Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

1 Introdução e Motivação

O objetivo deste curso é o estudo de técnicas para analisar e projetar de algoritmos. As técnicas utilizadas para analisar algoritmos, nos permitirão estudar o tempo e o espaço demandado pelo algoritmo para processar uma determinada entrada (complexidade de tempo e de espaço). As técnicas de projeto de algoritmos estão relacionadas ao trabalho da construção de algoritmos a partir de um dado problema. Neste contexto, a técnica de divisão e conquista resolve um problema a partir da sua divisão em problemas menores seguida de uma combinação adequada das soluções dos subproblemas para assim construir uma solução do problema original. Outras técnicas que serão estudadas incluem a chamada *programação dinâmica* e os *algoritmos gulosos*. Uma vez construído, a primeira preocupação que precisamos ter em relação a um algoritmo é se ele funciona da forma esperada. Em outras palavras, antes de qualquer coisa precisamos saber se o algoritmo é correto, e só depois faremos a análise da sua complexidade. Intuitivamente, dizemos que um algoritmo A é correto quando sempre fornece respostas corretas para qualquer entrada possível. Por exemplo, se A é um algoritmo de ordenação de inteiros, então espera-se que para qualquer lista de inteiros l, A retorne uma permutação de l que esteja ordenada. Isto significa que as respostas dadas pelo algoritmo são corretas para todas as entradas possíveis, e que estas respostas são geradas em tempo finito.

Uma ferramenta que será utilizada frequentemente nas provas de correção de algoritmos é a indução matemática. Ao analisarmos a eficiência (ou complexidade) dos algoritmos também faremos o uso de diversas ferramentas matemáticas (somatórios, conjuntos, funções, matrizes, etc). O apêndice VIII do livro [1, 2] pode ser usado para revisar estes temas.

Os computadores e seus algoritmos estão presentes na maioria das atividades quotidianas, utilizamos computadores para fazer compras, transações bancárias, etc. Os carros, aviões e equipamentos hospitalares modernos também possuem diversos sistemas embarcados. À medida que estes equipamentos se tornam mais comuns em nosso dia a dia, aumenta também o seu poder de processamento e de armazenamento. Diante desta realidade é natural perguntar: por que analisar algoritmos? Inicialmente porque precisamos garantir que são corretos. Adicionalmente, precisamos analisar a eficiência destes algoritmos. Mas não poderíamos simplesmente migrar para um computador mais potente e com mais capacidade de armazenamento quando fosse necessário? A resposta é não. Ainda que tivéssemos uma capacidade infinita de processamento e/ou de armazenamento, a análise da eficiência seria necessária. De fato, não faz sentido ter que esperar horas para a finalização de um processamento se for possível fazê-lo de forma mais eficiente em apenas alguns segundos, ou utilizar uma quantidade gigantesca de memória sem necessidade. Mais ainda, é a análise da complexidade quem vai nos mostrar como podemos aproveitar melhor a capacidade de processamento de um computador como veremos ao longo do curso.

Como primeiro exemplo, considere o problema de ordenar uma lista de inteiros. Existem diversos algoritmos que resolvem este problema, e uma abordagem simples consiste em inicialmente inserir elementos em uma lista ordenada. A função $insert\ x\ l$ definida a seguir, insere o elemento x na lista l:

$$insert\ x\ l := \begin{cases} x :: nil, & \text{se } l = nil \\ x :: l, & \text{se } x \leq h \text{ e } l = h :: tl \\ h :: (insert\ x\ tl), & \text{se } x > h \text{ e } l = h :: tl \end{cases}$$

A inserção é feita de forma que se l está ordenada então a lista resultante também está ordenada. Podemos enunciar esta propriedade como um lema:

Lema 1.1. *Se l é uma lista ordenada e x é um inteiro, então $(insert\ x\ l)$ é uma lista ordenada.*

*flaviomoura@unb.br

Exercício 1.2. *Prove o Lema 1.1.*

Na aula de *[2024-03-18 seg]*, iniciamos a prova do Lema 1.1 utilizando indução na estrutura da lista l. A base da indução corresponde ao caso em que a lista l é vazia. Neste caso, temos que provar que $insert\ x\ nil$ é uma lista ordenada, assumindo que a lista vazia está ordenada. Mas isto é trivial porque, pela definição de *insert*, $insert\ x\ nil$ é a lista unitária $x :: nil$ que está ordenada por definição.
O passo indutivo ocorre quando l tem a forma $h :: tl$. Neste caso, temos que provar que $insert\ x\ (h :: tl)$ é uma lista ordenada, assumindo que a lista $h :: tl$ está ordenada. A definição de *insert* nos dá dois subcasos:

1. $x \leq h$: Neste caso, $insert\ x(h :: tl)$ retorna $x :: h :: tl$ que está ordenada porque o elemento inserido x é menor ou igual ao primeiro elemento de uma lista ordenada.
2. $x > h$: Na aula de *[2024-03-20 qua]*, iniciamos com a escrita da hipótese de indução:

   > Se tl é uma lista ordenada e x é um inteiro, então $(insert\ x\ tl)$ é uma lista ordenada.

Temos como hipótese (do problema) que $h :: tl$ é uma lista ordenada, e queremos mostrar que a lista $insert\ x\ (h :: tl)$ é uma lista ordenada. Pela definição da função *insert*, temos que $insert\ x\ (h :: tl) = h :: (insert\ x\ tl)$. Agora observe que, por hipótese de indução, a lista $insert\ x\ tl$ está ordenada. Além disto, temos que $h < x$ e h é menor ou igual a todo elemento da lista tl já que $h :: tl$ está ordenada. Então h é menor ou igual a todo elemento de $(insert\ x\ tl)$, e portanto a lista $h :: (insert\ x\ tl)$ está ordenada. □

Agora vamos refazer esta prova mecanicamente utilizando o assistente de provas Coq. Primeiramente, vamos definir o predicado que caracteriza uma lista ordenada.

```
Inductive ordenada : list nat -> Prop :=
  | nil_ord : ordenada nil
  | one_ord : forall n:nat, ordenada (n::nil)
  | cons_ord : forall (n m:nat) (l:list nat), n <= m ->
               ordenada (m::l) -> ordenada (n::m::l).
```

A função *insert* pode ser definida em Coq como segue:

```
Fixpoint insert (x:nat) (l:list nat) : list nat :=
  match l with
  | nil => x::nil
  | h::tl => if x <=? h then x::l else h::(insert x tl)
  end.
```

Agora podemos enunciar e provar o lema 1.1 em Coq:

```
Lemma insert_ord : forall (x:nat) (l:list nat), ordenada l -> ordenada (insert x l).
```

Exercício 1.3. *Prove o lema* `insert_ord` *no Coq.*

A resolução do exercício anterior nos dá uma boa ideia do que consiste o processo de formalização (ou mecanização) de um algoritmo, ainda que bastante simples. A função principal do algoritmo de ordenação por inserção é dada a seguir:

$$insertion\_sort\ l := \begin{cases} l, & \text{se } l = nil \\ insert\ x\ (insertion\_sort\ tl), & \text{se } l = h :: tl \end{cases}$$

Exercício 1.4. *O algoritmo* `insertion_sort` *é correto? Como você responderia a esta pergunta com as devidas justificativas em papel e lápis? E no Coq?*

Tente resolver os exercícios apresentados, e traga as suas dúvidas para discutirmos na próxima aula. O arquivo Coq está disponível em http://flaviomoura.info/files/paa_2024_1_introducao.v

2 Leitura complementar:

- [1] (Capítulo 6)
- [2]
- [3] (Capítulo 6, secão 6.4)

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[3] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.