

Multiplication russe

Introduction

Dans le cadre de l'UE calculabilité et complexité nous avons dû réaliser plusieurs tâches sur un algorithme comprenant un contrat pré-conditions/post-conditions clair, et incluant au moins une boucle tant-que. Nous avons choisis de travailler sur l'algorithme représentant la multiplication russe et d'effectuer les tâches suivantes sur ce dernier :

- Écrire son code Dafny
- Montrer la correction totale :
 - Montrer la correction partielle : les invariants seront présentées et prouvés, ainsi que les post-conditions.
 - Prouver sa terminaison : fonctions de rang et éventuels invariants seront présentées et prouvés
- Déterminer sa complexité en temps qui dans le pire des cas sera justifiée et possiblement validée expérimentalement

Nous allons donc exposer nos travaux dans ce rapport en prenant soin de suivre l'ordre décrit ci-dessus

Code en Dafny et exemple d'utilisation

0.0.1 Dafny

```

1  method multiplicationRusse(x:nat,y:nat) returns (m:nat)
2  ensures m==x*y{
3    var a := x;
4    var b := y;
5    var r := 0;
6    while(a>0)
7    {
8      invariant a>=0
9      invariant r+a*b == x*y
10     decreases a
11     {
12       if(a%2 == 0){
13         b:=2*b;
14         a:=a/2;
15       }else{
16         r:=r+b;
17         a:=a-1;
18       }
19     }
20     m:=r;
  }
```

FIGURE 1 – code Dafny

```

1  Entrées : x et y, deux entiers naturels
2  Sorties : un entier
3  a : entier ← x
4  b : entier ← y
5  r : entier ← 0
6  tant que a > 0 faire
7    si a%2 =0 alors
8      b ← 2 * b
9      a ← a//2
10   sinon
11     r ← r + b
12     a ← a - 1
13   fin
14 fin
15 retourner r
```

FIGURE 2 – Algorithme de base

0.0.2 Code en Python et exemple d'utilisation

```

1  def MultiRusse(x:int,y:int):
2      a=x
3      b=y
4      r=0
5      while(a>0):
6          if(a%2 == 0):
7              b=2*b
8              a=a//2
9          else:
10             r=r+b
11             a=a-1
12     return r
13 print("MultiRusse(2,4)")
14 print(MultiRusse(2,4))
15 print("MultiRusse(3,5)")
16 print(MultiRusse(3,5))
17 print("MultiRusse(2,2)")
18 print(MultiRusse(2,2))

```

FIGURE 3 – code Python

```

MultiRusse(2,4)
8
MultiRusse(3,5)
15
MultiRusse(2,2)
4

```

FIGURE 4 – Exemple d'utilisation

Correction totale

0.0.3 Correction partielle

Méthode de détermination de la correction partielle

On commence par déterminer l'invariant de boucle en posant les cas de base et de récursivité.

Invariant en (*) $a \geq 0$:

- Cas de Base : x est affecté à a or x est un entier naturel ainsi par typage $a \geq 0$.
- Cas Inductif : nous supposons que l'invariant $a \geq 0$ est vrai, montrons alors $a' \geq 0$:

- 1^{er} cas : est pair. Nous savons que $\forall(a, a', b, b', r, r') \in \mathbb{N}^6$
$$\left[\begin{array}{l} a \geq 0 \wedge a > 0 \wedge \\ a' = (\frac{a}{2}) \wedge a \% 2 = 0 \wedge \\ b' = 2 * b \wedge r' = r \end{array} \right].$$

Ainsi nous avons $a \geq 0 \Leftrightarrow \frac{a}{2} \geq \frac{0}{2} \Leftrightarrow a' \geq 0$.

- 2^{ème} cas : a est impair. Nous savons que $\forall(a, a', b, b', r, r') \in \mathbb{N}^6$
$$\left[\begin{array}{l} a \geq 0 \wedge a > 0 \wedge \\ r' = r + b \wedge b' = b \wedge \\ a' = a + 1 \wedge a \% 2 = 1 \end{array} \right].$$
 Ainsi nous avons $a \geq 0$ d'après le test de boucle or
$$a \geq 0 \Leftrightarrow a - 1 \geq 0 - 1 \Leftrightarrow a' \geq 0.$$

Conclusion

Dans les deux cas, nous avons bien montré que $a' \geq 0$. $a \geq 0$ est donc bien un invariant de boucle en (*).

