PARTE B

# COMPLEXIDADE DE ALGORITMOS

CAPÍTULO I

# COMPLEXIDADE DE ALGORITMOS: NOÇÕES BÁSICAS

## 1. Introdução

Na Parte A, examinamos alguns aspectos importantes da síntese, descrição e correção de algoritmos. Todas estas questões são de grande interesse, não só do ponto de vista teórico, mas também prático. Certamente é fundamental sabermos como escrever programas corretos para algoritmos de nosso interesse, bem como demonstrar que os algoritmos calculam corretamente as funções desejadas, isto é, que após um número finito de passos nos fornecerão as respostas de acordo com as especificações. Em geral, porém, não basta garantir que um algoritmo forneça as respostas corretas em um número finito de passos. Para que o algoritmo possa ser útil, é necessário que este número de passos seja não só "finito", mas "muito finito". Este fato já foi ilustrado na Seção A.I. 4, onde exibimos um programa recursivo para o cálculo do $n$-ésimo número de Fibonacci, $F_n$, correto mas inútil para valores moderados de $n$. Assim, vimos que levaria mais de cem milhões de anos para calcular $F_{100}$ por meio deste algoritmo, dispondo de um computador que efetue um milhão de operações por segundo! Evidentemente, podemos calcular $F_{100}$ de maneira muito mais rápida utilizando-se o algoritmo iterativo descrito na mesma seção, ou melhor ainda, a fórmula fechada para $F_n$, $F_n = (\alpha^n - \beta^n)/\sqrt{5}$, onde $\alpha = (1 + \sqrt{5})/2$ e $\beta = (1 - \sqrt{5})/2$. Fica claro portanto, que considerações sobre a eficiência de desempenho de nossos algoritmos não podem ser ignoradas, e são estas as questões que são o objeto da Teoria da *Complexidade de Algoritmos*.

Embora a preocupação com eficiência de algoritmos seja possivelmente tão antiga quanto a própria noção de algoritmo, um estudo sistemático destas questões é relativamente recente. Na década de 1950 vários algoritmos foram analisados. Os primeiros trabalhos que estabeleceram as bases desta teoria, porém, só surgiram em torno de 1960 [97, 98, 54, 55].

Vale a pena ilustrar algumas das questões neste campo com mais um exemplo. Consideremos o seguinte programa:

```
procedimento mdc(m, n):
    início
        se m < n então d ← m senão d ← n;
        enquanto resto (m, d) ≠ 0 ∨ resto (n, d) ≠ 0 faça d ← d − 1;
        devolva d
    fim
```

Um pouco de reflexão deve convencer o leitor que, se $m$ e $n$ forem inteiros positivos, o procedimento acima calcula o máximo divisor comum entre $m$ e $n$. De fato, a condição do comando repetitivo garante que o valor final de $d$ é um divisor comum de $m$ e $n$, e como o valor inicial de $d$ é $min\,(m, n)$, claramente maior ou igual a qualquer divisor comum, e $d$ decresce de um em um, resulta que o valor final de $d$ é o maior de tais divisores. Na Parte A vimos que o Algoritmo de Euclides também calcula o máximo divisor comum de $m$ e $n$. É natural, portanto, perguntar qual destes dois algoritmos é melhor. Um critério para comparar estes dois algoritmos poderia ser a avaliação da sua eficiência pelo tempo que levam para calcular a resposta; esse tempo é diretamente proporcional ao número de operações efetuadas durante a computação[1]. Sejam $T_A(m, n)$ e $T_E(m, n)$ o número de operações efetuadas pelo algoritmo acima e o Algoritmo de Euclides respectivamente. Pelo Exercício 1, $T_A(m, n) = 7\,(min\,(m, n) - mdc\,(m, n)) + 8$. Em particular vemos que fixado $n$, $T_A(m, n)$ varia entre 8 e $7n + 1$, e portanto no pior caso vale $7n + 1$. O valor de $T_E(m, n)$ é mais difícil de determinar. Pelo Exercício 2 temos, no entanto, que $T_E(m, n) \leq 10\ log_2\ n + 8$. Para cada $n$, portanto, o Algoritmo de Euclides é, no pior caso, mais eficiente do que o algoritmo *mdc*, exceto para valores pequenos de $n$. Veja também os Exercícios 3 e 4, para outras questões interessantes sobre o desempenho destes dois algoritmos.

O exemplo simples acima ilustra que a análise do desempenho de nossos algoritmos pode nos fornecer subsídios úteis para a escolha do algoritmo mais adequado, além de sugerir muitas questões interessantes do ponto de vista matemático dignas de serem estudadas. Existe considerável conhecimento acumulado nos últimos vinte anos acerca das técnicas que se revelaram úteis na solução dos problemas que surgem na análise de algoritmos, e muitos algoritmos já foram analisados com êxito usando-se estas técnicas (veja [60, 61, 62]). Além de

---

(1) Vamos supor que cada operação elementar como soma, comparação, atribuição, etc., leve uma unidade de tempo.

ser útil na escolha do algoritmo mais adequado em cada caso, a análise de algoritmos aumenta a nossa compreensão dos algoritmos analisados, sugerindo muitas vezes idéias importantes para a sua melhoria.

