Universidade de Brasília

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Lógica Computacional 1

Relatório Radix Sort

Diego Vaz Fernandes 16/0117925
Gabriel dos Santos Martins 15/0126298

17 de Junho de 2019

1 Introdução

Atualmente algoritmos são comumente usados na solução de problemas do nosso dia a dia, sendo então crucial termos certeza de que esse algoritmo está correto e funciona como deveria. Sendo assim, após a elaboração de um algoritmo, é importante mostrar que o mesmo está funcionando corretamente. Existem várias formas de se testar um algoritmo, como por exemplo testes unitários de software ou testes de integração, porem é praticamente impossível testar todas as possibilidades e variações de um software, e testes convencionais não garantem uma certeza absoluta sobre o funcionamento do software testado. Então em sistemas críticos, que são sistemas que precisam funcionar perfeitamente, temos que ter uma maneira melhor de garantir esse funcionamento. Por isso, tem se a necessidade de mostrar que as propriedades de um algoritmo valem para qualquer valor. Neste relatório, iremos provar algumas propriedades do algoritmo de ordenação **Radix Sort**, utilizando a linguagem do assistente de demonstração PVS (*Specification and Verification System*), que é um software que auxilia na construção de provas formais.

2 Contextualização do Problema

O Radix sort é um algoritmo de ordenação rápida e estável que pode ser usado para ordenar chaves únicas.Cada chave é uma cadeia de caracteres ou cadeia de números, e o radix sort ordena estas chaves em qualquer ordem relacionada com a lexicografia. Um algoritmo é dito estável se a posição relativa de dois elementos iguais permanece inalterada durante o processo de ordenação.

Existem duas classificações para o Radix Sort: LSD (*Least significant digit – Dígito menos significativo*) e MSD (*Most significant digit – Dígito mais significativo*).

Como já dito anteriormente o Radix Sort ordena chaves em uma ordem qualquer utilizando como critério a lexicografia e para tal ordenação é preciso de um algortimo auxiliar pra realizar a ordenação propriamente dita. como queriamos ter um algortimo estável escolhemos o Merge Sort para isso. Ele que tem seu famoso processo de dividir para conquistar. Sua ideia principal consiste na divisão de um problema em vários partes menores e resolver essas partes menores dividindo-as novamente em mais partes menores e repetindo esse processo até a divisão não ser mais possível sendo esse um processo dito recursivo. Então no nosso problema, o papel do merge sorte é ordenar as partes das listas que forem enviadas pra ele seguindo esse processo descrito acima

3 Explicação das Soluçoes

3.1 Questão 1

Na questão 1, tinhamos que provar que a "conjecture"mostrada na figura 1 era verdeira.

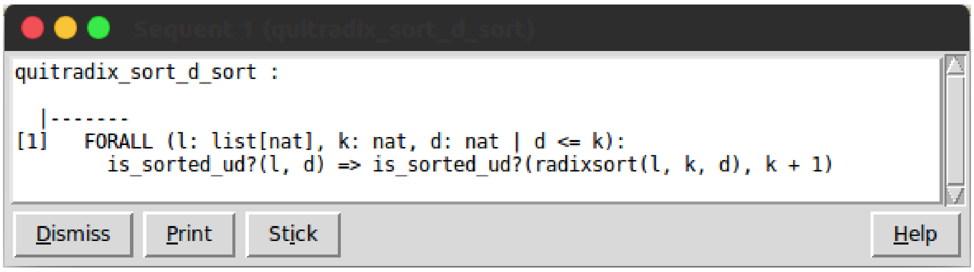


Figura 1: Conjecture da questão 1.

Começamos a prova utilizando **(measure-induct+ "n" ("n"))**, como foi pedido na questão, pois se tratava de indução forte. O measure-induct nos deu a hipótese mostrada na figura 2.

