INFO0947: Construction de programme

Groupe 27: Alexandru Dobre, Sami Ouazouz

Table des matières

1 Introduction

1.1 Contexte

Nous voulons construire une fonction dans laquelle nous allons passer en argument un tableau de nombres entiers et un nombre entier positif qui représente la taille du tableau. Nous voudrions que cette fonction nous retourne la taille du nombre d'éléments étant à la fois préfixe et suffixe de ce tableau.

1.2 Fonctionnement de la fonction

Soit un tableau T, contenant N valeur valeurs entières ($N \geq 0$). Nous voulons construire une fonction qui va retourner un entier k ($k \in [0, ..., N-1]$) tel que le sous tableau $T[0, k-1]$ est un préfixe de T et $T[N-k, N-1]$ est un suffixe de T, c'est-à-dire que leurs éléments soient identiques.

Extrait du prototype de la fonction :

```c
int prefixe_suffixe(int *T, const unsigned int N);
```

Extrait de Code 1 – Fonction souhaitée

où T et N ne sont pas modifiés.

1.3 Exemple d'utilisation

Soit F un tableau de 10 entiers et lg contient la taille du prefixe-suffixe :

```c
int F[10] = {1, 2, 3, 1, 2, 3, 1, 2, 3, 1};
unsigned int lg = prefixe_suffixe(F, 10);
printf("Le plus long prefixe-suffixe du tabLeau est de taille %u.\n", lg);
```

Extrait de Code 2 – Exemple d'utilisation

Le printf va nous afficher : Le plus long prefixe-suffixe du tableau est de taille 4.

2 Formalisation du Problème

Soit **prefixe_suffixe(*T, N)** , une notation telle que :
- T un tableau d'entiers.
- N est la taille du tableau ($N \geq 0$).

On a $prefixe\_suffixe(*T, N) \equiv \forall k, 0 \leq k \leq N-1 : T[0, k] = T[N-k, N-1]$.

Soit **est_pref_suff(T, k, N)** une notation telle que :
- T est un tableau d'entiers
- k est la taille du préfixe/suffixe à tester
- N est la taille du tableau

$est\_pref\_suff(T, k, N) \equiv \forall i, 0 \leq i < k : T[i] = T[N-k+i]$

Soit **max_prefixe_suffixe(*T, i , N)** une notation pour le plus long préfixe-suffixe de T telle que :
- T est un tableau d'entiers
- i est la position de comparaison
- N est la taille du tableau

$max\_prefixe\_suffixe(*T, i, N) \equiv \exists k, 0 \leq i \leq k < N : est\_pref\_suff(T, k, N)$

3 Définition et Analyse du Problème

3.1 Définition du Problème

3.1.1 Input

— T : Un tableau d'entiers
— N : La taille du tableau

3.1.2 Output

— La taille k du plus long préfixe qui est aussi un suffixe du tableau

3.1.3 Caractérisation des inputs

— **T** est un tableau non nul d'entiers
  — **int** *T ;
— **N** est un entier non signé tel que $N \geq 0$
  — **unsigned int** N ;

3.2 Analyse du Problème

On va découper le problème en sous-problèmes (SPs) afin de mieux l'analyser. Un sous-problème est une partie du problème qui peut être résolu indépendamment et qui peut être combiné avec d'autres sous-problèmes pour résoudre le problème global. Ils peuvent être de plusieurs types :
— Lecture au clavier.
— Affichage à l'écran
— Réaliser une action (Vérifier une propriété, une condition, sommer,...)
— Énumérer des valeurs et effectuer une action
Dans notre cas, nous avons affaire à deux sous-problèmes :
— **SP1** : Énumération des tailles possbiles de préfixe-suffixe.
— **SP2** : Comparer les valeurs de préfixe et de suffixe.

