Part II : Analyse de Programmes

Introduction

Correction : Mon code produit-il le résultat attendu ?

Terminaison : Mon code répond-il un jour ?

Complexité : A quelle vitesse mon programme répond-il ?

Solution 1 : Batteries de Tests. Limitation : On ne peut pas être exhaustif, il peut toujours se produire en situation réelle une configuration non testée.

Solution 2 : Analyse mathématique

Chapitre 1 : Correction

I - Introduction

```c
void swap(a,i,j) // échange les cases i et j de a

void mystery(int len, int* a) {
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < len; j++) {
            if (a[i] < a[j]) swap(a,i,j);
        }
    }
}
```

Ce programme est-il correct ?

- Correct : fait-il ce qu'on attend de lui ?
- Ici : qu'est ce qu'on attend de lui ?

Problème : Il faut préciser ce qu'on attend d'un programme, c'est sa **spécification**.

Entraînement : Écrivons la spécification d'un algorithme de tri.

- On demande que le tableau :
  - soit trié
  - soit une permutation du tableau initial

II - Vocabulaire

Pour préciser ce qu'un programme doit faire, on donne sa **spécification**. Elle est composée de :

- La **précondition** : ce sont les hypothèses que l'on fait sur les arguments.
- La **postcondition** : c'est ce que vérifie le résultat ou éventuellement les modifications effectuées en mémoire.

Un programme est alors **correct** pour une spécification donnée si pour toute entrée du programme qui vérifie la précondition alors la sortie vérifie la postcondition.

```c
int incr(int n) {
    return n+1;
}
```

Ce programme vérifie la spécification suivante :

- Précondition : `n` est pair

- Postcondition : `f(n)` est impair

```c
// Function to check if an array is sorted
bool is_sorted(int *a, int n) {
    while (--n >= 1) {
        if (a[n] < a[n - 1])
            return false;
    }
    return true;
}

// Function to shuffle the elements of an array
void shuffle(int *a, int n) {
    int i, t, r;
    for (i = 0; i < n; i++) {
        t = a[i];
        r = rand() % n;
        a[i] = a[r];
        a[r] = t;
    }
}

// BogoSort function to sort an array
void bogosort(int *a, int n) {
    while (!is_sorted(a, n))
        shuffle(a, n);
```

Le `bogosort` tire aléatoirement des permutations d'une liste (ou tableau) jusqu'à l'avoir trié.

> Remarque : On parle ici de **correction partielle**. Cela consiste à démontrer que le programme est correct en supposant qu'il termine (même si cette supposition est fausse.

On dit qu'un programme est **correct** lorsque l'on a **correction partielle** + **terminaison**.

III - Correction de programmes impératifs

```c
int max_arr(int len, int* a) {
    assert(len > 0);
    int m = a[0];
    for (int i = 1; i < len; i++) {
        m = max(a[i],m);
    }
    return m;
}
```

Spécification de `max_arr` :

- Précondition : `len > 0` (le tableau a est non vide)
- Postcondition : Renvoie la valeur maximale de a, c'est-à-dire $max_{i \in [0, len[} a[i]$.

Pour cela on utilise la notion **d'invariant de boucle**.

Un invariant de boucle est une propriété mathématique sur les variables du programme qui :

- Est vrai avant la boucle
- Est préservée par une itération de la boucle

Cette propriété sera donc vraie à la fin de l'exécution de la boucle.

> Remarque : Cette propriété doit impliquer la postcondition.

Sur l'exemple de `max_arr` : prenons comme invariant :

$$m = max_{j \in [0,i[} a[j]$$

Vérifions que c'est un bon invariant.

Avant la boucle :

$m = a[0]$ et $i = 1$

Or $max_{j \in [0,i[} a[j] = a[0] = m$

Donc l'invariant est vrai

Hérédité :

Si l'invariant est vrai **en début de boucle** montrons qu'il sera vrai en début de boucle suivante. En effet en début de boucle on a $max_{j \in [0,i[} a[j]$.

> Notation : Par convention on note m' et i' les valeurs des variables m et i après une itération de boucle.

On a $m' = max(a[i], m)$ et $i' = i + 1$

Donc $m' = max\left(a[i], max_{j \in [0,i[} a[j]\right) = max_{j \in [0,i]} a[j]$

Et donc comme $i' = i + 1$: $m' = max_{j \in [0,i'-1]} a[j]$

Puis on a $m' = max_{j \in [0,i'[} a[j]$

Donc $m = max_{j \in [0,len[} a[j]$

Finalement $m = max_{j \in [0,len[} a[j]$

C'est exactement la postcondition.

IV - Correction de programmes Récursifs

```c
int fibo(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return fibo(n-1) + fibo(n-2);
}
```

Spécification

- Précondition : $n \geq 0$
- Postcondition : renvoie u_n ou u est définie par $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$

La correction de programme récursifs se démontre par récurrence.

Prenons l'exemple du programme ci-dessus.

Pour tout $n \in \N$ on pose $H(n) : fibo(n) = u_n$

Initialisation

- Si $n = 0, fibo(0) = 1 = u_0$

- Si $n = 1$, $fibo(1) = 1 = u_1$

Hérédité

On suppose $n > 1$

$fibo(n)$ renvoie $fibo(n-1) + fibo(n-2)$

Par hypothèse de récurrence, comme $n - 1 < n$ et $n - 2 < n$ et $n - 1 \geq 0$, $n - 2 \geq 0$.

On a $fibo(n-1) = u_{n-1}$

Et $fibo(n-2) = u_{n-2}$

Or $u_n = u_{n-1} + u_{n-2}$

Donc $fibo(n) = u_n$

Le programme est donc correct.

Procédons à un autre exemple :

```
int sum_arr(int len, int* a) {
    if (len == 0) return 0;
    return sum_arr(len-1, a) + a[len-1];
}
```

Postcondition : renvoie $\sum_{j=0}^{len-1} a[j]$

On montre par récurrence sur `len` que la fonction est correcte c'est-à-dire elle vérifie la postcondition.

Si len = 0 : la fonction renvoie 0. Or $sum_{j=0}^{len-1} a[j] = 0$.

Si len > 0 : Par hypothèse de récurrence, `sum_arr(len-1, a)` renvoie $\sum_{j=0}^{len-1} a[j]$.

Donc `sum_arr(len, a)` renvoie $a[len-1] + \sum_{j=0}^{len-1} a[j] = \sum_{j=0}^{len-1} a[j]$.

L'invariant de boucle de la version impérative serait $S = \sum_{j=0}^{i-1} a[j]$