Universidade de Brasília

Relatório:

Formalização de Propriedades do Algoritmo Radix Sort

Grupo:

Leonardo Ribas do Nascimento Mat: 17/0038963
Amanda Augusto da Silva Mat: 18/0012053

Professor:
Flávio L. C. de Moura

June 17, 2019

1 Introdução

O projeto desenvolvido ao longo da disciplina de Lógica Computacional 1 visa a demonstração da correção do algoritmo de ordenação Radix Sort. Tal demonstração consiste na formalização de três propriedades do algoritmo utilizando o assistente de provas PVS, assim como técnincas dedutivas da lógica de predicados.

2 Especificação do Problema

O Radix sort utilizado no projeto ordena números inteiros, partindo da ordenação do dígito menos significativo ao mais significativo utilizando o merge sort como algoritmo auxiliar.

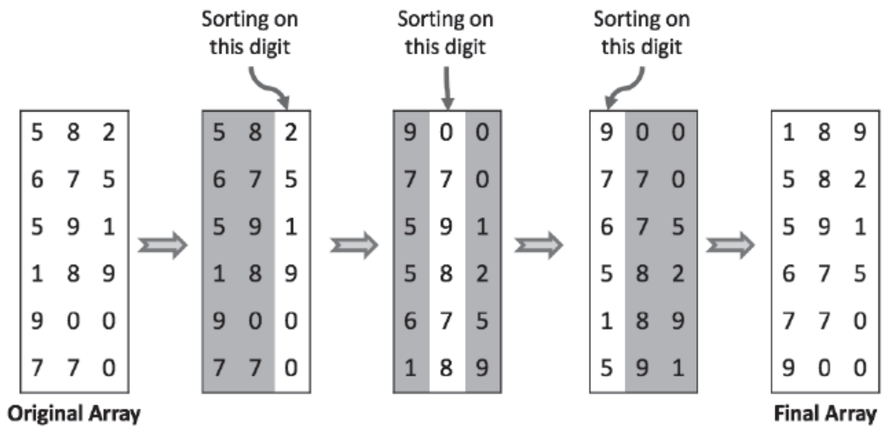

Figure 1: Funcionamento do Merge Sort

A primeira questão, consiste em demonstrar que se uma lista está ordenada até o d-ésimo dígito, e se aplica radixsort entre os dígitos d e k, então ficará ordenada até o dígito k + 1.

A segunda questão, está relacionada com o fato de que a função radixsort preserva os elementos das listas dadas como argumento

Finalmente, a terceira questão é demonstrar que função radixsort gera uma lista ordenada

3 Descrição das soluções e formalizações

3.1 Questão 1

Queremos provar a seguinte propriedade:

```
FORALL(l : list[nat], k : nat, d:nat | d <= k) :
    is_sorted_ud?(l,d) =>
       is_sorted_ud?(radixsort(l, k, d), k+1)
```

Iniciamos a prova com indução forte no intervalo k-d e obtemos como hipótese de indução que a propriedade vale para qualquer intervalo menor que k-d. Ficamos então com o seguinte sequente:

```
[-1]  FORALL (y_1: nat, y_2: d: nat | d <= y_1):
       FORALL (l: list[nat]):
         y_1 - y_2 < x!1 - x!2 IMPLIES
          is_sorted_ud?(l, y_2) =>
           is_sorted_ud?(radixsort(l, y_1, y_2), y_1 + 1)
[-2]  is_sorted_ud?(l, x!2)
 |-------
[1]   is_sorted_ud?(radixsort(l, x!1, x!2), x!1 + 1)
```

Aplicamos a definição de radixsort na formula 1, e dividiremos nossa prova em 2 subprovas de acordo com a implementação do próprio radixsort. A primeira subprova assume que k e d são iguais (PVS os denomina x!2 x!1 respectivamente). Pela definição do radixsort quando k e d são iguais é retornado apenas um merge_sort(l, d), ficamos com o seguinte sequente:

```
[-1]  x!2 = x!1
[-2]  is_sorted_ud?(l, x!2)
 |-------
[1]   is_sorted_ud?(merge_sort(l, x!2), 1 + x!1)
```

