ALGORITHMIQUE · CHAPITRE 29 · MISE EN PRATIQUE

Expliquer pourquoi un algorithme termine

Justifiez l'arrêt d'une boucle, vérifiez une propriété conservée et comptez une opération précise sans confondre résultat, coût et durée réelle.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Identifier une quantité qui progresse vers la sortie.
- Expliquer le résultat à l'aide d'une propriété de boucle.
- Dire exactement quelle opération est comptée.

Prérequis. Savoir suivre une boucle, une somme et des cas limites.

Pourquoi la somme finit-elle avec le bon résultat ?

Nous calculons encore la somme de 1 à n, pour n entier non négatif. Cette fois, la boucle TANT QUE permet d'observer séparément le test, l'addition et la mise à jour. Nous voulons justifier l'arrêt, expliquer le résultat et compter les additions effectuées par le corps.

Séparer résultat correct, arrêt et travail compté

La **correction** concerne le respect du résultat annoncé. La **terminaison** concerne l'arrivée à une fin du traitement. Le **coût** mesure un travail choisi, par exemple le nombre d'additions au total. Un programme peut terminer avec un mauvais résultat ; un programme peut aussi conserver une propriété utile sans jamais terminer.

Un **invariant de boucle** est une propriété vraie avant chaque nouveau test de la boucle. Ici, avant de traiter `value`, `total` contient la somme des entiers de 1 à `value - 1`. Initialement value vaut 1 et total vaut 0 : la somme sans terme vaut zéro. Ajouter value puis augmenter value de 1 conserve la propriété pour le test suivant.

Une **quantité de progression** permet de justifier l'arrêt. Tant que value ne dépasse pas n, le nombre d'entiers restant à traiter diminue à chaque tour. Pour n égal à 4, on traite quatre valeurs et on quitte avec value égal à 5. L'invariant permet alors de relier total à la somme complète recherchée.

Notre compteur compte seulement l'instruction qui ajoute value au total. Il ne compte pas l'incrément du compteur, celui de value ni les comparaisons. Une **durée mesurée** serait un temps observé sur une machine donnée ; ce n'est pas ce que mesure count. Préciser l'opération évite d'annoncer un nombre abstrait d'opérations sans convention.

Rendre les étapes visibles

Pseudo-code

```
// example.pseudo
// sum_with_count
n <- 4
value <- 1
total <- 0
count <- 0
WHILE value <= n DO
    total <- total + value
    count <- count + 1
    value <- value + 1
END WHILE
PRINT total, count
```

Python

```python
# sum_with_count.py
n = 4
value = 1
total = 0
count = 0
while value <= n:
    total = total + value
    count = count + 1
    value = value + 1
print(total, count)  # 10 4
```

MATLAB

```matlab
% sum_with_count.m
n = 4;
value = 1;
total = 0;
count = 0;
while value <= n
    total = total + value;
    count = count + 1;
    value = value + 1;
end
disp([total, count]); % [10 4]
```

`count` compte seulement les additions de la forme `total + value`. Nous ne comptons ni l'augmentation du compteur ni celle de value dans cette mesure. Sans cette convention, le mot "nombre d'opérations" serait ambigu.

Une quantité qui se rapproche de la sortie

Tant que le corps est exécuté, value est un entier compris entre 1 et n. Après chaque tour, il augmente de 1. Le nombre d'entiers restant à ajouter, `n - value + 1`, diminue donc de 1. Après l'ajout de n, value devient n + 1 et le test est faux. Pour n = 0, il est faux dès le début.

Une propriété conservée

Avant chaque tour, **total est la somme des entiers déjà parcourus**, de 1 à value − 1. Au départ, aucun entier n'a été parcouru et total vaut zéro. Le tour ajoute précisément value, puis prépare la valeur suivante : la propriété reste vraie. À la sortie, tous les entiers de 1 à n ont été parcourus ; total représente donc la somme recherchée.

Compter ce qui a été défini

Un ajout par entier parcouru

n	Additions total + value	Résultat
0	0	0
1	1	1
4	4	10
5	5	15

Ce comptage explique la croissance du travail sans mesurer un temps de machine. Un chronométrage dépendrait aussi de l'environnement, des affichages et du langage. Ici, nous relions directement la taille de l'entrée au nombre d'additions choisies.

Distinguer arrêt et résultat

▼ **Une boucle qui termine est-elle forcément correcte ?**

Non. Par exemple, partir de total = 1 au lieu de zéro ne changerait pas la progression de value ni l'arrêt, mais décalerait chaque résultat de un. La terminaison et la correction répondent à deux questions différentes.

▼ Si l'on supprime value ← value + 1, quel raisonnement casse ?

La quantité restante ne diminue plus. Pour n au moins égal à 1, le test reste vrai et le même entier est ajouté sans fin. Il manque à la fois la progression vers la sortie et le parcours de tous les termes attendus.

Garder le périmètre du module

Vous avez une première méthode pour justifier une boucle : domaine, progression, propriété conservée et sortie. L'étude systématique des notations de complexité et la comparaison des grandes familles d'algorithmes sont approfondies dans **DSA**. Ici, l'objectif est de pouvoir expliquer précisément un petit programme.

À retenir - fiche mémo

Invariant

Une propriété conservée avant chaque nouveau test de boucle.

Terminaison

Une progression explique pourquoi le parcours rejoint la sortie.

Coût

Annoncez exactement quelle opération est comptée.

Pseudo-code

```
// example.pseudo
// loop_invariant
// Before each test:
// total is the sum from 1 to value - 1.
// After the loop: value == n + 1 for n >= 0.
```

Point de vigilance. Un compteur d’additions ne donne pas directement une durée en secondes.

Consulter la cheatsheet : Vérifications et projets · **Télécharger la fiche du thème en PDF**

Version web : https://www.clearcode.fr/courses/programming/algorithmic/practice/termination-and-cost