La correction d'algorithmes

Un algorithme séquentiel est une suite d'instructions simples et/ou structurées. Il exprime un ensemble fini d'opérations liées par une structure de contrôle (boucle, choix, …) qui opèrent sur un ensemble fini de variables.

Entrée : *Algorithme* *Résultat :*

Etat initial des variables ⟶ état final d'un sous ensemble

Suite d'opérations finies des variables de l'état initial.

Définition : la validité d'un algorithme

> Un algorithme qui résout un problème P est valide si pour tout ensemble de valeurs de variables d'entrée, la suite des opérations exécutées est finie (condition de terminaison) et lorsque le programme termine, le sous-ensemble des variables de sortie contient le résultat recherché.

Donc prouver un algorithme consiste à :

a. démontrer qu'il termine
b. démontrer qu'il calcule bien ce qui est demandé (le résultat est valide)

Les questions qu'on se pose à propos d'un algorithme :

1. Prouver sa validité
   - Terminaison
   - Correction du résultat
2. Calculer sa complexité

1- Preuve par induction

Le principe d'une démonstration par récurrence (ou induction) d'une propriété P définie sur un ensemble D de données consiste à montrer que cette propriété est bien établie pour certaines valeurs particulières de D et que chaque fois que la propriété est vraie pour une valeur particulière dans D elle est vraie aussi pour la valeur suivante. Alors on peut ainsi affirmer que la propriété P est vraie pour toutes les valeurs de D.

Ainsi, dans une preuve par induction en fonction de n, on commence par vérifier le «**cas de base**» c'est-à-dire vérifier si la proposition est vraie pour certaines

valeurs *initiales* bien choisies de n (par exemple $n = 0, 1, 2$). Si le cas de base est vérifié, on pose « **l'hypothèse de récurrence** » autrement dit que la propriété est vraie à un ordre particulier n et on essaye de démontrer qu'elle est vraie à l'ordre $n+1$. Si on démontre cette implication, on déduit alors que la propriété est vraie quel que soit n (puisqu'elle est vraie pour n=0 vérifié explicitement, donc elle est vraie pour n=1 par implication; puisqu'elle est vraie pour n=1 alors elle est vraie pour n=2 par implication, puisqu'elle est vraie pour n=2 alors elle est vraie pour n=3 par implication et ainsi donc pour tout n).

En informatique, et plus particulièrement en algorithmique dès lors qu'il s'agit de raisonner formellement sur certaines propriétés d'algorithmes, les preuves par induction sont très souvent utilisées.

Preuve par induction :

Cas de base la proposition est vraie pour $n = 0$ ou n=1 ou n=2,….
Cas inductif si la proposition est vraie pour $n = k$, où $k \geq 0$, alors elle l'est pour $n=k+1$.

Exemple :

Montrer que $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

n=0 $\sum_{i=0}^{0} 2^0 = 2^1 - 1$

$2^0 = 2^1 - 1$ $2^0 = 1$ *et* $2^1 - 1 = 1$ *la proposition est vraie pour n=0*

$n=k$ $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$ *supposée vraie*

Montrons que pour n=k+1 la proposition est vraie c'est-à-dire $\sum_{i=0}^{k+1} 2^i = 2^{k+1} - 1$

$$\sum_{i=0}^{k+1} 2^i = \sum_{i=0}^{k} 2^i + 2^{k+1}$$
$$= 2^{k+1} - 1 + 2^{k+1}$$
$$= 2x2^{k+1} - 1$$
$$= 2^{k+2} - 1$$

ce qu'il fallait démontrer donc l'égalité est vraie.

2- Récurrence

Le terme « récurrence » recouvre en mathématiques l'idée de réutiliser l'objet qu'on est entrain de définir dans sa propre définition. En informatique, une fonction *f* est définie de façon *récursive*, lorsqu'il existe dans le corps de *f* au moins un appel (direct ou indirect) à la fonction *f*.