4 Specifications

4.1 Sous-problème 1

Nous voulons écrire une boucle dans laquelle k balaye toutes les valeurs possibles pour le préfixe-suffixe. Pour ceci définissons la fonction :
— **Input :**
  — N, la taille du tableau
— **Output :**
  — k, la taille du préfixe-suffixe
— **Caractérisation de l'input :**
  — N est un entier non signé tel que $N \geq 0$
  — **unsigned int** N

Il nous faut donc écrire une boucle while qui va balayer toutes les valeurs possibles de k de $N - 1$ à 0.
— **Déclaration du compteur :**
  — **unsigned int** $k = N - 1$

— **Nombres de tours dans la boucle :**
  — $N - 1$
— **Gardien de boucle :**
  — $k > 0$
— **Corps de Boucle :**
  — **Comparer les valeurs de préfixe et de suffixe :**
    — SP2
  — Décrémenter k

```
/*
 * @pre: N initialisé ∧ N > 0
 * @post: T = T_0 ∧ N = N_0 ∧ k = max_prefixe_suffixe(*T, k, N) ∧ k ≥ 0
 */
unsigned int k = N - 1;
while (k > 0) {
   // SP2
   k--;
}
```

Extrait de Code 3 – SP1

4.2 Sous-problème 2

Nous voulons écrire une boucle dans laquelle on compare les valeurs de préfixe et de suffixe. Pour ceci définissons la fonction :

— **Input :**
  — T, le tableau d'entiers
  — N, la taille du tableau
  — k, la taille du préfixe-suffixe
  — i, le compteur de la boucle
— **Output :**
  — k, la taille du préfixe-suffixe
— **Caractérisation des inputs :**
  — N est un entier non signé tel que $N \geq 0$
  — **unsigned int** N
  — k est un entier non signé tel que $0 \leq k < N$
  — **unsigned int** $k = N - 1$
  — i est un entier non signé tel que $0 \leq i < k$
  — **unsigned int** $i = 0$
  — T est un tableau d'entiers tel que $T[0, N - 1]$
  — **int** $T[N]$

Il nous faut donc écrire une boucle while qui va balayer toutes les valeurs possibles de i de 0 à k-1 et trouver des correspondances entre les valeurs de préfixe i et de suffixe $N - k + i$.

— **Déclaration du compteur :**
  — **unsigned int** $i = 0$
— **Nombres de tours dans la boucle :**
  — k
— **Gardien de boucle :**
— **Condition de sortie :**
  — $i < k$ && $T[i] == T[N - k + i]$
— **Corps de Boucle :**

- **Vérifier si i et k sont égaux :**
  - if (i == k)
- Incrémenter i

```c
/*
 * @pre: T initialisé ∧ N > 0 ∧ k initialisé ∧ 0 ≤ i < k
 * @post: T = T_0 ∧ N = N_0 ∧ k = max_prefixe_suffixe(*T, k, N) ∧ k ≥ 0
 */
unsigned int i = 0;
while (i < k && T[i] == T[N - k + i]) {
   i++;
}
if (i == k) {
   return k;
} else {
   i = 0;
}
```

Extrait de Code 4 – SP2

5 Invariants

5.1 Explications

La raison d'être de cette section est de définir les invariants nécessaires à la construction du programme. Les invariants sont des propriétés qui doivent être vérifiées à chaque étape de l'exécution du programme. Ils permettent de garantir que le programme fonctionne correctement et produit les résultats attendus. Il y a une règle qui est que chaque boucle nécessite un invriant. Nous aurons donc 2 invariants graphiques et 2 invariants formels

Nous allons donc définir les invariants graphiques. Le premier aura pour objectif de définir la boucle de décrementation de k et le seconde couvrira la boucle qui compare les valeurs du tableau entre $T[i]$ et $T[N - k + i]$.

5.2 SP1

5.2.1 Invariant Graphique

Pour le premier invariant, nous allons définir la boucle de décrementation de k. La valeur de k est initialisée à $N-1$ et elle est décrémentée jusqu'à ce qu'elle atteigne 0. Attention, elle atteint zéro uniquement si lors du SP2 nous n'avons trouvé aucune correspondance dans le tableau.

