Rapport du Challenge

Monotonic Segments and GHC Sort

Adrien BLASSIAU
Corentin JUVIGNY

ENSIIE - 22 décembre 2019

Sommaire

1. Introduction

Préliminaire Ce projet n'a pas de lien direct avec notre article de recherche *A Verified Implementation of the Bounded List Container*. Cependant, on remarque que les chercheurs n'avaient **pas spécifié ni vérifié les algorithmes associés aux tris**, car c'est une partie délicate et fastidieuse. Nous allons nous y atteler ici, sur une structure de données différentes et avec un autre langage d'annotation que celui de l'article de recherche.

Objectif L'objectif est de **développer** un algorithme de tri de liste de type patience sorting puis de le **vérifier**.
Cet algorithme a une complexité dans le pire des cas en $\mathcal{O}(n*log(n))$. Son fonctionnement, inspiré du jeu de cartes du même nom, utilise le principe de plus grands sous-ensembles croissants que nous expliquerons par la suite. Cette méthode de tri est aussi implémentée dans la bibliothèque standard d'Haskell.

Description du Challenge Le challenge est découpé en **deux parties à développer et à vérifier** :

1. pré-traiter la liste à trier `a` en succession de segments monotones, les points de coupe étant stockés dans une liste dédiée `cutpoints`
2. appliquer un algorithme de tri sur la liste `a` où les segments monotones ont tous été rendus croissants au préalable

Instructions L'ensemble des instructions pour lancer les tests et la vérification sont données dans le fichier `README.md` que nous vous invitons à lire.

2. DÉVELOPPEMENT

Soit l une liste non triée. Le développement est découpé en **4 étapes** qui forment une partie du Challenge :

1. découper la liste l en **segments monotones** $\sigma_1, \sigma_2, \ldots$
2. **inverser** les segments **décroissants**
3. **fusionner les deux premiers segments** de manière à préserver l'ordre
4. si **tous les segments n'ont pas été fusionnés**, retourner en **étape 3**.

On présente rapidement ces 4 parties.

2.1. Étape 1 : découpage de la liste en segments monotones

Soit l une liste où $l = l[0]l[1]\ldots l[n-1]$, $n \geq 0$. On pose $c = c[0]c[1]\ldots c[m-1]$. c est la liste de points de coupe associée à l si elle vérifie les propriétés suivantes :

- **non vide** : $m > 0$
- **début et fin de l** : $c[0] = 0$ et $c[m-1] = n$
- **indice de l** : $\forall i \in [\![0; m-1]\!] : 0 \leq c[i] \leq n$
- **monotonie** : $\forall i \in [\![0; m-2]\!]$:

$$l[c[i]\ldots c[i+1]] = l[c[i]]l[c[i]+1]\ldots l[c[i+1]-1]$$

est monotone, c'est-à dire :

  - soit $l[c[i]] < l[c[i]+1] < \ldots < l[c[i+1]-1]$
  - soit $l[c[i]] \geq l[c[i]+1] \geq \ldots \geq l[c[i+1]-1]$

Aucun travail n'a été effectué pour cette partie, l'algorithme de construction de segments monotones était déjà **développé** et **vérifié**.

Le code et les annotations associés à cette partie sont dans le fichier `monotonic_cutpoints.c`.

2.2. Étape 2 : inversion des segments décroissants

La deuxième étape consiste à inverser tous les segments de l qui sont décroissants de manière à n'avoir plus que des segments croissants. Voici le pseudo-code associé à cette étape :

Algorithme 2.1 Rendre croissants tous les segments de l décroissants

Require: l une liste de taille n composée de segments monotones et c la liste des points de coupe de taille m associée à l.
Ensure: Tous les segments de l sont croissants.

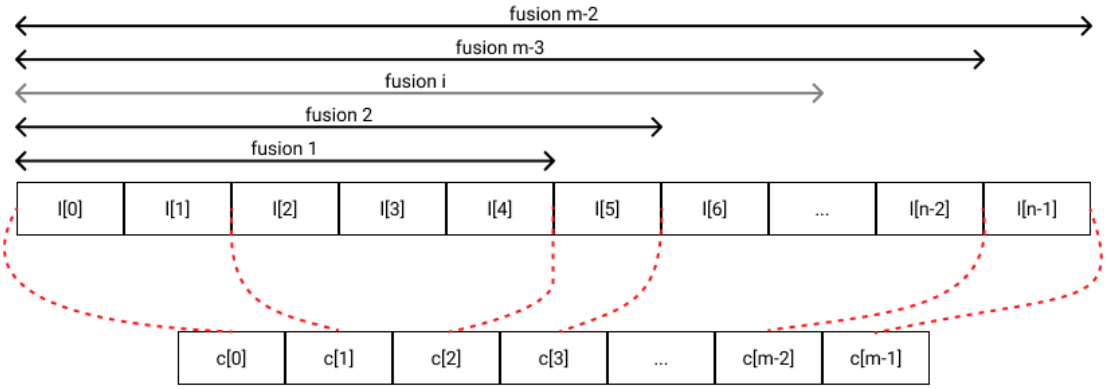
1: $i \leftarrow 0$
2: **Tant que** $i < m - 1$ **do**
3: **Si** l[c[i]] > l[c[i+1]-1] **Alors**
4: On inverse le segment $l[c[i]] \dots l[c[i+1]-1]$
5: **Fin Si**
6: $i \leftarrow i + 1$
7: **Fin Tant que**