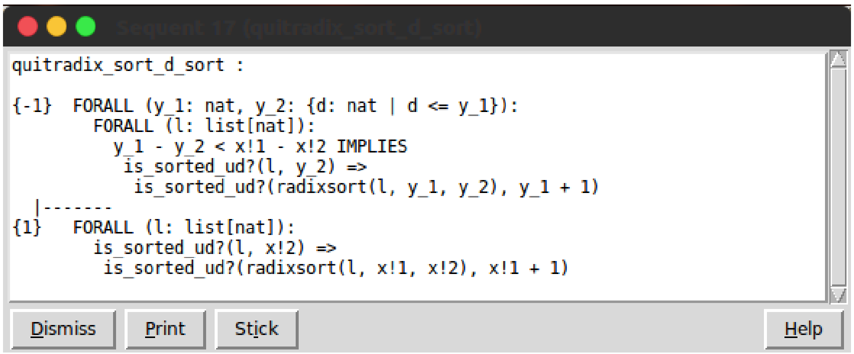


Figura 2: Hipótese fornecida.

Utilizamos o **(skeep)** para instanciar as variáveis presentes no consequente. Logo após, utilizamos o **(expand radixsort 1)** para aplicar a definição da função radixsort no sequente 1. Então obtvemos o seguinte:

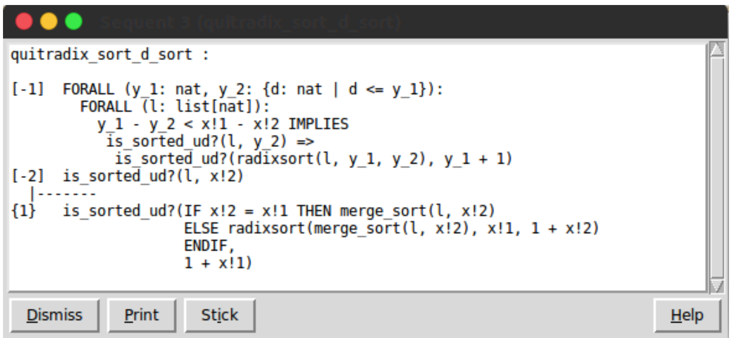
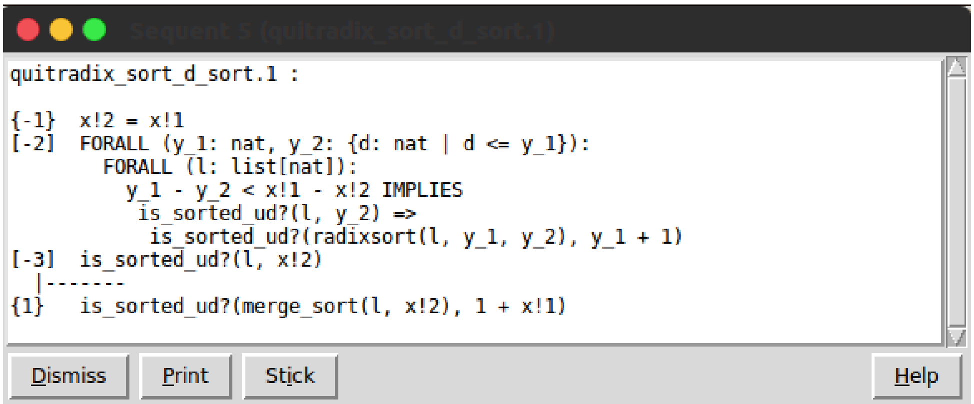


Figura 3: Expand no radisort.

Como radixsort é uma função recursiva, ao aplicar a sua definição, trazemos a condição de parada, o caso de x!1 = x!2, e o seu caso recursivo. Então para provar essa propriedade, precisamo provar esses dois casos. Para isso, aplicamos a regra **lift-if** para definir e separar cada caso da função. Após aplicar essa regra, podemos então usar a regra **prop**. Ela quebra cada um dos casos do *IF-THEN-ELSE* em um sequente. Deste modo, conseguimos provar cada um deles. Ao quebrar os casos, a árvore é dividida em duas Sub-árvores. A primeira está definida a seguir:



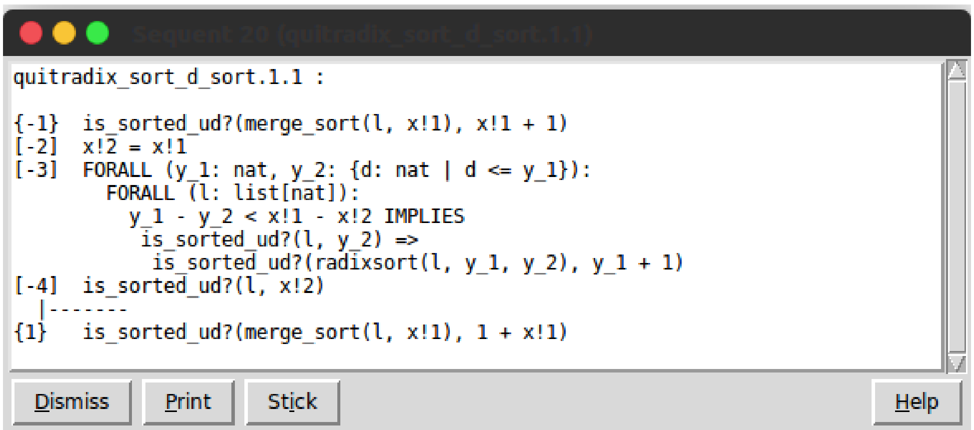
```
quitradix_sort_d_sort.1 :

{-1}  x!2 = x!1
[-2]  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)
[-3]  is_sorted_ud?(l, x!2)
  |-------
{1}   is_sorted_ud?(merge_sort(l, x!2), 1 + x!1)
```


Figura 3: Sub-árvore 1 - Caso em que x!1=x!2.

Neste ramo da árvore, queremos provar que uma lista continua sendo ordenada até o d-ésimo digito +1. Então usamos o comando **lemma merge_sort_d_sorts** para trazer o seguinte lemma:

```
merge_sort_d_sorts : LEMMA
    FORALL(l : list[nat], d : nat) :
      is_sorted_ud?(l,d)  =>
      is_sorted_ud?(merge_sort(l,d),d+1)
```

Este lemma nos garante que se uma lista está ordenda até o d-ésimo digito implica que ela também estará ordenada para o d-ésimo digito + 1. Justamente o que queriamos provar, não ?! Então bastou usarmos o comando **inst? -1** para eliminarmos o o *FORALL* do sequente -1, usar o comando **split** para eliminar a implicação a esquerda e depois fazer um **replace -2 (-1 1)** para que todos os x!2 fossem substituidos por x!1, ja que estamos no caso que os dois são iguais. Deste modo, ficamos com o mesmo argumento na esquerda e na direita da prova, como mostrado a seguir:



```
quitradix_sort_d_sort.1.1 :

{-1}  is_sorted_ud?(merge_sort(l, x!1), x!1 + 1)
[-2]  x!2 = x!1
[-3]  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)
[-4]  is_sorted_ud?(l, x!2)
  |-------
{1}   is_sorted_ud?(merge_sort(l, x!1), 1 + x!1)
```


Figura 3: Sequentes -1 e 1 iguais.

Quando chegamos neste ponto em uma prova na Lógica de Primeira Ordem, quer dizer que terminamos a prova naquele ramo da árvore, pois exite um axioma que fala que se você possui uma variável no lado esquerdo do sequente e a mesma variável no lado direito do sequente, então você fechou sua prova. Que é o nosso caso aqui.