Exemple : on peut définir mathématiquement le calcul de la factorielle n (n !) de deux façons. La première :

$$n! = \begin{cases} 1 * 2 * 3 * \dots * n-1 * n & si\ n > 1 \\ 1 & si\ n = 0\ ou\ n = 1 \end{cases}$$

Elle se traduit par l'algorithme itératif suivant :

Algorithme1 :
Fonction fact(entier n) :entier ;
Debut
f, i :entier ;
f←1 ;
pour i←1 à n faire
*f←f*i ;*
finpour ;
retourner(f) ;
Fin ;

On peut aussi la définir par récurrence :

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

Cette définition se traduit par une fonction informatique récursive :

Algorithme2 :
Fonction fact(entier n) :entier ;
Debut
si n=0 ou n=1 retourner 1;
*sinon retourner n*fact(n-1) ;*
finsi ;
Fin ;

Prouver un programme récursif, c'est :

- prouver que le résultat produit dans le cas d'arrêt est correct
- prouver que les résultats produits dans les cas d'appels récursifs, *sous hypothèse* que ces appels récursifs soient corrects et produisent bien ce qu'il est attendu d'eux, sont corrects
- prouver la convergence, c'est-à-dire que toute séquence d'appels récursifs conduit toujours à une situation d'arrêt.

3- Validation d'un algorithme

Pour prévoir l'effet de k itérations d'une boucle donnée, il faut en général connaître l'effet de $k-1$ itérations.
En effet, la $k^{ème}$ itération s'exécute après les $k-1$ première itérations, et son effet dépend donc des valeurs des variables après $k-1$ itérations.

Pour prouver qu'une certaine proposition est vraie après un nombre arbitraire d'itérations d'une certaine boucle, on montrera d'abord que la proposition est vraie avant la première itération ; puis on montrera que si la proposition est vraie pour $k-1$ itérations, elle l'est aussi pour k itérations.

Exemple : *Ecrire un algorithme qui calcul la somme des éléments d'un tableau A de n entiers (n>0)*

1) Ecrire un algorithme itératif qui calcule la somme des éléments de A et prouver sa validité
2) Déterminer sa complexité
3) Ecrire un algorithme récursif pour le même problème et prouvez-le
4) Déterminer sa complexité

Solution :

1) L'algorithme itératif

```
fonction  somme (A : tableau ; n :entier) :entier ;
debut
  s, i : entier;
  S← 0 ;
  pour i←1 à n faire
          s←s+A[i] ;
  finpour;
  retourner s;
fin;
```

Pour prouver la validité de cet algorithme on démontre sa terminaison et la validité du résultat

a. *La terminaison :*

La preuve est triviale car : n>0 par hypothèse ; i=1 par initialisation et est incrémenté de 1 à chaque tour de boucle ; donc i atteindra la valeur d'arrêt n après n-1 itérations, aussi le corps de la boucle ne contient que l'addition et l'affectation qui s'exécutent chacune en un temps fini donc l'algorithme se termine.

b. *La validité :*

On la démontre par récurrence (sur la taille des données). On montre qu'à chaque itération le résultat est celui recherché et ceci en formalisant l'expression du résultat d'une itération.

« L'invariant de la boucle »

« A la fin de l'itération i, la variable S contient la somme des i premiers éléments du tableau A »

Plus formellement : $S_i = \sum_{k=1}^{i} A[k]$

Les invariants représentent un outil précieux pour expliquer le comportement d'une boucle (et donc pour prouver la correction de la boucle). Il est important de choisir un invariant suffisamment précis, pour pouvoir raisonner sur l'algorithme considéré.

Définition d'un invariant :
Un **invariant** est une propriété logique sur les valeurs des variables qui caractérise tout ou partie de l'état interne du programme et qui doit être « toujours » vraie. Dans le cas d'un programme récursif, cet invariant est la relation de récurrence ; dans le cas d'un programme itératif, cet invariant est l'invariant de boucle.

