Projeto e Análise de Algoritmos

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

8 de setembro 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, 13, 12]. 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

TBC Vamos iniciar considerando um problema bem simples:

2.1 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:

  - O caso $x = h$ não ocorre já que estamos assumindo que x não ocorre em l.

– 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. Observe que a hipótese de indução diz que "se x não ocorre em tl então $bseq\ tl = false$;", e como x não ocorre em l, temos que x também não ocorre em tl e portanto, $bseq\ tl = false$. Logo, $bseq\ x\ l = bseq\ x\ tl = false$. □

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

Prova da parte2: A prova é por indução na estrutura da lista l.

- O caso em que l é a lista vazia não é possível porque estamos assumindo que x ocorre em l.
- Suponha que l tem a forma $h :: tl$. Temos 2 subcasos:
  - Se $x = h$ então o algoritmo retorna true como queríamos.
  - 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. Observe que a hipótese de indução diz que "se x ocorre em tl então $bseq\ tl = true$;", e como x ocorre em l, temos que x tem que ser igual a algum elemento da cauda tl, e portanto, $bseq\ tl = true$. Logo, $bseq\ x\ l = bseq\ x\ tl = true$. □

Prova alternativa da parte2: A prova alternativa é por indução no tamanho $|l|$ da lista l. Observe que queremos provar que "se x ocorre em l então $bseq\ x\ l = true$, para qualquer $|l| > 0$".

- O caso base é $|l| = 1$, ou seja, $l = x :: nil$. Neste caso, $bseq\ x\ (x :: nil) = true$ como queríamos.
- Suponha que $|l| > 1$, ou seja, l tem pelo menos dois elementos. Temos 2 subcasos:
  - Se $x = h$ então o algoritmo retorna true como queríamos.
  - 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. Observe que a hipótese de indução diz que "se x ocorre em tl então $bseq\ x\ tl = true$, para qualquer $|tl| > 0$", e como l tem pelo menos dois elementos, então $|tl| > 0$ e $bseq\ tl = true$. Logo, $bseq\ x\ l = bseq\ x\ tl = true$. □

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.

2.2 Elemento Mínimo

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,

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

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.*

2.3 A Torre de Hanoi

A Torre de Brahma

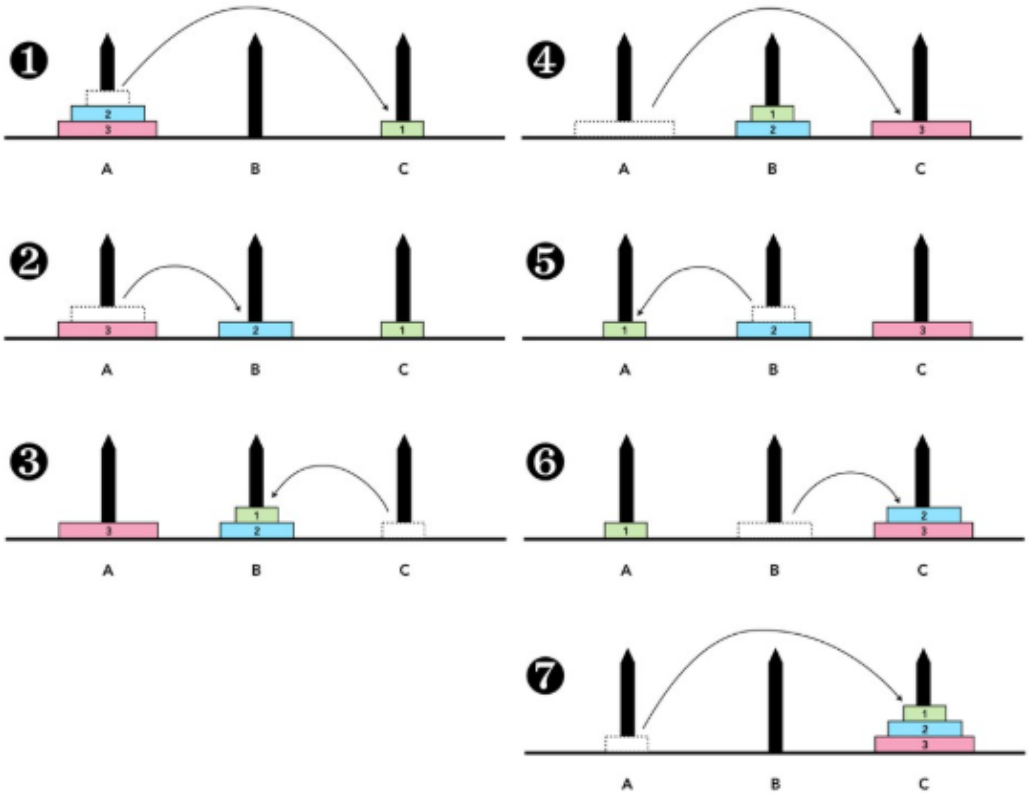
Conta a lenda que a Torre de Brahma era formada por 64 discos de ouro com diâmetros distintos e três hastes de diamante. No início dos tempos, Deus colocou todos os 64 discos sobre uma única haste, empilhados em ordem decrescente, do maior na base ao menor no topo, e ordenou que um grupo de sacerdotes os movesse para outra haste, seguindo as seguintes regras:

1. Um disco não pode ser colocado sobre outro de diâmetro menor;
2. Apenas um disco pode ser movido por vez;
3. Só é permitido mover o disco que estiver no topo de uma das torres.

Os sacerdotes trabalhariam ininterruptamente, noite e dia, para cumprir a tarefa. Dizia a lenda que, quando finalmente a concluíssem, a torre desmoronaria e o mundo chegaria ao fim.

O problema da Torre de Hanói é um famoso desafio inventado em 1883 pelo matemático francês Édouard Lucas (1842-1891). Ele consiste em uma base com três hastes. Em uma delas, está disposta uma torre formada por discos de diâmetros distintos, empilhados em ordem decrescente, do maior na base ao menor no topo.

O objetivo do desafio é mover toda a torre de discos para uma das outras duas hastes com o menor número de movimentos e obedecendo as regras acima. Observe que, se a torre tiver um único disco, o desafio é resolvido com um único movimento. Com dois discos, a solução também é simples e requer três movimentos. Com três discos, o problema começa a ficar interessante, sendo necessários pelo menos sete movimentos:



É claro que quanto maior o número de discos, maior será o número de movimentos necessários para transferir a torre de uma haste para outra dentro das regras estabelecidas.

Quantos movimentos são necessários para mover a torre com 64 discos? A melhor forma de resolver este problema consiste em tentar generalizar o número de discos. Vejamos o que acontece se tivermos $n > 0$ discos.

O problema da Torre de Hanói consiste em mover n discos de diâmetros diferentes de uma torre A para uma torre C usando uma torre auxiliar B.