Então com isso, conseguimos fechar a primeira sub-árvore. Já a segunda sub-árvore, que é o caso de x!1 != x!2, está representada a seguir:

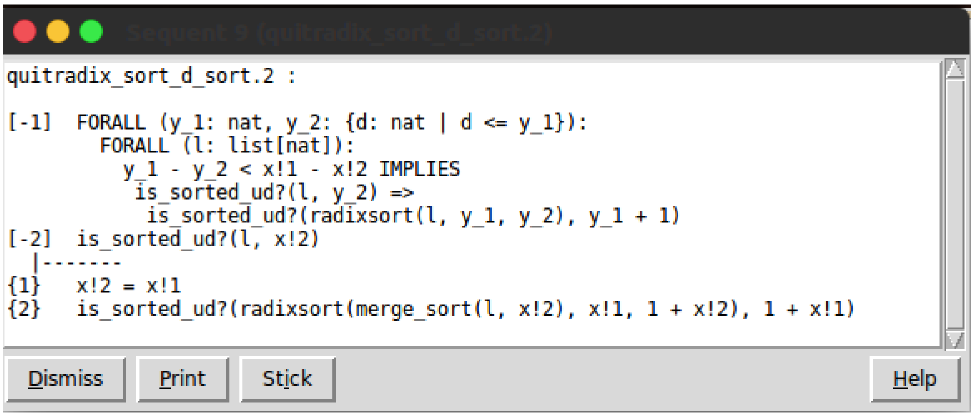


Figura 3: Caso que x!1 != x!2.

Agora iremos utilizar a hipotese que o questão nos dá. Para isso, utilizamos o comando **inst?** para instaciar os *FORALL* do sequente -1 com as variáveis que estamos utilizando no sequente que queremos provar, vulgo o sequente 2. Após isso, utilizamos o comando **split** para quebrar a implicação, e nisso ficamos com 3 sub-árvores. A primeira delas bastou usar o comando **assert** para fechar, visto que tinhamos as condições necessárias para a aplicação do axioma descrito anteriormente. Já a segunda sub-afvore, repetimos o mesmo processo para provar o caso em que *x!1 = x!2*. Já a terceira sub-árvore é um dilema. Tentamos provar ela e não conseguimos, porém, ao dar o comando **assert**, a prova fechava.

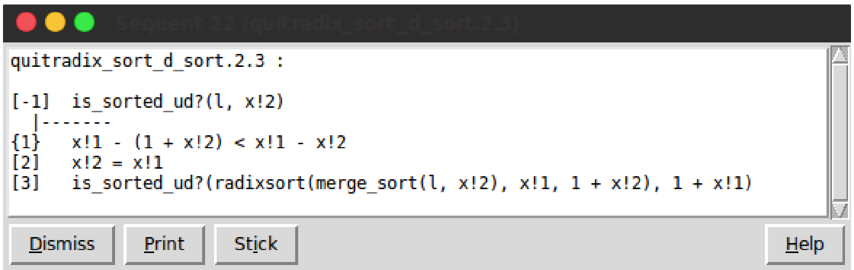
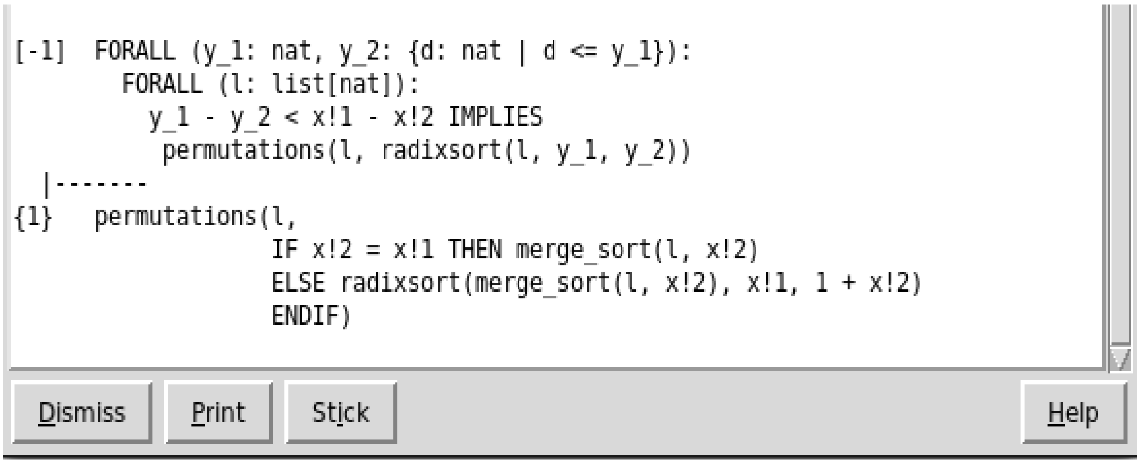


Figura 3: Caso que fechou com um assert.

