Projeto e Análise de Algoritmos

Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília[1]

19 de agosto de 2025

[1]flaviomoura@unb.br

Capítulo 1

Introdução

Este material serve de suporte para a disciplina Projeto e Análise de Algoritmos do curso de graduação em Computação. Dois aspectos fundamentais no estudo de algoritmos são estudados em detalhe: a correção e a análise assintótica dos algoritmos.

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 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**! Esta é a abordagem que utilizaremos aqui, e que tem se mostrado cada vez mais importante

para o desenvolvimento da Matemática[7, 6, 1, 2] e Computação[8, 11, 10]. Para concluir esta seção e começarmos a colocar a mão na massa, listamos três exemplos famosos de erros em sistemas computacionais:

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.

Já deixamos claro que vamos **provar** muita coisa aqui. No próximo capítulo estudaremos as técnicas utilizadas para mostrar a correção de algoritmos.

Capítulo 2

A Correção de Algoritmos

Vamos iniciar considerando um problema bem simples: busca em uma lista. Formalmente, podemos enunciar este problema da seguinte forma:

> Dados um natural x e uma lista l, queremos saber se x ocorre em l. A resposta é apenas sim ou não.

A função *bseq* a seguir, recebe um natural `x` e uma lista `l` como argumentos, e resolve este problema:

$$bseq\ x\ l := \begin{cases} false & \text{se } l = nil \text{ (lista vazia);} \\ true & \text{se } l = h :: tl \text{ e } x = h; \\ bseq\ x\ tl & \text{se } l = h :: tl \text{ e } x \neq h; \end{cases}$$

O que significa dizer que este algoritmo (função) é correto? Uma possibilidade consiste em mostrar que qualquer que seja a lista l e o natural x, se x ocorre em l então $bseq\ x\ l = true$, e se x não ocorre em l então $bseq\ l = false$. Como mostrar isto? Podemos separar esta afirmação em duas partes:

1. se x não ocorre em l então $bseq\ l = false$;

2. se x ocorre em l então $bseq\ x\ l = true$.

Vamos provar cada uma destas partes separadamente.

Prova da parte1: A prova é por indução na estrutura da lista l.

- Se l for lista vazia então $bseq\ x\ nil = false$ e a afirmação é verdadeira uma vez que x não ocorre na lista vazia.

- Suponha que l tem a forma $h :: tl$. Temos 2 subcasos:

  - Se $x = h$ então encontramos uma ocorrência de x na lista l, e o algoritmo retorna `true` como esperado.

  - Se $x \neq h$ então o algoritmo continua a busca na cauda da lista l, e a hipótese de indução nos permite concluir que a afirmação está correta.

Exercício 1. *Escreva a prova da parte 2.*

A prova que acabamos de fazer é bem simples, mas veremos outras bem mais complicadas ao longo do semestre. A prova acima é dita informal em contraposição com provas mecânicas, isto é, feitas em sistema de provas implementado em um computador. Estes sistemas de provas são chamados de assistentes de provas e existem vários disponíveis, como o Rocq, PVS e Isabele/HOL. Vamos refazer a prova acima no Rocq (`https://rocq-prover.org/`). Alguns poucos ajustes são necessários para que a função esteja de acordo com a sintaxe da linguagem funcional do Rocq:

```
Fixpoint bseq x l :=
 match l with
 | nil => false
 | h::tl => if (x =? h) then true else bseq x tl
 end.
```

A propriedade correspondente a parte 1 pode ser escrita da seguite forma:

```
Lemma bseq_correto_parte1: forall l x, ~(In x l) -> bseq x l = false.
```

A prova deste lema está disponível no arquivo bseq.

Agora considere o seguinte problema:

Encontre o menor elemento de uma lista de números naturais.

