Projeto e Análise de Algoritmos (2025-1)

Flávio L. C. de Moura*

24 de março de 2025

Introdução

O foco deste curso é na construção/design/projeto e na análise de algoritmos. Para análisar um algoritmo precisaremos de um ferramental matemático que nos permita, em um certo sentido, medir a qualidade dos algoritmos. Podemos dizer que a qualidade de um algoritmo está associada a dois aspectos fundamentais:

1. **Correção**: Um algoritmo deve funcionar corretamente sempre, caso contrário, terá pouca ou nenhuma utilidade.

2. **Eficiência**: A eficiência de um se refere aos recursos computacionais utilizados durante a sua execução que são subdivididos em dois tópicos:

   (a) **Eficiência de tempo**: Em linhas gerais, consiste na mensuração do tempo de execução do algoritmo. Esta análise requer diversos cuidados de forma que os resultados possam ser utilizados para comparar algoritmos distintos que resolvem o mesmo problema.

   (b) **Eficiência de espaço**: Consiste na mensuração da memória requerida durante a execução do algoritmo.

No contexto de algoritmos e desenvolvimento de *software* é comum a utilização de testes como método de validação. Ou seja, o programa (ou *software*) é executado com diversas entradas distintas, e se nenhum problema é encontrado, o programa é considerado bom o suficiente para ser utilizado. De fato, a primeira coisa que fazemos após implementar um algoritmo é testá-lo para diversas entradas, e caso alguma resposta seja incorreta, uma revisão da implementação é feita para corrigir o erro, e então novos testes são realizados. Este processo é repetido até que o programador sinta confiança na implementação, mas depois de todos estes testes é possível dizer que o programa é correto? Certamente não! Pensando no caso particular da implementação de um algoritmo de

*flaviomoura@unb.br

ordenação de naturais ou inteiros (ou qualquer estrutura munida de uma ordem total), sabemos que existe uma infinidade de listas de inteiros que podem ser utilizadas nos testes, e portanto não é possível testar todas elas. Em se tratando de programas utilizados em sistemas críticos (aviação, medicina, sistemas bancários, etc), por menores que sejam as chances de erros, falhas não são toleradas. O que fazer então para garantir a correção de um programa? Uma abordagem possível consiste em **provar** a correção do programa! Uma prova de uma propriedade de um programa fornece a garantia de que o programa satisfaz a propriedade provada **sempre**! A importância deste processo pode ser exemplificada pelos casos listados a seguir:

1. **Therac-25**: Uma máquina de radioterapia controlada por computador causou a morte de pelo menos 6 pacientes entre 1985 e 1987 por overdose de radiação.

2. **Pentium FDIV**: Um erro na construção da unidade de ponto flutuante do processador Pentium da Intel causou um prejuízo de aproximadamente 500 milhões de dólares para a empresa que se viu forçada a substituir os processadores que já estavam no mercado em 1994.

3. **Ariane 5**: Um foguete que custou aproximadamente 7 bilhões de dólares para ser construído explodiu no seu primeiro voo em 1996 devido ao reuso sem verificação apropriada de partes do código do seu predecessor.

A correção de algoritmos

Como veremos, a prova de que um algoritmo é correto nem sempre é uma tarefa fácil. No caso de algoritmos recursivos, as provas costumam ser mais simples porque podemos utilizar indução. Os exemplos a seguir são definidos sobre a estrutura de listas (encadeadas) que são definidas pela seguinte gramática recursiva:

$$l ::= nil \mid h :: l$$

onde nil representa a lista vazia, e $h :: l$ representa a lista que tem h como primeiro elemento, e cauda l. A seguir, definimos algumas funções (algoritmos) sobre a estrutura de listas.

O comprimento de uma lista, isto é, o número de elementos que a lista possui, é definido recursivamente por:

$$|l| = \begin{cases} 0, & \text{se } l = nil \\ 1 + |l'|, & \text{se } l = a :: l' \end{cases}$$

Uma operação importante que nos permite construir uma nova lista a partir de duas listas já construídas é a concatenação. Podemos definir a concatenação de duas listas por meio da seguinte função recursiva:

$$l_1 \circ l_2 = \begin{cases} l_2, & \text{se } l_1 = nil \\ a :: (l' \circ l_2), & \text{se } l_1 = a :: l' \end{cases}$$

Por fim, o reverso de uma lista é definido recursivamente por:

$$rev(l) = \begin{cases} l, & \text{se } l = nil \\ (rev(l')) \circ (a :: nil), & \text{se } l = a :: l' \end{cases}$$

Os exercícios a seguir expressam diversas propriedades envolvendo estas operações. Resolva cada um deles utilizando indução.

1. Prove que $|l_1 \circ l_2| = |l_1| + |l_2|$, quaisquer que sejam as listas l_1, l_2.

2. Prove que $l \circ nil = l$, qualquer que seja a lista l.

3. Prove que a concatenação de listas é associativa, isto é, $(l_1 \circ l_2) \circ l_3) = l_1 \circ (l_2 \circ l_3)$ quaisquer que sejam as listas l_1, l_2 e l_3.

4. Prove que $|rev(l)| = |l|$, qualquer que seja a lista l.

5. Prove que $rev(l_1 \circ l_2) = (rev(l_2)) \circ (rev(l_1))$, quaisquer que sejam as listas l_1, l_2.

6. Prove que $rev(rev(l)) = l$, qualquer que seja a lista l.