Invariant (*) $r + a * b = x * y$:

- Cas de Base : 0 est affecté à r, x est affecté à a et y est affecté à b ainsi $r + a * b = 0 + a * b = x * y$. On a donc bien $r + a * b = x * y$.
- Cas Inductif : nous supposons que l'invariant $r + a * b = x * y$ est vrai, montrons alors que $r' + a' * b' = x' * y'$ est vrai :
 - 1^{er} cas : a est pair. Nous savons que $\forall(a, a', b, b', r, r', x, x', y, y') \in \mathbb{N}^{10}$

$$\left[\begin{array}{l} r + a * b = x * y \wedge a > 0 \wedge \\ a \% 2 = 0 \wedge a' = \frac{a}{2} \wedge r' = r \wedge \\ b' = 2 * b \wedge x' = x \wedge y' = y \end{array} \right].$$

Ainsi nous avons $r' + a' * b' = r' + \frac{a}{2} * 2 * b = r + a * b$ or d'après invariant $r + a * b = x * y$ et $x * y = x' * y'$. Nous avons donc bien $r' + a' * b' = x' * y'$.
 - 2^{ème} cas : a est impair. Nous savons que $\forall(a, a', b, b', r, r', x, x', y, y') \in \mathbb{N}^{10}$

$$\left[\begin{array}{l} r + a * b = x * y \wedge a > 0 \wedge \\ a \% 2 = 1 \wedge a' = a - 1 \wedge b' = b \wedge \\ r' = r + b \wedge b' = b \wedge x' = x \wedge y' = y \end{array} \right].$$

Ainsi nous avons $r' + a' * b' = r + b + (a - 1) * b = r + a * b + b - b = r + a * b$ or d'après l'invariant nous avons $r + a * b = x * y$ et $x * y = x' * y'$. Nous avons donc bien $r' + a' * b' = x' * y'$.

Conclusion

Dans les deux cas, nous avons montré que $r' + a' * b' = x' * y'$. $r + a * b = x * y$ est donc un invariant de boucle en (*).

Terminaison

Méthode de détermination de la terminaison

On détermine une fonction de rang et on prouve que celle-ci est valide. C'est-à-dire, on vérifie que la fonction de rang $\in \mathbb{R}$ au point T1 et qu'elle décroît lorsque l'exécution passe entre les points T1 et T2.

Fonction de rang à valeur dans \mathbb{N} :

- À chaque passage au point T1, montrons que $a \in \mathbb{N}$. D'après l'invariant de boucle $a \geq 0$ nous savons donc que $a \in \mathbb{N}$.
- À chaque fois que l'exécution passe entre le point T1 et le point T2, montrons que a décroît strictement.
 - 1^{er} cas : a est pair. Nous savons que $\forall(a, a', b, b', r, r', x, x', y, y') \in \mathbb{N}^{10}$

$$\left[\begin{array}{l} r + a * b = x * y \wedge a > 0 \wedge \\ a \% 2 = 0 \wedge a' = \frac{a}{2} \wedge r' = r \wedge \\ b' = 2 * b \wedge x' = x \wedge y' = y \end{array} \right].$$

Ainsi nous avons $a' = \frac{a}{2}$ donc $a > \frac{a}{2} \Leftrightarrow a > a'$.
Nous avons bien a strictement décroissante.

- 2^{ème} cas : a est impair. Nous savons que $\forall (a, a', b, b', r, r', x, x', y, y') \in \mathbb{N}^{10}$

$$\left[\begin{array}{l} r + a * b = x * y \wedge a > 0 \wedge \\ a \% 2 = 1 \wedge a' = a - 1 \wedge b' = b \wedge \\ r' = r + b \wedge b' = b \wedge x' = x \wedge y' = y \end{array} \right].$$
Ainsi nous avons $a' = a - 1$ donc $a > a - 1 \Leftrightarrow a > a'$. Nous avons bien a strictement décroissante.

Conclusion

Dans les deux cas, nous avons montré que a est strictement décroissante. a est donc une fonction de rang à valeur dans \mathbb{N} .

Complexité

0.0.4 Détermination de la complexité

Méthode de détermination de la complexité

Afin de calculer la complexité de cet algorithme, nous allons utiliser les propriétés 1,2,3 et 5 du cours de complexité.

Cout en opération élémentaire

```
def MultiRusse(x:int, y:int):
    a=x          —————> 1
    b=y          —————> 1
    r=0          —————> 1
    while(a>0):  —————> val init Fonction Rang * T(Corps) = 3x
                  —————> 1 + max (if, else) = 1 + 2 =3
        if(a%2 == 0):
            b=2*b    —————> 1
            a=a//2    —————> 1
        else:
            r=r+b     —————> 1
            a=a-1     —————> 1
    return r        —————> 1
```

On a donc un cout élémentaire de $1 + 1 + 1 + 1 + 3 * x = 3x + 4 = O(x)$. En approfondissant on se rend compte que c'est aussi un $\theta(\ln(x))$.

On a donc une complexité d'ordre de grandeur de $\theta(\ln(x))$

Expérimentation

0.0.5 Détermination des valeurs C_1 et C_2

Méthode de détermination des valeurs C_1 et C_2

Pour déterminer ces valeurs, nous allons faire une moyenne de temps d'exécution sur l'algorithme. Puis à l'aide d'un système d'équations, nous allons retrouver les valeurs de C_1 et C_2 .

