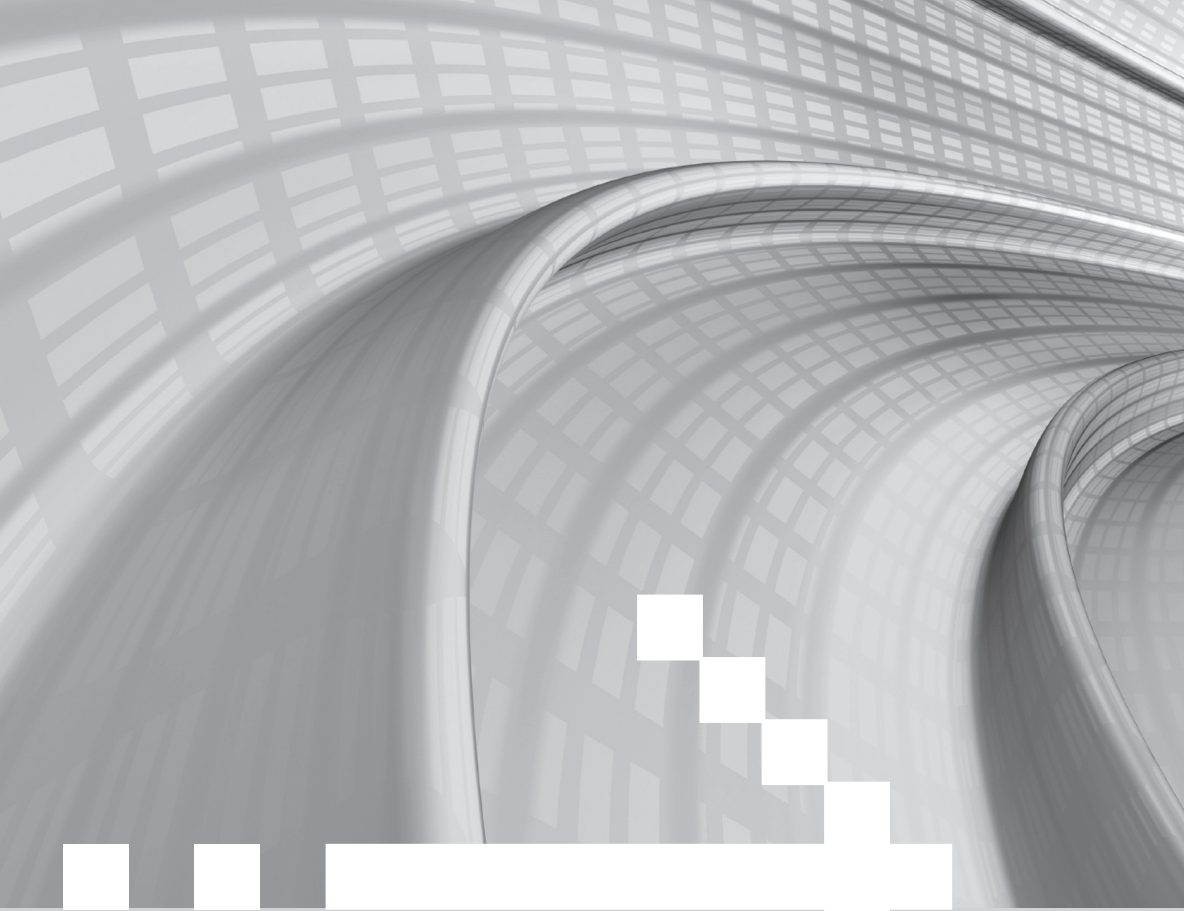

capítulo 1

introdução

Os algoritmos são o cerne da computação. Este capítulo introdutório procura ressaltar a importância da complexidade e dos métodos de projeto e análise de algoritmos. Partindo da ideia de que um programa codifica um algoritmo para ser executado em um computador e assim resolver um problema, é fundamental que os resultados sejam produzidos com dispêndios de tempo e de memória razoáveis. É a análise da complexidade que permite a rejeição de algoritmos mais simples, cujo esforço de computação tornam impraticáveis e permite a escolha de algoritmos mais elaborados e eficientes. Daí depreende-se a importância de projeto e análise de algoritmos, com ênfase em complexidade.