Un invariant peut servir à concevoir, expliquer, documenter, comprendre et tester un algorithme

Preuve par récurrence de la propriété :
On note S_i cette somme. On a alors :

$S_0 = 0$

Pour i=1 $S_1 = S_0 + A[1]$

$S_1 = A[1]$ donc la propriété est vraie pour i=1

On suppose que la propriété est vraie à l'itération i, c'est-à-dire :

$S_i = \sum_{k=1}^{i} A[k]$ vraie et montrons qu'elle est vraie à l'itération i+1

« A la fin de l'itération i+1 la variable S contient ce qu'elle contenait à l'étape i, plus A[i+1] »

$S_{i+1} = S_i + A[i+1]$

$= \sum_{k=1}^{i} A[k] + A[i+1]$

$S_{i+1} = \sum_{k=1}^{i+1} A[k]$ donc la propriété est vraie à i+1

L'algorithme se termine à la fin de l'itération *n* et on aura donc calculé :

$S_n = \sum_{k=1}^{n} A[k]$

L'algorithme est donc valide.

c. La complexité

On compte le nombre d'opérations dans la boucle : $\underbrace{\text{addition + affectation}}_{= 1 \text{ opération}}$

$T(n) = n \Rightarrow T(n) = O(n)$ et plus précisément $T(n) = \Theta(n)$

L'algorithme récursif

```
fonction  somRec (A : tableau ; n :entier) :entier ;
debut
   si n=0 retourner 0
   sinon  retourner (A[n] + somRec (A, n-1));
   finsi ;
fin;
```

a. La terminaison

Par récurrence sur n :

- Si n=0, cas évident, se termine sans appel récursif.
- On suppose qu'à l'étape n, l'algorithme termine donc somRec (A, n) termine et évaluons somRec (A, n+1)

somRec (A, n+1) appelle somRec (A, n) + A[n+1]

termine par hypothèse de récurrence

addition opération finie et termine

donc somRec (A, n+1) termine ∀ la valeur du paramètre n

b. La validité

On note S_n la valeur retournée par somRec(A, n). La fonction reproduit la relation de récurrence suivante :

$$\begin{cases} S_0=0 \\ S_n= S_{n-1} + A[n] \quad \text{si } n>0 \end{cases}$$

On montre comme dans le cas itératif $S_n= \sum_{j=1}^{n} A[j]$

Et donc que l'algorithme calcule bien la somme des éléments du tableau A.

c. La complexité

On compte le nombre d'addition :

$$T(n) = \begin{cases} 0 \quad \text{si } n=0 \text{ (on a zéro addition)} \\ T(n-1) + 1 \quad \text{si } n>0 \end{cases}$$

Qui a pour solution T(n) = n. Et on en déduit que $T(n)=\Theta(n)$

Méthode générale :

Pour résoudre une équation de récurrence, on remplace le terme T(k) par le membre droit de l'équation k fois pour obtenir une formule qui ne contient plus « T » à droite et on intègre à la fin les cas particuliers dans la formule.

Exemple : *Calcul de n !*

```
fonction fact (n :entier)
debut
  1. Si (n<=1)  alors retourner 1
  2. Sinon retourner(n*fact(n-1) ;
  finsi ;
fin ;
```

Soit T(n) le temps d'exécution de fact(n)

- La ligne 1 : s'exécute en O(1)=constante = d
- La ligne 2 : s'exécute en O(1) + T(n-1) = constante +T(n-1)
  = c+T(n-1)

Donc il existe 2 constantes c et d telles que :

$$T(n) = \begin{cases} d & si \quad n<=1 \\ c + T(n-1) & si \ n>1 \end{cases} \qquad (1)$$

Soit n>2, appliquons la formule (1)

Pour n-2 :

$$\left.\begin{array}{l} T(n)= c + T(n-1) \\ T(n-1)= c + T(n-2) \end{array}\right\} \Rightarrow \quad T(n)= 2*c + T(n-2)$$

$T(n-2)= c + T(n-3) \Rightarrow \quad T(n)= 3*c + T(n-3)$

Donc $T(i)= i*c + T(n-i) \quad n>i$

Pour $i=n-1 \Rightarrow \quad T(n)= (n-1)*c + T(1) \Rightarrow \quad T(n)= (n-1)*c + d = O(n)$

Donc $T(n)=O(n)$