Résolution d'équation :

$$\begin{aligned} \begin{cases} T_1 = C_1 * n_1 + C_2 \\ T_2 = C_1 * n_2 + C_2 \end{cases} &\Leftrightarrow \begin{cases} T_1 = C_1 * n_1 + C_2 \\ C_2 = T_2 - C_1 * n_2 \end{cases} \Leftrightarrow \begin{cases} T_1 = C_1 * n_1 + (T_2 - C_1 * n_2) \\ C_2 = T_2 - C_1 * n_2 \end{cases} \\ &\Leftrightarrow \begin{cases} \frac{T_1 - T_2}{n_1 - n_2} = C_1 \\ C_2 = T_2 - \left(\frac{T_1 - T_2}{n_1 - n_2}\right) * n_2 \end{cases} \Leftrightarrow \begin{cases} C_1 = \frac{T_1 - T_2}{n_1 - n_2} \\ C_2 = T_2 - \left(\frac{T_1 - T_2}{n_1 - n_2}\right) * n_2 \end{cases} \end{aligned}$$

Détermination de C_1 et C_2 à l'aide d'une expérimentation :

On détermine C_1 et C_2 à l'aide des équations ci-dessus.

```
def calculTime(n,m):
    T= 0
    for i in range(0,20):
        debut = perf_counter_ns()
        m(n,100)
        T+= perf_counter_ns()-debut
    return T/20

def c1(t1,t2,n1,n2):
    return (t1-t2)/(np.log(n1)-np.log(n2))

def c2(c1,t2,n2):
    return t2-c1*np.log(n2)

def Texp(c1,c2,n):
    return c1*np.log(n)+c2
```

```
n1 = 10**6
n2 = 8*10**9
t1 = calculTime(n1,MultiRusse)
t2 = calculTime(n2,MultiRusse)
C1 = c1(t1,t2,n1,n2)
C2 = c2(C1,t2,n2)
T1 = Texp(C1,C2,n1)
```

Ainsi on trouve : $C_1 = 833.964154737176$ et $C_2 = -3206.640584735207$. On peut maintenant comparer une valeur expérimentale et théorique.

Valeur théorique et valeur expérimentale :

```
n1 = 10**7
t1 = calculTime(n1,MultiRusse)
T1 = Texp(c1,c2,n1)
```

Les résultats dépendent de la charge du processeur de l'ordinateur. Ainsi, lorsque l'ordinateur effectue peu de tâches le temps de calcul est très proche du temps théorique.

```
Temps expérimental = 10190.0
Temps Calcul = 10235.273430789202
```

Alors que s'il effectue des tâches plus ou moins lourde le temps peut être très éloigné du temps théorique.

Temps expérimental = 11790.0
Temps Calcul = 10235.273430789202

Temps expérimental = 18945.0
Temps Calcul = 10235.273430789202

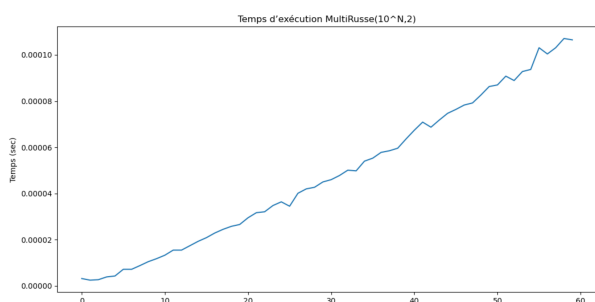


FIGURE 5 – Graphique correcte

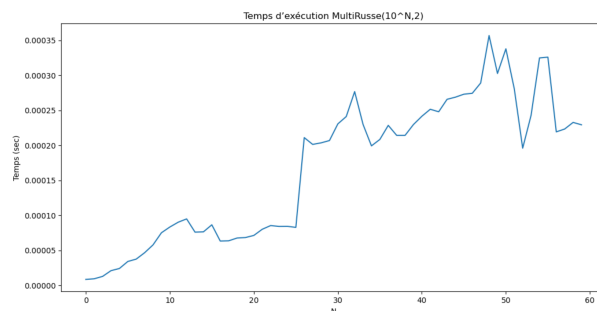


FIGURE 6 – Graphique erroné

Conclusion

En conclusion on a pu démontré la correction totale de l'algorithme et sa rapidité d'exécution du fait de sa complexité en $\theta \ln(x)$. On peut également tirer plusieurs choses de ce projet. Il faut lorsque c'est possible (comme c'était le cas ici) effectuer une vérification expérimentale, les résultats ressortant de cette expérimentation peuvent confirmer d'avantages ou réfuter les résultats obtenues de manière calculatoire. Il ne faut pas se limiter à une seule expérimentation, dans notre cas des résultats plutôt bons se montrent après une série de tests.