISON
2 :     tant que TERMINE ?(SOUS_TEST)) est faux répéter
3 :         ne rien faire.
```

Théorème 1 — Problème de l'arrêt. Il n'existe pas de programme permettant de dire si un algorithme termine toujours ou non.

■ **Exemple 2 — Suite de Syracuse .** Prenons par exemple l'algorithme de calcul des éléments de la suite de Syracuse (cf. algorithme 2).

Une implémentation dans le langage de votre choix vous montrera que cette fonction renvoie la valeur 1 au bout d'un certain nombre d'itérations [a]. Cet algorithme se termine donc, au moins en apparence.

Il existe d'ailleurs une conjecture nommée «conjecture de Syracuse» qui formule l'hypothèse que la suite de Syracuse de n'importe quel entier strictement positif atteint 1.

Cependant, aucune théorie mathématique ou informatique n'a pu le démontrer à ce jour. Il se peut même qu'on ne puisse pas le démontrer.

a. À partir de ce rang, la suite de Syracuse est cyclique.

Algorithme 2 Calcul des éléments de la suite de Syracuse

```
1 : Fonction SYRACUSE(n)                    ▷ n est un paramètre formel d'entrée
2 :     s ← n
3 :     tant que s > 1 répéter
4 :         si s est pair alors
5 :             s ← s/2
6 :         sinon
7 :             s ← 3 * s + 1
8 :     renvoyer s
```

R D'une manière générale, on en peut donc pas conclure sur la terminaison d'un algorithme. Néanmoins, dans de nombreux cas particuliers et en utilisant des propriétés mathématiques, on peut démontrer que des algorithmes se terminent. Ceci est l'objet des deux sections suivantes.

B Variant de boucle

■ **Définition 1 — Variant de boucle (fonction de terminaison).** Un variant de boucle v : $\mathcal{D} \longrightarrow \mathbb{N}$ est une fonction :

1. à valeurs entières et positives ($\mathbb{N}$),
2. strictement décroissante à chaque itération de la boucle,
3. qui dépend des variables de la boucle.

R Le variant de boucle est une suite de valeurs entières strictement décroissante et positive et donc minorée par 0. Le **théorème de la limite monotone** affirme qu'il peut **atteindre n'importe quelle valeur positive inférieure à sa valeur de départ en un nombre d'itérations fini**. Dès lors, si la condition d'arrêt de la boucle est le franchissement de cette valeur, l'arrêt se déclenche et le programme termine.

R D'une manière générale, la condition d'arrêt d'une boucle doit impliquer le franchissement d'une valeur particulière par le variant de boucle.

C Démontrer la terminaison d'un algorithme

M **Méthode 1 — Démontrer la terminaison d'un algorithme** Pour prouver la terminaison d'un algorithme, si cela est possible, il suffit souvent de prouver que les boucles se terminent et donc de :

1. trouver un variant de boucle (entier, positif, strictement décroissant),
2. montrer que le variant est minoré, qu'il franchit nécessairement une valeur limite liée à la condition d'arrêt.

R La boucle **pour** ne pose généralement [a] pas de problèmes au niveau de la terminaison car le nombre d'itérations est connu et explicité clairement dans la syntaxe de la boucle. Il suffit alors de choisir un variant lié à la variable d'itération de la boucle, typiquement $n-i$ ou i. Ce n'est pas le cas des boucles tant que.

a. généralement, car on peut aussi mal formuler une boucle **pour** dans n'importe quel langage et faire que cela ne soit pas explicite…

■ **Exemple 3 — Fonction produit.** Prenons l'exemple simple du calcul du produit de deux nombres entiers naturels a et b par l'algorithme 3 : la condition d'arrêt de la boucle tant que $c < b$ peut s'écrire $b - c > 0$. Par ailleurs, c est incrémenté à chaque tour de boucle. La fonction

$$\begin{aligned} v : \mathbb{N} &\longrightarrow \mathbb{N} \\ c &\longmapsto b - c \end{aligned}$$

est une fonction à valeurs entières, strictement décroissante, initialisée à une valeur positive, c'est à dire un variant de boucle. v atteint nécessairement la valeur 0, au bout d'un

certain nombre d'itérations. Il s'en suit que la condition d'arrêt $b-c>0$ devient fausse et déclenche l'arrêt de la boucle. Donc, l'algorithme 3 se termine.

On procède de la même manière pour l'algorithme 4. Construisons un variant de boucle à partir de la variable i. On peut poser :

$$\begin{aligned} v : \mathbb{N} &\longrightarrow \mathbb{N} \\ i &\longmapsto b-i \end{aligned}$$

Il s'agit bien d'une fonction à valeurs entières. Au démarrage, $v(0)$ vaut b qui est une valeur entière positive. À chaque tour de boucle, i est incrémenté implicitement de 1. Donc v est une fonction entière strictement décroissante, un variant de boucle. La condition de sortie de la boucle est $i>b-1$. On a $v(b)=b-b=0$. D'après le théorème de la limite monotone, v atteindra la valeur 0 au bout d'un certain nombre d'itération. Donc l'algorithme se termine.

Algorithme 3 Produit de deux nombres entiers naturels, quel pourrait-être un variant?