A função recursiva $min\_list\ h\ l$ que recebe um natural h e uma lista de números naturais l como argumento, e retorna o menor elemento da lista $(h :: l)$:

$$min\_list\ h\ l = \begin{cases} h, & \text{se } l = nil \\ min\_list\ h\ l', & \text{se } l = h' :: l' \text{ e } h \leq h' \\ min\_list\ (h', l'), & \text{se } l = h' :: l' \text{ e } h > h' \end{cases}$$

A função é definida recursivamente na estrutura da lista l, segundo argumento da função min_list. De fato, quando l é a lista vazia, notação nil, a função retorna h. Quando a lista é não vazia com primeiro elemento h' e cauda l', notação $h' :: l'$, temos dois subcasos a considerar:

1. $h \leq h'$: Neste caso, a chamada recursiva é feita com os argumentos h e l';

2. $h > h'$: Neste caso, a chamada recursiva é feita com os argumentos h' e l'.

Afirmação. A função $min\_list\ h\ l$ retorna o menor elemento da lista $(h :: l)$.

Podemos ver facilmente que a propriedade expressa nesta afirmação é verdadeira em diversos casos. Por exemplo, se $h = 1$ e $l = 2 :: 3 :: 4 :: nil$ então esperamos que $min\_list1(2 :: 3 :: 4 :: nil)$ retorne 1, ou seja, o menor elemento da lista $(1 :: 2 :: 3 :: 4 :: nil)$. De fato, temos

$$\begin{aligned} &min\_list\ 1\ (2 :: 3 :: 4 :: nil) &&= \\ &min\_list\ 1\ (3 :: 4 :: nil) &&= \\ &min\_list\ 1\ (4 :: nil) &&= \\ &min\_list\ 1\ nil &&= \ 1 \end{aligned}$$

Ou ainda,

$$\begin{array}{lll} min\_list\ 10\ (2 :: 3 :: 4 :: nil) & = & \\ min\_list\ 2\ (3 :: 4 :: nil) & = & \\ min\_list\ 2\ (4 :: nil) & = & \\ min\_list\ 2\ nil & = & 2 \end{array}$$

E assim, podemos testar diversos argumentos para verificar se a função (algoritmo) *min_list* está funcionando corretamente, mas esses testes seriam suficiente para garantirmos que a afirmação acima é verdadeira? Observe que existe uma infinidade de argumentos distintos possíveis... Certamente, não! Para garantirmos que função *min_list h l* retorna o menor elemento da lista $(h :: l)$ quaisquer que sejam h e l precisamos construir uma prova. Utilizando indução na estrutura da lista l, provaremos que a afirmação é verdadeira.

Prova da afirmação. Temos dois casos a considerar:

1. A lista l é vazia, isto é, $l = nil$: Neste caso, *min_list h nil* retorna h, que é o menor elemento da lista $(h :: nil)$.

Exercício 2. *Complete a prova da afirmação acima. Ou seja, mostre que min_list h l retorna o menor elemento da lista* $(h :: l)$ *quando l é uma lista não vazia.*

2.1 Indução

Indução é uma ferramenta fundamental que será utilizada com frequência para provar a correção de algoritmos. Além disso, na análise da eficiência dos algoritmos, usaremos várias ferramentas matemáticas, como somatórios, conjuntos, funções e matrizes. O apêndice VIII do livro [4, 5] pode ser consultado para revisar esses tópicos. A próximas subseções fazem uma revisão de indução.

2.1.1 Indução Matemática

Indução matemática é uma técnica de prova muito poderosa que desempenha um papel fundamental tanto em Matemática quanto em Computação. Se $P(n)$ denota uma propriedade dos números naturais $\mathbb{N} = \{0, 1, 2, \ldots\}$ então o princípio da indução matemática (PIM) é dado por:

$$\frac{P\ 0 \qquad \forall k, P\ k \Longrightarrow P\ (k+1)}{\forall n, P\ n}\ (PIM)$$

Na descrição acima, chamamos $P\ 0$ de *base da indução* e $\forall k, P\ k \Longrightarrow P\ (k+1)$ de *passo indutivo*. No passo indutivo, $P\ k$ é chamado de *hipótese de indução*. Vejamos um exemplo:

Considere a seguinte propriedade sobre os números naturais:

$$\text{A soma dos } n \text{ primeiros números naturais ímpares é igual a } n^2. \tag{2.1}$$

Esta propriedade vale trivialmente para o 0 (a soma dos 0 primeiros números ímpares é igual a 0^2), o que corresponde à base da indução. Agora seja k um natural arbitrário, e suponha que a soma dos k primeiros números ímpares seja igual a k^2 (hipótese de indução). Precisamos provar que a soma dos $k+1$ primeiros números ímpares é igual a $(k+1)^2$. De fato, o $(k+1)$-ésimo número ímpar é igual a $2.k+1$ (por que?), e portanto $k^2+2.k+1 = (k+1)^2$ como queríamos provar.

Uma prova mais detalhada de (2.1) pode ser feita da seguinte forma: a soma dos n primeiros números ímpares pode ser escrita por meio do somatório $\sum_{i=1}^{n}(2.i-1)$, que por definição é igual a 0, se $n = 0$. Queremos provar que

$$\sum_{i=1}^{n}(2.i-1) = n^2, \forall n \tag{2.2}$$

Aplicando o princípio da indução matemática (PIM), temos 2 casos para analisar:

- (**Base da indução**): Para $n = 0$, a igualdade (2.2) é trivial porque o lado esquerdo da igualdade é igual a 0 por definição.

- (**Passo indutivo**): No passo indutivo assumimos que (2.2) vale para um número natural arbitrário, digamos k, e provamos que esta propriedade continua valendo para o natural $k+1$. Ou seja, assumimos que $\sum_{i=1}^{k}(2.i-1) = k^2$, e vamos provar que $\sum_{i=1}^{k+1}(2.i-1) = (k+1)^2$. Partindo do lado esquerdo desta última igualdade, podemos decompor o somatório da seguinte forma $\sum_{i=1}^{k+1}(2.i-1) = \sum_{i=1}^{k}(2.i-1) + (2.k+1)$, e agora podemos utilizar a hipótese de indução (h.i.) para assim chegarmos ao lado direito da mesma: $\sum_{i=1}^{k+1}(2.i-1) = \sum_{i=1}^{k}(2.i-1) + (2.k+1) \overset{h.i.}{=} k^2 + (2.k+1) = (k+1)^2$.

Observe que o passo indutivo é a parte interessante de qualquer prova por indução. A base da indução consiste apenas na verificação de que a propriedade vale para uma situação particular. Agora resolva os exercícios a seguir:

Exercício 3. *Mostre que* $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Exercício 4. *Prove que* $\sum_{i=0}^{n} i(i+1) = \frac{n.(n+1).(n+2)}{3}$.

Exercício 5. *Prove que* $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

Exercício 6. *Prove que* $3 \mid (2^{2n} - 1)$ *para todo* $n \geq 0$.

Existem propriedades que valem apenas para um subconjunto próprio dos números naturais:

Por exemplo, $2^n < n!$ só vale para $n \geq 4$. Para este tipo de problema utilizamos uma generalização do PIM onde a base de indução não precisa ser o 0. Chamaremos esta variação de *Princípio da Indução Generalizado (PIG)*:

$$\frac{P\ m \qquad \forall k, P\ k \Longrightarrow P\ (k+1)}{\forall n \geq m, P\ n}\ (PIG)$$

Exemplo 1. *Prove que* $2^n < n!, \forall n \geq 4$.

1. *(Base de indução) A propriedade vale para* $n = 4$*, o que é trivial, e;*

2. *(Passo indutivo) Mostraremos que* $2^{k+1} < (S\ k)!$ *assumindo que* $2^k < k!, \forall k \geq 4$*. De fato, temos que* $2^{k+1} = 2.2^k \overset{(h.i)}{<} 2.k! \overset{(*)}{<} (k+1).k! = (k+1)!$*, onde a desigualdade (*) se justifica pelo fato de* k *ser maior ou igual a 4.*

Exercício 7. *Prove que a soma dos* n *primeiros números naturais é igual a* $\frac{n(n+1)}{2}$.

