Algorithmique	Chap 04	Correction et terminaison des algorithmes

Pourquoi prouver un algorithme ?

Écrire un algorithme qui "fonctionne sur quelques exemples"ne suffit pas.
En informatique, on doit pouvoir garantir que :

— le résultat produit est **toujours correct**, pour **toutes** les entrées valides;

— l'algorithme ne risque pas de **boucler indéfiniment**.

Or, un algorithme peut :

— sembler correct sur des tests simples, mais échouer dans des cas particuliers;

— donner le bon résultat, mais ne jamais s'arrêter;

— s'arrêter, mais produire un résultat incorrect.

L'objectif de ce chapitre est donc d'apprendre à vérifier qu'un algorithme est **totalement correct**.

4.1 Spécifier un algorithme

Définition 1 — Spécification d'un algorithme

Pour pouvoir dire si un algorithme est correct, il faut pouvoir le décrire. On utilise pour cela une **spécification** qui décrit précisément :

— les **entrées** (et les entrées valides);

— les **sorties** attendues;

— la **propriété** que la sortie doit vérifier en fonction des entrées.

Exemple 1 — Spécification d'une recherche dans une liste

Entrées : une liste `tab` de taille n et une valeur `x`.
Sortie : un booléen b.
Propriété attendue :

$b$vaut `True` si et seulement si `x` apparaît au moins une fois dans `tab`.

Exemple 2 — Spécification d'un maximum

Entrée : une liste `tab` de taille $n \geq 1$.

Sortie : un entier `m`.

Propriété attendue :

`m` appartient à `tab` et `m` est supérieur ou égal à tous les éléments de `tab`.

Exercice 1 — Écrire une spécification (entrées / sorties / propriété)

Écrire la spécification complète pour :

1. un algorithme qui compte le nombre d'occurrences de `x` dans `tab` ;
2. un algorithme qui teste si une chaîne `s` est un palindrome ;
3. un algorithme qui renvoie l'indice du premier `x` dans `tab` (ou `-1` si absent).

4.2 Correction

Définition 2 — Correction d'un algorithme

Un algorithme est **correct** si :

pour toute entrée valide, il produit une sortie valide qui respecte la spécification.

Rapide $\neq$ correct

La correction ne dépend pas du temps d'exécution. Un algorithme peut être rapide et faux, ou lent et correct.

Définition 3 — Correction partielle

Un algorithme est **partiellement correct** si :

lorsqu'il termine, son résultat est conforme à la spécification.

Cela ne garantit pas qu'il termine.

Définition 4 — Terminaison

Un algorithme **termine** si, pour toute entrée valide, il s'arrête après un nombre fini d'étapes.

Correction totale

Un algorithme est **totalement correct** s'il est :

— partiellement correct;

— et s'il termine.

Plan-type de justification

Pour rédiger une justification complète, on suit **toujours** ce plan :

1. **Spécification :** rappeler ce qui est demandé (entrées/sorties/propriété).
2. **Correction partielle :** expliquer pourquoi le résultat est correct **si** l'algorithme s'arrête. Souvent avec un **invariant** $P(i)$.
3. **Terminaison :** expliquer pourquoi l'algorithme s'arrête toujours. Souvent avec un **variant** (quantité monotone et bornée).
4. **Conclusion :** donc l'algorithme est (ou n'est pas) **totalement correct**.

4.3 Preuve de correction partielle

Définition 5 — Invariant de boucle

Soit une boucle indexée par i. Un **invariant** est une propriété $P(i)$ telle que :

— **Initialisation :** $P(i_0)$ est vraie avant la première itération (souvent $i_0 = 0$) ;

— **Hérédité :** pour tout i, si $P(i)$ est vraie au début de l'itération i, alors $P(i+1)$ est vraie au début de l'itération suivante ;

— **Conclusion :** à la fin de la boucle, $P(i)$ permet de prouver la propriété demandée.

C'est une preuve par récurrence.

Exemple 3 — Somme d'une liste

Algorithme :

```
s = 0
for i in range(n):
    s = s + tab[i]
```

Spécification (objectif).
À la fin, `s` doit être la somme de tous les éléments de `tab`.
Propriété $P(i)$.
Au début de l'itération d'indice i,

$$P(i): \quad \texttt{s} = \sum_{k=0}^{i-1} \texttt{tab}[k].$$

Initialisation.
Avant la première itération, $i = 0$ et s=0. Or $\sum_{k=0}^{-1}(\cdots) = 0$ (somme vide), donc $P(0)$ est vraie.
Hérédité.
Supposons $P(i)$ vraie au début de l'itération i : $\texttt{s} = \sum_{k=0}^{i-1} \texttt{tab}[k]$. Après l'instruction `s = s + tab[i]`, on obtient

$$\texttt{s} = \sum_{k=0}^{i-1} \texttt{tab}[k] + \texttt{tab}[i] = \sum_{k=0}^{i} \texttt{tab}[k],$$

ce qui est exactement $P(i+1)$.
Conclusion (correction partielle).
Quand la boucle finit, $i = n$, donc $P(n)$ donne $\texttt{s} = \sum_{k=0}^{n-1} \texttt{tab}[k]$: la somme totale est correcte.
Donc l'algorithme est **totalement correct**.

