

Chapitre ITC2

Preuves d'algorithmes

Comment obtenir de la mayonnaise

- ❶ Prendre un œuf entier dans le frigo et le mélanger à une cuillère de moutarde. Ajouter une cuillère d'huile puis fouetter jusqu'à ce que la mayonnaise prenne. Continuer de fouetter en ajoutant de l'huile petit à petit.
- ❷ Faire chauffer un peu d'huile dans une poêle. Battre les œufs dans un saladier, puis verser dans la poêle et laisser cuire à feu doux pendant 7min.
- ❸ Prendre un œuf, de la moutarde et de l'huile à température ambiante. Séparer le jaune du blanc d'œuf. Placer le blanc dans un verre, le faire tourner 3 fois dans le sens trigonométrique. Mélanger la moutarde avec le jaune, puis faire 2 fois le tour de la table. Ajouter une cuillère d'huile puis fouetter jusqu'à ce que la mayonnaise prenne. Faire une pirouette. Continuer de fouetter en ajoutant de l'huile petit à petit ; si au bout de 5 minutes la mayonnaise n'a pas pris, arrêter..

Recette 1

Prendre un œuf entier dans le frigo et le mélanger à une cuillère de moutarde. Ajouter une cuillère d'huile puis fouetter jusqu'à ce que la mayonnaise prenne. Continuer de fouetter en ajoutant de l'huile petit à petit.

Cette recette...

- ne se termine jamais

Recette 2

Faire chauffer un peu d'huile dans une poêle. Battre les œufs dans un saladier, puis verser dans la poêle et laisser cuire à feu doux pendant 7min.

Cette recette...

- se termine
- ne donne pas le résultat attendu

Recette 3

Prendre un œuf, de la moutarde et de l'huile à température ambiante. Séparer le jaune du blanc d'œuf. Placer le blanc dans un verre, le faire tourner 3 fois dans le sens trigonométrique. Mélanger la moutarde avec le jaune, puis faire 2 fois le tour de la table. Ajouter une cuillère d'huile puis fouetter jusqu'à ce que la mayonnaise prenne. Faire une pirouette. Continuer de fouetter en ajoutant de l'huile petit à petit ; si au bout de 5 minutes la mayonnaise n'a pas pris, arrêter..

Cette recette...

- se termine
- donne normalement le résultat attendu
- est inutilement compliquée

Outils d'étude des algorithmes

Un algorithme doit :

- ❶ **se terminer**
- ❷ **faire ce qu'on veut qu'il fasse**
- ❸ **s'exécuter en un temps raisonnable**

Outils d'étude des algorithmes

Un algorithme doit :

- ❶ **se terminer**
→ étude de la **terminaison** à l'aide d'un **variant de boucle**
- ❷ **faire ce qu'on veut qu'il fasse**
- ❸ **s'exécuter en un temps raisonnable**

Outils d'étude des algorithmes

Un algorithme doit :

- ❶ **se terminer**
→ étude de la **terminaison** à l'aide d'un **variant de boucle**
- ❷ **faire ce qu'on veut qu'il fasse**
→ étude de la **correction** à l'aide de la logique de Hoare
(pour nous simplement par un **invariant de boucle**)
- ❸ **s'exécuter en un temps raisonnable**

Outils d'étude des algorithmes

Un algorithme doit :

- ❶ **se terminer**
→ étude de la **terminaison** à l'aide d'un **variant de boucle**
- ❷ **faire ce qu'on veut qu'il fasse**
→ étude de la **correction** à l'aide de la logique de Hoare
(pour nous simplement par un **invariant de boucle**)
- ❸ **s'exécuter en un temps raisonnable**
→ étude de la **complexité** de l'algorithme

Outils d'étude des algorithmes

Un algorithme doit :

- ❶ **se terminer**
→ étude de la **terminaison** à l'aide d'un **variant de boucle**
- ❷ **faire ce qu'on veut qu'il fasse**
→ étude de la **correction** à l'aide de la logique de Hoare
(pour nous simplement par un **invariant de boucle**)
- ❸ **s'exécuter en un temps raisonnable**
→ étude de la **complexité** de l'algorithme