```
1 : Fonction PRODUIT(a, b)                          ▷ a ∈ ℕ et b ∈ ℕ.
2 :     p ← 0
3 :     c ← 0                                       ▷ c est un entier.
4 :     tant que c < b répéter
5 :         p ← p + a
6 :         c ← c + 1
7 :     renvoyer p
```

Algorithme 4 Produit de deux nombres entiers naturels, quel pourrait-être un variant?

```
1 : Fonction PRODUIT(a, b)                          ▷ a ∈ ℕ et b ∈ ℕ.
2 :     p ← 0
3 :     pour i = 0 à b − 1 répéter                  ▷ i est la variable d'itération de la boucle pour
4 :         p ← p + a
5 :     renvoyer p
```

R Il est important de noter que le test de la boucle **tant que** s'exprime avec le signe inférieur. On veillera à procéder de même, c'est à dire à exprimer ces tests à l'aide de l'opérateur $<$ ou $>$, non pas avec l'opérateur $\neq$. Ceci afin d'éviter l'erreur suivante illustrée par l'exemple 4.

■ **Exemple 4 — Boucle infinie (à ne pas reproduire !).** Reprenons l'algorithme 3 en introduisant une **petite erreur** à la ligne 6 : on incrémente de 2. La condition d'arrêt de la boucle $c \neq b$ ne garantit pas que la fonction $v(c)=b-c$ soit minorée par zéro, notamment si b est impair : v peut prendre des valeurs négatives, ce n'est donc pas un variant de boucle. Cet algorithme 5 ne termine pas toujours.

Algorithme 5 Boucle infinie (à ne pas reproduire !)

```
1 : Fonction BINF(a, b)                          ▷ a ∈ ℕ et b ∈ ℕ.
2 :     p ← 0
3 :     c ← 0                                    ▷ c est un entier.
4 :     tant que c ≠ b répéter
5 :         p ← p + a
6 :         c ← c + 2
7 :     renvoyer p
```

R De manière similaire, on évitera les tests d'égalité sur les flottants dans les conditions d'arrêt : on a toutes les chances de ne pas la vérifier, comme le montre l'exemple 5 et l'algorithme 6.

■ **Exemple 5 — Boucle infinie, de la théorie à la pratique (à ne pas reproduire !).** Dans l'algorithme 6, la condition d'arrêt doit pouvoir être vérifiée, en théorie. Dans les faits, elle ne l'est pas pour deux raisons :

1. on ne sait pas si r est un multiple de 0.1,
2. même si c'était le cas, l'utilisation des flottants (norme IEEE 754) ne permet pas d'atteindre la valeur 0.1 exactement, quelque soit le langage de programmation comme cela est expliqué au chapitre **??**.

Il faut donc éviter les conditions d'arrêt de boucle comprenant des flottants et, si jamais on y est contraint, utiliser les opérateurs de comparaison $<$ ou $<$, jamais les opérateurs $\neq$ ou ==.

Algorithme 6 Boucle infinie, condition d'arrêt sur un flottant (**ne pas reproduire !!!**))

