INFO0947: Projet1

Groupe 06: Maxime Deravet, Luca Matagne

Table des matières

1 Formalisation du problème 3
1.1 Objets utilisés . . . 3
1.2 Prédicats . . . 3

2 Spécifications du module 3

3 Découpe en sous-problèmes 3

4 Invariants de boucle 3
4.1 Invariant graphique . . . 4
4.2 Invariant formel . . . 4

5 Implémentations des Sous-Problèmes 4
5.1 SP0 . . . 4
5.2 SP1 . . . 4
5.3 SP2 . . . 4

6 Construction du code 5
6.1 Code complet . . . 5

7 Démonstration de la complexité de $filtrer()$ 6
7.1 T(A) . . . 6
7.2 T(B) . . . 6
7.3 T(C) . . . 7
7.4 Complexité du module complet . . . 7

1 Formalisation du problème

1.1 Objets utilisés

T est un tableau d'entiers de taille non nulle
N est la taille du tableau T, (N >0)
taille_utile est le nombre d'éléments du tableau qui satisfont $p(.)(0 \leq taille\_utile \leq N)$

1.2 Prédicats

p(x) est un prédicat déjà défini

$Zone\_utile(T_0, N) \equiv \#j \cdot (0 \leq j < N | p(T_0[j]))$

$Filtrage(T_0, N, taille\_utile) \equiv \forall j, 0 \leq j < taille\_utile < N, p(T_0[j])$

$Zone\_morte(T, N, taille\_utile) \equiv \forall j, taille\_utile \leq j < N, T[j] = 0$

$Sous\_suite(T_0, taille\_utile, T, N) \equiv (\forall k, 1 \leq k < taille\_utile, (\exists j, 0 \leq j < N, T_0[j] = T[k])) \wedge (\exists l, o \leq l < j, T_0[l] = T[k-1])$

2 Spécifications du module

```
/**
 *
 * @préconditions : ∀i, 0 ≤ i < N, T[i] ∈ ℤ ∧ N > 0
 * @postconditions : N = N₀ ∧ filtrer(T, N) = taille_utile ∧ T = [(Sous_suite(T₀, taille_utile, T, N) ∧ Filtrage(T₀, N, taille_utile)||Zone_morte(T, N, taille_utile))]
 */
int filtrer(int *T, int N);
```

3 Découpe en sous-problèmes

SP0 : Parcours du tableau et vérification de $p(\cdot)$ pour chaque valeur
SP1 : Si p n'est pas vérifié, on place un 0 dans la case actuelle, et on continue le parcours du tableau
SP2 : Si p est vérifié, on vérifie si on a déjà rencontré des valeurs qui ne vérifient pas p
SP2.1 : Si c'est le cas, alors on met la valeur actuelle dans la case à l'index taille_utile, puis 0 dans la case actuelle. On incrément taille_utile. On poursuit la boucle ensuite
SP2.2. Si on n'a pas encore rencontré de valeur ne vérifiant pas p, alors on incrémente taille_utile, et on poursuis la boucle.

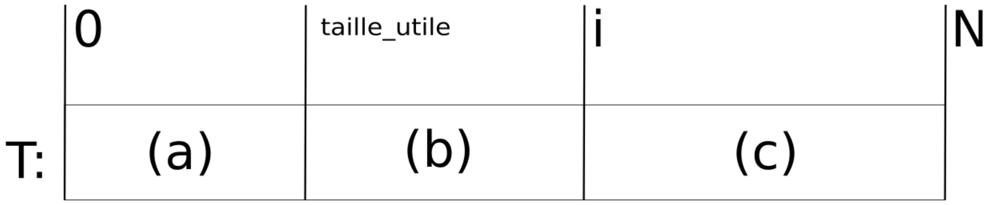
$SP0 \supset (SP1, SP2 \supset (SP2.1, SP2.2))$

4 Invariants de boucle

Le SP0 fait apparaître la nécessité d'une boucle pour résoudre le problème. Voici les invariants de cette boucle :

4.1 Invariant graphique

Invariant graphique de la fonction filtrer()



a) Les valeurs vérifient $p(\cdot)$ et sont dans le même ordre que T_0
b) Contient des 0 (il y en a i - taille_utile)
c) Le tableau n'est pas encore filtré

4.2 Invariant formel

$N = N_0$
$\wedge\, 0 \leq taille\_utile \leq i < N$
$\wedge\, \forall x, 0 \leq x < taille\_utile, Filtrage(T_0, i, taille\_utile), Sous\_suite(T_0, taille\_utile, T, i)$
$\wedge\, \forall y, taille\_utile \leq y < i, Zone\_morte(T, i, taille\_utile)$

fonction de terminaison : N-i

5 Implémentations des Sous-Problèmes

5.1 SP0

```
while(i<N){ //SP0
        if(!test(T[i])){ //SP0
        //SP1
        }else{
        //SP2
        }
    }
```