Le travail effectué est le développement de cet algorithme ainsi que sa vérification (plus de détails dans la partie vérification).

Le code et les annotations sont dans le fichier `monotonic_reverse.c`.

2.3. Étape 3 et 4 : fusion des segments deux à deux par la gauche

La troisième étape consiste à **fusionner les segments triés croissants deux à deux par la gauche** jusqu'à ce qu'il ne reste plus qu'un seul segment entièrement trié.



À noter que ce projet est réalisé en C. Il a fallu revoir le pseudo-code qui était proposé dans le sujet :

- `.extend` n'existe pas en C pour concaténer deux listes, on a donc choisi d'utiliser une liste en plus pour stocker la fusion courante notée `sl` dans le pseudo-code.

- on ne fait pas de récursion de manière à faciliter la preuve

Voici le pseudo-code associé à cette étape :

Algorithme 2.2 Trier par ordre croissant les éléments de la liste l

Require: l une liste de taille n composée de segments croissants, c la liste des points de coupe de taille m et sl la liste utilisée pour la fusion.
Ensure: l est croissante donc triée.

```
k ← 1
i, j, ls, lt, x, y ← 0
Tant que k < m − 1 do
    i, x, y ← 0
    ls ← c[k]
    lt ← c[k + 1] − ls
    Tant que x < ls et y < lt do
        Si a[x] < a[y + ls] Alors
            sl[i] ← a[x];
            x ← x + 1;
        Sinon
            sl[i] ← a[y + ls];
            y ← y + 1;
        Fin Pour
        i ← i + 1;
    Fin Tant que
    Tant que x < ls do
        sl[i] ← a[x]
        x ← x + 1
        i ← i + 1
    Fin Tant que
    Tant que y < lt do
        sl[i] ← a[y + c[k]]
        y ← y + 1
        i ← i + 1
    Fin Tant que
    Tant que j < lt + ls do
        a[j] ← sl[j]
        j ← j + 1
    Fin Tant que
    k ← k + 1
Fin Tant que
```

Le code et les annotations de cette partie sont dans le fichier `merge.c`.

2.4. Application de l'algorithme de tri sur un exemple

On considère la liste l suivante :

$$l = [6; 3; 4; 2; 5; 3; 7]$$

La liste de ses points de coupe est

$$c = [0; 2; 4; 6; 7]$$

Les segments monotones associés sont donc :

$$63|42|53|7$$

On inverse les segments décroissants (ceux en rouge) :

$$36|24|35|7$$

Les étapes de l'algorithme de tri sont les suivantes :

$$36|24|35|7 \rightarrow 2346|35|7 \rightarrow 233456|7 \rightarrow 2334567$$

2.5. Bilan du développement de l'algorithme

On a effectué 100 000 tests unitaires sur des listes de 100 éléments aléatoires avec `Cunit`. Cela nous assure la **complétude** mais pas la **correction**. La démarche de **vérification** devrait nous assurer la **correction** et la **non complétude**.

La commande pour lancer les tests est disponible dans le fichier `README.md`.

3. VÉRIFICATION

Deux fichiers devaient être vérifiés : `monotonic_reverse.c` et `merge.c`. Le premier fichier contient la fonction `reverse` que inverse tous les segments décroissants. Le deuxième fichier contient la fonction de tri `merge` qui trie une liste pré-traitée avec la fonction `reverse` au préalable.

3.1. Spécification de la fonction reverse

On définit tout d'abord deux prédicats qui spécifient qu'une liste `a` est croissante (respectivement monotone) entre deux indices `low` et `up`.

```
/*@ predicate increasing_slice(int* a, size_t low, size_t up) =
(\forall integer i,j; low <= i < j < up ==> a[i] <= a[j]);
*/

/*@ predicate monotone_slice(int* a, size_t low, size_t up) =
(\forall integer i,j; low <= i < j < up ==> a[i] < a[j]) ||
(\forall integer i,j; low <= i <= j < up ==> a[i] >= a[j]);
*/
```

On s'intéresse maintenant aux spécifications de la fonction `reverse`.