1.1 a complexidade no desempenho de algoritmos

Um algoritmo (por exemplo, um programa) é um procedimento, consistindo em um conjunto de regras não ambíguas, as quais especificam, para cada entrada, uma sequência finita de operações, terminando com uma saída correspondente.

Um algoritmo resolve um problema quando, para qualquer entrada, produz uma resposta correta, se forem concedidos tempo e memória suficientes para sua execução. O fato de um algoritmo resolver (teoricamente) um problema não significa que seja aceitável na prática. Os recursos de espaço e tempo requeridos têm grande importância em casos práticos.

tabela 1.1.1 Tamanho do problema × tempo de execução

n	Método de Cramer	Método de Gauss
2	22 μs	50 μs
3	102 μs	150 μs
4	456 μs	353 μs
5	2,35 ms	666 μs
10	1,19 min.	4,95 ms
20	15225 séculos	38,63 ms
40	5.10^{33} séculos	0,315 s

Às vezes, o algoritmo mais imediato está longe de ser razoável em termos de eficiência. Um exemplo é o caso da solução de sistemas de equações lineares. O método de Cramer, calculando o determinante por sua definição, requer dezenas de milhões de anos para resolver um sistema 20 × 20. Um sistema como esse pode ser resolvido em tempo razoável pelo método de Gauss. A tabela 1.1.1 mostra o desempenho desses dois algoritmos que calculam o determinante de uma matriz n × n, considerando tempos de operações de um computador real.

O crescente avanço tecnológico, permitindo a criação de máquinas cada vez mais rápidas, pode, ingenuamente, parecer ofuscar a importância da complexidade de tempo de um algoritmo. Entretanto, acontece exatamente o inverso. As máquinas, tornando-se mais rápidas, passam a poder resolver problemas maiores, e é a complexidade do algoritmo que determina o novo tamanho máximo de problema que pode ser resolvido. Para um algoritmo rápido, qualquer melhoria na velocidade de execução das operações básicas é sentida, e o conjunto de problemas que podem ser resolvidos por ele aumenta sensivelmente. Esse impacto é menor no caso de algoritmos menos eficientes.

Para ilustrar, consideremos o impacto de um aumento de velocidade sobre alguns algoritmos. Inicialmente, examinemos um algoritmo com tempo linear: o tempo de execução é proporcional ao tamanho da entrada. Suponhamos que, em uma máquina, num certo período máximo de tempo tolerável, ele resolve problemas de tamanho máximo x_1. Em um computador

dez vezes mais rápido, o mesmo algoritmo, no mesmo tempo, resolverá um problema de tamanho dez vezes maior, isto é, $10\ x_1$.

Considere agora um algoritmo com tempo quadrático, levando tempo proporcional a n^2 para uma entrada de tamanho n. Suponha um problema cujo tamanho máximo de problema que pode ser resolvido num tempo t na máquina mais lenta é x_3: ${x_3}^2 = t$. Agora suponha a máquina dez vezes mais rápida, ou equivalentemente, um tempo 10t na máquina mais lenta. O tamanho de problema resolvido será y, tal que

$$y^2 = 10t \therefore y^2 = 10\,(x_3)^2 \therefore y = x_3\sqrt{10}.$$

Portanto, agora y é aproximadamente $3{,}16\ x_3$. Finalmente, consideremos um algoritmo exponencial, levando tempo 2^n para uma entrada de tamanho n. Se x_5 é o tamanho máximo de problema resolvível num tempo t na máquina mais lenta e y na máquina mais rápida, tem-se

$$2^{x_5} = t \text{ e } 2^y = 10t \therefore 2^y = 10 \cdot 2^{x_5} \cdot \therefore y = \log_2 10 + x_5 = x_5 + 3{,}3.$$

O avanço tecnológico da máquina foi mais bem aproveitado pelo primeiro algoritmo. Já um algoritmo de complexidade de tempo exponencial, sob esse aspecto, praticamente não tira proveito da rapidez da segunda máquina, pois, se o tamanho máximo de problema resolvível na máquina lenta é x_5, na máquina mais rápida será

$$x_5 + 3{,}3 \qquad \text{(isto é, } x_5 + \log_2 10\text{)}.$$

A tabela 1.1.2 completa esse estudo com outros casos.