Existem, porém, questões importantes que a simples análise dos algoritmos já conhecidos não pode responder, pois existem infinitos algoritmos para calcular cada função computável. Por exemplo, em geral queremos não só saber qual a quantidade de recursos computacionais utilizada por algum algoritmo conhecido, mas também se este algoritmo pode ainda ser melhorado. Enquanto a análise do algoritmo conhecido nos fornece um *limite superior* para a quantidade de recursos que é suficiente para resolver esta tarefa, o primeiro passo para responder se este algoritmo pode ser melhorado é estabelecer um *limite inferior* na quantidade de recursos necessária. Naturalmente, temos interesse em estabelecer o maior limite inferior que conseguirmos, e analogamente, o menor limite superior possível. Na situação ideal os dois limites deveriam ser iguais, caso este em que conheceríamos exatamente a quantidade de recursos que é tanto necessária como suficiente. Se dispusermos de um algoritmo que utilize exatamente esta quantidade de recursos, então, obviamente, temos um *algoritmo ótimo* para a tarefa, no sentido de que a quantidade de recursos utilizada por qualquer outro algoritmo para a tarefa será maior, ou no melhor dos casos igual, à do algoritmo que temos.

Na prática, são raros os casos em que temos esta situação ideal. Pode-se mesmo demonstrar que existem problemas que não possuem algoritmos ótimos. Normalmente, a diferença entre o limite superior e o limite inferior é uma indicação de quanto poderíamos, no máximo, melhorar o algoritmo de que dispomos. É possível, no entanto, que a melhoria efetivamente obtenível seja bem menor do que a diferença entre o limite superior e inferior nos faria acreditar, porque o limite inferior é demasiadamente baixo. Para responder este tipo de questão, é preciso, pois, estudar classes de algoritmos em vez de algoritmos individuais.

Estabelecer limites inferiores para a quantidade de recursos necessária para uma tarefa é, em geral, um problema muito difícil. Nos três capítulos seguintes voltaremos a este problema sob vários pontos de vista. No Capítulo II mostraremos que existem tarefas naturais de computação para as quais não existem algoritmos. Isto pode ser encarado como um problema extremo de limites inferiores, no sentido de que nenhuma quantidade de recursos, por maior que seja, é suficiente para computar estas funções. No Capítulo III estudaremos a comple-

xidade de algoritmos que calculam o produto de duas matrizes. A medida de complexidade aqui, será o número de multiplicações utilizadas. Como veremos, o algoritmo clássico de multiplicação de matrizes, surpreendentemente, não é o melhor algoritmo, e apresentaremos um algoritmo que lhe é assintoticamente superior. A seguir esboçaremos uma teoria algébrica que permite provar limites inferiores para este tipo de problema. A relativa sofisticação do método, quando comparada com os resultados, às vezes modestos, que podem ser provados mediante o seu uso, ilustra bem a dificuldade de se provar limites inferiores.

No Capítulo IV isolaremos uma classe de problemas que num certo sentido preciso têm aproximadamente a mesma complexidade. Esta classe inclui, como veremos, muitos problemas de interesse em diversas áreas do conhecimento, no entanto não se sabe se estes problemas podem ou não ser resolvidos eficientemente. Temos aqui, portanto, a interessante situação em que todos os limites superiores conhecidos para resolver estes problemas são "ineficientes", mas não se conseguiu até hoje provar limites inferiores que mostrem que este é necessariamente o caso.

Como indica a discussão acima, um dos problemas fundamentais da Teoria de Complexidade é a estimativa dos recursos computacionais necessários e suficientes para o cálculo de funções computáveis específicas de nosso interesse. Poderia parecer à primeira vista que, em virtude disto, a teoria tenderia a ser uma coleção de técnicas, cada uma adequada a um problema específico. Na realidade, porém, existem relações entre as complexidades de muitos problemas aparentemente diferentes, permitindo reduzir um problema a outro. Esta técnica de reduções aparece com destaque nos três capítulos seguintes sob diversas formas.

Existem naturalmente outras questões importantes da Teoria de Complexidade que não examinaremos nos capítulos seguintes. Por exemplo, uma pergunta fundamental da teoria é determinar qual a relação entre as diversas medidas de complexidade de algoritmos, tais como tempo e memória utilizados. Não estudaremos aqui tais questões, embora acreditemos que os capítulos seguintes constituam uma amostra significativa não só de alguns dos resultados mais relevantes da teoria, mas também dos métodos nela utilizados.

## 2. Máquinas de Turing

Para se poder estudar a complexidade computacional de uma tarefa é necessário escolher um modelo formal de algoritmos. Já vimos uma possível formalização por meio da linguagem de programação *LP* descrita na Parte A. Em muitos casos, porém, este tipo de formalização é por demais complexo para ser útil do ponto de vista teórico. No Capítulo III, por exemplo, é introduzida uma simplificação deste modelo para eliminar os aspectos da linguagem, irrelevantes para a questão lá estudada, facilitando assim o estudo do número de operações algébricas utilizadas pelos algoritmos.