Exercício 8. *Prove que a soma dos* n *primeiros quadrados é igual a* $\frac{n(n+1)(2n+1)}{6}$*. Ou seja, mostre que* $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

Exercício 9. *Prove que a soma dos* n *primeiros cubos é igual ao quadrado da soma de 1 até* n*, ou seja, que* $1^3 + 2^3 + \ldots + n^3 = (1 + 2 + \ldots + n)^2$.

Exercício 10. *Prove que* $2^n - 1$ *é múltiplo de 3, para todo número natural* n *par.*

Exercício 11. *Prove que* $3^n \geq n^2 + 3$ *para todo* $n \geq 2$.

Exercício 12. *Prove que* $n^2 < 4^{n-1}$ *para todo* $n \geq 3$.

Exercício 13. *Prove que* $n! > 3^n$ *para todo* $n \geq 7$.

Exercício 14. *Prove que* $n! \leq n^n$ *para todo* $n \geq 1$.

Uma variação do PIM bastante útil é conhecida como *Princípio da Indução Forte (PIF)*:

$$\frac{\forall k, (\forall m, m < k \Longrightarrow P\ m) \Longrightarrow P\ k}{\forall n, P\ n}\ (PIF)$$

Exercício 15. *Prove que qualquer inteiro* $n \geq 2$ *é um número primo ou pode ser escrito como um produto de primos (não necessariamente distintos), i.e. na forma* $n = p_1.p_2.\cdots.p_r$, *onde os fatores* p_i $(1 \leq i \leq r)$ *são primos.*

Exercício 16. *Mostre que PIM e PIF são princípios equivalentes.*

2.1.2 Indução Estrutural

Nesta seção veremos que o princípio de indução matemática (PIM) visto anteriormente é um caso particular de um princípio geral que está associado a qualquer conjunto definido indutivamente. Vimos dois tipos de regras utilizadas na construção de um conjunto definido indutivamente:

1. As regras não recursivas, ou seja, aquelas que definem diretamente um elemento do conjunto definido indutivamente;

2. As regras recursivas, ou seja, aquelas que constroem novos elementos a partir de elementos já construídos.

Como veremos no próximo exemplo, estas regras podem fazer uso de elementos de outros conjuntos previamente definidos. Formalmente, se $A_1, A_2, \ldots$ são conjuntos então a estrutura geral das regras de um conjunto definido indutivamente B é como a seguir:

1. Inicialmente temos as regras não recursivas que definem diretamente os elementos $b_1, \ldots, b_m$ de B:

$$\frac{a_1 \in A_1 \qquad a_2 \in A_2 \ldots\ a_{j_1} \in A_{j_1}}{b_1[a_1, \ldots, a_{j_1}] \in B} \qquad \cdots \qquad \frac{a_1 \in A_1 \qquad a_2 \in A_2 \ldots\ a_{j_m} \in A_{j_m}}{b_m[x_1, \ldots, x_{j_m}] \in B}$$

2. Em seguida, temos as regras recursivas que constroem novos elementos a partir de elementos já construídos:

$$\frac{a_1 \in A_1 \ldots\ a_{j_1'} \in A_{j_1'}\ d_1, \ldots, d_{k_1} \in B}{c_1[x_1, \ldots, x_{j_1'}, d_1, \ldots, d_{k_1}] \in B} \quad \ldots \quad \frac{a_1 \in A_1 \ldots\ a_{j_n} \in A_{j_n}\ d_1, \ldots, d_{k_n} \in B}{c_n[a_1, \ldots, a_{j_n}, d_1, \ldots, d_{k_n}] \in B}$$

Qualquer elemento de um conjunto definido indutivamente pode ser construído após um número finito de aplicações das regras que o definem (e somente com estas regras). Os elementos $d_1, d_2, \ldots, d_{k_i}$ são ditos *estruturalmente menores* do que o elemento $c_i[a_1, \ldots, a_{j_i}, d_1, \ldots, d_{k_i}]$. Isto significa que os elementos $d_1, d_2, \ldots, d_{k_i}$ são subtermos próprios de $c_i[a_1, \ldots, a_{j_i}, d_1, \ldots, d_{k_i}]$.