5.2 SP1

```
if(!test(T[i])){//SP0
   T[i]= 0;//SP1
   i++;//SP1
   }
```

5.3 SP2

```
else{//SP0
    if(taille_utile!= i){//SP2
        T[taille_utile]= T[i];//SP2.1
        T[i]= 0;//SP2.1
        i++;//SP2.1
```

```
        taille_utile++;//SP2.1
    }else{//SP2
        taille_utile++;//SP2.2
        i++;//SP2.2
    }
```

6 Construction du code

6.1 Code complet

```
int filtrer(int *T, int N){
    assert(T != NULL && N >0);
    // $\forall i, 0 \leq i < N, T[i] \in \mathbb{Z} \wedge N < 0$
    int i= 0;
    int taille_utile = 0;
    // $\forall i, 0 \leq i < N, T[i] \in \mathbb{Z} \wedge N < 0 \wedge i = 0 \wedge taille\_utile = 0$

    // $N = N_0$
    // $\wedge$
    // $0 \leq taille\_utile \leq i < N$
    // $\wedge$
    // $\forall x, 0 \leq x < taille\_utile, Filtrage(T_0, i, taille\_utile), Sous\_Suite(T_0, taille\_utile, T, i)$
    // $\wedge$
    // $\forall y, taille\_utile \leq y < i, Zone\_morte(T, i, taille\_utile)$
    while(i<N){
    // $N = N_0$
    // $\wedge$
    // $0 \leq taille\_utile \leq i < N$
    // $\wedge$
    // $\forall x, 0 \leq x < taille\_utile, Filtrage(T_0, i, taille\_utile), Sous\_Suite(T_0, taille\_utile, T, i)$
    // $\wedge$
    // $\forall y, taille\_utile \leq y < i, Zone\_morte(T, i, taille\_utile)$
    // $\wedge$
    // $i < N$
        if(!test(T[i])){
        // $Filtrage(T_0, i-1, taille\_utile-1) \wedge Zone\_Morte(T, i-1, taille\_utile-1) \wedge \neg p(T[i])$
            T[i]= 0;
            // $Filtrage(T_0, i, taille\_utile-1) \wedge Zone\_Morte(T, i, taille\_utile-1)$
            i++;
            // $Filtrage(T_0, i-1, taille\_utile-1) \wedge Zone\_Morte(T, i-1, taille\_utile-1)$
        }else{
        // $Filtrage(T_0, i-1, taille\_utile-1) \wedge Zone\_Morte(T, i-1, taille\_utile-1) \wedge p(T[i])$
            if(taille_utile!= i){
            // $Filtrage(T_0, i-1, taille\_utile-1) \wedge Zone\_Morte(T, i-1, taille\_utile-1)$
            // $\wedge$
            // $p(T[i]) \wedge taille\_utile \neq i$
                T[taille_utile]= T[i];
                T[i]= 0;
                // $Filtrage(T_0, i, taille\_utile) \wedge Zone\_Morte(T, i, taille\_utile)$
                // $\wedge$
                // $taille\_utile \neq i$
                i++;
                taille_utile++;
                // $Filtrage(T_0, i-1, taille\_utile-1) \wedge Zone\_Morte(T, i-1, taille\_utile-1)$
                // $\wedge$
                // $taille\_utile \neq i$
            }else{
            // $Filtrage(T_0, i, taille\_utile) \wedge Zone\_Morte(T, i, taille\_utile)$
```

```
                // ∧
                // p(T[i]) ∧ taille_utile = i
                    taille_utile++;
                    i++;
                    // Filtrage(T_0, i − 1, taille_utile − 1) ∧ Zone_Morte(T, i − 1, taille_utile − 1)
                    // ∧
                    // p(T[i]) ∧ taille_utile = i
                }
            }
        }//fin while
        // N = N_0
        // ∧
        // 0 ≤ taille_utile ≤ i < N
        // ∧
        // ∀x, 0 ≤ x < taille_utile, Filtrage(T_0, i, taille_utile), Sous_Suite(T_0, taille_utile, T, i)
        // ∧
        // ∀y, taille_utile ≤ y < i, Zone_morte(T, i, taille_utile)
        // ∧
        // i ≥ N
        return taille_utile;
        // N = N_0
        // ∧
        // filtrer(T, N) = taille_utile
        // ∧
        // T = [(Sous_suite(T_0, taille_utile, T, N)∧Filtrage(T_0, N, taille_utile))||Zone_morte(T, N, taille_utile)]
}//fin filtrer
```

7 Démonstration de la complexité de $filtrer()$

Construisons la fonction T(·), qui est la somme des complexités de chaque partie de $filtrer()$:

La première partie du code regroupe les initialisations de variables et est appelée T(A). La boucle *while* et son contenu seront regroupés dans la fonction T(B). Et pour terminer, l'instruction *return* sera reprise dans la fonction T(C).