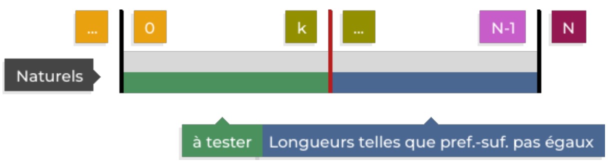

FIGURE 1 – Invariant graphique 1

Le critère d'arrêt de la boucle est $k == 0$. Donc, le gardien de boucle sera $k > 0$.

5.2.2 Invariant Formel

$$\boxed{\begin{array}{c} N = N_0 \\ \wedge \\ 0 \leq k \leq N-1 \\ \wedge \\ k \neq est\_pref\_suff(*T, k, N) \end{array}}$$

5.3 SP2

5.3.1 Invariant Graphique

Pour le second invariant, nous allons définir la boucle de comparaison entre les valeurs du tableau entre $T[i]$ et $T[N-k+i]$. La valeur de i est initialisée à 0 et elle est incrémentée jusqu'à ce qu'elle atteigne k.

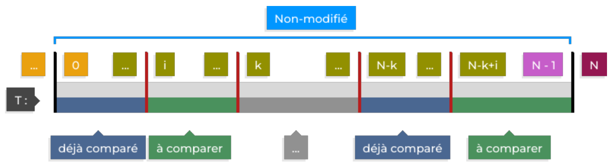

FIGURE 2 – Invariant graphique 2

Le critère d'arrêt de la boucle est $i == k$. Donc, le gardien de boucle sera $i < k$.

5.3.2 Invariant Formel

$$\boxed{\begin{array}{c} N = N_0 \wedge T = T_0 \\ \wedge \\ 0 \leq i < k < N \\ \wedge \\ i = max\_pref\_suff\_(*T, i, N) \end{array}}$$

6 Approche Constructive

6.1 Sous-problème 1

Voici les instructions ainsi que la construction instruction par instruction pour le sous-problème 1.

```
{Pré ≡ $N = N_0 \wedge 0 \leq N$}
int k = N - 1;
{Inv ≡ $N = N_0 \wedge 0 \leq N \wedge k = N - 1 \wedge k \neq$ est_pref_suff$(*T, i, N)$}
while (k > 0) {
  {Inv ∧$B \equiv N = N_0 \wedge 0 \leq N \wedge k \leq N - 1 \wedge k \neq$ est_pref_suff$(*T, i, N) \wedge k > 0$}

  {$N = N_0 \wedge 0 \leq N \wedge 0 < k \leq N - 1 \wedge k \neq$ est_pref_suff$(*T, i, N)$}
  // SP2
  if (i == k){
    {$N = N_0 \wedge 0 \leq N \wedge 0 < k \leq N - 1 \wedge k =$ est_pref_suff$(*T, i, N)$}
    return k;
  }
  else{
    i = 0;
  }
```

```
        k --;
        {Inv ∧B ≡ N = N_0 ∧ 0 ≤ N ∧ k ≤ N − 1 ∧ k ≠ est_pref_suff(*T, i, N)}
    }
    {Inv ∧¬B ≡ T = T_0 ∧ N ≥ 0 ∧ k = 0}
```

Extrait de Code 5 – Sous-problème 1

6.2 Sous-problème 2

Voici les instructions ainsi que la construction instruction par instruction pour le sous-problème 2.

```
    {Pré ≡ N = N_0 ∧ T = T_0 ∧ 0 < k ≤ N − 1}
    int i = 0;
    {Inv ≡ N = N_0 ∧ T = T_0 ∧ i = 0 < k ≤ N − 1}
    while(i < k && T[i] == T[N - k + i]){
      {Inv ∧B ≡ N = N_0 ∧ T = T_0 ∧ 0 ≤ i < k < N − 1 ∧ T[i] = T[N − k + i]}
      i++;
      {N = N_0 ∧ T = T_0 ∧ 0 < i ≤ k < N − 1 ∧ T[i] = T[N − k + i]}
    }
    {Inv ∧¬B ≡ T = T_0 ∧ i == k ∨ T[i] ≠ T[N − k + i] ∧ i = max_pref_suff(*T, i, N)}
```