Il existe 2 types de correction :

- si on prouve que l'algorithme fait ce qu'il faut lorsqu'il s'arrête, on a prouvé la **correction partielle**
- si en plus on prouve qu'il se termine, on a prouvé la **correction totale**

Section 1

Les variants de boucles

Une boucle **pour** se termine toujours

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : $somme$

$somme \leftarrow 0$

pour i allant de 1 à n faire

$somme \leftarrow somme + i$

fin pour

Une boucle **tant que** peut se terminer ou non

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$somme \leftarrow 0$

$i \leftarrow 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$somme \leftarrow somme + i$

fin tant que

Une boucle **tant que** peut se terminer ou non

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$somme \leftarrow 0$

$i \leftarrow 0$

tant que $i < n$ **faire**

!!!!

$somme \leftarrow somme + i$

fin tant que

Définition

Pour prouver qu'une boucle se termine, on introduit un **variant de boucle** ou **fonction de terminaison**. Il s'agit d'une fonction dépendant des variables intervenant dans la boucle, et ayant les propriétés suivantes :

- elle prend des **valeurs entières**
- elle est **strictement décroissante** à chaque exécution de la boucle
- si cette fonction devient négative ou nulle, la boucle se termine

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$somme \leftarrow 0$

$i \leftarrow 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$somme \leftarrow somme + i$

fin tant que

Terminaison de la boucle

Soit $T = n - i$ la fonction de terminaison :

- Elle a des valeurs entières.
- À chaque étape de la boucle, n ne change pas et i est incrémenté de 1, donc T diminue de 1.
- Lorsque $T \leq 0$ alors la boucle se termine.

Somme des termes d'une liste (pop enlève le dernier terme de la liste et le renvoie)

```
1 def somme_liste(liste):  
2     somme=0  
3     while len(liste)>0:  
4         somme+=pop(liste)  
5     return somme
```

Somme des termes d'une liste (pop enlève le dernier terme de la liste et le renvoie)

```
1 def somme_liste(liste):  
2     somme=0  
3     while len(liste)>0:  
4         somme+=pop(liste)  
5     return somme
```

Terminaison de la boucle

Posons $T = \text{len}(L)$.

- Cette fonction est à valeurs entières.
- À chaque exécution de la boucle, l'instruction *pop* enlève un terme à la liste, donc T décroît strictement.
- Lorsque T devient négatif ou nul, la boucle se termine.

Algorithme : Tri d'un tableau de 0 et de 1

Entrée : une liste L de 0 et de 1

Sortie : L triée

$i, j \leftarrow 0, \text{longueur de } L - 1$

tant que $i < j$ **faire**

si $L[i] = 0$ **faire**

$i \leftarrow i + 1$

sinon

 échanger $L[i]$ et $L[j]$

$j \leftarrow j - 1$

fin si

fin tant que

Algorithme : Tri d'un tableau de 0 et de 1

Entrée : une liste L de 0 et de 1

Sortie : L triée

$i, j \leftarrow 0, \text{longueur de } L - 1$

tant que $i < j$ **faire**

si $L[i] = 0$ **faire**

$i \leftarrow i + 1$

sinon

 échanger $L[i]$ et $L[j]$

$j \leftarrow j - 1$

fin si

fin tant que

Terminaison de la boucle

Posons $T = j - i$ comme fonction de terminaison.

- Cette fonction est à valeurs entières positives. À chaque exécution de la boucle, si le test est vrai, i augmente de 1 donc T diminue de 1 ; sinon j diminue de 1 donc T diminue de 1. Dans tous les cas T décroît de 1.
- Lorsque T devient négatif ou nul, la boucle se termine.

Recherche du zéro d'une fonction croissante par dichotomie

```
1 def zero(f,a,b,precision):  
2     while (b-a)>precision:  
3         m=(a+b)/2  
4         if f(m)<0:  
5             a=m  
6         else:  
7             b=m  
8     return (a,b)
```

Terminaison de la boucle

$T = b - a$ est décroissante mais n'est pas à valeurs entières.