Provavelmente o modelo formal de algoritmo mais utilizado em Teoria da Computação é o de Turing. Esta preferência se deve ao fato de que este modelo é suficientemente simples, facilitando as demonstrações, mas ao mesmo tempo suficientemente poderoso para que os resultados provados se apliquem a modelos aparentemente com mais recursos. Em particular, o modelo é suficientemente poderoso para que qualquer algoritmo possa ser nele representado. Passaremos agora a definir este modelo.

Uma *máquina de Turing determinística* é um sistema formal que pode ser visualizado como um computador consistindo de uma *fita de entrada/trabalho/saída* manipulada por um *controle central* (Figura 1). A fita é subdividida em *células*, cada uma das quais contém um símbolo de um alfabeto finito $\Sigma$[2], e é infinita para a direita. Embora isto não seja essencial, suporemos que $\Sigma$ contém pelo menos os quatro símbolos $\{\vdash, \not b, 0, 1\}$, podendo, no entanto, conter símbolos adicionais. A primeira célula à esquerda contém sempre o símbolo $\vdash$, indicando que esta é a célula mais à esquerda da fita. Este símbolo não é usado para nenhum outro propósito. O controle central tem acesso a informações na fita, e pode modificar tais informações, por meio de uma *cabeça* móvel de *leitura* e *gravação* que, em cada instante, encontra-se sobre uma das células da fita. O controle central, por sua vez, está em algum *estado* $q$, de um conjunto finito de estados $Q$. Inicialmente, o controle central está no estado $q_0$, chamado de *estado inicial*, a cabeça encontra-se

---

(2) Um *alfabeto* $\Sigma$ é um conjunto de símbolos. Uma seqüência finita de símbolos de $\Sigma$ é uma *palavra* sobre $\Sigma$. O conjunto de todas as palavras sobre $\Sigma$ é denotado por $\Sigma^*$. O número de símbolos de uma palavra $x$ é denotado por $|x|$, e é chamado o *comprimento* de $x$. A palavra de comprimento zero é denotado por $\Lambda$ e é chamada a *palavra vazia*.

sobre a célula mais à esquerda, contendo o símbolo $\vdash$, e as $n$ células seguintes contém uma palavra $x$, chamada de *entrada*, sobre o *alfabeto de entrada e saída*, $\Sigma_{e/s} \subseteq \Sigma\backslash\{\vdash, \not b\}$. As demais células da fita contêm o símbolo *branco*, $\not b$. A computação da máquina começa então, prosseguindo passo a passo. Um *passo* da máquina consiste das seguintes operações:

(a) ler o símbolo $\sigma$ sob a cabeça;
(b) escrever um símbolo $\sigma'$ no lugar de $\sigma$;
(c) reposicionar a cabeça, que pode ser movida uma célula à direita ou à esquerda, ou deixada no mesmo lugar;
(d) mudar o controle central para um novo estado $q'$.

As operações (b), (c) e (d) dependem apenas do símbolo $\sigma$ lido em (a) e do estado $q$ do controle central antes deste passo. Ademais, o símbolo $\sigma'$ é necessariamente diferente de $\vdash$ sempre que $\sigma$ for diferente de $\vdash$; no caso em que $\sigma = \vdash$, a cabeça encontra-se sobre a célula mais à esquerda da fita, e então $\sigma' = \vdash$ obrigatoriamente e a cabeça não pode ser movida para a esquerda.

A computação prossegue passo a passo, só sendo interrompida se o controle central assumir um dos dois *estados finais* $q_a$ ou $q_r$. Se isto acontecer, dizemos que a computação *pára*. Neste caso, se o estado final for $q_a$, dizemos que a máquina *aceita* a entrada $x$, e se for $q_r$, dizemos que a máquina *rejeita* a entrada $x$. Se a máquina nunca parar, $x$ não é nem aceita nem rejeitada. A *saída* da máquina, definida apenas se a máquina parar, é a palavra sobre $\Sigma_{e/s}$, escrita após a primeira célula à esquerda contendo $\vdash$, até o primeiro símbolo de $\Sigma\backslash\Sigma_{e/s}$ na fita, exclusive.