Extrait de Code 6 – Sous-problème 2

7 Code Complet

7.1 Code du header

```
 #ifndef __PREFIXE_SUFFIXE__
 #define __PREFIXE_SUFFIXE__

 /*
  * @pre: T initialisé ∧ N >= 0
  * @post: T = T_0 ∧ N = N_0 ∧ k = max_prefixe_suffixe(*T, i, N)
  */
int prefixe_suffixe(int *T, const unsigned int N);

#endif
```

Extrait de Code 7 – prefixe_suffixe.h

7.2 Code du module

```
    #include <assert.h>
    #include <stdlib.h>

    #include "prefixe_suffixe.h"


    int prefixe_suffixe(int *T, const unsigned N){
      assert(T != NULL && N >= 0);

      int k = N-1;
      int i = 0;

      while (k > 0){
```

```
    while (i < k && T[i] == T[N - k +i]) {
      i++;
    } // fin while
    if (i == k) {
      return k;
    }
    else{
      i = 0;
    }
    k--;
  } // fin while
  return 0;
} // fin prefixe_suffixe
```

Extrait de Code 8 – prefixe_suffixe.c

7.3 Code du programme principal

```
#include <stdio.h>

#include "prefixe_suffixe.h"

#define N1 9
#define N2 10
#define N3 9

int main(){

  int T1[N1] = {1,4,2,4,5,1,4,2,4};
  int T2[N2] = {1,2,3,2,1,1,2,3,2,1};
  int T3[N3] = {3,2,3,2,1,2,3,2,1};
  int T4[N1] = {1,1,1,1,1,1,1,1,1};

  printf("Longueur plus long préfixe/suffixe de T1: %u\n",
        prefixe_suffixe(T1, N1));
  printf("Longueur plus long préfixe/suffixe de T2: %u\n",
        prefixe_suffixe(T2, N2));
  printf("Longueur plus long préfixe/suffixe de T3: %u\n",
        prefixe_suffixe(T3, N3));
  printf("Longueur plus long préfixe/suffixe de T4: %u\n",
        prefixe_suffixe(T4, N1));
}
```

Extrait de Code 9 – main-prefixe_suffixe.c

8 Complexité

8.1 Analyse du code

Décomposons le code de la fonction prefixe_suffixe en plusieurs blocs :

```
int k = N-1;
int i = 0;
```

Extrait de Code 10 – T(A)

```
  while (k > 0){
    // T(B2)
    // T(C)
    // T(D)
 }
```

Extrait de Code 11 – T(B1)

```
 while(i < k && T[i] == T[N - k +i]){
    i++; // T(B2')
 }
```

Extrait de Code 12 – T(B2) et T(B2')

```
 if (i == k){
    return k;
 }
```

Extrait de Code 13 – T(C)

```
 else{
    i = 0;
 }
```

Extrait de Code 14 – T(D)

```
  k--;
```

Extrait de Code 15 – T(E)

8.2 Calcul de la complexité

Dans le pire des cas, on entre dans la première boucle N - 1 fois (lorsque k va de $N-1$ à 1). Dans la boucle imbriquée, on entre jusqu'à k fois. On a donc

On a donc, pour chaque valeur de k :

$$T(B1) = T(B2) + max(T(C), T(D)) + T(E) = k + 2$$

On a aussi que :

$$T(N) = T(A) + T(B1)$$

La complexité totale est donc :

$$T(N) = T(A) + \sum_{k=1}^{N-1}(k+2) = 1 + \sum_{k=1}^{N-1} k + \sum_{k=1}^{N-1} 2$$

$$T(N) = 1 + \frac{(N-1) \cdot N}{2} + 2(N-1)$$

$$T(N) = \frac{2 + N^2 - N + N - 1}{2}$$

$$T(N) = \frac{N^2 + 1}{2}$$

Cette fonction peut donc être bornée par $O(N^2)$.

9 Conclusion