tabela 1.1.2 Complexidade do algoritmo × tamanho máximo de problema resolvível

Complexidade de tempo	Tamanho máximo de problema resolvível na máquina lenta	Tamanho máximo de problema resolvível na máquina rápida
$\log_2 n$	x_0	$(x_0)^{10}$
x	x_1	$10\ x_1$
$n.\log_2 n$	x_2	$10\ x_2$(para x_2 grande)
n^2	x_3	$3{,}16\ x_3$
n^3	x_4	$2{,}15\ x_4$
2^n	x_5	$x_5 + 3{,}3$
3^n	x_6	$X_6 + 2{,}096$

exercício 1.1.1 Verifique os valores da tabela 1.1.2 para $\log_2 n$, n, $n.\log_2 n$, n^3 e 3^n. ↲

Supondo-se que um computador execute uma operação em 1 s, chega-se à tabela 1.1.3, que descreve os tamanhos limites de problemas resolvíveis por algoritmos de diferentes complexidades.

tabela 1.1.3 Complexidade do algoritmo × tempo de execução

	Tamanho de problema executável		
Complexidade de tempo	1 segundo	1 minuto	1 hora
$\log_2 n$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{3,6 \cdot 10^9}$
x	10^6	6.10^7	$3,6.10^9$
$n\log_2 n$	62.746	$2,8.10^6$	$1,3.10^8$
n^2	10^3	$7,746.10^3$	6.10^4
n^3	10^2	$3,9.10^2$	$1,5.10^3$
2^n	20	25	32
3^n	13	16	20

As tabelas 1.1.1, 1.1.2 e 1.1.3 mostram que a potencialidade, isto é, a capacidade de um algoritmo, é função de sua complexidade.

exercício 1.1.2 Considere dois algoritmos $\mathbf{a}_1$ e $\mathbf{a}_2$ com complexidades $8n^2$ e n^3. Qual o maior valor de n, para o qual o algoritmo $\mathbf{a}_2$ é mais eficiente que o algoritmo $\mathbf{a}_1$? ↲

exercício 1.1.3 Um algoritmo tem complexidade $2n^2$. Num certo computador, num tempo t, o algoritmo resolve um problema de tamanho 25. Imagine agora que você tem disponível um computador 100 vezes mais rápido. Qual o tamanho máximo de problema que o mesmo algoritmo resolve no mesmo tempo t no computador mais rápido? ↲

exercício 1.1.4 Considere o mesmo problema anterior para um algoritmo de complexidade 2^n. ↲

exercício 1.1.5 Suponha que uma empresa utiliza um algoritmo de complexidade n^2 que, em um tempo t, na máquina disponível, resolve um problema de tamanho x. Suponha que o tamanho do problema a ser resolvido aumentou em 20%, mas o tempo de resposta deve ser mantido. Para isso, a empresa pretende trocar a máquina por uma mais rápida. Qual percentual de melhoria no tempo de execução das operações básicas é necessário para atingir sua meta? ↲

exercício 1.1.6 Suponha que, no problema anterior, ainda se queira reduzir em 50% o tempo de resposta. Qual a melhoria esperada para a nova máquina? ↲

1.2 ⇢ complexidade de algoritmos e de problemas

A complexidade vem ganhando destaque a ponto de que afirmações, como a de Aho, Hopcroft e Ullman (1974), de que a complexidade é o coração da computação, já não surpreendem. Assim, uma apresentação da complexidade como uma tarefa sistemática faz-se necessária para torná-la uma prática corriqueira no projeto de algoritmos.

1.2.1 metodologia de cálculo da complexidade

A análise da complexidade de um algoritmo é realizada, usualmente, de maneira muito particular, o que é compreensível, já que a complexidade é uma medida que tem parâmetros bem particulares do algoritmo. Apesar disso, algumas ideias são gerais, no sentido de que só dependem da estrutura do algoritmo. Conceitos e métodos nessa linha são apresentados no capítulo 3.

1.2.2 complexidade na fase de projeto do algoritmo

O estudo das técnicas no desenvolvimento de algoritmo permite prever alguns aspectos da complexidade do algoritmo resultante. Muitas dessas ideias são empiricamente conhecidas do programador experiente, mas complicadas de passar para um iniciante, por não estarem fundamentadas.

Mas, o projeto de algoritmos eficientes é sempre um propósito de qualquer projetista e é uma preocupação já nessa fase. Portanto, subsídios a respeito da complexidade, se puderem ser antecipados para essa fase, são muito bem-vindos.