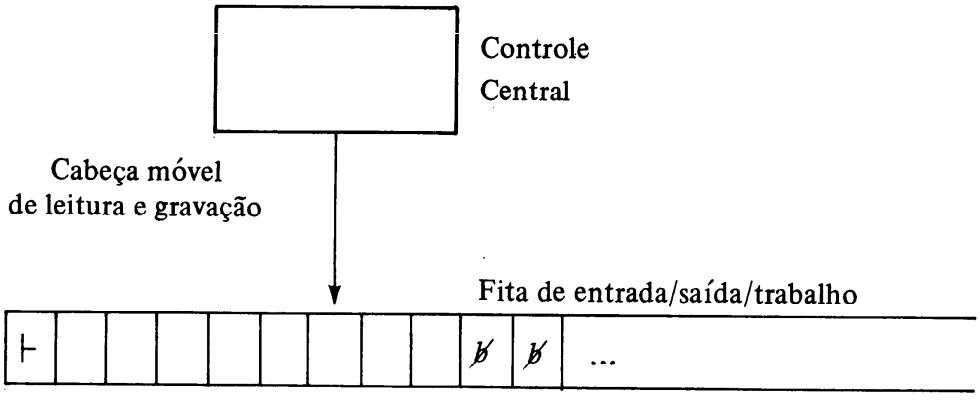


**Figura 1** — Uma máquina de Turing.

Uma máquina de Turing $M$ *reconhece* um conjunto $A \subseteq \Sigma^*_{e/s}$ se para toda entrada $x \in \Sigma^*_{e/s}$, $M$ pára, e aceita $x$ sse $x \in A$. Uma máquina

de Turing $M$ *calcula uma função* $f$: $\Sigma^*_{e/s} \to \Sigma^*_{e/s}$ se para toda entrada $x \in \Sigma^*_{e/s}$, $M$ pára, e a saída de $M$ é a palavra $f(x)$.

Resumindo formalmente a discussão acima: uma máquina de Turing $M$ é um sistema $(Q, q_0, q_a, q_r, \Sigma, \Sigma_{e/s}, \vdash, \not b, \delta)$ onde $q_0, q_a, q_r \in Q$, $\vdash, \not b \in \Sigma$, $\Sigma_{e/s} \subseteq \Sigma \backslash \{\vdash, \not b\}$ e $\delta$ é a *função de transferência* de $M$, $\delta: Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$. A interpretação de $\delta(q, \sigma) = (q', \sigma', \Delta)$ é que a máquina no estado $q$, tendo lido $\sigma$, escreve $\sigma'$, move a cabeça $\Delta$ células para a direita, e muda para o estado $q'$, sendo portanto sujeita às restrições:

$$\begin{array}{l} \text{se } \sigma = \vdash \text{ então } \sigma' = \vdash, \text{ e } \Delta = +1 \text{ ou } \Delta = 0, \\ \text{se } \sigma \neq \vdash \text{ então } \sigma' \neq \vdash, \\ \text{e} \quad \delta(q_a, \sigma) = (q_a, \sigma, 0), \\ \quad\quad \delta(q_r, \sigma) = (q_r, \sigma, 0). \end{array}$$

Finalmente, a transformação da fita associada à máquina $M$ pode ser formalmente definida pela relação $\vDash_M$ abaixo sobre $\hat{\Sigma}^*$, onde $\hat{\Sigma} = \Sigma \cup Q \times \Sigma$. Para $x, y \in \Sigma^*$ e $\delta(q, \sigma) = (q', \sigma', \Delta)$:

(i) se $\Delta = 0$ então $x(q, \sigma)y \vDash_M x(q', \sigma')y$;
(ii) se $\Delta = +1$ e $y = \tau y'$, ou se $\Delta = +1$, $y' = y = \Lambda$ e $\tau = \not b$, então $x(q, \sigma)y \vDash_M x\sigma'(q', \tau)y'$;
(iii) se $\Delta = -1$ e $x = x'\tau$ então $x(q, \sigma)y \vDash_M x'(q', \tau)\sigma' y$.

Uma *máquina de Turing não determinística* é um sistema formal como o acima descrito, com a única diferença que a máquina não determinística possui uma fita adicional chamada *fita de sugestões*. O controle central tem acesso a esta fita, mas não pode modificar as informações nela armazenadas. O acesso é por meio de uma *cabeça de leitura apenas*, que não pode gravar. Esta cabeça encontra-se inicialmente sobre a primeira célula à esquerda da fita, que contém o símbolo $\vdash$, as células seguintes contêm uma palavra $y$ de $\Sigma^*_{e/s}$, e as demais células da fita de sugestões contém brancos $\not b$. A computação da máquina não determinística prossegue passo a passo como descrito antes. Naturalmente, a função de transferência que descreve um passo da máquina é agora uma função $\delta$: $Q \times \Sigma \times \Sigma \to Q \times \Sigma \times \{-1, 0, +1\}^2$, refletindo que cada passo depende agora não só do estado $q$ do controle central e do símbolo $\sigma$ lido da fita de entrada/trabalho/saída, mas também do símbolo lido da fita de sugestões, com a seguinte interpretação: se $\delta(q, \sigma_1, \sigma_2) = (q', \sigma', \Delta_1, \Delta_2)$ então o controle central no estado $q$, lendo $\sigma_1$ na fita de entrada/trabalho/saída e $\sigma_2$ na fita de sugestões, escreve $\sigma'$ no lugar de $\sigma_1$, move as duas cabeças respectivamente $\Delta_1$ e $\Delta_2$ células para a direita e muda para o estado $q'$.

A idéia da fita de sugestões é dar à máquina informações auxiliares, daí o nome de fita de sugestões, para ajudar a determinar se a entrada $x$ da máquina deve ou não ser aceita. Isto sugere a definição abaixo.