3.2 Questão 2

Na questão 2, queremos provar o algoritmo onde seja l uma lista de naturais, k e d naturais tal que d <= k, onde essa lista será ordenado de d até k utilizando o algoritmo *merge_sort* dentro do *radixsort*. Para todas essas propriedades, temos que uma lista l com essa mesma lista l depois de aplicado o radixsort são iguais através da permutations. Começamos a prova utilizando o measure induct como pedido no enunciado da questão no PVS. Utilizamos o (skeep) para remover o FORALL que tínhamos na parte de baixo da nossa prova, e então foi utilizado o comando (expand "radixsort" 1) para termos mais opções para trabalhar na prova.

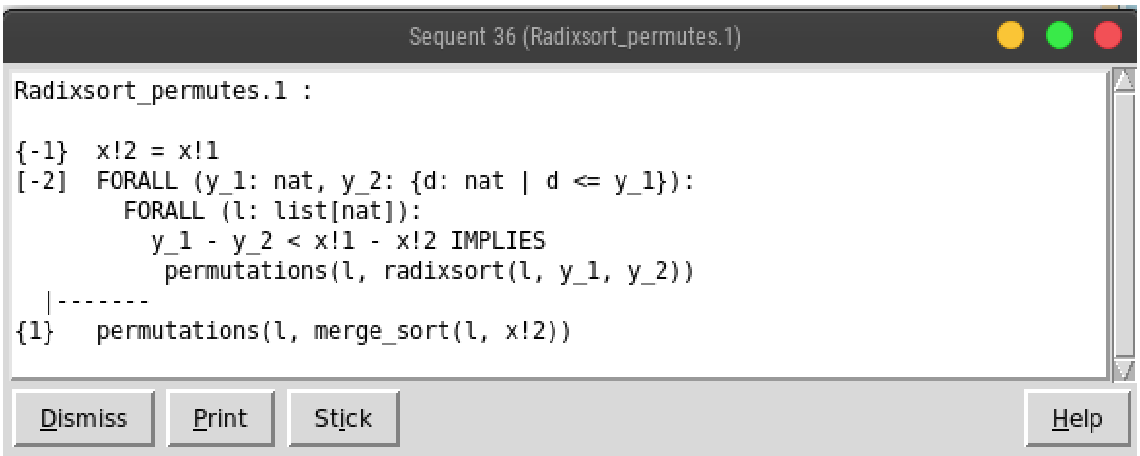


```
[-1]  FORALL (y_1: nat, y_2: {d: nat | d <= y_1}):
        FORALL (l: list[nat]):
          y_1 - y_2 < x!1 - x!2 IMPLIES
           permutations(l, radixsort(l, y_1, y_2))
  |-------
{1}   permutations(l,
                   IF x!2 = x!1 THEN merge_sort(l, x!2)
                   ELSE radixsort(merge_sort(l, x!2), x!1, 1 + x!2)
                   ENDIF)
```

Dismiss Print Stick Help

Figura 11

Após isso foi temos um IF ELSE e então utilizamos o comando (lift-if) e logo em seguida (prop) sendo o primeiro para propagarmos para dentro do IF os termos que ficaram fora e (prop) para quebrar o IF nos casos possíveis. Com esse (prop) nossa árvore se abriu em 2 ramos distintos.