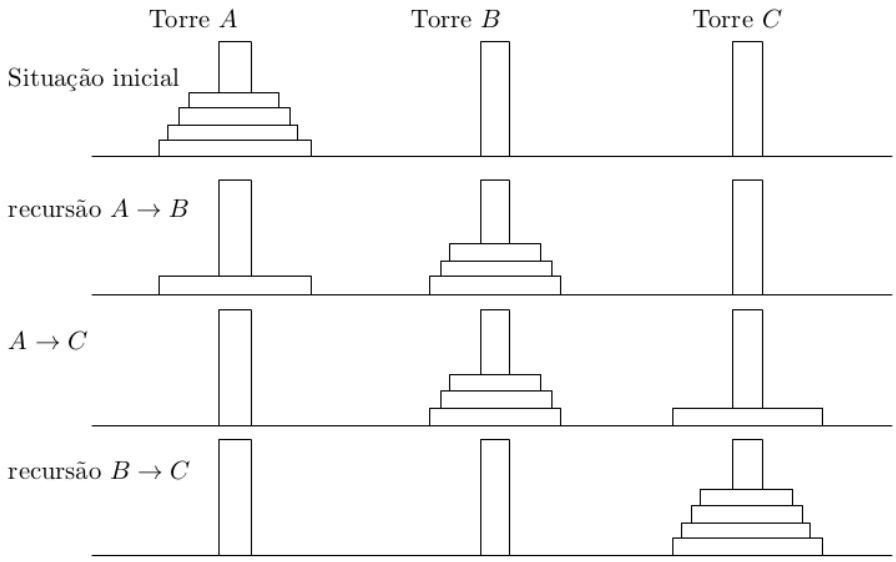


Figura 1: Torres de Hanoi

Os discos estão inicialmente organizados na torre A em ordem decrescente segundo o diâmetro e devem terminar na torre C com a mesma organização. Os discos são movidos de uma torre a outra de acordo com as três regras descritas acima (ver A Torre de Brahma). Esta descrição sugere o seguinte algoritmo:

Algoritmo

1. Transladar recursivamente os $n-1$ discos de menor diâmetro, da torre A para a torre B, usando a torre C como torre auxiliar;
2. Mover o disco de maior diâmetro de A a C;
3. Transladar recursivamente os $n-1$ discos de B a C, usando a torre A como torre auxiliar.

Exercício

Seja $M(n)$ o número de movimentos necessários para transladar n discos segundo o algoritmo anterior.

1. (1.0 ponto) Construa a relação de recorrência que expressa o número de movimentos $M(n)$ necessários para mover os discos da torre A para a torre C;

2. (2.0 pontos) Resolva a relação de recorrência do item anterior;

3. (1.0 ponto) Qual o tempo necessário para os sacerdotes terminarem o trabalho com os 64 discos, assumindo que eles movimentam 1 disco por segundo e sempre acertam o próximo movimento.

2.4 Tópico de revisão (opcional): 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.

2.4.1 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.*

2.4.2 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*.*

2.4.3 Leitura complementar:

- [11] (Capítulo 2)
- [3] (Capítulo 1)
- [10]

2.5 Invariantes de laço

- Leitura complementar: Capítulo 2 ([4]).

Invariantes de laço são utilizadas para provar propriedades de laços. Como o nome sugere, a propriedade a ser provada deve permanecer válida ao longo de toda a execução do laço. Concretamente, a prova de uma invariante de laço é dividida em três etapas:

1. **Inicialização**: A propriedade deve ser verdadeira antes da primeira iteração do laço;

2. **Manutenção**: Se a propriedade é verdadeira antes de uma iteração (qualquer) do laço então ela permanece verdadeira antes da próxima iteração;

3. **Terminação**: Quando o laço termina, a conclusão da invariante nos fornece uma informação útil para concluirmos que o algoritmo é correto.

Uma versão não recursiva do algoritmo *min_list*

Estudaremos diversos algoritmos não-recursivos neste curso, e a seguir, apresentamos uma versão não-recursiva do algoritmo *min_list*. Neste caso, o algoritmo recebe o vetor $A[1..n]$ como argumento:

```
1 min ← A[1];
2 for i = 2 to n do
3 |   if A[i] < min then
4 |   |   min ← A[i];
5 |   end
6 end
7 return min;
```

Algoritmo 1: *min_list_norec*$(A[1..n])$

Afirmação. Ao final de sua execução o algoritmo *min_list_norec*$(A[1..n])$ retorna o menor elemento do vetor $A[1..n]$.

Como provar esta afirmação? Em programas contendo laços utilizamos as chamadas *invariantes de laço*, que são propriedades satisfeitas durante toda a execução de um laço. A prova de uma invariante de laço consiste em três etapas:

1. **Inicialização**: Nesta etapa mostramos que a propriedade é satisfeita antes da primeira execução do laço. Esta etapa é equivalente à base da indução em uma prova indutiva;

2. **Manutenção**: Esta é a etapa mais delicada da prova (veja a semelhança com um passo indutivo). Aqui assumimos por hipótese (de indução) que a invariante vale antes de uma iteração arbitrária do laço (depois da primeira) e mostramos que a invariante continua válida antes da próxima iteração. Ou seja, assumimos que a invariante vale antes da k-ésima iteração ($k > 0$) e mostramos que ao final desta iteração, isto é, antes da $(k+1)$-ésima iteração, a invariante continua valendo;

3. **Finalização**: Nesta etapa concluímos que a invariante é satisfeita durante toda a execução do laço, e esta informação é utilizada para estabelecer a correção do algoritmo.

O laço **for** do algoritmo *min_list_norec*$(A[1..n])$ percorre os todos os elementos do vetor A a partir da segunda posição. Não podemos garantir que a variável *min* armazena o menor elemento de A ao longo de toda a execução do algoritmo, mas certamente *min* armazena o menor elemento dos que já foram considerados. Assim, provaremos a seguinte invariante de laço:

Invariante. Antes de cada iteração indexada por i, a variável min contém o menor elemento do subvetor $A[1..i-1]$.

Prova da invariante. Consideraremos as três etapas descritas acima:

1. **Inicialização**: Antes da primeira iteração, temos $i = 2$, e $min = A[1]$ (linha 1) contém o menor elemento do subvetor $A[1]$.

2. **Manutenção**: Assuma que antes da k-ésima iteração, isto é, quando $i = k + 1$, min contém o menor elemento do subvetor $A[1..k]$. Ao longo da k-ésima iteração o elemento $A[i] = A[k + 1]$ será comparado com min (linha 3). Se $A[k+1]$ for menor do que min então encontramos um valor menor do que o atual no subvetor $A[1..k + 1]$, e este novo valor é armazenado em min (linha 4). Assim, antes da (k+1)-ésima iteração min contém o menor elemento do subvetor $A[1..k+1]$, como queríamos provar.