Um conjunto $A \subseteq \Sigma_{e/s}^*$ é *aceito* por uma máquina não determinística $M$, se com entrada $x$:

(a) se $x \in A$ então com alguma sugestão $y$ a máquina $M$ pára e aceita $x$;

(b) se $x \notin A$ então qualquer que seja a sugestão $y$, $M$ ou nunca pára, ou rejeita $x$.

O leitor atento terá observado que a definição de máquina de Turing não determinística estende a definição do modelo determinístico. De fato, qualquer máquina de Turing determinística pode ser encarada como uma máquina não determinística que ignora a sua fita de sugestões. Deste modo, uma máquina determinística $M$ aceita um conjunto $A$ se $M$, encarada como uma máquina não determinística, aceitar $A$. Note que se uma máquina determinística reconhecer $A$, então ela também aceita $A$. A recíproca não é verdadeira: é possível a uma máquina determinística $M$ aceitar um conjunto $A$ sem reconhecê-lo.

Vamos ver agora um exemplo simples aplicando os conceitos acima. Considere a máquina de Turing determinística

$M = (Q, q_0, q_a, q_r, \Sigma, \Sigma_{e/s}, \vdash, \not b, \delta)$ que reconhece[3] $\{0^n1^n \mid n \geq 1\}$, onde

$Q = \{q_0, q_1, q_2, q_3, q_4, q_a, q_r\}$, $\Sigma = \{\vdash, \not b, 0, 1, A, B\}$, $\Sigma_{e/s} = \{0, 1\}$ e $\delta$ é dado por:

(i) $\delta(q_0, \vdash) = (q_0, \vdash, +1)$
$\delta(q_0, 0) = (q_1, A, +1)$

($M$ verifica se a entrada é da forma $0^n1^n$ substituindo alternadamente um 0 por um $A$ e um 1 por um $B$. Assim no estado $q_0$, o 0 lido é substituído por um $A$ e em seguida $M$ move para a direita no estado $q_1$ procurando o primeiro símbolo 1.);

(ii) $\delta(q_1, 0) = (q_1, 0, +1)$
$\delta(q_1, B) = (q_1, B, +1)$
$\delta(q_1, 1) = (q_2, B, -1)$

---

(3) Para $\sigma \in \Sigma$ a notação $\sigma^n$ denota a palavra $\sigma\sigma \ldots \sigma$ de comprimento $n$.

(No estado $q_1$, $M$ passa por cima dos símbolos 0 e $B$ até encontrar o primeiro símbolo 1, que é substituído por $B$. $M$ agora moverá à esquerda no estado $q_2$.);

(iii) $\delta(q_2, B) = (q_2, B, -1)$
$\delta(q_2, 0) = (q_3, 0, -1)$
$\delta(q_2, A) = (q_4, A, +1)$

(No estado $q_2$ a máquina vai à esquerda, passando por cima de $B's$. Se houver mais zeros a serem substituídos, o primeiro símbolo que não seja $B$ será um 0, e neste caso a máquina procurará o 0 mais à esquerda no estado $q_3$. Caso contrário, se o primeiro símbolo diferente de $B$ for um $A$, então não há mais zeros, e precisamos apenas verificar se todos os $1's$ foram substituídos por $B's$.);

(iv) $\delta(q_3, 0) = (q_3, 0, -1)$
$\delta(q_3, A) = (q_0, A, +1)$

(No estado $q_3$ movemos por cima dos zeros até um símbolo $A$. Quando este é encontrado movemos à direita no estado $q_0$ para substituir o próximo símbolo 0 por $A$.);

(v) $\delta(q_4, B) = (q_4, B, +1)$
$\delta(q_4, \not b) = (q_a, \not b, 0)$

($M$ verifica se todos os $1's$ foram substituídos por $B's$.);

(vi) Para todo $\sigma \in \Sigma$,
$\delta(q_a, \sigma) = (q_a, \sigma, 0)$
$\delta(q_r, \sigma) = (q_r, \sigma, 0)$;

(vii) Para todo $(q, \sigma) \in Q \times \Sigma$ tal que $\delta(q, \sigma)$ não foi definido anteriormente, $\delta(q, \sigma) = (q_r, \sigma, 0)$.

Para a entrada 0011 temos a seguinte seqüência de passos que aceitam esta entrada:

$$
\begin{aligned}
(q_0, \vdash)0011 &\vDash_M \vdash(q_0, 0)011 \vDash_M \vdash A(q_1, 0)11 \vDash_M \vdash A0(q_1, 1)1 \\
&\vDash_M \vdash A(q_2, 0)B1 \vDash_M \vdash(q_3, A)0B1 \vDash_M \vdash A(q_0, 0)B1 \\
&\vDash_M \vdash AA(q_1, B)1 \vDash_M \vdash AAB(q_1, 1) \vDash_M \vdash AA(q_2, B)B \\
&\vDash_M \vdash A(q_2, A)BB \vDash_M \vdash AA(q_4, B)B \vDash_M \vdash AAB(q_4, B) \\
&\vDash_M \vdash AABB(q_4, \not b) \vDash_M \vdash AABB(q_a, \not b).
\end{aligned}
$$

O Exercício 15 mostra que esta máquina de Turing reconhece o conjunto $\{0^n1^n \mid n \geq 1\}$.