No capítulo 5, alguns métodos de desenvolvimento de algoritmos são apresentados, e a complexidade dos algoritmos gerados é analisada.

1.2.3 a complexidade do problema e a intratabilidade

A complexidade também pode ser vista como uma propriedade do problema, o que significa dar uma medida independente do tratamento dado ao problema, independente do caminho percorrido na busca da solução, portanto independente do algoritmo. Alguns problemas são "bem comportados" e permitem chegar a limites de complexidade, como o problema de classificação, para qual é conhecida a ordem de complexidade mínima dos algoritmos que o resolvem.

Outros problemas parecem ficar tão difíceis de serem resolvidos para instâncias grandes que parecem tornar-se intratáveis. Nesse contexto, surgem os problemas NP-completos, que, segundo Cook (1983), podem também ser considerados como um limite de complexidade. Entretanto, esses problemas existem, são usados no dia a dia e, portanto, precisam ser tratados, e o que é ainda mais importante, precisam ser identificados. Esses problemas são apresentados no capítulo 6.

1.3 → revisão matemática

No cálculo da complexidade de um algoritmo, é necessário o uso de conceitos e resultados matemáticos, alguns dos quais serão aqui relembrados. Somatórios e produtórios são úteis no cálculo da complexidade envolvendo estruturas iterativas. Por definição, o fatorial n! é o

produto dos n primeiros naturais. Soma dos termos de uma progressão aritmética: a_0, $a_0 + r$, $a_0 + 2r$, . . . , $a_0 + n.r$.

$$\sum_{i=0}^{n} a_i = \frac{(a_0 + a_n) \cdot (n+1)}{2}$$

com $a_i = a_0 + i.r$

Por exemplo, a soma dos n primeiros naturais é:

$$\sum_{j=1}^{n} j = \frac{(1+n) \cdot n}{2}$$

exercício 1.3.1 Mostre que a soma dos n primeiros ímpares é;

$$\sum_{i=1}^{n} (2i+1) = (n+1)^2$$

↲

Soma dos termos de uma progressão geométrica: a, a . q, a . q^2, …, a . q^n

$$\sum_{i=0}^{n-1} a \cdot q^i = \frac{a \cdot (q^n - 1)}{q - 1} \quad \text{para } q \neq 1$$

exercício 1.3.2 Por que a expressão acima para a soma dos termos de uma progressão geométrica requer $q \neq 1$? Quanto é a soma

$$\sum_{i=0}^{n-1} a \cdot q^i, \quad \text{quando } q = 1?$$

↲

exercício 1.3.3 Mostre que

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \quad \text{e} \quad \sum_{i=0}^{n} m^i = \frac{m^{n+1} - 1}{m - 1} \quad \text{para } q \neq 1$$

↲

Outras somas:

$$\sum_{i=0}^{n} i^2 = \frac{1}{6} \cdot n \cdot (n+1) \cdot (2n+1) \quad \text{e} \quad \sum_{i=0}^{n} i\,2^i = 2 + (n-1) \cdot 2^{n+1}$$

Outras expressões interessantes são dadas a seguir.

Combinação:

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

Binômio de Newton:

$$(a+b)n = \sum_{i=0}^{n} \binom{n}{k} \cdot a^{n-1} \cdot b^{i}$$

Somas combinatoriais também são comuns em cálculos de complexidade:

$$\sum_{k=0}^{n} \binom{n}{k} \cdot (x-1)^{k} = x$$

exercício 1.3.4 Mostre que

$$\sum_{k=0}^{n} \binom{n}{k} = 2^{n}$$

↲

Logaritmos também são funções comuns, e suas propriedades merecem ser lembradas, pois serão úteis.

Por definição, $a^{\log_a n} = n$

Propriedade do produto: $\log_a(n \cdot m) = \log_a n + \log_a m$

Propriedade da divisão: $\log_a \left(\frac{n}{m}\right) = \log_a n - \log_a m$

Propriedade da potência: $\log_a(n^m) = m \cdot \log_a n$

Expoentes: $a^{\log_c b} = b^{\log_c a}$

Troca de base: $\log_a n = \log_b n \cdot \log_a b$

exercício 1.3.5 Demonstre a propriedade dos expoentes. ↲

Também são usados com frequência os conceitos de piso e teto.