3. **Finalização**: Ao final da execução do laço, isto é, quando $i = n + 1$, temos que min contém o menor elemento do vetor $A[1..n]$, e portanto a invariante é verdadeira. □

Exercício 23. *Resolva os exercícios a seguir:*

1. *Apresente o pseudocódigo de um algoritmo não-recursivo que verifica se um determinado número está presente em um vetor $A[1..n]$ de números naturais. O algoritmo deve retornar* `true` *se o número estiver na lista e* `false` *caso contrário. Em seguida, prove a correção do seu algoritmo.*

2. *Apresente o pseudocódigo de um algoritmo recursivo que retorna o número de ocorrências de um dado elemento em uma lista. Em seguida, prove a correção do seu algoritmo.*

3. *Apresente o pseudocódigo de um algoritmo não-recursivo que retorna o número de ocorrências de um dado elemento em um vetor. Em seguida, prove a correção do seu algoritmo.*

4. *Apresente o pseudocódigo de um algoritmo recursivo que retorna o maior elemento de uma lista de números naturais. Em seguida, prove a correção do seu algoritmo.*

5. *Apresente o pseudocódigo de um algoritmo não-recursivo que retorna o maior elemento de um vetor $A[1..n]$ de números naturais. Em seguida, prove a correção do seu algoritmo.*

Capítulo 3

A complexidade de algoritmos

Como avaliar o tempo de execução de um algoritmo? Antes de respondermos esta pergunta, devemos fazer outra: por que avaliar o tempo de execução de um algoritmo? Em outras palavras, qual a necessidade de estudar análise de algoritmos? Como veremos, diversas razões para isso. Inicialmente, observe que um algoritmo tem como objetivo resolver um problema específico, como ordenação, busca ou cálculo de um determinado valor. No entanto, para um mesmo problema, podem existir múltiplos algoritmos diferentes. Por exemplo, no caso da ordenação há diversos algoritmos conhecidos, como *Insertion Sort*, *Merge Sort*, *Selection Sort*, *Heapsort*, *Quicksort*, etc. Qual deles escolher? O objetivo da análise de algoritmos é fornecer as ferramentas teóricas que nos permitam responder essa pergunta com base em critérios matemáticos, garantindo a escolha da solução mais eficiente para cada situação.

O que significa dizer que um algoritmo é eficiente? Podemos analisar a eficiência de um algoritmo de duas formas: eficiência temporal e eficiência espacial. No primeiro caso, estamos interessados no tempo de execução do algoritmo, enquanto que no segundo caso, queremos analisar a quantidade de espaço extra (memória) que é utilizado pelo algoritmo durante sua execução [5, 3, 11, 9]. A forma de determinar a eficiência de um algoritmo deve permitir a comparação de algoritmos distintos que resolvam o mesmo problema. Inicialmente, poderíamos pensar em utilizar o tempo de execução de um programa que implementa um algoritmo, mas esta não é uma boa medida porque depende tanto do computador (*hardware*) quanto da implementação (*software*). Precisamos de um método que nos informe sobre a eficiência do algoritmo independentemente do computador em que ele venha a ser implementado, da linguagem de programação e do estilo de programação utilizados. O método deve ser preciso e geral de forma que possa ser utilizado para diversos algoritmos e aplicações.

Antes de apresentarmos o ferramental matemático que utilizaremos, precisamos estabelecer alguns critérios:

- **Independência de *hardware***: A escolha do algoritmo deve ser baseada em sua eficiência teórica, não no desempenho de um hardware específico;
- **Independência de *software***: A análise deve ser independente da linguagem de programação ou das habilidades do programador;
- **Princípio da invariância**: Diferentes implementações de um mesmo algoritmo não devem variar em eficiência além de um fator constante multiplicativo.

Uma possibilidade é contar todas as operações realizadas pelo algoritmo, mas veremos que esta abordagem possui alguns problemas, além de ser muito trabalhosa.

3.0.1 Busca Sequencial

Considere o problema de buscar um elemento em um vetor arbitrário. O pseudocódigo a seguir recebe como argumentos o vetor $A[0..n-1]$ contendo n elementos e o valor x procurado, e faz a buscar sequencial de x em A: se A possui uma ocorrência de x então o algoritmo retorna a posição $0 \leq i \leq n-1$ da primeira ocorrência encontrada. Caso contrário, o algoritmo retorna o valor -1.

```
1 i ← 0;
2 while i < n and A[i] ≠ x do
3 |   i ← i + 1;
4 end
5 if i < n then
6 |   return i;
7 else
8 |   return -1
9 end
```

Algoritmo 2: SequentialSearch($A[0..n-1], x$)

Observe que a execução do algoritmo não demanda espaço adicional, ou seja, o espaço utilizado para a sua execução é o espaço alocado para armazenar o vetor A e nada mais. Neste caso, dizemos que a complexidade em espaço do algoritmo SequentialSearch é constante. Uma complexidade, seja em tempo ou espaço, constante é a melhor situação que podemos ter, ou seja, a mais eficiente possível. As classes básicas de eficiência que utilizaremos para analisar algoritmos são listadas a seguir em ordem crescente de complexidade em função do tamanho n da entrada[5]:

Classe	**Nome**
1	constante
$\log n$	logarítmica
n	linear
$n.\log n$	linearítmica
n^2	quadrática
n^3	cúbica
n^k	polinomial ($k \geq 1$ e finito)
a^n	exponencial ($a \geq 2$)
$n!$	fatorial

A análise da complexidade de tempo do algoritmo SequentialSearch não é tão imediata quanto a análise feita para a complexidade de espaço, ainda que seja simples. Podemos começar com a seguinte pergunta: qual o custo de execução de cada linha do algoritmo SequentialSort? A linha 1 faz uma atribuição, cujo custo não depende do tamanho n do vetor A, e portanto é razoável dizer que este custo é constante, digamos c_1, uma constante positiva. Observe que esta constante não depende do parâmetro n, mas do computador e da linguagem de programação. As linhas 2-4 constituem um laço cujo corpo contém apenas uma atribuição. Ainda que o custo da linha 3 possa ser o mesmo da linha 1, vamos denotá-lo pela constante positiva c_3. Quantas vezes a linha 3 é executada? Isto depende tanto do vetor A quanto da chave x. De fato, se x ocorre na primeira posição de A, isto é, se $A[0]$ é igual a x então a condição do laço é executada uma única vez, mas a linha 3 não é executada nenhuma vez independente de existirem outras ocorrências de x em A. Esta é a situação constitui o melhor caso possível, e por isso é chamada de *análise do melhor caso*. Se $A[0] \neq x$ e x ocorre na segunda posição de A então a linha 3 é executada uma única vez, enquanto que a linha 2 é executada duas vezes. Em geral, observe que a linha que define um laço é sempre executada uma vez a mais do que as linhas que compõem o seu corpo. Por fim, se x não ocorre no vetor A então a linha 2 será executada $n+1$ vezes enquanto que a linha 3 será executada n vezes. Esta situação vai configurar a *análise do pior caso*. Por fim, o condicional da linha 5 será executado uma única vez a um custo constante, digamos c_5, e apenas uma das linhas 6 ou 8 será executada uma única vez. Juntando todas estas informações podemos então dividir a análise em 2 casos:

Análise do melhor caso na busca sequencial

Como vimos anteriormente, o melhor caso ocorre quando o elemento procurado ocorre na primeira posição do vetor A. Os custos associados por linha nesta situação são apresentados na seguinte tabela:

Linha	Custo	Observação
1	c_1	
2	c_2	
3	0	não é executada
5	c_5	
6	c_6	
8	0	não é executada
Total	$c_1 + c_2 + c_5 + c_6$	

Denotando por $T_b(n)$ o custo no melhor caso (*best case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_b(n) = c_1 + c_2 + c_5 + c_6$. Neste caso dizemos que o custo da busca sequencial é constante em função do tamanho n da entrada.