Pelo exemplo acima podemos notar que descrever uma máquina de Turing pela sua função de transferência é bastante trabalhoso, mesmo para resolver problemas relativamente simples. Para facilitar a construção de máquinas de Turing, certas "técnicas de programação" destas máquinas tornam-se indispensáveis. Com tais construções podemos descrever máquinas de Turing mais complexas sem precisarmos dar explicitamente funções de transferência envolvendo possivelmente centenas de estados. Por falta de espaço não daremos aqui estas técnicas. Quando precisarmos mais adiante construir máquinas de Turing, daremos descrições que esperamos que sejam suficientemente pormenorizadas para que o leitor se convença de que, seguindo a descrição, pode construir a função de transferência da máquina desejada.

Existem vários textos de computabilidade mencionados nas notas bibliográficas em que o leitor interessado pode encontrar estas técnicas. A resolução dos Exercícios 7 a 12 pode também ser de utilidade para adquirir alguma prática na construção de máquinas de Turing.

## 3. A tese de Church

"Qualquer procedimento efetivo pode ser realizado por uma máquina de Turing". Esta proposição é conhecida como a *Tese de Church*.

Uma máquina de Turing tem uma descrição finita, pois o conjunto de estados $Q$ e o alfabeto $\Sigma$ são conjuntos finitos por definição, e portanto a função de transferência $\delta$ também é um objeto finito. Cada operação da máquina é claramente efetiva. Assim, examinando as propriedades que exigimos de um procedimento descritas na parte A, fica intuitivamente claro que toda máquina de Turing descreve um procedimento efetivo. O modelo de Turing, porém, é tão simples que à primeira vista pode parecer injustificado supor que qualquer procedimento efetivo possa nele ser implementado. No entanto é exatamente isto que afirma a Tese de Church.

A noção de procedimento efetivo é um conceito informal, e assim a Tese de Church afirma a equivalência de um conceito informal e um conceito formal, não sendo portanto passível de demonstração. Não obstante, existe considerável evidência que suporta esta proposição. De um lado temos uma vasta evidência empírica: cada um dos algoritmos e procedimentos efetivos usados em matemática pode ser

descrito por meio de uma máquina de Turing, ou diretamente, ou através de um formalismo mais conveniente equivalente ao formalismo de Turing. Em particular, pode-se demonstrar que para cada programa em linguagem *LP* existe uma máquina de Turing que calcula a mesma função. Mais ainda, existe um algoritmo que tendo por entrada um programa na linguagem *LP* produz por saída a descrição de uma máquina de Turing correspondente.

Por outro lado, todas as tentativas independentes de formalizar os conceitos de procedimento efetivo e algoritmo se mostraram equivalentes, apesar de haver uma grande variedade de tais formalizações, aparentemente muito dissimilares.

Em virtude da evidência acima, a Tese de Church é geralmente aceita pelos matemáticos.

Supondo que a Tese de Church seja aceita, podemos utilizá-la para dois tipos de aplicações. Às vezes, para economizar tempo e esforço, podemos especificar um procedimento informalmente e, invocando a Tese de Church, concluir que existe um programa formal correspondente. Este tipo de aplicação pode ser evitado, especificando-se formalmente o procedimento em questão.

Existe uma certa analogia entre este tipo de aplicação da Tese de Church e demonstrações em matemática. Em lógica define-se precisamente o conceito de demonstração num sistema formal. Mesmo assim, na maioria dos casos em matemática, usamos métodos informais em nossas demonstrações, subentendendo-se que, se desejado, pode-se transformar as demonstrações informais em demonstrações formais numa teoria formal conveniente.

O segundo tipo de aplicação é porém inevitável e ocorre quando demonstrarmos que não existe um programa formal de uma certa espécie, mas pela Tese de Church afirmarmos que não exista tampouco nenhum procedimento informal do tipo desejado. Usaremos nas seções abaixo ambas as formas de aplicação.

## 4. Medidas de complexidade de máquinas de Turing

A eficiência de um algoritmo pode ser descrita pela função que para cada entrada dá por resultado o tempo de execução do algoritmo com esta entrada. No nosso modelo, "tempo" corresponde de modo natural ao número de passos tomados pela máquina até parar. Assim,

para cada máquina de Turing determinística $M$ definimos a função $t_M: \Sigma^*_{e/s} \to \mathbb{N} \cup \{\infty\}$ dada por[4]:

(a) Se $M$ com entrada $x$ parar então $t_M(x)$ é o número de passos tomados por $M$ até a sua parada;

(b) Se $M$ com entrada $x$ não parar então $t_M(x) = \infty$.

