

INFO0947: Prefixe and Suffixe

Groupe 33: Aleksandr PAVLOV, Alexandre GENDEBIEN

Table des matières

| | | |
|----------|---|----------|
| 1 | Introduction | 3 |
| 2 | Formalisation du Problème | 3 |
| 3 | Définition et Analyse du Problème | 3 |
| 3.1 | Input/Output | 3 |
| 3.2 | Découpe en sous-problèmes | 3 |
| 4 | Specifications | 3 |
| 4.1 | SP1 : Recherche du plus grand prefixe-suffixe : | 3 |
| 4.2 | SP2 : Vérification que le préfixe et suffixe de longueur k sont égaux : | 3 |
| 5 | Invariants | 3 |
| 5.1 | SP1 : | 3 |
| 5.2 | SP2 : | 4 |
| 6 | Approche Constructive | 4 |
| 7 | Code Complet | 5 |
| 8 | Complexité | 6 |
| 9 | Conclusion | 6 |

1 Introduction

Dans le cadre du cours INFO-0947 nous avons du résoudre un problème donné et en créer un algorithme en C capable de : trouver la longueur du plus grand sous-tableau qui soit à la fois le préfixe et le suffixe d'un tableau donné. Ce problème sera complètement documenté en LaTeX. Pour ce faire, nous devons prendre compte de plusieurs contraintes : ne pouvons pas utiliser de tableau intermédiaire et nous avons l'obligation d'utiliser uniquement des boucles de type while.

2 Formalisation du Problème

$$\text{prefixe_suffixe}(T, N) \equiv \max\{k \mid k \in 0 \dots N-1 \wedge \forall i \cdot (i \in 0 \dots k-1 \Rightarrow T[i] = T[N-k+i])\}$$

3 Définition et Analyse du Problème

3.1 Input/Output

- **Input** : Un tableau d'entiers T de taille N , $N \geq 0$
- **Output** : Un entier k représentant la longueur du plus grand sous-tableau préfixe et suffixe 0 si aucun sous-tableau non trivial ne satisfait la condition

3.2 Découpe en sous-problèmes

Nous décomposons le problème en deux Sp :

1. Recherche du plus grand préfixe-suffixe de longueur k possible de $N-1$ à 1
2. Vérification que le préfixe et suffixe de longueur k sont égaux

4 Specifications

4.1 SP1 : Recherche du plus grand préfixe-suffixe :

- **Précondition** : T pointer vers un tableau, $N \geq 0$
- **Postcondition** : $T = T_0$
- **Retour** : le plus grand $k \in [0, N-1]$ tel que $T[0..k-1] = T[N-k..N-1]$

4.2 SP2 : Vérification que le préfixe et suffixe de longueur k sont égaux :

- **Précondition** : T pointer vers un tableau, $0 < N, 0 < k < N$
- **Postcondition** : $T = T_0$
- **Retour** : 1 si $T[0..k-1] = T[N-k..N-1]$ sinon 0

5 Invariants

5.1 SP1 :

Invariant formel :

$$0 < k < N$$

\wedge

$$T[0..k-1] \neq T[N-k..N-1]$$

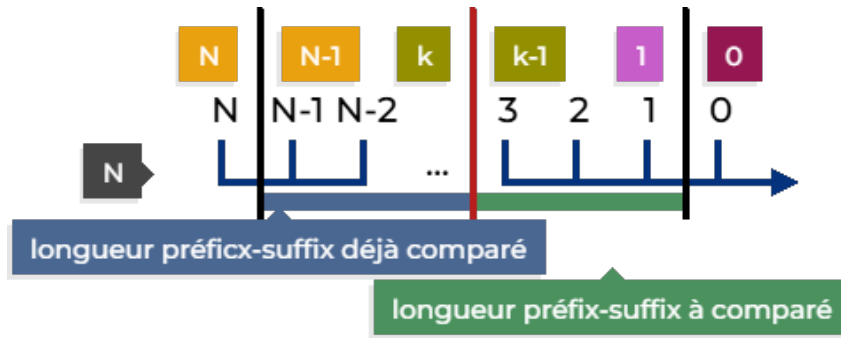


FIGURE 1 – Invariant graphique SP1

5.2 SP2 :

Invariant formel :