Análise do pior caso na busca sequencial

Agora vamos compilar as informações discutidas anteriormente considerando que o laço da linha 2 é executado o maior número de vezes possível, o que acontece quando o elemento procurado não ocorre no vetor:

Linha	Custo
1	c_1
2	$c_2.(n+1)$
3	$c_3.n$
5	c_5
6	0
8	c_8
Total	$c_1 + c_2.(n+1) + c_3.n + c_5 + c_8$

Denotando por $T_w(n)$ o custo no pior caso (*worst case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_w(n) = c_1 + c_2.(n+1) + c_3.n + c_5 + c_8$. Neste caso dizemos que o custo da busca sequencial é linear em função do tamanho n da entrada. Antes de refinarmos a análise e apresentarmos as definições precisas das análises de melhor e pior caso, vejamos um outro exemplo considerando agora o problema da ordenação de um vetor.

Vejamos um outro exemplo antes de refinarmos nossa análise.

3.0.2 Ordenação por inserção

Nesta seção estamos interessados em ordenar $n > 0$ números naturais em ordem crescente. Suponha que estes números estão armazenados no vetor $A[0..n-1]$. Então, ao final do processo queremos obter uma permutação de $A[0..n-1]$, digamos $A'[0..n-1]$ tal que $A'[i-1] \leq A'[i]$, para todo $1 \leq i < n$. Estudaremos diversas formas distintas de abordar este problema nas próximas seções, mas aqui vamos iniciar com o algoritmo de ordenação por inserção (InsertionSort), cujo pseudocódigo é dado a seguir:

```
1 for  j = 1 to n − 1 do
2 |   key ← A[j];
3 |   i ← j − 1;
4 |   while  i ≥ 0 and A[i] > key do
5 |   |   A[i + 1] ← A[i];
6 |   |   i ← i − 1;
7 |   end
8 |   A[i + 1] ← key;
9 end
```

Algoritmo 3: InsertionSort($A[0..n-1]$)

Faremos uma análise da complexidade do algoritmo de ordenação por inserção semelhante análoga à feita para a busca sequencial. Certamente, ordenar um vetor com 1000 demanda mais tempo do que ordenar apenas 3 elementos, assim é usual descrever o tempo de execução de um algoritmo em função do tamanho da entrada que neste caso é o número n de elementos a serem ordenados. Novamente assumiremos que cada linha do pseudocódigo é executada em tempo constante, mas este tempo pode diferir de uma linha para outra. Assim, denotaremos por c_i a constante que corresponde ao tempo de execução da i-ésima linha do pseudocódigo. Vejamos, então, o custo de execução do algoritmo InsertionSort. O laço **for** da linha 1 é executado n vezes, enquanto que o corpo do laço é executado $n-1$ vezes, uma vez para cada $j = 1, \ldots, n-1$. Denotaremos por t_j o número de vezes que o teste do laço **while** da linha 4 é executado, de forma que temos o seguinte custo por linha:

Linha	Custo	Número de execuções	Custo total
1	c_1	n	$c_1.n$
2	c_2	$n-1$	$c_2.(n-1)$
3	c_3	$n-1$	$c_3.(n-1)$
4	c_4	$\sum_{j=1}^{n-1} t_j$	$c_4.\sum_{j=1}^{n-1} t_j$
5	c_5	$\sum_{j=1}^{n-1}(t_j-1)$	$c_5.\sum_{j=1}^{n-1}(t_j-1)$
6	c_6	$\sum_{j=1}^{n-1}(t_j-1)$	$c_6.\sum_{j=1}^{n-1}(t_j-1)$
8	c_8	$n-1$	$c_8.(n-1)$

Portanto, o custo total, que denotaremos por $T(n)$ é dado por:

$$T(n) = c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.\sum_{j=2}^{n} t_j + c_5.\sum_{j=2}^{n}(t_j-1) + c_6.\sum_{j=2}^{n}(t_j-1) + c_8.(n-1)$$

Agora note que, mesmo para entradas de mesmo tamanho, o tempo de execução pode mudar. De fato, um vetor que tenha mais elementos a serem reposicionados terá um custo maior para ser ordenado. Portanto, a análise do melhor caso se dá quando o vetor já estiver ordenado pois $t_j = 1$, para todo $2 \leq j \leq n$:

$$\begin{aligned} T_b(n) &= c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.(n-1) + c_8.(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8).n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

ou seja, uma função linear de n. Por outro lado, a análise do pior caso se dá quando o vetor estiver ordenado decrescentemente pois $t_j = j$ (por que?), e portanto

$$T_w(n) = c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\tfrac{(n-1).n}{2}) + (c_5 + c_6).(\tfrac{(n-2).(n-1)}{2})$$

ou seja, uma função quadrática de n.

A forma de análise feita para InsertionSort acima, assim como para SequentialSearch na seção anterior, apresenta alguns problemas porque as constantes utilizadas podem mudar dependendo do computador,

da linguagem de programação ou mesmo do estilo de programação utilizados. Uma maneira de ignorar estas especificidades, e fazer uma análise que seja independente destes aspectos, consiste na utilização de uma notação adequada, a *notação assintótica*, que considera o comportamento de funções no limite, isto é, para valores suficientemente grandes do parâmetro n. A ideia é que possamos pegar uma função como $T_w(n) = c_1 + c_2.n + c_3.(n-1) + c_6 + c_8$ que expressa o custo no pior caso do algoritmo de busca sequencial, e dizer que ela cresce como n, sem a necessidade de considerar as constantes. Faremos isto considerando o conjunto das funções que são limitadas superiormente por um múltiplo constante de n. Observe que podemos facilmente construir uma cota superior para a função $T_w(n)$ da seguinte forma $T_w(n) = c_1+c_2.n+c_3.(n-1)+c_6+c_8 \leq c_1.n+c_2.n+c_3.n-c_3+c_6.n+c_8.n \leq (c_1+c_2+c_3+c_6+c_8).n \leq c.n$ para qualquer constante $c \geq c_1 + c_2 + c_3 + c_6 + c_8$ e $n \geq 1$. Neste caso, dizemos que a função $T_w(n)$ é $O(n)$, ou seja, que $T_w(n)$ é de ordem n. Formalmente, temos a seguinte definição para o conjunto $O(g(n))$ que contém todas as funções que são da ordem de $g(n)$:

Definição 2. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $O(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $f(n) \leq c \cdot g(n), \forall n \geq n_0$. Alternativamente, $O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0.\}$*

Observe que uma função $f(n)$ pode estar em $O(g(n))$ mesmo que $f(n) > g(n), \forall n$. O ponto importante é que $f(n)$ tem que ser limitada por um múltiplo constante de $g(n)$. A relação entre $f(n)$ e $g(n)$ para valores pequenos de n também é desconsiderada. Intuitivamente, os termos de menor ordem de uma função assintoticamente positiva podem ser ignorados na determinação da cota superior porque são insignificantes para valores grandes do parâmetro n. Assim, quando n é grande qualquer porção ou fração do termo de maior ordem é suficiente para dominar os termos de menor ordem.