Posons $T = E\left(\frac{b-a}{precision}\right)$.

- Elle est à valeurs entières.
- À chaque exécution de la boucle, on constate que $b - a$ est divisée par 2. Or si un réel supérieur à 1 est divisé par 2, sa partie entière décroît au moins de 1. Ici tant qu'on est dans la boucle, on a forcément $b - a > precision$ donc $T \geq 1$ donc T décroît strictement à chaque étape.
- Lorsque T s'annule, la boucle se termine.

Exemple de boucle double

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Calcul de $\sum_{i=1}^N \sum_{j=1}^M \frac{1}{(i+j)^2}$

Entrée : deux entiers N et M

Sortie : *somme*

```
somme, i  $\leftarrow$  0, 1
tant que  $i \leq N$  faire
    j  $\leftarrow$  1
    tant que  $j \leq M$  faire
        somme  $\leftarrow$  somme +  $\frac{1}{(i+j)^2}$ 
        j  $\leftarrow$  j + 1
    fin tant que
    i  $\leftarrow$  i + 1
fin tant que
```

Exemple de boucle double

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Calcul de $\sum_{i=1}^N \sum_{j=1}^M \frac{1}{(i+j)^2}$

Entrée : deux entiers N et M

Sortie : somme

```
somme, i ← 0, 1
tant que i ≤ N faire
    j ← 1
    tant que j ≤ M faire
        somme ← somme +  $\frac{1}{(i+j)^2}$ 
        j ← j + 1
    fin tant que
    i ← i + 1
fin tant que
```

Terminaison de la boucle

Si on veut une seule fonction de terminaison pour les deux boucles, on peut vérifier que $T = (N - i) * M - j$ fonctionne.

Exemple de boucle double

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Calcul de $\sum_{i=1}^N \sum_{j=1}^M \frac{1}{(i+j)^2}$

Entrée : deux entiers N et M

Sortie : somme

```
somme, i  $\leftarrow$  0, 1
tant que  $i \leq N$  faire
    j  $\leftarrow$  1
    tant que  $j \leq M$  faire
        somme  $\leftarrow$  somme +  $\frac{1}{(i+j)^2}$ 
        j  $\leftarrow$  j + 1
    fin tant que
    i  $\leftarrow$  i + 1
fin tant que
```

Terminaison de la boucle

Sinon $T_1 = N - i$ pour la boucle extérieure et $T_2 = M - j$ pour la boucle intérieure conviennent bien.

Peut-on prouver toujours si un programme se termine ?

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Suite de Syracuse

Entrée : un entier a

Sortie : le plus petit entier n tel que $u = 1$

$u \leftarrow a$

tant que $u > 1$ **faire**

si u est pair **faire**

$u \leftarrow u/2$

sinon

$u \leftarrow 3u + 1$

fin si

fin tant que

À l'heure actuelle, on ne sait pas prouver que cet algorithme se termine toujours quelle que soit la valeur de a

Section 2

Les invariants de boucles

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : $somme$

$i, somme \leftarrow 0, 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$somme \leftarrow somme + i$

fin tant que

On sait montrer que cet algorithme se termine

Mais renvoie-t-il le bon résultat si les arguments sont corrects ?

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$i, \text{somme} \leftarrow 0, 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$\text{somme} \leftarrow \text{somme} + i$

fin tant que

On sait montrer que cet algorithme se termine

Mais renvoie-t-il le bon résultat si les arguments sont corrects ?

Précondition, postcondition

- on appelle **précondition** une proposition qui est supposée toujours vraie avant le début de la boucle
- on appelle **postcondition** une proposition qui est toujours vraie après la fin de la boucle

Définition

On considère une boucle **tant que** qui se répète tant qu'une condition C est vraie. On appelle **invariant de boucle** une proposition logique qui :

- est vraie avant d'entrer dans la boucle
- est vraie à la fin de chaque itération de la boucle à condition que C soit vraie au départ

La démonstration de la correction à l'aide d'un invariant de boucle ressemble à un raisonnement par récurrence.