Exercice 2 — Pareil sur le produit

On considère :

```python
p = 1
for i in range(n):
    p = p * tab[i]
```

1. Écrire une propriété $P(i)$ adaptée.

2. **Initialisation :** montrer que $P(0)$ est vraie.

3. **Hérédité :** supposer $P(i)$ vraie et montrer $P(i+1)$.

4. Conclure quant à la correction partielle

Exercice 3 — Maximum : rédiger la preuve par invariant

On considère l'algorithme ci-dessous dont l'objectif est de trouver le maximum d'une liste tab.

```
m = tab[0]
for i in range(1, n):
    if tab[i] > m:
        m = tab[i]
```

1. Proposer une propriété $P(i)$ en tant qu'invariant pour justifier la correction partielle du programme.

2. **Initialisation :** montrer que $P(1)$ est vraie.

3. **Hérédité :** supposer $P(i)$ vraie et montrer $P(i+1)$.

4. Conclure quant à la correction partielle.

Exercice 4 — Recherche dans une liste

On considère le programme suivant dont l'objectif est de trouver si oui ou non l'élément x est dans la liste tab.

```
i = 0
trouve = False
while i < n and not trouve:
    if tab[i] == x:
        trouve = True
    else:
        i = i + 1
```

1. Proposer une propriété $P(i)$ qui peut servir d'invariant pour vérifier la correction partielle du programme.

2. **Initialisation :** montrer que $P(1)$ est vraie.

3. **Hérédité :** supposer $P(i)$ vraie et montrer $P(i+1)$.

4. Conclure quant à la correction partielle.

4.4 Terminaison

Terminaison d'une boucle `for`

Par essence, une boucle `for` termine toujours, car :

— le nombre d'itérations est **fixé à l'avance** ;

— l'indice de boucle parcourt un **ensemble fini de valeurs** ;

— aucune instruction à l'intérieur de la boucle ne peut augmenter ce nombre d'itérations.

Ainsi, la terminaison d'une boucle `for` est immédiate et ne nécessite pas de raisonnement supplémentaire.

Définition 6 — Variant de boucle `while`

Pour prouver la terminaison d'une boucle `while`, on exhibe souvent un **variant** : une quantité V telle que :

— V est un entier naturel (≥ 0) ;

— à chaque itération, V **décroît strictement** (ou croît strictement) ;

— V est borné (par exemple, il ne peut pas devenir négatif).

Donc on ne peut pas effectuer une infinité d'itérations.

Attention : "borné"ne suffit pas

Une quantité peut rester bornée et pourtant la boucle être infinie (oscillation). Ce qui garantit la terminaison, c'est la **monotonie stricte** vers une borne.

Exemple 4 — Variant simple : compteur

```
i = 0
while i < n:
    i = i + 1
```

Variant possible : $V = n - i$.
À chaque itération, V décroît de 1 et reste ≥ 0, donc la boucle termine.

Exemple 5 — Variant : division par 2

```
x = N
while x > 0:
    x = x // 2
```

Variant possible : $V = x$.
x décroît et reste ≥ 0. Dès que $x = 0$, la boucle s'arrête, donc terminaison.

Exercice 5 — Termine ou non ? Justifier proprement

Pour chaque code, dire s'il termine **pour toute entrée valide** et justifier avec un variant (ou un contre-exemple).

1.
```python
i = n
while i > 0:
    i = i - 2
```

2.
```python
i = 1
while i < n:
    i = 2 * i
```

3.
```python
i = 0
while i < n:
    if i % 2 == 0:
        i = i + 1
    else:
        i = i - 1
```

4.5 Prouver la correction totale d'un algorithme

Exercice 6 — Compléter une preuve

On considère :

```
c = 0
for i in range(n):
    if tab[i] % 2 == 0:
        c = c + 1
```

1. Donner la spécification.

2. Proposer une propriété $P(i)$, un invariant de boucle.

3. Initialisation : montrer $P(0)$.

4. Hérédité : supposer $P(i)$ et montrer $P(i+1)$.

5. Etablir la termination

6. Conclure