TD 1 : Analyse des Programmes

24 septembre 2023

1 Méthode

1.1 Terminaison

Pour prouver la terminaison d'un algorithme, on montre que ses boucles terminent. Pour cela, on exhibe un **variant de boucle**, qui est une valeur **entière et positive** à chaque itération de la boucle et qui **décroît strictement**.

Exemple (algorithme d'exponentiation naïve) :

```
int exp_naive(int a, int n)
    {
    int r = 1;
    for (int i = 0; i < n; i++){
        r = r*a;
    }
    return r;
    }
```

Dans l'algorithme ci-dessus, la valeur $n-i$ est entière, positive à chaque itération de la boucle (c'est-à-dire que la condition pour rester dans la boucle implique la positivité de $n-i$), et i croît strictement, donc $n-i$ décroît strictement. La boucle termine pour toute valeur de n, donc l'algorithme termine.

1.2 Correction

Pour prouver la correction partielle d'un algorithme, on doit prouver que pour les entrées pour lesquelles l'algorithme termine (c'est-à-dire pour lesquelles toutes ses boucles terminent), les sorties correspondent à la spécification de l'algorithme. On doit pour cela être capable de déterminer l'état des variables à la sortie de chaque boucle. Pour cela, on utilise un **invariant de boucle**, qui est une **propriété vraie à chaque itération de la boucle en fonction des variables de l'algorithme.**

Pour prouver qu'une propriété est un invariant de boucle, on doit montrer :

- Qu'elle est vraie avant la première itération de la boucle ;
- Que si elle est vraie au début de la i-ème itération de la boucle, elle l'est également à la fin de la i-ème/au début de la la $i+1$-ème.

Si c'est le cas, elle sera également vraie en fonctions des variables du programme avec leurs valeurs finales en sortie de boucle. Si on a un ensemble de variables `a`, `b`, `c`... pour prouver la correction, on pourra noter a_i, b_i, c_i... les valeurs des variables à la fin de la i-ème itération de la boucle (et a_0, b_0, c_0... les valeurs avant la première itération).

Exemple : si l'on reprend l'algorithme d'exponentiation naïve, on note r_i la valeur de r à la fin de la i-ième itération de la boucle. On peut montrer qu'à chaque itération i, de la boucle `for`, on a :

$r_i = a^i$

Initialisation : Avant la première itération de la boucle, $i = 0$ et $r_0 = 1 = a^0 = a^i$.

Conservation : Supposons que $r_i = a^i$. On a $r_{i+1} = r_i \times a = a^i \times a = a^{i+1}$.

Conclusion : A la fin de chaque itération i de la boucle, $r_i = a^i$. La propriété est un invariant de boucle, elle est donc vraie en sortie de boucle avec les valeurs de sortie des variables : à la sortie de la boucle, $r = r_n = a^n$. L'algorithme renvoie donc la valeur a^n en fonction de a et n.

On retiendra dans la méthode les deux choses suivantes pour les invariants de boucles :

- Prouver la vérité de l'invariant de boucle à chaque itération avec initialisation, conservation et conclusion ;
- Définir les valeurs v_i de chaque variable v utilisée dans la boucle à chaque itération i.

1.3 Complexité

Pour calculer la complexité asymptotique d'un algorithme en fonction de la taille d'entrée n, on doit calculer en particulier calculer la complexité des boucles.

Pour cela, il faut :

- Calculer le nombre d'itérations de la boucle en fonction de la taille d'entrée n
- Calculer la complexité (majorant, ordre de grandeur) de la i-ième itération en fonction de i
- Faire la somme des complexités en fonction de i pour i entre 1 et le nombre d'itérations de la boucle.

2 Exercices

Exercice 1

1) La fonction suivante en C prend en entrée un entier n et renvoie $n!$:

```
int fact(int n)
    {
    int r = 1;
    for (int i = 1; i <= n; i++){
        r = r*i;
    }
    return r;
```

}

2) La valeur $n - i$ est un entier.

$i \leq n \Leftrightarrow n - i \geq 0$: c'est une valeur positive à l'intérieur de la boucle.

A chaque itération de la boucle, i croît strictement, d'où $n - i$ décroît strictement.

$n - i$ est donc un **variant de boucle**. Par conséquent, la boucle de l'algorithme termine, donc l'algorithme termine.

3) Soit r_k et i_k les valeurs de r et i à la fin de la k-ième itération de la boucle, pour k compris entre 1 et n, et r_0 et i_0 les valeurs de r et i avant la première itération. Montrons que la propriété "$r_k = k!$ et $i = k + 1$" est un invariant de boucle.

Initialisation : Avant la première itération, $i_0 = 1 = 0 + 1$ et $r_0 = 1 = 0!$.

Conservation : Supposons que $r_k = k!$ et $i_k = k + 1$.

$r_{k+1} = r_k \times i_k = k! \times (k + 1) = (k + 1)!$ et $i_{k+1} = i_k + 1 = k + 2$.

Conclusion : A la fin de chaque itération de boucle k, $r_k = k!$ et $i_k = k + 1$: c'est un invariant de boucle. La propriété est donc vraie en sortie de boucle avec les valeurs de sortie des variables.