7.1 T(A)

```
    assert(T != NULL && N >0);
    int i= 0;
    int taille_utile = 0;
```

T(A) regroupe uniquement des déclarations et vérifications de variables. Selon la règle 1 (cfr. INFO0946, chapitre 4),nous pouvons dire que $T(A) = 1$.

7.2 T(B)

```
while(i<N){

        if(!test(T[i])){
            T[i]= 0;
            i++;

        }else{
            if(taille_utile!= i){
                T[taille_utile]= T[i];
                T[i]= 0;
                i++;
```

```
                taille_utile++;
            }else{
                taille_utile++;
                i++;
            }
        }
    }//fin while
```

Selon la règle 5, la complexité d'une boucle *while* est la complexité du contenu multipliée par le nombre de tours.

Intéressons-nous donc au contenu : Pour commencer, nous avons une première structure *if else*. Sous le premier *if* se trouvent uniquement des affectations et incrémentations de variables. Selon la règle 2, la complexité théorique est donc $T_1(n)$.

Ensuite, sous le premier *else*, se trouve une deuxième structure *if else*. Regardons ce qu'il s'y trouve :

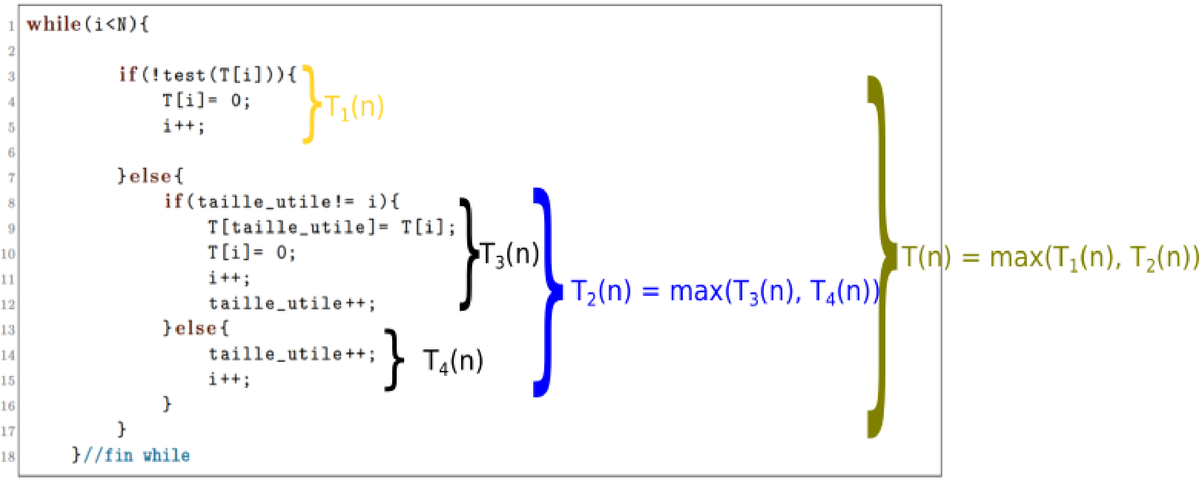
Encore une fois, sous le second *if*, se trouvent uniquement des opérations d'incrémentations et d'affectations de variables. La complexité théorique est $T_3(n)$(selon la règle 2).

Sous le second *else*, de nouveau, seulement des incrémentations de variables, et donc une complexité théorique $T_4(n)$ (selon la règle 2).

En additionnant le tout, cela nous donne : $T_2(n) = max(T_3(n), T_4(n))$
i est le nombre de tours de boucle

$$\begin{aligned} T(B) &= max(T_1(n), T_2(n)) * i \\ &= T(n) * i \\ &= T(n) \end{aligned}$$

Voici un schéma reprenant les explications ci-dessus :



7.3 T(C)

Étant donné que la fonction T(C) ne reprend qu'une seule instruction, qui est le *return*, sa complexité est égale à 1 (règle 1).

7.4 Complexité du module complet

Au final, nous obtenons :

$$
\begin{aligned}
T(N) &= T(A) + T(B) + T(C) \\
&= 1 + n + 1 \\
&= n + 2 \\
&= n
\end{aligned} \tag{1}
$$

La complexité du module $filtrer$ est donc **linéaire.**