Piso (*floor*) de n: $\lfloor r \rfloor := $ maior inteiro k tal que $k \leq r$.

Teto (*ceiling*) de n: $\lceil r \rceil := $ menor inteiro k tal que $k \geq r$

Por exemplo: $\lfloor 2{,}3 \rfloor = 2$ e $\lfloor 2 \rfloor = 2$, enquanto $\lceil 2{,}3 \rceil = 3$ e $\lceil 2 \rceil = 2$.

exercício 1.3.6 Mostre ou dê um contraexemplo $\lfloor r \rfloor \leq \lceil r \rceil$. Quando $\lceil r \rceil = \lfloor r \rfloor$? ↲

Uma desigualdade não trivial é

$$\sum_{i=1}^{n} \frac{1}{i} \leq \log n + 1$$

1.4 ⇢ resumo do capítulo

A finalidade deste capítulo é ressaltar a importância da complexidade e de métodos de projeto e análise de algoritmos.

Nem sempre o algoritmo mais imediato é viável na prática.

Algoritmos mais eficientes costumam tirar maior proveito de máquinas melhores.

A análise de complexidade de um algoritmo pode ser feita com base em sua estrutura.

Há métodos gerais de desenvolvimento de algoritmos com amplo espectro de aplicação.

A complexidade de um problema é a do melhor algoritmo imaginável (conhecido ou não) que possa resolvê-lo.

Fórmulas matemáticas envolvendo somatórios, como na seção 1.3, são úteis para a análise de complexidade de algoritmos.

Os exercícios adicionais a seguir tratam de questões similares às examinadas neste capítulo.

1.5 ⇢ exercícios adicionais

1.1 São dados um problema a ser resolvido e um tempo máximo de resposta aceitável x. O algoritmo disponível, que tem complexidade 4^n, resolve, nesse tempo, um problema de tamanho máximo n. Entretanto, o tamanho do problema a ser resolvido agora é 2n. Deseja-se um novo algoritmo que resolva o novo problema no mesmo tempo x. Qual deve ser a complexidade do novo algoritmo?

1.2 Seja um algoritmo **a** que requer $4n^2 - 3n$ operações para resolver um problema de tamanho n. Deseja-se, mantendo o mesmo tempo de resposta, resolver um problema 30% maior, desenvolvendo um algoritmo mais rápido. Qual deverá ser a complexidade do novo algoritmo?

1.3 No problema anterior, se o algoritmo tiver complexidade $3n^3$, qual deverá ser a complexidade do novo algoritmo? Será necessário mudar a ordem de complexidade? E se o algoritmo tiver complexidade $4n^4$?

1.4 Um algoritmo tem complexidade $9\ 2^n$. Um computador, num tempo t, usando tal algoritmo resolve um problema de tamanho máximo x. Suponha agora que você quer resolver um problema duas vezes maior, qual o tempo (em função de t) necessário?

1.5 Um algoritmo tem complexidade $\log_2 n$. Num certo computador, num tempo t, o algoritmo resolve um problema de tamanho 16. Imagine agora que você tem dispo-

nível um computador 3 vezes mais rápido. Qual o tamanho máximo de problema que o mesmo algoritmo resolve no mesmo tempo t no computador mais rápido?

1.6 Um algoritmo tem complexidade $3n^3$. Num certo computador, num tempo $t = 3000$, resolve um problema de tamanho x. Imagine agora que você tem disponível um algoritmo mais rápido, de complexidade $2n^2$ e uma máquina seis vezes mais rápida. Que tempo precisará para resolver o mesmo problema de tamanho x?

1.7 Suponha que uma empresa utiliza um algoritmo de complexidade n^2 que, em um tempo t, na máquina disponível, resolve um problema de tamanho x. Suponha que o tamanho do problema a ser resolvido aumentou em 30%, mas o tempo de resposta deve ser mantido. Para isso, a empresa pretende trocar a máquina por uma mais rápida.

a) Qual percentual de melhoria no tempo de execução das operações básicas é necessário para atingir sua meta?
b) Considere o mesmo problema, mas com um algoritmo de complexidade $5n^2$.
c) Que conclusão se pode tirar dos dois itens anteriores?

