FREI Florian, MOREL—DORIDAT Stefan, TALLANDIER Benoît, THESET Guillaume

Projet MALG

Groupe 7



Table of Contents

[I. Conversion 3](#_Toc481937108)

[1. Concept de l’algorithme 3](#_Toc481937109)

[2. Vérification Frama-c 3](#_Toc481937110)

[3. TLA+ 3](#_Toc481937111)

[II. Radix sort 3](#_Toc481937112)

[1. Concept de l’algorithme 3](#_Toc481937113)

[2. Frama-c 3](#_Toc481937114)

[3. Tla+ 4](#_Toc481937115)

[III. Cube 4](#_Toc481937116)

[1. Concept de l’algorithme 4](#_Toc481937117)

[1. Frama-c 5](#_Toc481937118)

1. Conversion
2. Concept de l’algorithme

Cet algorithme a pour de convertir un nombre décimal en écriture binaire. Pour cela, il parcourt les 31bits sur lesquels est codé le nombre décimal, il récupère la valeur binaire par un shift et affiche la valeur du bit.

1. Vérification Frama-c

La première difficulté pour ce programme fut que le résultat du shift est simplement affiché et non stocké. Nous avons donc utilisé une variable pour stocker dans la boucle les résultats successifs du shift. Cette variable nous permet de générer, au fur et à mesure de la boucle, le nombre décimal.

La boucle étant du type « for », nous somme sûr de sa terminaison. Reste à montrer que l’on se rapproche du résultat à chaque itération. Nous avons utilisé l’invariant de boucle suivant : « ». La variable n étant le nombre décimal recherché, cumul est la variable qui est le nombre décimal calculé au fil de la boucle avec les bits trouvés de n, et c est l’incrément de la boucle. Dans cette boucle, nous avons rajouté le calcul suivant : cumul=cumul+(k&1)\*pow(2,c). Ainsi, cumul s’incrémente bien et se rapproche de n.

A la fin de la boucle, nous plaçons une assertion : « //@ assert cumul==n; » pour vérifier que nous avons bien affiché le bon nombre binaire.

1. TLA+
2. Radix sort
3. Concept de l’algorithme

L’objectif de cet algorithme est de trié les éléments d’un tableau (array dans le code). IL va procéder de la façon suivante : le tableau va être trié suivant les unités des éléments du tableau, puis des dizaines, puis des centaines jusqu’à ce qu’il n’y ait plus de chiffre non traité dans les éléments du tableau.

1. Frama-c

Pour cet algorithme, nous avons utilisés l’invariant suivant sur la boucle principale :

«  »

Ceci permet de vérifier qu’à chaque itération, le tableau est trié selon le chiffre de la puissance désignée par significantDigit.

Nous avons également placé un autre invariant sur la boucle interne qui change les éléments de place (lors de l’utilisation de semiSorted). Nous avons utilisé l’invariant suivant :

Il fut difficile de trouver un bon invariant pour cette boucle car l’insertion ne se fait pas dans l’ordre des places de semiSorted. C’est qu’une fois sorti de la boucle que l’on peut vérifier le tri entièrement le tri. Pour quand même vérifier le travail effectué dans la boucle, il faut distinguer 3 cas lors de l’assertion à la position k : soit la case k-1 a déjà été affecté auquel cas nous pouvons comparer les cases k et k-1. Soit la case k-1 n’a pas encore été affecté et nous somme à la première itération (ie significantDigit=1), donc semiSorted est encore partiellement vide. Soit la case k-1 n’a pas encore été affecté et nous avons déjà fait le tri sur les unités (ie significantDigit>1), alors semiSorted contient les anciennes valeurs de array, et donc nous ne devons pas comparer les valeurs.

1. Tla+
2. Cube
3. Concept de l’algorithme

L’objectif de cet algorithme est de calculer le cube d’un nombre par addition. Il calcul les cubes des entiers jusqu’à la valeur demandée. Pour cela, il utilise le résultat suivant : et . Les variables z,v,t,w contiennent respectivement , , et . L’algorithme peut donc calculer 33 en utilisant 23 qui nécessite le calcul de 13.

1. Frama-c

Nous avons utilisé l’invariant suivant pour la boucle principale :

Nous avons également placé une précondition pour la fonction : et comme postcondition :