A la fin de la boucle, $i = k + 1 > n$: $k + 1 = n + 1$, d'où $k = n$, donc $r = n!$. L'algorithme renvoie donc la valeur $n!$ en sortie.

Exercice 2

1) $a^21 = a \times (a^2)^10$

$= a \times (a^4)^5$

$= a \times a^4 \times (a^8)^2$

$= a \times a^4 \times (a^{16})^1$

$= a \times a^4 \times a^{16} \times (a^{32})^0 = a \times a^4 \times a^{16}$

2)

```
int exp_rapide(int a, int n)
    {
    int r = 1;

    while (n>0){
    if (n % 2) == 1{
        r = r*a;
    }
    n = n/2;
    a = a*a;
    }
    return r;
    }
```

3) n est un entier naturel. C'est une valeur positive à l'intérieur de la boucle.

A chaque itération de la boucle, la nouvelle valeur de n est $\lfloor \frac{n}{2} \rfloor$: n décroît strictement.

Donc n est un variant de boucle. Par conséquent, la boucle `while` termine, donc l'algorithme termine.

4) Soit a_i, n_i et r_i les valeurs des variables `a`, `n` et `r` à la fin de la i-ième itération de la boucle (et a_0, n_0 et r_0 leurs valeurs au début de la première itération). Montrons que la propriété $r_i \times a_i^{n_i} = a_0^{n_0}$ est un invariant de boucle.

Initialisation : $r_0 \times a_0^{n_0} = 1 \times a_0^{n_0} = a_0^{n_0}$.

Conservation : Supposons que $r_i \times a_i^{n_i} = a_0^{n_0}$.

$a_{i+1} = a_i^2$.

Si n est pair, $r_{i+1} = r_i$ et $n_{i+1} = \dfrac{n_i}{2}$, d'où $r_{i+1} \times a_{i+1}^{n_{i+1}} = r_i \times (a_i^2)^{\frac{n_i}{2}} = r_i \times a_i^{n_i} = a_0^{n_0}$.

Si n est impair, $r_{i+1} = r_i \times a_i$ et $n_{i+1} = \dfrac{n_i - 1}{2}$, d'où $r_{i+1} \times a_{i+1}^{n_{i+1}} = r_i \times a_i \times (a_i^2)^{\frac{n_i - 1}{2}} = r_i \times a_i \times a_i^{n_i - 1} = r_i \times a_i^{n_i} = a_0^{n_0}$.

Conclusion : la propriété $r_i \times a_i^{n_i} = a_0^{n_0}$ est vraie à chaque itération de la boucle, c'est donc un invariant de boucle.

Elle est donc vraie pour les valeurs finales r_F, a_F et n_F des variables en sortie de boucle, or $n_F = 0$: $r_F = r_F \times a_F^{n_F} = a_0^{n_0}$.

L'algorithme renvoie donc la valeur $a_0^{n_0}$ en sortie.

5) Soit $n_0 = 2^k$: on a $n_{i+1} = \dfrac{n_i}{2}$ à chaque itération de la boucle `while` pour i inférieur à k.

On a donc, $n_i = \dfrac{2^k}{2^i}$ pour $i \leq k$.

Au bout de $i = k$ itérations, $n_i = \dfrac{2^k}{2^k} = 1$. A la $k+1$-ième itération, $n_{k+1} = \lfloor \dfrac{n_k}{2} \rfloor = \lfloor \dfrac{1}{2} \rfloor = 0$: c'est la dernière itération ; L'algorithme itère la boucle $k+1$ fois.

6) Soit $n \in \mathbb{N}^*$. On pose $k = \lfloor \log_2(n) \rfloor$. On a $2^k \leq n < 2^{k+1}$.

En appelant `exp_naive` sur des valeurs a et n, à la i-ième itération de la boucle, pour $i \leq k$, on a $\dfrac{2^k}{2^i} \leq n_i < \dfrac{2^k}{2^{i+1}}$.

En particulier, à la k-ième itération, on a $1 \leq n_k < 2$, d'où $n_k = 1$ (car entier) : $n_{k+1} = 0$, il y a donc $k + 1 = \lfloor \log_2(n) \rfloor + 1$ itérations de boucles. Comme le coût de chaque itération est d'ordre de grandeur constant, la complexité de l'algorithme est en $\Theta(\log(n))$ (là où celle de l'exponentiation naïve est en $\Theta(n)$).

7) on considère une fonction dans laquelle un accumulateur est initialisé à 0 et, pour i compris entre 1 et n, on calcule a^i à l'aide de l'exponentiation rapide pour incrémenter l'accumulateur de a^i.

- L'algorithme itère la boucle n fois ;
- A la i-ième itération, on applique l'exponentiation rapide sur a et i : on a un coût partiel $C_p(i) = \Theta(\log(i))$
- $\displaystyle\sum_{i=1}^{n} \log(i) = \Theta(n \log(n))$: le coût total en fonction de n est donc $C(n) = \Theta(n \log(n))$.