1.8 Um algoritmo tem complexidade $2n^3$. Num certo computador, num tempo $t = 1600$, resolve um problema de tamanho x. Imagine agora que você tem disponível um algoritmo mais rápido, de complexidade $2n^2$ e uma máquina cinco vezes mais rápida. Que tempo precisará para resolver o mesmo problema de tamanho x?

1.9 Um algoritmo tem complexidade $2n^2$. Num certo computador, num tempo t, o algoritmo resolve um problema de tamanho x. Imagine agora que você tem disponível um computador 30 vezes mais rápido. Que parcela do tempo t precisará para resolver um problema 3 vezes maior?

1.10 Um algoritmo tem complexidade $\log_2 n$. Num certo computador, num tempo t, o algoritmo resolve um problema de tamanho 16. Imagine agora que você tem disponível um computador 8 vezes mais rápido. Qual o tamanho máximo de problema que o mesmo algoritmo resolve no mesmo tempo t no computador mais rápido?

1.11 Um algoritmo **a** tem complexidade $2n^2$ e o algoritmo **aa** complexidade 2^n. Num certo computador, num tempo t, o algoritmo **a** resolve um problema de tamanho x e o algoritmo **aa** um problema de tempo x'. Imagine agora que você tem disponível um computador 20 vezes mais rápido. Que tamanho de problema resolverão os algoritmos **a** e **aa**, no mesmo tempo t? Analise a resposta.

1.12 Mostre, por indução em n, que:

a) $\sum_{i=0}^{n} i^2 = \frac{1}{6} \cdot n \cdot (n+1) \cdot (2n+1)$

b) $\sum_{i=0}^{n} i2^i = 2 + (n-1) \cdot 2$

1.13 Seja a progressão aritmética: $a_0, a_1, \ldots a_i, \ldots, a_n$, com $a_i = a_0 + i.r$. Mostre que as somas: $a_0 + a_n$, $a_{01} + a_{n-1} + a_n$, etc. são todas iguais. Use esse fato para obter a expressão da soma

$$\sum_{i=0}^{n} a_i$$

1.14 Dada uma progressão geométrica de razão $q \neq 1$, chame

$$S(n) = \sum_{i=0}^{n-1} aq^i$$

Mostre que $a + q \cdot S(k) = a \cdot q^k$. Use isso para obter a expressão dada para a soma

$$\sum_{i=0}^{n-1} aq^i$$

1.15 As expressões dadas em 1.3 para as somas dos termos de progressões aritméticas e geométricas ainda se aplicam para razões nulas? Explique.

1.16 Mostre a desigualdade

$$\sum_{i=1}^{n} \frac{1}{i} \leq \log n + 1$$

usando

$$\sum_{i=1}^{n} \frac{1}{i} \leq \sum_{i=0}^{\lfloor \log n \rfloor} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j} \leq \sum_{i=0}^{\lfloor \log n \rfloor} \sum_{j=0}^{2^i - 1} \frac{1}{2^i} \leq \sum_{i=0}^{\lfloor \log n \rfloor} 1$$

1.17 Considere o algoritmo $q \leftarrow 0$; `para` i `de` 0 `até` n–1 `faça` $q \leftarrow q + 2 . i + 1$ `fim-para`. Pode-se garantir que o valor final de q seja o quadrado do natural n? Explique.

1.18 Pode-se garantir que $2 . n \leq n^2$, para todo natural $n > 1$?

1.19 Compare o fatorial de n com $\frac{(n-1) \cdot n}{2}$: pode-se garantir que $\frac{(n-1) \cdot n}{2} \leq n!$ sempre?

1.20 Mostre ou dê um contraexemplo:

a) para $n > 0 : n! \leq 2^n$;
b) para $n > 1 : n! < 2^n$

1.21 Mostre ou de um contraexemplo para $m \geq 0$.:

a) Para $n > 0 : n^m \leq 2^{m.n}$
b) Para $n > 1 : n^m < 2^{m.n}$

{*Sugestão*: Considere o número de funções e relações entre dois conjuntos.}

1.22 Considere uma função f de $\mathbb{N}$ em $\mathbb{N}$ crescente: $f(m) > f(n)$ sempre que $m > n$. Mostre (por indução) que $n \leq f(n)$, para todo natural n. Pode-se garantir que $n < f(n)$, para todo natural n?