Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

117366 - Lógica Computacional Turma A

Relatório sobre **Provas de conjecturas do Merge Sort e Radix Sort**

Alexandre Mitsuru Kaihara - 18/0029690
Christian Luis Marcondes Costa - 15/0153538

14 de novembro de 2019

1 Introdução

Na computação, ao desenvolver algoritmos, muitas vezes depara-se com o problema de ordenação de listas. Para ordenar uma lista qualquer, temos à disposição uma grande quantidade de algoritmos, entre eles o merge sort e o radix sort.

O merge sort funciona da seguinte forma: tendo uma lista qualquer, é feita a divisão desta lista em elementos unitários, em seguida pegamos esses elementos unitários e formamos uma sublista fazendo a combinação ordenada desses elementos 2 à 2 (se possível), em seguida fazemos a ordenação entre cada uma dessas sublistas, pegando sempre o menor elemento (ou o maior para o caso de ordenação decrescente) entre as duas cabeças dessa lista e adicionando esse elemento na cauda de uma nova sublista. Quanto chegar ao ponto que temos apenas uma sublista, esta lista será uma permutação ordenada da lista inicial.

O radix sort é implementado da seguinte maneira: considere uma lista não ordenada de inteiros. Para ordenar essa lista é observado o dígito menos significativo de cada um desses números, e a ordenação é feita a partir desse valor. Em seguida pegamos o digito subsequente e realizamos a mesma operação ordenação, até que seja alcançado o dígito mais significativo do maior número da lista. Ao completar essa sequência de ordenações, obteremos uma lista ordenada da lista inicial. Entretanto, é necessário ter um algoritmo auxiliar para fazer a ordenação da lista em cada um desses passos.

Para que o algoritmo seja estável, ou seja, que não haja a troca da ordem original entre elementos iguais, é necessário um algoritmo estável para ordenar os dígitos em cada um dos passos do radix sort. Para resolver este problema é utilizado o merge sort nos passos auxiliares.

O objetivo deste projeto é provar a corretude dos algoritmos de merge sort e radix sort utilizando técnicas dedutivas da lógica de predicados por meio do assistente de demonstração PVS. Nos capítulos abaixo é possível verificar a intuição por trás de cada prova.

2 Questão 1: Merge sort gera uma permutação da lista L

O objetivo desta questão é provar que para toda lista L, o mergesort de L é uma permutação de L. Para isso, faremos indução no tamanho da lista L.

Para o caso em que a lista tenha tamanho menor ou igual a um é trivial de se provar, pois pela definição, o merge sort de uma lista desse tamanho retorna a própria lista. Portanto, as ocorrências de duas listas iguais são as mesmas. Para os demais casos, partimos da seguinte fórmula:

```
occurrences(merge_sort(l))(x) = occurrences(merge_sort(l))(x)
```

O intuito é alcançar a igualdade de que ocorrências de x em L são as mesmas das ocorrências de x do merge sort de L. Dado que o merge sort é um merge do prefixo e sufixo da lista L, obtém-se:

```
occurrences(merge(merge_sort(prefix(x ,floor(length(x)/2)))),
merge_sort(suffix(x ,floor(length(x)/2)))))(x)
= occurrences(merge_sort(l))(x)
```

Com o lema de que as ocorrências de um merge de duas listas quaisquer é a soma das ocorrências da primeira com as ocorrências da segunda, substitui-se na fórmula.

```
occurrences(merge_sort(prefix(x ,floor(length(x)/2))))(x) +
occurrences(merge_sort(suffix(x ,floor(length(x)/2))))(x)
= occurrences(merge_sort(l))(x)
```

A partir da nossa hipótese de indução, cuja definição garante que para toda lista y de tamanho menor que a lista x implica que as ocorrências do merge sort y são as mesmas que na lista y, assim, para a lista do prefixo de x e sufixo de x tem-se:

```
occurrences(merge_sort(prefix(x ,floor(length(x)/2))))(x) =
occurrences(prefix(x ,floor(length(x)/2)))(x)
```

```
occurrences(merge_sort(suffix(x ,floor(length(x)/2))))(x) =
occurrences(suffix(x ,floor(length(x)/2)))(x)
```

A partir dessas hipóteses de indução chegamos em:

```
occurrences(prefix(x ,floor(length(x)/2)))(x) +
occurrences(suffix(x ,floor(length(x)/2)))(x) =
occurrences(merge_sort(l))(x)
```

Por fim, utiliza-se o lema de que a soma de duas ocorrências é o mesmo da ocorrência do apêndice das duas listas, para enfim aplicar outro lema que diz que o apêndice do prefixo e sufixo da lista L é a própria lista L, resultando em:

```
occurrences(l)(x) = occurrences(merge_sort(l))(x)
```

Enfim, prova-se que o merge sort gera uma permutação da lista L.

3 Questão 2: Radix sort gera uma permutação da lista L

Nesta questão, o objetivo é provar que para toda lista L de um tipo T qualquer, temos que radixsort de L retorna uma permutação de L. Entretanto, sabe-se que o radixsort de L nada mais é do que um mergesort da pré-ordem « de outro mergesort de pré-ordem <= da lista L. Para provar, é necessário considerar que o mergesort de L gera uma permutação da lista L - provado na questão 1. Também, utilizando-se desse lema, é possível a aplicação para as duas possíveis pré-ordens « e <= - sendo duas ordenações distintas quaisquer.