Normalmente, escrevemos $T(n) \in O(n^2)$ para dizer que $T(n)$ é $O(n^2)$ já que $O(n^2)$ é um conjunto. No entanto, é comum encontrarmos o uso da igualdade $T(n) = O(n^2)$ ao invés de $T(n) \in O(n^2)$ (Ver [5]). A conveniência do uso da igualdade será vista posteriormente, mas o importante aqui é entender que esta igualdade é unidirecional, e portanto não pode ser confundida com a igualdade tradicional. Por exemplo, escrevemos $T(n) = O(n^2)$, mas $O(n^2) = T(n)$ não é correto. O número de funções anônimas em uma expressão é igual ao número de vezes que a notação assintótica aparece: por exemplo, na expressão $\sum_{i=1}^{n} O(i)$ contém apenas uma função anônima (a função que tem parâmetro i), e portanto esta expressão não é o mesmo que $O(1) + O(2) + \ldots + O(n)$ (que não possui uma interpretação clara). A notação assintótica também pode aparecer do lado esquerdo de uma equação: $2n^2 + O(n) = O(n^2)$. Neste caso, independentemente da forma como as funções anônimas são escolhidas do lado esquerdo da equação, existe uma forma de escolher funções anônimas do lado direito da equação de forma que a equação se verifique. No caso do exemplo acima, temos que para qualquer $f(n) = O(n)$, existe uma função $g(n) = O(n^2)$ tal que $2n^2 + f(n) = g(n), \forall n$.

Equações também podem ser encadeadas como em $2n^2 + 3n + 1 = 2n^2 + O(n) = O(n^2)$, e podem ser interpretadas separadamente de acordo com as regras anteriores. Assim, a primeira equação nos diz que existe alguma função $f(n) = O(n)$ para a qual a equação se verifica para todo n. A segunda equação nos diz que para toda função $g(n) = O(n)$, existe uma função $h(n) = O(n^2)$ tal que a equação se verifica para todo n. Este encadeamento é transitivo, ou seja, podemos concluir que $2n^2+3n+1 = O(n^2)$.

O lema a seguir nos permite utilizar limites para utilizar a notação assintótica:

Lema 3. *Uma função $f(n) = O(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, incluindo o caso em que $c = 0$.*

Demonstração. Exercício.

Solução

Pela definição de limite, temos $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ significa que $\forall \epsilon > 0, \exists \delta > 0 : |\frac{f(n)}{g(n)} - c| < \epsilon, \forall n > \delta$. Ou seja, que $-\epsilon < \frac{f(n)}{g(n)} - c < \epsilon, \forall n > \delta$, uma vez que $f(n)$ e $g(n)$ são funções não-negativas. Podemos então tomar $\epsilon = 1$ (ou igual a qualquer outro valor positivo), e considerando apenas a desigualdade da direita, temos que $\frac{f(n)}{g(n)} < c + 1, \forall n > \delta \iff f(n) < (c+1).g(n), \forall n > \delta \implies f(n) < (c+1).g(n), \forall n \geq \delta + 1 \implies f(n) \leq (c+1).g(n), \forall n > \delta + 1$. Assim, temos que existem constantes positivas c_0 e n_0 tais que $f(n) \leq c_0.g(n), \forall n \geq n_0 \iff f(n) = O(g(n))$.

□

Observe que a outra direção do lema anterior não vale: de fato, considere $f(n) = n$ e $g(n) = 2^{\lfloor \lg n \rfloor}$, onde $\lg n$ é o logaritmo de n na base 2. Temos que $f(n) = O(g(n))$ porque $f(n) = n = 2^{\lg n} \leq 2^{\lfloor \lg n \rfloor + 1} = 2.2^{\lfloor \lg n \rfloor} = 2.g(n), \forall n$. No entanto, o limite $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ não existe, já que o quociente $\frac{f(n)}{g(n)}$ oscila.

Teorema 4. *1. Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$*, onde* c *é uma constante real positiva, então* $f(n) = O(g(n))$ *e* $g(n) = O(f(n))$*;*

2. Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ *então* $f(n) = O(g(n))$*, mas* $g(n) \neq O(f(n))$*;*

3. Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ *então* $f(n) \neq O(g(n))$*, mas* $g(n) = O(f(n))$*.*

Demonstração. Exercício.

□

No caso de InsertionSort, a análise do pior caso nos dá a função

$$T_w(n) \quad = \quad c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\tfrac{(n-1).n}{2}) + (c_5 + c_6).(\tfrac{(n-2).(n-1)}{2})$$

que é $O(n^2)$.

Como exercício, mostre em detalhes que a complexidade do pior caso de InsertionSort é $O(n^2)$.

Assim, considerando as expressões (ou polinômios) construídas(os) até agora, observamos que a classe de complexidade é obtida considerando-se o monômio de maior grau sem levar em conta o coeficiente. Portanto, a construção do polinômio a partir do custo de cada linha do algoritmo não é uma estratégia eficiente porque no final consideraremos apenas a parcela mais significativa, ou seja, o monômio de maior grau. Vamos então buscar diretamente a parte do algoritmo que nos dá este monômio de maior grau. Observando a Tabela 9 concluímos que o termo quadrático vem da linha 4, mais precisamente da comparação $A[i] > key$ que é executada em cada iteração do laço **for**. Então podemos fazer uma análise bem mais direta do que a feita anteriormente para chegarmos à mesma conclusão. Como durante a i-ésima iteração do laço **for**, a linha 4 é executada i vezes, temos:

$$T_w(n) = \sum_{i=1}^{n-1} i = \frac{(n-1).n}{2} = O(n^2).$$

A análise do melhor caso também pode ser feita da mesma forma considerando que a cada iteração do laço **for**, a linha 4 é executada uma única vez:

$$T_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n).$$

Da mesma forma, na busca sequencial o custo linear do pior caso pode ser obtido calculando diretamente o número de comparações feitas na linha 2. A notação O nos dá uma cota superior para o custo de execução de algoritmos, mas ela também pode ser utilizada para estabelecer uma cota para a complexidade de espaço utilizado durante a execução de um algoritmo. Tanto a busca sequencial quanto o algoritmo de ordenação por inserção não necessitam de espaço adicional de armazenamento, e portanto, em ambos os casos a complexidade é constante, ou seja, é igual a $O(1)$. Dizemos que algoritmos de ordenação que não demandam espaço adicional fazem a ordenação *in place*. Posteriormente estudaremos algoritmos que necessitam de espaço adicional.

Depois de alguns exercícios, e de apresentarmos mais alguns detalhes sobre a notação assintótica, estudaremos um pouco da chamada análise do caso médio. A análise do melhor caso nos dá uma ideia de situações específicas em que o algoritmo tem a melhor performance possível, mas a análise do melhor caso não costuma ser muito informativa e normalmente não é relevante. A análise do pior caso, por outro lado, tem bastante relevância e será explorada exaustivamente nas próximas seções. Ela é importante porque nos fornece o pior cenário possível para o algoritmo. Com isto sabemos que o algoritmo não pode ter um comportamento menos eficiente do que o apresentado pela análise do pior caso. No entanto, esta análise pode ser excessivamente pessimista considerando uma situação mais realista. Por exemplo, pode ser que o pior cenário só ocorra para uma ou duas entradas específicas dentre uma infinidade de possibilidades igualmente possíveis. A análise do caso médio pode nos fornecer uma ideia da eficiência do algoritmo considerando uma média dentre todos os tempos de execução possíveis, o que não corresponde à média entre as análises do melhor e pior casos.

Por fim, é importante ter em mente que a notação assintótica nos permite analisar a taxa de crescimento, ou ordem de crescimento do tempo de execução de um algoritmo, e portanto as simplificações feitas na obtenção da cota superior não devem ser esquecidas em situações práticas. Por exemplo, considere dois algoritmos A e B com complexidades, respectivamente, iguais a $O(n^2)$ e $O(n^3)$. Qual dos dois algoritmos é mais eficiente? Para valores grandes de n certamente o algoritmo A é mais eficiente, mas devemos levar em consideração que no cálculo destas classes de complexidade diversas constantes foram ignoradas. Se soubéssemos, por exemplo, que o algoritmo A realiza $100.n^2$ operações, enquanto que o algoritmo B realiza $5.n^3$ operações para resolver o mesmo problema, então agora sabemos que para $n < 20$ o algoritmo B é mais eficiente.