Algorithme : Division euclidienne de a par b

Entrée : deux entiers a et b strictement positifs

Sortie : le quotient q et le reste r , tels que $a = q.b + r$ et $r < b$

$q, r \leftarrow 0, a$

tant que $r \geq b$ **faire**

$r \leftarrow r - b$

$q \leftarrow q + 1$

fin tant que

Preuve de la correction avec $I : \{a = q.b + r \text{ et } r \geq 0\}$

Exemple simple

Algorithme : Division euclidienne de a par b

Entrée : deux entiers a et b strictement positifs

Sortie : le quotient q et le reste r , tels que $a = q.b + r$ et $r < b$

$q, r \leftarrow 0, a$

tant que $r \geq b$ **faire**

$r \leftarrow r - b$

$q \leftarrow q + 1$

fin tant que

Preuve de la correction avec $I : \{a = q.b + r \text{ et } r \geq 0\}$

Initialisation : au début $q = 0$ et $r = a$ donc $r \geq 0$ et $a = q.b + r$.

Ensuite, notons q_i et r_i les valeurs au début d'une itération et q_f et r_f à la fin. Supposons que I soit vrai au début :

$a = q_i.b + r_i$ et $r_i \geq 0$. De plus, par la condition de la boucle, $r_i \geq b$. À la fin on a $r_f = r_i - b \geq 0$ et $q_f = q_i + 1$ donc $q_f.b + r_f = (q_i + 1).b + r_i - b = q_i.b + r_i = a$ donc l'invariant reste vrai.

Algorithme : Division euclidienne de a par b

Entrée : deux entiers a et b strictement positifs

Sortie : le quotient q et le reste r , tels que $a = q.b + r$ et $r < b$

```
 $q, r \leftarrow 0, a$   
tant que  $r \geq b$  faire  
     $r \leftarrow r - b$   
     $q \leftarrow q + 1$   
fin tant que
```

Preuve de la correction avec $I : \{a = q.b + r \text{ et } r \geq 0\}$

On a montré que $I : \{a = q.b + r \text{ et } r \geq 0\}$ est un invariant de la boucle, donc à la fin on a $a = q.b + r$ et de plus on est sorti de la boucle donc $r < b$: CQFD.

C'est la correction partielle de cet algorithme; si on prouve aussi qu'il se termine, on aura la correction totale.

Exemple moins simple

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$i, \text{somme} \leftarrow 0, 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$\text{somme} \leftarrow \text{somme} + i$

fin tant que

Montrons que cet algorithme calcule bien $\sum_{k=1}^n k$

Proposons comme invariant : $I : \text{somme} = \sum_{k=0}^i k$.

- Au départ, $i = 0$ et $\text{somme} = 0$ donc I est vraie
- Notons i_i et somme_i les valeurs au début d'une itération et i_f et somme_f à la fin. Si I est vraie au début d'une itération de la boucle, alors, à la fin, $i_f = i_i + 1$ et $\text{somme} = \sum_{k=1}^{i_f-1} k + i_f = \sum_{k=0}^{i_f} k$ donc I est encore vraie.

Exemple moins simple

Preuves
d'algorithmes

Les variants
de boucles

Les invariants
de boucles

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$i, \text{somme} \leftarrow 0, 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$\text{somme} \leftarrow \text{somme} + i$

fin tant que

Montrons que cet algorithme calcule bien $\sum_{k=1}^n k$

$I : \text{somme} = \sum_{k=0}^i$ est un invariant de cette boucle. À la fin de la boucle on sait donc que :

- $\text{somme} = \sum_{k=0}^i k$
- $i \geq n$

mais cela ne prouve pas que $\text{somme} = \sum_{k=0}^n k$

Exemple moins simple

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$i, \textit{somme} \leftarrow 0, 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$\textit{somme} \leftarrow \textit{somme} + i$

fin tant que

Montrons que cet algorithme calcule bien $\sum_{k=1}^n k$

Modifions un peu l'invariant : $I : \{\textit{somme} = \sum_{k=0}^i k \text{ et } i \leq n\}$.