A partir da definição de permutação, temos que para todo item x de uma lista L qualquer temos a mesma quantidade de ocorrências deste mesmo item x em outra lista L'. A partir disso, pode-se afirmar que as ocorrências de x em mergesort de L na pré-ordem <= são iguais as ocorrências de x na lista L (questão 1). Obtém-se a seguinte formula:

```
occurrences(merge_sort[T, <=](l))(x)
= occurrences(l)(x)
```

A partir do enunciado do problema e dado que a permutação é a ocorrência dos mesmos elementos de duas listas e a definição de radix sort, obtém-se a formula:

```
occurrences(merge_sort[T,<<](merge_sort[T, <=](l)))(x)
= occurrences(merge_sort[T, <=](l))(x)
```

Ao substituir a primeira fórmula na segunda obtém-se:

```
occurrences(merge_sort[T,<<](merge_sort[T, <=](l)))(x)
= occurrences(l)(x)
```

Dado que a definição de radix sort é um merge sort de um merge sort da lista L com as pré-ordem « e <=, respectivamente, conclui-se a prova.

```
occurrences(radixsort(l)))(x) = occurrences(l)(x)
```

4 Questão 3: Radix sort ordena

Aqui é necessário provar que radixsort está ordenado na ordem lexicográfica, ou seja, uma ordenação similar à do dicionário. Para iniciar essa prova, usaremos dois lemas: o lema de que mergesort é ordenado na ordenação « e o lema de que uma lista ordenada implica que os elementos seguem uma função monótona, ou seja, que todos os elementos estão seguindo uma mesma ordem, seja ela crescente ou decrescente.

A intuição por trás do uso dos lemas é deixar o antecedente com a mesma forma da fórmula do consequente. Com isso, temos um sequente da forma A implica B e outro sequente A no antecedente, podemos realizar uma simplificação e obter o sequente A e o sequente B apenas, eliminando a implicação.

O próximo passo realizado foi expandir a definição de radixsort está ordenado. Para fins de organização e legibilidade do relatório, foram substituidas as ocorrências de mergesort[T, «](mergesort[T, <=](l)) por radixsort(l) nas fórmulas.

```
FORALL (
j: below[length(radixsort(l))],
i: below[j]):
    nth(radixsort(l), i) << nth(radixsort(l), j)
|---------
FORALL (k: below[length(radixsort(l))]):
k <= length(radixsort(l)) - 2
=> lex(nth(radixsort(l), k), nth(radixsort(l), 1 + k))
```

Como tanto o antecedente e o consequente são fórmulas de ordenação, faremos a expansão das fórmulas até que se chegue em algo comum no consequente e antecedente. Para isso, na fórmula do consequente foi assumido um k qualquer, e para instanciado para j o valor de k + 1 e para i o valor de k, por fim expandimos a definição de lex. Lex nos diz que:

```
(x << y AND NOT (y << x))
 OR
(( x << y AND y << x) AND x <= y )
```

Em nossa prova, x e y são, respectivamente:

```
(nth(radixsort(l)), k + 1)
(nth(radixsort(l)), k)
```

Queremos dividir essa prova em dois casos, um no qual x « y é verdadeiro e outro no qual x « y é falso. Com isso, teremos uma ramificação da árvore com x « y no antecedente e outra ramificação x « y no consequente.

A ramificação com x « y no consequente pode ser resolvida com uma manipulação de fórmulas. Considere x « y como A e y « x como A^{-1}, além disso, considere y <= x como B. Em nosso consequente, temos:

NOT (A^{-1}) OR A OR ((A^{-1}) AND NOT A) OR (A^{-1} AND A AND B)

Manipulando esta fórmula com as propriedades distributivas temos:

((NOT (A^{-1}) OR (A^{-1}) OR A) AND (NOT (A^{-1}) OR A OR NOT A)) OR (A^{-1} AND A AND B)

O que nos gera uma tautologia no consequente. Passando esta tautologia para o antecedente, temos um absurdo que nos leva ao axioma e conclui este ramo da prova.

Já no outro ramo da árvore temos x « y no antecedente, realizando manipulações e simplificações de fórmula conseguimos obter B no consequente como única fórmula. Nosso objetivo é conseguir fazer que B apareça no antecedente e para isso aconteça usaremos o lema que diz que mergesort é conservativo, pois esse lema implica em uma fórmula que pode simplificar B. Além disso, temos x « y e y « x na esquerda dessa implicação do lema e temos essas mesmas fórmulas no antecedente. Simplificando o antecedente, obtemos a fórmula abaixo.

```
EXISTS (i, j: below[length(merge_sort[T, <=](l))]):
  i < j AND
   nth(merge_sort(merge_sort[T, <=](l)), k) =
    nth(merge_sort[T, <=](l), i)
    AND
    nth(merge_sort(merge_sort[T, <=](l)), 1 + k) =
     nth(merge_sort[T, <=](l), j)
```

Tomando i e j como valores que satisfazem a fórmula, temos as igualdade que necessitamos para simplificar a fórmula B. Reescrevendo B temos:

```
nth(merge_sort[T, <=](l), i) <=
nth(merge_sort[T, <=](l), j)
```

Nesta parte da prova, usaremos o lema que diz que lista ordenada implica em função monótona e o lema que diz que mergesort é ordenado. Com a fusão desses dois lemas, utilizando a ordem $<=$ em mergesort, temos apenas a função monótona, da seguinte maneira.

```
FORALL (
j: below[length(merge_sort[T, <=](l))],
i: below[j]):
nth(merge_sort[T, <=](l), i) <=
nth(merge_sort[T, <=](l), j)
```

Que é uma generalização de B no consequente. Tomando i e j como valores quaisquer temos um axioma e assim concluímos a prova da questão 3.