```
1 : Fonction CAFLOT(r)                           ▷ r ∈ ℝ.
2 :     x ← 0.0                                  ▷ x ∈ ℝ.
3 :     tant que x ≠ r répéter
4 :         x ← x + 0.1
```

R Dans le cas où l'algorithme est récursif, on peut démontrer la terminaison par récurrence sur la variable de la récursivité si celle-ci décroît par sous-traction. On initialise pour la condition d'arrêt de l'algorithme, puis on fait l'hypothèse de récurrence est que l'algorithme p se termine pour $p(n)$ et on montre qu'il se termine pour $p(n+1)$. On conclut sur la terminaison pour tout $n \in \mathbb{N}$.

Si la récurrence décroit par division, on peut essayer de montrer que les paramètres des appels récursifs forment une suite des $(u_n)_{n\in\mathbb{N}}$ strictement décroissante, à valeurs positives et minorée par zéro.

■ **Exemple 6 — Terminaison de l'algorithme récursif 7 de calcul de $n!$.** On procède par récurrence sur n.

Initialisation : pour n = 0 ou 1, l'algorithme renvoie la valeur 1 et termine.

Hérédité : Supposons que l'algorithme termine pour le paramètre n. Le calcul de fact(n+1) termine, car par hypothèse de récurrence fact(n) termine et l'algorithme renvoie alors la multiplication de n et de ce résultat.

Conclusion : L'algorithme de factoriel termine pour tout $n \in \mathbb{N}$.

On aurait également pu dire que la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_{n+1} = u_n - 1$ et $u_0 = N$ est à valeurs entières, strictement décroissante et initialisée à une valeur positive. Le nombre d'appels récursifs avant d'atteindre la condition d'arrêt est donc fini. L'algorithme se termine.

Algorithme 7 Factoriel récursif

```
1 : Fonction FACT(n)
2 :     si n ⩽ 1 alors                          ▷ Condition d'arrêt
3 :         renvoyer 1
4 :     sinon
5 :         renvoyer n × FACT(n-1)              ▷ Appel récursif
```

D Le résultat est-il correct ?

Le résultat d'un algorithme est-il le bon ? Cette question hante de nombreux informaticiens et plus récemment tout ceux qui sont passés par Parcoursup !

■ **Définition 2 — Correction d'un algorithme.** La correction d'un algorithme est sa capacité à :

1. se terminer,
2. produire un résultat correct, conforme à ce qu'on attend de lui, quelles que soient les entrées.

On dit que la correction est :

partielle si le résultat est correct lorsque l'algorithme se termine,

totale si elle est partielle et que l'algorithme se termine.

Dans le cadre de la programmation structurée, on peut prouver formellement la correction des opérations d'affectation, des enchaînements d'instructions, des structures conditionnelles et des boucles. Néanmoins, nous nous intéressons dans ce qui suit plus particulièrement aux boucles.

E Invariant de boucle

Si la terminaison d'une boucle permet de conclure sur le fait qu'elle s'arrête, elle ne prouve en aucun cas la validité de son résultat. La correction d'une boucle dans le cadre de tests unitaires non exhaustifs n'est pas une preuve non plus. Or, il est difficile voire impossible de tester de manière exhaustive un algorithme et il faut être capable de garantir la correction, qu'elles que soient les entrées.

La démarche générale de démonstration de la correction des boucles est une forme de démonstration par récurrence. La propriété à démontrer est nommée *invariant de boucle*. On procède donc en trois phases : l'initialisation de l'invariant, l'hérédité et la conclusion.

■ **Définition 3 — Invariant de boucle.** Un invariant de boucle est une **propriété** liée aux variables d'un algorithme qui :

1. est vraie avant la boucle,
2. est invariante par les instructions de la boucle à chaque itération,
3. donne le résultat escompté si la condition de boucle est invalidée.

F Démontrer que le résultat d'une boucle est correct

■ **Exemple 7 — Correction de la division euclidienne.** On considère l'algorithme de la division euclidienne (cf. algorithme 8). Pour démontrer sa correction, on peut choisir l'invariant de boucle $\mathcal{I}$ *: on a l'égalité* $a = bq + r$. On procède alors en trois temps :

1. $\mathcal{I}$ est vérifié avant la boucle car $a = qb + r = r$ car $q = 0$.
2. Au cours de l'exécution de la boucle, on choisit une itération quelconque. On suppose qu'au début de cete itération, l'invariant $\mathcal{I}$ est vérifié et on a $a = bq + r$. À la fin de l'itération, $b(q+1) + (r-b) = bq + b + r - b = bq + r = a$. Donc, $\mathcal{I}$ est vérifié à la fin de l'itération, si elle est vraie à l'entrée de celle-ci.
3. Comme l'invariant de boucle $\mathcal{I}$ est vrai au démarrage de la boucle et invariant par les instructions de la boucle, on a donc, à la fin de la boucle, $a = bq + r$ et $0 \leqslant r < b$. Cet algorithme délivre donc bien le résultat escompté.

Algorithme 8 Division euclidienne $a = bq + r$