Podemos associar um princípio de indução a qualquer conjunto definido indutivamente. No contexto genérico acima, teremos um caso base (base da indução) para cada regra não recursiva, e um passo indutivo para cada regra recursiva. O esquema simplificado (omitindo os parâmetros por falta de espaço) tem a seguinte forma:

$$\frac{\overbrace{P(b_1) \ldots P(b_m)}^{\text{casos base}}\ \overbrace{(\forall d_1 \ldots d_{k_1}, P(d_1), \ldots, P(d_{k_1}) \Rightarrow P(c_1)) \ldots (\forall d_1 \ldots d_{k_1}, P(d_1), \ldots, P(d_{k_n}) \Rightarrow P(c_n))}^{\text{casos indutivos}}}{\forall x \in B, P\ x}$$

Retornando ao caso do conjunto dos números naturais, temos um princípio indutivo com apenas um caso base e um caso indutivo:

$$\frac{P\ 0 \qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, P\ n}$$

O conjunto dos booleanos possui um princípio indutivo com dois casos base, e nenhum caso indutivo:

$$\frac{P \text{ true} \qquad P \text{ false}}{\forall b, P\ b}$$

Futuramente estudaremos diversos algoritmos que utilizam a estrutura de lista encadeada, definida pela seguinte gramática $l ::= nil \mid a :: l$, onde nil representa a lista vazia, e $a :: l$ representa a lista com primeiro elemento a e cauda l. Como esta gramática possui um construtor não recursivo, e um construtor recursivo, teremos um princípio de indução com um caso base, e um passo indutivo:

$$\frac{P\ nil \qquad \forall l\ h, P\ l \Longrightarrow P\ (h :: l)}{\forall l, P\ l}$$

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.

Exercício 17. *Prove que* $|l_1 \circ l_2| = |l_1| + |l_2|$*, quaisquer que sejam as listas* l_1, l_2*.*

Exercício 18. *Prove que* $l \circ nil = l$*, qualquer que seja a lista* l*.*

Exercício 19. *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*.*

Exercício 20. *Prove que* $|rev(l)| = |l|$*, qualquer que seja a lista* l*.*

Exercício 21. *Prove que* $rev(l_1 \circ l_2) = (rev(l_2)) \circ (rev(l_1))$*, quaisquer que sejam as listas* l_1, l_2*.*

Exercício 22. *Prove que* $rev(rev(l)) = l$*, qualquer que seja a lista* l*.*

2.1.3 Leitura complementar:

- [9] (Capítulo 2)
- [3] (Capítulo 1)

Referências Bibliográficas

[1] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2–es, December 2007.

[2] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, April 2014.

[3] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA, 1996.

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

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

[6] G. Gonthier. A computer-checked proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2008.

[7] T. Hales, M. Adams, G. Bauer, D. Tat Dang, J. Harrison, T. Le Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. Tat Nguyen, T. Quang Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. Hoai Thi Ta, T. N. Tran, D. Thi Trieu, J. Urban, K. Khac Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *ArXiv e-prints*, January 2015.

[8] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107, 2009.

[9] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[10] R. B. Nogueira, A. C. A. Nascimento, F. L. C. de Moura, and M. Ayala-Rincón. Formalization of Security Proofs Using PVS in the Dolev-Yao Model. In *Booklet Proc. Computability in Europe - CiE*, 2010.

[11] Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J Autom Reasoning*, 55(1):1–37, 2015.