Na tabela abaixo, resumimos as análises feitas até agora:

Algoritmo	tempo (melhor caso)	tempo (pior caso)	espaço
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$

Uma ferramenta bastante útil na análise assintótica é conhecida como *regra do máximo*:

$$O(f(n) + g(n)) = O(\max(f(n), g(n))) \tag{3.1}$$

Exercício 24. *Prove a regra do máximo.*

Solução

Definimos $\max(f(n), g(n)) = \begin{cases} f(n), & se\ f(n) \geq g(n); \\ g(n), & se\ f(n) < g(n). \end{cases}$ *Ou seja, para cada valor de x tomamos o maior dos valores entre* $f(n)$ *e* $g(n)$*. Inicialmente, mostraremos que* $O(f(n) + g(n)) \subseteq O(max(f(n), g(n)))$*. Seja* $t(n) \in O(f(n) + g(n))$*, isto é,* $t(n)$ *é um elemento arbitrário do conjunto* $O(f(n) + g(n))$*. Por definição, existem constantes positivas* c *e* n_0 *tais que* $t(n) \leq c.(f(n) + g(n)), \forall n \geq n_0$*. Adicionalmente, temos para todo* n*, que* $f(n) \leq \max(f(n), g(n))$ *e* $g(n) \leq \max(f(n), g(n))$*, e somando as duas desigualdades, temos*

$$f(n) + g(n) \leq 2.\max(f(n), g(n)), \forall n \tag{3.2}$$

Logo, $t(n) \leq 2.c.\max(f(n), g(n)), \forall n \geq n_0$ *pela equação (3.2), e portanto,* $t(n) \in O(\max(f(n), g(n)))$*. Reciprocamente, observe que*

$$\max(f(n), g(n)) \leq f(n) + g(n) \tag{3.3}$$

e seja, $t(n) \in O(\max(f(n), g(n)))$*, um elemento arbitrário. Então, por definição, existem constantes positivas* c *e* n_0 *tais que* $t(n) \leq c.\max(f(n), g(n)), \forall n \geq n_0$*, e pela equação (3.3), concluímos que* $t(n) \leq c.(f(n) + g(n)), \forall n \geq n_0$*. Portanto,* $t(n) \in O(f(n) + g(n))$ *e isto completa a prova.*

Exercício 25. *Considere os pseudocódigos a seguir e complete a tabela:*

```
1 for i = 0 to n − 2 do
2 |   for j = 0 to n − 2 − i do
3 |   |   if A[j + 1] < A[j] then
4 |   |   |   swap A[j] and A[j + 1];
5 |   |   end
6 |   end
7 end
```

Algoritmo 4: BubbleSort($A[0..n-1]$)

```
1 for i = 0 to n − 2 do
2 |   min ← i;
3 |   for j = i + 1 to n − 1 do
4 |   |   if A[j] < A[min] then
5 |   |   |   min ← j;
6 |   |   end
7 |   end
8 |   swap A[i] and A[min];
9 end
```

Algoritmo 5: SelectionSort($A[0..n-1]$)

Algoritmo	*tempo (melhor caso)*	*tempo (pior caso)*	*espaço*
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$
Bubble sort			
Selection sort			

Exercício 26. *Mostre que* $n = O(n^2)$*.*

Exercício 27. *Mostre que* $100n + 5 = O(n^2)$*.*

Exercício 28. *Mostre que* $\frac{n(n-1)}{2} = O(n^2)$*.*

Exercício 29. *Mostre que* $n^3 \neq O(n^2)$.

Solução

Suponha que $n^3 = O(n^2)$. *Por definição existem constantes positivas* c *e* n_0 *tais que* $n^3 \leq c.n^2, \forall n \geq n_0 \iff n \leq c, \forall n \geq n_0$ *(absurdo).*

Exercício 30. *Sejam* $f(n), g(n)$ *e* $h(n)$ *funções dos inteiros não-negativos nos reais positivos. Mostre que se* $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$ *então* $f(n) = O(h(n))$.

Assim, como $O(g(n))$ estabelece uma cota superior para funções, o conjunto $\Omega(g(n))$ estabelece uma cota inferior para as funções:

Definição 5. *Seja* $g(n)$ *uma função dos inteiros não-negativos nos reais positivos. Então* $\Omega(g(n))$ *é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real* $c > 0$ *e uma constante inteira* $n_0 > 0$ *satisfazendo a desigualdade* $c \cdot g(n) \leq f(n), \forall n \geq n_0$. *Alternativamente,* $\Omega(g(n)) = \{f(n) :$ *existem constantes positivas* c *e* n_0 *tais que* $0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0.\}$

Quando dizemos que o tempo de execução de um algoritmo é $\Omega(g(n))$, queremos dizer que independentemente da entrada de tamanho n, o tempo de execução desta entrada é pelo menos uma constante multiplicada por $g(n)$ para n suficientemente grande. Ou seja, estamos fornecendo uma cota inferior no melhor caso. Por exemplo, no melhor caso, o algoritmo InsertionSort é $\Omega(n)$, e portanto, o tempo de execução do algoritmo InsertionSort está entre $\Omega(n)$ e $O(n^2)$. A definição alternativa para o conjunto $\Omega(g(n))$ em termos de limites é dada pelo lema a seguir:

Lema 6. *Uma função* $f(n) = \Omega(g)$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, *incluindo o caso em que o limite é igual a* ∞.

Demonstração. Exercício.

□

A forma mais precisa de expressar o comportamento assintótico de um algoritmo é fornecendo cotas superiores e inferiores ao mesmo tempo. No parágrafo anterior, apresentamos uma cota superior e uma cota inferior para o algoritmo InsertionSort. No entanto, estas cotas são de classes diferentes, o conjunto $\Theta(g(n))$, definido a seguir, é utilizado quando ambas as cotas são da mesma classe.

Definição 7. *Seja* g *uma função dos inteiros não-negativos nos reais positivos. Então* $\Theta(g(n))$ *é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem constantes reais positivas* c_1 *e* c_2, *e uma constante inteira* $n_0 > 0$ *satisfazendo a desigualdade* $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$. *Alternativamente,* $\Theta(g(n)) = \{f(n) :$ *existem constantes positivas* c_1, c_2 *e* n_0 *tais que* $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.\}$

Como qualquer constante pode ser vista como um polinômio de grau 0, podemos representar funções constantes como $\Theta(n^0)$, ou simplesmente, $\Theta(1)$. O lema a seguir apresenta uma caracterização do conjunto $\Theta(g(n))$ em termos de limite:

Lema 8. *Uma função* $f(n) = \Theta(g(n))$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, *para alguma constante* $0 < c < \infty$.

Demonstração. Exercício.

□

Teorema 9. *1. Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$*, onde* c *é uma constante real positiva, então* $f(n) = \Theta(g(n))$*;*

2. Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ *então* $f(n) = O(g(n))$*, mas* $f(n) \neq \Theta(g(n))$*;*

3. Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ *então* $f(n) = \Omega(g(n))$*, mas* $f(n) \neq O(g(n))$*.*

Demonstração. Exercício.