**Para uma máquina de Turing não determinística $M$ a função $t_M: \Sigma^*_{e/s} \to \mathbb{N} \cup \{\infty\}$ é definida por:**

(a) Se para alguma sugestão $y$ a máquina $M$ aceitar a entrada $x$ então $t_M(x)$ é o número de passos mínimo dentre todas as computações de $M$ que aceitam $x$. (Note que podemos ter várias sugestões $y$ com as quais $M$ aceita $x$, tomando um número maior ou menor de passos conforme a sugestão.);

(b) Se $M$ não aceitar $x$ com nenhuma sugestão, mas parar com alguma sugestão, então $t_M(x)$ é o número de passos mínimo dentre todas as computações de $M$ que rejeitam $x$;

(c) Se $M$ não parar com nenhuma sugestão então $t_M(x) = \infty$.

É muitas vezes mais conveniente medir a eficiência de um algoritmo em termos do tamanho de sua entrada. Assim, dada uma função $g: \mathbb{N} \to \mathbb{N}$ dizemos que uma máquina de Turing $M$ é *limitada em tempo* por $g$ se $t_M(x) \leq g(|x|)$ para toda entrada $x$. Seja $G$ uma família de funções $g: \mathbb{N} \to \mathbb{N}$. Um conjunto $A$ é *reconhecível em tempo G* se existir uma máquina de Turing determinística $M$ que reconhece $A$ e é limitada em tempo por algum $g \in G$. Um conjunto $A$ é *aceitável em tempo G* se existir uma máquina de Turing não determinística $M$ que aceita $A$ e é limitada em tempo por algum $g \in G$.

Uma outra medida de complexidade de algoritmos importante é a "memória" utilizada na computação. Isto corresponde naturalmente ao número de células da fita visitadas pela cabeça de leitura/gravação da máquina de Turing durante a computação. Por convenção define-se que o espaço utilizado numa computação que não pára é infinito, mesmo se a máquina visita indefinidamente um trecho finito da fita. Deixamos ao leitor, como exercício, as definições da função análoga a $t_M(x)$ para a medida de complexidade de espaço, bem como os conceitos de conjunto reconhecível e aceitável em espaço $G$.

Os limites superiores de tempo e espaço para máquinas de Turing podem, sem perda de generalidade, ser expressos a menos de uma

(4) $n < \infty$ para todo $n \in \mathbb{N}$.

constante multiplicativa, conforme o Exercício 11. Para isto freqüentemente é muito conveniente utilizar-se da notação $O$ definida do seguinte modo: dadas as funções $f$, $g$: $\mathbb{N} \to \mathbb{N}$ dizemos que $f \in O(g)$, leia-se $f$ é da *ordem* de $g$, se existir uma constante $K > 0$ e $n_0 \in \mathbb{N}$ tal que $f(n) \leq K \cdot g(n)$ para todo $n \geq n_0$.

## EXERCÍCIOS

1. Mostre que o número de operações efetuadas pelo procedimento *mdc* é $T_A(m, n) = 7\,(min\,(m, n) - mdc\,(m, n)) + 8$. Conte cada soma (+), subtração (−), atribuição (←), cálculo de resto, comparação, $\vee$, etc. como uma operação. Assim, por exemplo, cada execução de **se** $m < n$ **então** $d \leftarrow m$ **senão** $d \leftarrow n$ conta como duas operações.
2. Mostre que o número de operações efetuadas pelo Algoritmo de Euclides, $T_E(m, n)$, é limitada superiormente por $10\,log_2\,n + 8$. (Sugestão: Mostre que para cada duas execuções do comando repetitivo o valor de $y$ decresce pelo menos para $y/2$.)
3. No texto comparamos os algoritmos *mdc* e de Euclides pelo seu desempenho no pior caso, para cada $n$ fixo. Defina o conceito de *tempo médio* de execução para $n$ fixo, para o algoritmo *mdc*. Demonstre que existe uma função $\overline{t}_A(n)$ que corresponde à sua definição.
4. Estude as propriedades de $\overline{t}_A(n)$ do Exercício 3.
   (a) Determine o valor de $\overline{t}_A(45)$;
   (b) Determine o valor de $\overline{t}_A(p)$ onde $p$ é primo;
   (c) Determine o valor de $\overline{t}_A(p \cdot q)$ onde $p$ e $q$ são primos;
   (d) O que você pode dizer no caso geral?
5. Considere um problema para o qual dispomos de quatro algoritmos de tempos de execução $200n$, $40n\,\lfloor log_2\,n \rfloor$, $10\,n^2$ e $2^n$. Para cada algoritmo indique a faixa de valores de $n$ para os quais ele é o melhor dos quatro algoritmos. (Para um número real $\alpha$, $\lfloor \alpha \rfloor$ denota o maior inteiro contido em $\alpha$, isto é, o maior inteiro não maior do que $\alpha$.)
6. Contraste as definições de conjunto reconhecido por uma máquina de Turing determinística, e conjunto aceito por uma máquina de Turing não determinística.