requires Ces spécifications précisent notamment que `a` et `cutpoints` ne doivent pas occuper le même espace mémoire (`a_valid`, `res_valid` et `sep`). On a aussi besoin de certaines propriétés sur `cutpoints` que l'on réécrit (`bounds`, `beg`, `end` et `monot`).

```
/*@
  requires length < 100;
  requires cutlength > 0;
  requires a_valid: \valid(a + (0 .. length - 1));
  requires res_val: \valid(cutpoints + (0 .. cutlength-1));
  requires sep: \separated(a + (0 .. length - 1),
  cutpoints + (0 .. length));
  requires bounds: \forall integer i; 0 <= i < cutlength
  ==> 0 <= cutpoints[i] < length;
  requires beg: cutpoints[0] == 0;
  requires end: cutpoints[cutlength-1] == length;
  requires monot: \forall integer i; 0 <= i < cutlength-1
  ==> monotone_slice(a,cutpoints[i],cutpoints[i+1]);
*/
```

assigns Cette spécification précise que seule la liste `a`, que l'on a fourni en entrée de la fonction, doit être modifiée.

```
/*@
  assigns a[0 .. length-1];
*/
```

ensures Cette spécification précise que la liste `a` ne doit être composée que de segments croissants en sortie de la fonction.

```
/*@
  ensures increasing: \forall integer i; 0 <= i < cutlength-1 ==>
  increasing_slice(a,cutpoints[i],cutpoints[i+1]);
*/
```

résultats Tous les goals passent (50/50, moins d'une seconde), cette partie ne nous a pas posé de problème.

La commande pour lancer la vérification sur le fichier `monotonic_reverse.c` est disponible dans le fichier `README.md`.

3.2. Spécification de la fonction merge

requires Ces spécifications précisent notamment que `a`, `sorted_list` et `cutpoints` ne doivent pas occuper le même espace mémoire (`a_valid`, `sorted_valid`, `res_valid` et `sep`). On précise que `sorted_list` est la liste dans laquelle on effectue le merge de deux sous listes triées. On a aussi besoin de certaines propriétés sur `cutpoints` que l'on réécrit, ce sont quasiment les mêmes que pour la fonction `reverse`.
Enfin, on souhaite que les segments soient tous croissants (`inc`). Ceci est garanti si on a pré-taité la liste `a` en entrée avec la fonction `reverse` spécifiée plus haut.

```
/*@
    requires length < 100;
    requires cutlength <= length + 1;
    requires a_valid: \valid(a + (0 .. length - 1));
    requires sorted_valid: \valid(sorted_list + (0 .. length - 1));
    requires res_val: \valid(cutpoints + (0 .. cutlength-1));
    requires sep: \separated(a + (0 .. length - 1),
    sorted_list + (0 .. length - 1),cutpoints + (0 .. cutlength-1));
    requires inc: \forall integer i; 0 <= i < cutlength-1
    ==> increasing_slice(a,cutpoints[i],cutpoints[i+1]);
    requires beg: cutpoints[0] == 0;
    requires end: cutpoints[cutlength-1] == length;
    requires \forall integer i; 0 <= i < cutlength-1 ==>
    cutpoints[i] < cutpoints[i+1];
*/
```

assigns Ces spécifications précisent que seules les listes `a` et `sorted_list` que l'on a fourni en entrée de la fonction, doivent être modifiées. `a` contiendra en sortie la liste triée et `sorted_list` est utilisée pour effectuer le merge de deux listes triées.

```
/*@
    assigns sorted_list[0 .. length-1];
    assigns a[0 .. length-1];
*/
```

ensures La première spécification formalise le fait que les éléments de la liste de sortie sont les mêmes que ceux de la liste d'entrée. La deuxième spécification précise que la liste `a` est triée en sortie. Elle est commentée car nous ne sommes pas parvenus à la vérifier.

```
/*@
    ensures same_elements: \forall integer i; 0 <= i < length
    ==> \exists integer j; 0 <= j < length ==> a[i] == \old(a[j]);
    // ensures sorted_result: \forall integer i; 0 <= i < length-1
    ==> a[i] <= a[i+1];
*/
```

résultats Tous les goals passent (79/79, quelques secondes), mais cette partie nous a posé de nombreux problèmes et tout n'est pas vérifié. En effet, il manque l'invariant de boucle qui spécifie que lorsque l'on merge des sous listes triées entre elles, on forme une liste triée.

La commande pour lancer la vérification sur le fichier `merge.c` est disponible dans le fichier `README.md`.

4. Conclusion

Pour conclure, la **complétude** a été montrée grâce aux tests unitaires effectués. Concernant la **correction**, nous avons rencontré beaucoup de difficultés avec frama-c. Ainsi, seule une partie du code a été complètement vérifiée. Les fonctions utilisées pour le pré-traitement de la liste à trier le sont mais pas entièrement la fonction liée au tri.