□

Exercício 31. *Prove que* $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$ *para qualquer inteiro* $k \geq 0$ *fixado.*

Solução

A prova é dividida em duas etapas. Inicialmente, mostraremos que $\sum_{i=1}^{n} i^k = O(n^{k+1})$*, ou seja, mostraremos que existem constantes positivas* c *e* n_0 *tais que* $\sum_{i=1}^{n} i^k \leq c.(n^{k+1}), \forall n \geq n_0 \iff 1^k + 2^k + 3^k + \ldots + n^k \leq c.(n^{k+1}), \forall n \geq n_0$*. De fato,* $1^k + 2^k + 3^k + \ldots + n^k \leq n^{k+1} = n.n^k$*. Assim, tomando* $c = 1$ *e* n_0 *qualquer, temos a desigualdade desejada.*

Teorema 10. *Dadas funções* $f(n)$ *e* $g(n)$*, temos que* $f(n) = \Theta(g(n))$ *se, e somente se,* $f(n) = O(g(n))$ *e* $f(n) = \Omega(g(n))$*.*

Demonstração. Exercício.

□

Lema 11. *1.* $f(n) = O(g(n))$ *se, e somente se* $g(n) = \Omega(f(n))$*;*

2. Se $f(n) = \Theta(g(n))$ *então* $g(n) = \Theta(f(n))$*;*

3. Θ *define uma relação de equivalência sobre as funções. Cada conjunto* $\Theta(f(n))$ *é uma classe de equivalência que chamamos de classe de complexidade;*

4. $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$*;*

5. $\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$*;*

Definição 12. *Seja* $g(n)$ *uma função dos inteiros não-negativos nos reais positivos. Definimos,* $o(g(n)) = \{f(n) :$ *para qualquer constante positiva* c*, existe uma constante positiva* n_0 *tal que* $0 \leq f(n) < c \cdot g(n), \forall n \geq n_0.\}$

Lema 13. *Uma função* $f(n) = o(g)$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$*.*

Definição 14. *Seja* $g(n)$ *uma função dos inteiros não-negativos nos reais positivos. Definimos,* $\omega(g(n)) = \{f(n) :$ *para qualquer constante positiva* c*, existe uma constante positiva* n_0 *tal que* $0 \leq c \cdot g(n) < f(n), \forall n \geq n_0.\}$

Lema 15. *Uma função* $f(n) = \omega(g(n))$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, *se este limite existir.*

Lema 16. *Se* $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$ *então* $f(n) = O(h(n))$, *ou seja, a notação* O *é transitiva. Também são transitivos* Ω, Θ, o *e* ω.

Teorema 17. $\lg n = o(n^\alpha), \forall \alpha > 0$. *Ou seja, a função logaritmo cresce mais lentamente do que qualquer potência de* n *(incluindo potências fracionárias)*

Prova

Pelo Lema (13), basta mostrarmos que $lim_{n\to\infty} \frac{\lg n}{n^\alpha} = 0$. *De fato,* $lim_{n\to\infty} \frac{\lg n}{n^\alpha} = lim_{n\to\infty} \frac{\frac{1}{n}}{\alpha.n^{\alpha-1}} = lim_{n\to\infty} \frac{1}{\alpha.n^\alpha} = 0$.

Teorema 18. $n^k = o(2^n), \forall k > 0$. *Ou seja, potências de* n *crescem mais lentamente que a função exponencial* 2^n. *Mais ainda, potências de* n *crescem mais lentamente do que qualquer função exponencial* $c^n, c > 1$.

Exercício 32. *Mostre que* $\frac{n^2}{2} - 3n = \Theta(n^2)$.

Exercício 33. *Mostre que* $6n^3 \neq \Theta(n^2)$.

Exercício 34. *Sejam* $f(n)$, $g(n)$ *e* $h(n)$ *funções não-negativas tais que* $f(n) = O(h(n))$ *e* $g(n) = O(h(n))$. *Prove que* $f(n) + g(n) = O(h(n))$.

Capítulo 4

O algoritmo *mergesort*

Algoritmos recursivos desempenham um papel fundamental em Computação. O algoritmo de ordenação *mergesort* é um exemplo de algoritmo recursivo, que se caracteriza por dividir o problema original em subproblemas que, por sua vez, são resolvidos recursivamente. As soluções dos subproblema são então combinadas para gerar uma solução para o problema original. Este paradigma de projeto de algoritmo é conhecido com *divisão e conquista*. Este algoritmo foi inventado por J. von Newmann em 1945.

O algoritmo *mergesort* é um algoritmo de ordenação que utiliza a técnica de divisão e conquista, que consiste das seguintes etapas:

1. **Divisão**: O algoritmo divide a lista (ou vetor) l recebida como argumento ao meio, obtendo as listas l_1 e l_2;

2. **Conquista**: O algoritmo é aplicado recursivamente às listas l_1 e l_2 gerando, respectivamente, as listas ordenadas l'_1 e l'_2;

3. **Combinação**: O algoritmo combina as listas l'_1 e l'_2 através da função *merge* que então gera a saída do algoritmo.

Por exemplo, ao receber a lista $(4 :: 2 :: 1 :: 3 :: nil)$, este algoritmo inicialmente divide esta lista em duas sublistas, a saber $(4 :: 2 :: nil)$ e $(1 :: 3 :: nil)$. O algoritmo é aplicado recursivamente às duas sublistas para ordená-las, e ao final deste processo, teremos duas listas ordenadas $(2 :: 4 :: nil)$ e $(1 :: 3 :: nil)$. Estas listas são, então, combinadas para gerar a lista de saída $(1 :: 2 :: 3 :: 4 :: nil)$.

```
1 if p < r then
2 |   q = ⌊(p+r)/2⌋;
3 |   mergesort(A, p, q);
4 |   mergesort(A, q + 1, r);
5 |   merge(A, p, q, r);
6 end
```

Algoritmo 6: mergesort(A, p, r)

A etapa de combinar dois vetores ordenados (algoritmo *merge*) é a etapa principal do algoritmo *mergesort*. O procedimento $merge(A, p, q, r)$ descrito a seguir recebe como argumentos o vetor A, e os índices p, q e r tais que $p \leq q < r$. O procedimento assume que os subvetores $A[p..q]$ e $A[q+1..r]$ estão ordenados.

```
1  n_1 = q − p + 1 ;                                   // Qtd. de elementos em A[p..q]
2  n_2 = r − q ;                                       // Qtd. de elementos em A[q + 1..r]
3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays;
4  for i = 1 to n_1 do
5  |  L[i] = A[p + i − 1];
6  end
7  for j = 1 to n_2 do
8  |  R[j] = A[q + j];
9  end
10 L[n_1 + 1] = ∞;
11 R[n_2 + 1] = ∞;
12 i = 1;
13 j = 1;
14 for k = p to r do
15 |  if L[i] ≤ R[j] then
16 |  |  A[k] = L[i];
17 |  |  i = i + 1;
18 |  end
19 |  else
20 |  |  A[k] = R[j];
21 |  |  j = j + 1;
22 |  end
23 end
```

Algoritmo 7: merge(A, p, q, r)