7. Construa uma máquina de Turing determinística que reconhece o subconjunto de $\{0, 1\}^*$:
   (a) $\{xx^R \mid x \in \{0, 1\}^*\}$ onde $x^R$ denota a palavra reversa correspondente a $x$ definida por: $\Lambda^R = \Lambda$, $(x0)^R = 0x^R$, $(x1)^R = 1x^R$ (exemplo: $(01001)^R = 10010$);
   (b) $\{x \mid x$ é a representação binária de um número par$\}$;
   (c) $\{x \mid x$ é a representação binária de um número divisível por 3$\}$;
   (d) $\{1^{n^2} \mid n \in \mathbb{N}\}$.
8. Mostre que as linguagens (b) e (c) do Exercício 7 são reconhecíveis no sentido definido no Capítulo D. II.
9. Mostre que o conjunto (d) do Exercício 7 é reconhecível em espaço $|x|$, onde $x$ é a entrada da máquina de Turing.
10. No texto foram dadas duas definições para $t_M(x)$, uma para $M$ determinística e uma para $M$ não determinística. Mostre que se $M$ for uma máquina determinística então $t_M(x)$ calculada por ambas as definições dá a mesma função. Considere uma máquina determinística como uma máquina não determinística que ignora a sua fita de sugestões.
11. (a) Mostre que se $L$ é reconhecido por uma máquina de Turing limitada em tempo por $t(n)$, onde $n$ é o comprimento da entrada, e $\lim_{n\to\infty} t(n)/n^2 = \infty$ então para qualquer constante $c > 0$ o conjunto $L$ é reconhecido por uma máquina de Turing de fita **única limitada em tempo por** $max(n, c, \cdot t(n))$.
    (Sugestão: construa uma nova máquina de Turing que reconheça $L$ cujo alfabeto tem mais símbolos do que o alfabeto da máquina original. Note que basta demonstrar a proposição para $c = 1/2$.)
    (b) Mostre que se $L$ é reconhecido por uma máquina de Turing limitada em espaço por $e(n)$ então para qualquer constante $c > 0$ o conjunto $L$ é reconhecido por uma máquina de Turing limitada em espaço por $max(n, c \cdot e(n))$.
12. Seja $f: \mathbb{N} \to \mathbb{N}$ uma função. Diremos que $f$ é computável por uma máquina de Turing se existir uma máquina de Turing determinística que com entrada $1^n$ produz a saída $1^{f(n)}$. Demonstre que
    (a) $f(n) = n + 1$ é computável;
    (b) se $f$ e $g$ são computáveis então $f \circ g$ também é computável;
    (c) se $f$ é uma função computável e $K \in \mathbb{N}$ então a função $g$ definida por

$$\begin{cases} g(0) = K \\ g(n) = f^n(K) \text{ para } n \geq 1, \end{cases}$$

também é computável.

13. (a) Seja $f(n) = 15n^2 + 7n \lfloor log_2 n \rfloor + 5n + 3$. Mostre que $f \in O(n^2)$ mas $f \notin O(n \lfloor log_2 n \rfloor)$.
    (b) Mostre que $\lfloor log_a n \rfloor \in O(\lfloor log_b n \rfloor)$ para quaisquer $a, b > 1$.
14. Mostre que
    (a) toda função constante pertence a $O(1)$;
    (b) existem funções $f$ e $g$ tais que $g \in O(f)$ mas $f \notin O(g)$.
    (c) Seja $O(f) + O(g) = \{h: \mathbb{N} \to \mathbb{N} \mid h = f' + g' \text{ e } f' \in O(f) \text{ e } g' \in O(g)\}$. Então $O(f+g) = O(f) + O(g) = O(max(f, g))$.
15. Demonstre que a máquina de Turing construída na Seção 2 reconhece o conjunto $\{0^n1^n \mid n \geq 1\}$.

## NOTAS BIBLIOGRÁFICAS

O modelo de Turing foi definido em [119]. A Tese de Church exprime a convicção de que este é um modelo correto de procedimento efetivo, e coroa os esforços desenvolvidos na primeira metade deste século em lógica matemática para isolar este conceito. [39] é uma referência sobre a evolução histórica de artefatos de computação mecânica. Algumas destas contribuições já anteviam a importância da complexidade dos cálculos, muito antes do aparecimento dos primeiros computadores digitais.

A análise de algoritmos específicos, em muitos casos antecedeu o estudo mais abstrato e geral de questões de complexidade computacional, e se desenvolveu em paralelo com esta teoria. Muito do que se conhece sobre técnicas de análise de algoritmos pode ser encontrado nos livros de Knuth [60, 61, 62].

A primeira tentativa para medir a dificuldade de computação de um ponto de vista axiomático foi feito por Rabin [97]. Mais tarde o trabalho de Hartmanis e Stearns em [54, 55] estabeleceu alguns resultados fundamentais da teoria, e constitui o primeiro estudo sistemático de uma medida específica de complexidade. Desde então, o campo de complexidade tornou-se uma das áreas de estudo autônomo de Teoria da Computação, e vem se desenvolvendo vigorosamente.

Uma formalização elegante e abstrata do conceito de medida de complexidade é a teoria axiomática de Blum [5].