Essa subprova é fechada utilizando-se o lema "merge_sort_d_sorts" que afirma que qualquer lista ordenada pelo merge-sort até o d-ésimo digito tambem está ordenada até o d-ésimo digito + 1. Na segunda subprova temos que $k \neq d$ e pela definição do radixsort temos que provar:

```
is_sorted_ud?(radixsort(merge_sort(l, x!2), x!1, 1 + x!2), 1 + x!1)
```

Temos por hipótese de indução que para qualquer lista e qualquer intervalo menor que k-d a propriedade vale, portantao podemos instanciamor nossa hipótese como se segue:

- l recebe merge_sort(l, x!2),

- y_2 recebe 1+x!2,

- y_1 recebe x!1,

- pois $x!1 - (1 + x!2) < x!1 - x!2$.

Ficamos com o seguinte sequente:

```
 is_sorted_ud?(merge_sort(l, x!2), 1 + x!2) =>
        is_sorted_ud?(radixsort(merge_sort(l, x!2), x!1, 1 + x!2), x!1 +
```

Portanto se provarmos o antecedente is_sorted_ud?(merge_sort(l,x!2), 1 + x!2) fecharemos a prova. E provamos isso da mesma maneira que provamos a primeira subprova do problema.

3.2 Questão 2

Queremos provar a seguinte propriedade:

```
FORALL(l : list[nat], k : nat, d : nat | d <=k):
  permutations(l, radixsort(l,k,d))
```

Assim como na propriedade anterior, aplicamos indução forte no intervalo k-d, e dividiremos nossa prova em duas subprovas: caso $k = d$ e caso $k \neq d$.

Caso $k = d$:
Por definição, quando $k = d$ radixsort retorna merge_sort(l, x!2), e portanto temos que provar:

```
permutations(l, merge_sort(l, x!2))
```

Provamos isso a partir do lema "merge_sort_permutes" que afirma que o merge_sort preserva os elementos da lista.

Caso $k \neq d$:
Instanciamos nossa hipótese de indução da mesma maneira que instanciamos na questão anterior e obtemos:

```
-1  permutations(merge_sort(l, x!2),
                   radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
  |-------
[1]   x!2 = x!1
[2]   permutations(l, radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
```

Sabemos que pelo lemma "merge_sort_permutes" merge_sort(l, x!2) é uma permutação de l.

```
-1  permutations(l, merge_sort(l, x!2))
[-2]  permutations(merge_sort(l, x!2),
                   radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
```

```
  |-------
[1]   x!2 = x!1
[2]   permutations(l, radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
```

E pelo lema "permutations_is_transitive" temos que se l permuta com merge_sort(l, x!2) e que merge_sort(l, x!2) permuta com radixsort(merge_sort(l, x!2), x!1, 1+x!2) então l permuta com radixsort(merge_sort(l, x!2), x!1, 1 + x!2).

```
-1  permutations(l, merge_sort(l, x!2))
[-2]  permutations(merge_sort(l, x!2),
                    radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
  |-------
[1]   x!2 = x!1
[2]   permutations(l, radixsort(merge_sort(l, x!2), x!1, 1 + x!2))
```

Assim fechamos a prova da segunda propriedade.

4 Conclusão

Com esse projeto tivemos a oportunidade de colocar em prática os fundamentos explorados em sala de aula na disciplina de Lógica Computacional 1, completando a formalização da carreção do algoritmo radix sort. Alem de revisitarmos conceitos de indução, utilizamos conceitos da lógica dedutiva e lógica de predicados.

As dificuladades em relação ao projeto giram em torno, principalmente, da ferramenta PVS, que a principio não dialoga diretamente com os conceitos vistos em sala. É necessário um certo período de adaptação até que se domine os conceitos básicos da ferramenta assim como a propria linguagem utilizada no PVS.

No geral a experiência em relação ao projeto foi boa, embora não tenhamos conseguido completar a questão 3, finalizar e formalizar propriedades num ambiente que garante a correção de tais provas é gratificante.

5 Referências

M. Ayala-Rincón and F. L. C. de Moura. Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs. Undergraduate Topics in Computer Science. Springer, 2017