```
1 : Fonction DIVISER(a,b)                    ▷ les entrées sont des entiers naturels.
2 :     r ← a
3 :     q ← 0
4 :     tant que r ≥ b répéter
5 :         r ← r − b
6 :         q ← q + 1
7 :     renvoyer (q, r)
```

R La plupart du temps, l'invariant de boucle est une propriété liée à l'expression que l'on souhaite calculer grâce à cette boucle. Il est nécessaire de l'exprimer soit mathématiquement comme dans l'exemple 7 soit en langage naturel. Par exemple, lors de l'étude des graphes, la preuve de la correction du parcours en largeur d'un graphe est donnée. L'invariant utilisé est le suivant : *«Pour chaque sommet v ajouté à V et enfilé dans F, il existe un chemin de s à v.»*. Comme cet invariant n'est pas une expression arithmétique, il est d'autant plus important de soigner la rédaction.

R Dans tous les cas, une démonstration de correction est une rédaction et demande, comme toute preuve, un effort de structuration du propos et de l'intelligibilité.

R D'une manière pratique, on peut vérifier un invariant de boucle en ajoutant une assertion à chaque tour de boucle qui exige de le vérifier, comme dans le code ci-dessous.

```
def euclid_div(a,b):
    r=a
    q=0
    assert a == b*q + r
    while r >= b:
        r = r - b
        q = q + 1
        assert a == b*q + r
    return q,r
```

■ **Exemple 8 — Correction de l'exponentiation rapide.** L'invariant de la boucle *tant que* de l'algorithme 9 est le suivant :
$\mathcal{I} : i \times e^n = a^n$.

1. Initialisation : $\mathcal{I}$ est vérifié avant la boucle, car $i \times e^n = 1 \times a^n = a^n$.
2. Hérédité : supposons que $\mathcal{I}$ est vérifié à l'entrée de la boucle. On a donc $i \times e^n = a^n$.
   - Si n est pair, i n'est pas modifié par les instructions. Par contre, n est divisé par deux et e devient e^2. On a donc, à la suite des instructions de la boucle : $i \times e^n \longrightarrow i \times (e^2)^{n//2} = i \times e^n = a^n$, car n est pair et $2 \times (n//2) = n$.
   - Si n est impair, les instructions font évoluer i, e et n. Ainsi
   $$i \times e^n \longrightarrow (i \times e) \times (e^2)^{n//2} = i \times e^{1+n//2} = i \times e^n = a^n$$
   car n est impair.
3. Conclusion : $\mathcal{I}$ est invariant par les instructions de la boucle. Comme il est vérifié à l'entrée, il est donc vérifié à la fin de la boucle. Comme, on sort de la boucle lorsque $n = 0$, à la fin on a $i \times e^1 = ie = a^n$. L'algorithme est donc correct.

```
def ite_quick_exp(a, n):
```

Algorithme 9 Exponentiation rapide, version itérative

```
1 : Fonction EXP(a,n)                                   ▷ n est un entier naturel.
2 :     si n==0 alors
3 :         renvoyer 1
4 :     sinon si n==1 alors
5 :         renvoyer a
6 :     sinon
7 :         e ← a
8 :         i ← 1
9 :         tant que n > 1 répéter
10 :            si n est pair alors
11 :                e ← e × e
12 :                n ← n//2
13 :            sinon
14 :                i ← e × i
15 :                e ← e × e
16 :                n ← n//2
17 :        renvoyer i ∗ e
```

```
2       if n == 0:
3           return 1
4       elif n == 1:
5           return a
6       else:
7           e = a
8           i = 1
9           N = n
10          while n > 1:
11              assert i*e**n == a**N  # Loop invariant
12              if n % 2 == 0:
13                  e = e * e
14                  n = n // 2
15              else:
16                  i = i * e
17                  e = e * e
18                  n = n // 2
19          # last computed n was 1 -> i*e**1 = a**n
20          return e*i
```