1.6 Validation d'un algorithme

Lorsque nous construisons un algorithme, nous devons répondre aux trois questions suivantes :

a) l'algorithme donne t-il toujours un résultat ? c'est la *terminaison*

b) l'algorithme donne t-il le bon résultat ? c'est la *correction*

c) l'algorithme donne t-il le bon résultat en temps raisonnable ? c'est la *complexité*

Les deux premières questions répondent à la notion de *validation* de l'algorithme.

Remarque 1.6.1 *Pourquoi chercher à valider un algorithme au lieu de simplement le tester ?*

> *" Tester un algorithme montre la présence d'erreurs et non leur absence"*
>
> *[E.W.Dijkstra 1969]*

En effet :

le test :

1. valide une instance ;
2. permet une vérification rapide qui est utile au cours du développement d'un programme ;
3. fait apparaître les limites de la solution.

la preuve :

1. donne une garantie incontestable sur le principe de l'algorithme ;
2. nécessite l'utilisation d'outils formels
3. ne garantit pas contre les erreurs de programmation

1.6.1 Types d'algorithmes

Un algorithme est soit *itératif*, soit *récursif*.

L'algorithme itératif est constitué d'actions simples et de structures de contrôle (les boucles et les choix).
Son analyse consiste à estimer le nombre de répétitions des itérations d'actions (voir les exemples précédents).

Les algorithmes récursifs consistent à ramener la solution d'un problème de taille n à celle d'un nombre réduit de sous-problèmes de même type que le problème initial mais chacun de taille m, n (dans une stratégie "*Diviser pour régner*".
Son analyse de complexité consiste à estimer le temps d'exécution de tous les sous-problèmes et leur composition pour résoudre entièrement le problème. La combinaison des analyses de chaque sous-problème fournit une relation de récurrence qui sera transformée en une equation qui relie le nombre d'opérations total à la taille du problème initial.

Exemple 1.6.2

$$n! = \begin{cases} 1 & si\ n = 0, 1 \\ n * (n-1)! & si\ n > 0 \end{cases}$$

Si on note $T(n)$ le temps d'exécution de cette fonction alors on peut écrire :

$$T(n) = \begin{cases} 0 & si\ n = 0, 1 \\ 1 + T(n-1) & si\ n > 0 \end{cases}$$

La résolution de cette équation de récurrence donne $T(n) = \Theta(n)$

1.6.2 La validation : terminaison et correction

Rappelons qu'un algorithme est une suite finie d'actions qui opèrent sur un ensemble fini de variables et les modifient de leur état initial vers un état final. Un sous-ensemble de ces variables de l'état final constitue le résultat de l'algorithme.

Un algorithme résout un problème P si pour tout ensemble de valeurs des variables d'entrée, la suite des opérations exécutées est finie, c'est ce qu'on appelle la *condition de terminaison*, et lorsque l'algorithme termine le sous-ensemble des variables en sortie contient le résultat recherché, c'est *la correction* de l'algorithme.

Ainsi, avant d'évaluer la quantité de ressources nécessaire à l'exécution de l'algorithme, il est nécessaire de prouver qu'il est *valide*.

Prouver qu'un algorithme est valide consiste à :

1. démontrer qu'il *termine* quelles que soit les données d'entrée ;
2. démontrer qu'il est *correcte*, c'est-à-dire qu'il calcule bien ce qui est demandé.

1.6.3 La terminaison d'un algorithme

Pour prouver qu'un traitement termine, il suffit de trouver une mesure (fonction) qui prend des valeurs dans $\mathbb{N}$ et qui décroit strictement à chaque appel itération ou appel récursif.
La terminaison est garantie puisqu'il est impossible de construire une suite d'entiers infinie et strictement décroissante.

Quelques regles pratiques

1. La terminaison d'un algorithme sans boucle ni récursion, ayant un nombre fini de lignes est évidente.
2. La terminaison d'une boucle "**Pour**" dont le compteur ne varie pas dans le corps de la boucle est aussi évidente. (Donc il ne faut jamais modifier la variable du compteur de la boucle dans le corps de cette boucle).

3. La terminaison d'une boucle " Tant que " ou d'une fonction récursive nécessite une preuve.

Exemple 1.6.3 *La conjecture de Syracuse*

```
Tant que  n>1
faire
   si n pair  Alors   n<-- n/2
              Sinon   n<-- 3*n+1
    fsi

 fait
 retourner (n)
```

Il n'existe pas encore de preuve que Syracuse(n) s'arrête $\forall n \in \mathbb{N}$

La démarche

On prouve la terminaison de beaucoup de boucle " Tant que" en mettant en évidence l'existence d'une suite finie d'entiers naturels qui décroit à chaque itération de la boucle.

Exemple 1.6.4

```
i <--- 0;
Tant que i< N
faire
   i <--- i+1
fait
```

Si $U_i = N - i > 0$*, alors* $N, N-1, N-2, \ldots, N-i, \ldots, 0$ *est une suite décroissante. Ce cas décrit par une boucle "Pour" est évident.*

Exemple 1.6.5 *Calcul de la somme* $S = a + b$ *par itération* $(a, b > 0)$

```
S  <--- a;
Tant que b>0
faire
   S <--- S+1;
```

```
   b <--- b-1;
fait
retourner S;
```

La suite décroissante est : $b, b-1, b-2, \ldots, 1, 0$

Exemple 1.6.6 *Le calcul de la division euclidienne* $div(a,b) = (q,r)$

```
q <--- 0;
Tant que a>= b
faire
  a <--- a-b;
  q <--- q+1
fait;

retourner (a, q)
```

La suite décroissante est : $a, a-b, a-2*b, a-3*b, \ldots a-q*b-r=0$, *ce qui donne :* $a = b*q+r$, *l'algorithme s'arrête avec* $q > 0$ *et* $r >= 0$.

EXEMPLE

Écrire un algorithme qui calcule la somme des elements d'un tableau de n entiers ($n > 0$).

1. Écrire un algorithme itératif et prouver sa validité
2. Déterminer sa complexité
3. Écrire un algorithme récursif pour le même problème et prouvez-le ;
4. déterminer sa complexite