Exercício 35. *Prove que o algoritmo merge é correto.*

Exercício 36. *Prove que o algoritmo mergesort é correto.*

Exercício 37. *Faça a análise assintótica do algoritmo merge.*

Exercício 38. *Faça a análise assintótica do algoritmo mergesort.*

4.0.1 Equações de recorrência

Nesta seção estudaremos as equações de recorrência utilizadas no paradigma de divisão de conquista [9]:

Definição 19. *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* eventualmente não-decrescente *se existir um número inteiro n_0 tal que $f(n)$ é não-decrescente no intervalo $[n_0, \infty)$, ou seja,*

$$f(n_1) \leq f(n_2), \forall n_2 > n_1 \geq n_0.$$

Definição 20. *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* suave *se for eventualmente não-decrescente e*

$$f(2.n) = \Theta(f(n))$$

Teorema 21. *Sejam $f(n)$ uma função suave, e c e n_0 constantes positivas. Se $f(2n) \leq c.f(n), \forall n \geq n_0$ então $f(2^k n) \leq c^k.f(n), \forall n \geq n_0$ e $k \geq 1$.*

Teorema 22. *Seja $f(n)$ uma função suave. Então para qualquer $b \geq 2$ fixado,*

$$f(b.n) = \Theta(f(n))$$

O teorema a seguir é conhecido como *regra da suavização*

Teorema 23. *Seja $T(n)$ uma função eventualmente não-decrescente, e $f(n)$ uma função suave. Se $T(n) = \Theta(f(n))$ para valores de n que são potências de b ($b \geq 2$), então*

$$T(n) = \Theta(f(n)), \forall n.$$

A regra da suavização nos permite expandir a informação sobre a ordem de crescimento estabelecida para $T(n)$ de um subconjunto de valores (potências de b) para o domínio inteiro. O teorema a seguir é um resultado muito útil nesta direção conhecido como *teorema mestre*:

Teorema 24. *Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência*
$T(n) = a.T(n/b) + f(n), \quad$ *para* $n = b^k, k = 1, 2, 3, \ldots$
$T(1) = c$
onde $a \geq 1, b \geq 2$ e $c \geq 0$. Se $f(n) = \Theta(n^d)$, onde $d \geq 0$, então

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{se } a > b^d \\ \Theta(n^d . \lg n), & \text{se } a = b^d \\ \Theta(n^d), & \text{se } a < b^d \end{cases}$$

Demonstração. Considere que $f(n) = n^d$. Aplicando o método da substituição para a recorrência do teorema, obtemos:

$$T(b^k) = a^k.[T(1) + \sum_{j=1}^{k} f(b^j)/a^j]$$

Como $a^k = a^{\log_b n} = n^{\log_b a}$, podemos reescrever a equação acima como:

$$T(n) = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} f(b^j)/a^j]$$

e para $f(n) = n^d$, temos:

$$T(n) = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} (b^j)^d/a^j] = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} (b^d/a)^j]$$

A soma acima forma uma série geométrica, e portanto:

$$\sum_{j=1}^{\log_b n} (b^d/a)^j = (b^d/a)\frac{(b^d/a)^{\log_b n} - 1}{(b^d/a) - 1}, \text{ se } b^d \neq a.$$

Quando $b^d \neq a$, temos que $\sum_{j=1}^{\log_b n} (b^d/a)^j = \log_b n$. Agora basta analisarmos cada um dos casos: $a < b^d$, $a > b^d$ e $a = b^d$.

□

Apresentaremos agora uma versão um pouco mais geral do teorema mestre[5]. Consideraremos como anteriormente uma recorrência da forma:

$$T(n) = a.T(n/b) + f(n)$$

on $a \geq 1$ e $b > 1$ são constantes, e $f(n)$ é uma função assintoticamente positiva.

Teorema 25. *Sejam $a \geq 1$ e $b \geq 2$ constantes, $f(n)$ uma função assintoticamente positiva, e $T(n)$ definida nos inteiros não-negativos pela recorrência $T(n) = a.T(n/b) + f(n)$, onde n/b deve ser interpretado como $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Então $T(n)$ tem as seguintes cotas assintóticas:*

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;*

2. *Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a}.\lg n)$;*

3. *Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $a.f(n/b) \leq c.f(n)$ para alguma constante $c < 1$, então para todo n suficientemente grande, temos que $T(n) = \Theta(f(n))$.*

A prova será dividida em três lemas, onde inicialmente consideraremos que n é potência de b.

Lema 26. *Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função não-negativa definida para potências de b. Defina $T(n)$ para potências de b pela recorrência:*

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1; \\ a.T(n/b) + f(n), & \text{se } n = b^i \end{cases}$$

onde i é um inteiro positivo. Então

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j.f(n/b^j).$$

Demonstração. Analise a árvore de recorrência da equação dada.

□

Em termos da árvore de recorrência, os três casos do teorema mestre correspondem aos casos onde o custo total da árvore é:

1. dominado pelo custo das folhas;

2. uniformemente distribuído ao longo da árvore;

3. dominado pelo custo da raiz.

Lema 27. *Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função não-negativa definida para potências de b. A função $g(n)$ definida para potências de b por:*

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j . f(n/b^j).$$

tem as seguintes cotas assintóticas para potências de b:

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $g(n) = O(n^{\log_b a})$;*

2. *Se $f(n) = \Theta(n^{\log_b a})$, então $g(n) = \Theta(n^{\log_b a} . \lg n)$;*

3. *Se $a.f(n/b) \leq c.f(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande, então $g(n) = \Theta(f(n))$.*

Demonstração. Exercício.

□

Exercício 39. *Resolva as seguintes relações de recorrência:*

1. $T(1) = 1$, $T(n) = 3T(n/2) + n^2$, $n \geq 2$

2. $T(1) = 1$, $T(n) = 2T(n/2) + n$, $n \geq 2$

3. $T(1) \in \Theta(1)$, $T(n) = 3T(n/3 + 5) + n/2$

4. $T(1) = 1$, $T(n) = 2T(n-1) + 1$, $n \geq 2$

5. $T(1) \in \Theta(1)$, $T(n) = 9T(n/3) + n$

6. $T(1) \in \Theta(1)$, $T(n) = T(2n/3) + 1$

7. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + 1$

8. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + \sqrt{n}$

9. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + \sqrt{n}\lg^2 n$

10. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + n$

11. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + n^2$

12. $T(1) \in \Theta(1)$, $T(n) = 3T(n/2) + nln(n)$

13. $T(1) \in \Theta(1)$, $T(n) = 3T(n/4) + nln(n)$

14. $T(1) \in \Theta(1)$, $T(n) = 2T(n/2) + nln(n)$

15. $T(1) \in \Theta(1)$, $T(n) = 2T(n/2) + n/ln(n)$

16. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + 1/n$

17. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + ln(n)$

18. $T(1) \in \Theta(1)$, $T(n) = \sqrt{n}T(\sqrt{n}) + n$

19. $T(n) = 8T(n/2) + \Theta(n^2)$

20. $T(n) = 8T(n/2) + \Theta(1)$

21. $T(n) = 7T(n/2) + \Theta(n^2)$

Índice Remissivo

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

[10] Udi Manber. Using Induction To Design Algorithms. *Communications of the ACM*, 31(11):1300–1313, 1988.

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

[12] 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.

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