Sequent 36 (Radixsort_permutes.1)

```
Radixsort_permutes.1 :

{-1}  x!2 = x!1
[-2]  FORALL (y_1: nat, y_2: {d: nat | d <= y_1}):
        FORALL (l: list[nat]):
          y_1 - y_2 < x!1 - x!2 IMPLIES
           permutations(l, radixsort(l, y_1, y_2))
  |-------
{1}   permutations(l, merge_sort(l, x!2))
```

Dismiss Print Stick Help

Ramo 1 da Árvore

O primeiro deles, foi preciso trazer o lemma "merge_sort_permutes" que diz se o merge_sort aplicado em uma lista l, com essa mesma lista sem o merge_sort, continua a mesma.

merge_sort_permutes : LEMMA
FORALL(l : list[nat], d : nat):
permutations
(l,merge_sort(l,d))

Quando trouxemos esse lemma, ficamos com duas coisas parecidas em cima e em baixo, porém com nome de variáveis diferentes, e foi precisa utilizar um (inst? -1) para instanciar essas variáveis com mesmo nome, e assim o PVS reconhecer a mesma coisas e fecha a prova por ser um AXIOMA.

```
Sequent 1 (Radixsort_permutes.1)

Radixsort_permutes.1 :

{-1}  FORALL (l: list[nat], d: nat): permutations(l, merge_sort(l, d))
[-2]  x!2 = x!1
[-3]  FORALL (y_1: nat, y_2: {d: nat | d <= y_1}):
        FORALL (l: list[nat]):
          y_1 - y_2 < x!1 - x!2 IMPLIES
           permutations(l, radixsort(l, y_1, y_2))
  |-------
[1]   permutations(l, merge_sort(l, x!2))

Dismiss  Print  Stick  Help
```

Ramo final, antes do instanciar

Já do outro lado da árvore (lado direito), começamos utilizando o (inst? -1) para instanciar as variáveis logo no inicio, e em seguida trouxemos o mesmo lemma utilizado no ramo da árvore já citado acima (lado esquerdo). Após trazer esse lema, quebramos o IMPLIES utilizando o comando (split), quebrando nossa árvore em outros 2 ramos.

```
Sequent 2 (Radixsort_permutes.2)

Radixsort_permutes.2 :

{-1}  FORALL (l: list[nat], d: nat): permutations(l, merge_sort(l, d))
[-2]  x!1 - (1 + x!2) < x!1 - x!2 IMPLIES
       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))

Dismiss  Print  Stick  Help
```

Antes de ser aplicado o split

O primeiro deles (lado esquerdo), começamos instanciando as variáveis na linha -2 utilizando o comando inst. Em seguida, trouxemos o lemma permutations_is_transitive que diz que a transitividade entre a permutação de 3 listas l, l1 e l2 é verdadeira.

permutations_is_transitive : LEMMA
permutations(l,l1) AND permutations (l1,l2) IMPLIES permutations (l,l2)

Em seguida instanciamos na linha -1 como segue abaixo:

(inst -1 "lmerge_sort(l, x!2)"
"radixsort(merge_sort(l, x!2), x!1, 1 + x!2)")

Pois assim teríamos um axioma no próximo passo, por ter a mesma coisa em cima e em baixo, fechando a prova com um (assert).

4 Conclusões

Ao fim das resoluções das questões 1 e 2 do projeto, tivemos uma importante experiência de como que é feita uma prova formal de um algoritmo suas dificuldades sua importância seus resultados e como deve ser utilizada. Para esse projeto utilizamos o PVS que se mostrou uma ferramenta bastante poderosa na elaboração de provas formais de um algoritmo. Uma das principais dificuldades foi não estar totalmente familiarizado com a sintaxe e com os códigos em LISP, e além disso a interface e usabilidade do pvs não ser muito boa nem intuitiva, mas mesmo com todas essas adversidades foi possível compreender a importância de uma ferramenta que auxilie em provas formais de algoritmos, pois através dela conseguimos assegurar que uma prova está correta e que não estamos utilizando uma regra ou propriedade de maneira leviana através das regras do cálculo de Gentzen. E utilizando indução estrutural, foi possível a prova das questões já vistas neste documento.

5 Lista de referências

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