On a déjà montré que si la première partie est vraie au début d'une itération, elle est vraie à la fin. Qu'en est-il de la partie $i \leq n$? Eh bien, si on est dans la boucle, c'est que la condition initiale est vraie : $i_i < n$ soit (puisque i et n sont entiers) $i_i \leq n - 1$. Au cours de la boucle, $i_f = i_i + 1$, donc à la fin $i_f \leq n$.

I est donc encore un invariant de boucle.

Exemple moins simple

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$i, \text{somme} \leftarrow 0, 0$

tant que $i < n$ **faire**

$i \leftarrow i + 1$

$\text{somme} \leftarrow \text{somme} + i$

fin tant que

Montrons que cet algorithme calcule bien $\sum_{k=1}^n k$

$I : \{ \text{somme} = \sum_{k=1}^i k \text{ et } i \leq n \}$ est un invariant de cette boucle. À la fin de la boucle on sait donc que :

- $\text{somme} = \sum_{k=1}^i k$
- $i \leq n$
- $i \geq n$

On en déduit facilement que $i = n$ et $\text{somme} = \sum_{k=1}^n k$.

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$i, \text{somme} \leftarrow 0, 0$

tant que $i < n$ **faire**

$\text{somme} \leftarrow \text{somme} + i$

$i \leftarrow i + 1$

fin tant que

Cet algorithme ne fonctionne pas

Il va calculer $1 + 2 + \dots + (n - 1)$. L'invariant de boucle précédent n'est plus bon.

Algorithme : Somme des entiers de 1 à n

Entrée : un entier n

Sortie : *somme*

$i, \text{somme} \leftarrow 0, 0$

tant que $i \leq n$ **faire**

$i \leftarrow i + 1$

$\text{somme} \leftarrow \text{somme} + i$

fin tant que

Cet algorithme ne fonctionne pas

L'invariant de boucle précédent est faux : la partie $i \leq n$ n'est plus vérifiée.

Algorithme : Recherche du maximum d'une liste

Entrée : une liste L

Sortie : max

```
 $max \leftarrow L[0]$   
pour  $i$  variant de 1 à  $\text{longueur}(L)-1$  faire  
    si  $L[i] > max$  faire  
         $max = L[i]$   
    fin si  
fin pour
```

Algorithme : Recherche du maximum d'une liste

Entrée : une liste L

Sortie : max

```
 $max \leftarrow L[0]$   
pour  $i$  variant de 1 à  $\text{longueur}(L)-1$  faire  
    si  $L[i] > max$  faire  
         $max = L[i]$   
    fin si  
fin pour
```

Preuve de la correction de cet algorithme

Un bon invariant de boucle est $V : \{max \text{ est le maximum de } \{L[0], \dots, L[i]\}\}$.

Algorithme de Horner : évaluation d'un polynome

```
1 def horner(P,x):
2     """ calcule P(x) par l'algorithme de Horner
3     entrees: x: valeur de la variable
4             P: coefficients du polynome
5             sous la forme [a0,a1,...,an] """
6     k=len(P)-1
7     valeur=P[k]
8     while k>0:
9         k=k-1
10        valeur=valeur*x+P[k]
11    return valeur
```

Algorithme de Horner : évaluation d'un polynome

```
1 def horner(P,x):
2     """ calcule P(x) par l'algorithme de Horner
3     entrees: x: valeur de la variable
4             P: coefficients du polynome
5             sous la forme [a0,a1,...,an] """
6     k=len(P)-1
7     valeur=P[k]
8     while k>0:
9         k=k-1
10        valeur=valeur*x+P[k]
11    return valeur
```

Preuve de la correction de cet algorithme

$V : \{valeur = a_k + a_{k+1}x + a_{k+2}x^2 + \dots + a_n x^{n-k} \text{ et } k \geq 0\}.$