$$T = T_0 \wedge k = k_0$$

\wedge

$$0 \leq i < k$$

\wedge

$$T[i] = T[N - k + i]$$

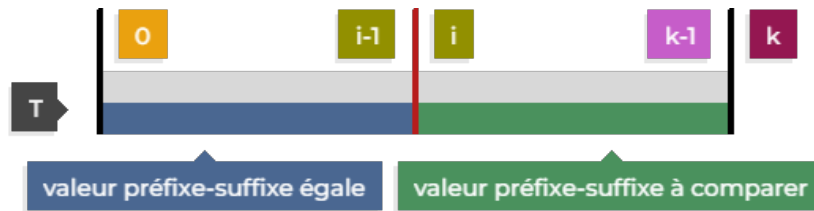


FIGURE 2 – Invariant graphique SP2

6 Approche Constructive

```

1 int prefixe_suffixe(int *T, const unsigned int N) {
2   unsigned int k = N - 1;
3   // T = T_0 ∧ N = N_0 ∧ k = N - 1
4   while (k > 0) {
5     // 0 < k < N ∧ T = T_0 ∧ N = N_0
6     if (pref_equal_suff(T, N, k)) return k;
7
8     // T[0..k-1] ≠ T[N-k..N-1]
9     k--;
10    // k = k - 1
11  }
12  // k = 0
13  return 0;
14  // T = T_0 ∧ N = N_0
15 }
```

Extrait de Code 1 – SP1

```

1 static int pref_equal_suff(int *T, const unsigned int N, const unsigned int k) {
2
3     unsigned int i = 0;
4     //  $T = T_0 \wedge N = N_0 \wedge k = k_0$ 
5     while (i <= k - 1) {
6         //  $T = T_0 \wedge N = N_0 \wedge k = k_0 \wedge 0 \leq i < k$ 
7         if (T[i] != T[N - k + i]) return 0;
8         //  $T[i] = T[N - k + i]$ 
9
10        i++;
11        //  $i = i + 1$ 
12    }
13    //  $i = k$ 
14    return 1;
15    //  $T = T_0 \wedge N = N_0 \wedge k = k_0 \wedge T[0..k-1] = T[N-k..N-1]$ 
16 }

```

Extrait de Code 2 – SP2

7 Code Complet

```

1 #include <assert.h>
2 #include <stdlib.h>
3
4 #include "prefixe_suffixe.h"
5
6 // ===== Prototypes =====
7
8 static int pref_equal_suff(int *T, const unsigned int N, unsigned int k);
9
10 // ===== Code =====
11
12 /**
13  * Sp 1
14  * Checking all prefixes starting from the longest one until we find a match
15  * or exhaust all possibilities.
16  */
17 int prefixe_suffixe(int *T, const unsigned int N) {
18     assert((T != NULL) && (0 < N));
19
20     unsigned int k = N - 1;
21     while (k > 0) {
22         if (pref_equal_suff(T, N, k)) return k;
23         k--;
24     }
25     return 0;
26 }
27
28 /**
29  * Sp 2
30  * Comparing the prefix and suffix of the given length, element by element.
31  */
32 static int pref_equal_suff(int *T, const unsigned int N, const unsigned int k) {
33     assert((T != NULL) && (0 < N) && (0 < k && k < N));
34
35     unsigned int i = 0;
36     while (i <= k - 1) {
37         if (T[i] != T[N - k + i]) return 0;

```

```

38     i++;
39 }
40 return 1;
41 }

```

Extrait de Code 3 – Implémentation de `prefixe_suffixe`

8 Complexité

Complexité :

- Complexité de la fonction `pref_equal_suff` :
 - Dans le pire cas, la fonction effectue k comparaisons
 - Dans cas maximal ($k = N - 1$), la complexité est $\mathcal{O}(N)$
- Complexité de la fonction `prefixe_suffixe` :
 - Dans le pire cas, itère sur toutes les valeurs de k de $N - 1$ à 1
 - Appelle `pref_equal_suff` pour chaque k

Complexité totale :

- $\sum_{k=1}^{N-1} \mathcal{O}(k) = \mathcal{O}(N^2)$

9 Conclusion

Pour terminer ce rapport nous pouvons ajouter que nous avons réussi à créer un algorithme en C capable de trouver le plus long préfixe qui est aussi un suffixe d'une liste donnée. Et tout cela en respectant les contraintes données tel que l'utilisation exclusives des boucles `while` et l'absence de l'utilisation de list intermédiaires.