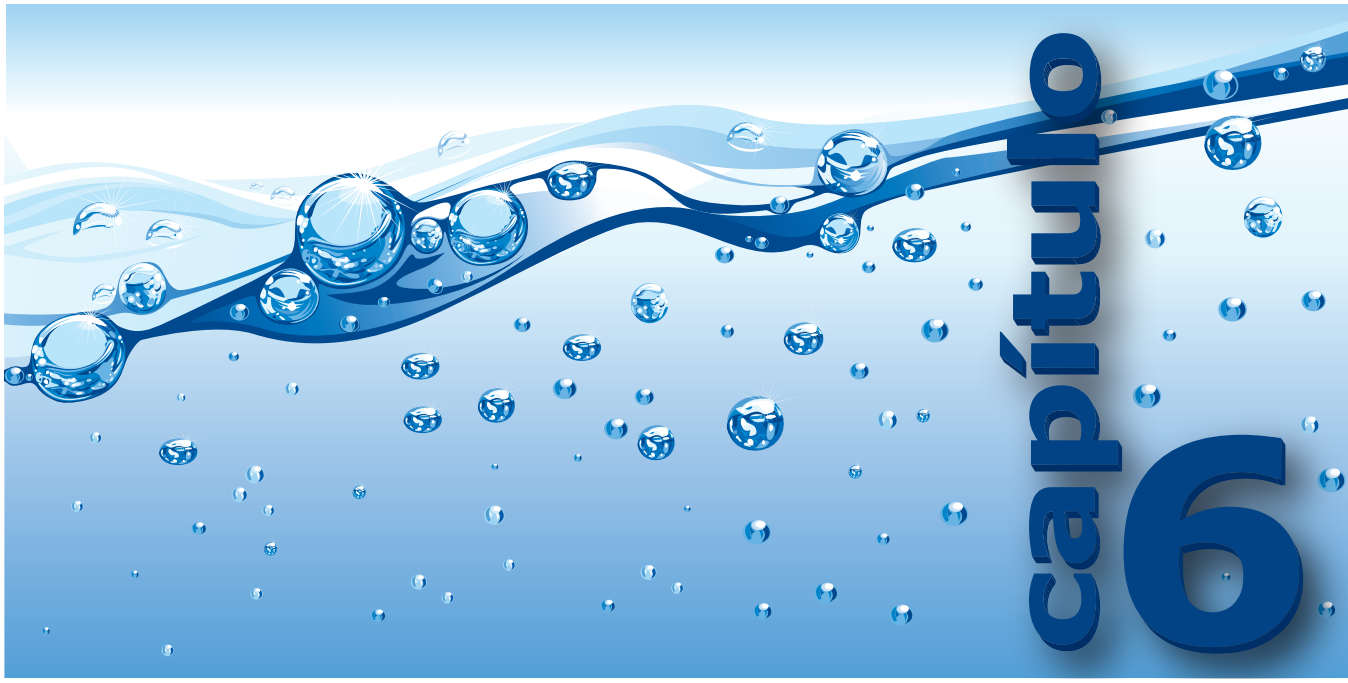


Análise de Algoritmos

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Análise de algoritmo
  - ❒ Complexidade
  - ❒ Custo temporal
  - ❒ Custo espacial
  - ❒ Pior caso
  - ❒ Melhor caso
  - ❒ Caso médio
  - ❒ Regra da soma
  - ❒ Regra do produto
  - ❒ Análise assintótica
  - ❒ Custo constante
  - ❒ Custo linear
  - ❒ Custo quadrático
  - ❒ Custo cúbico
  - ❒ Custo polinomial
  - ❒ Custo exponencial
  - ❒ Custo logarítmico
  - ❒ Custo linear logarítmico
  - ❒ Custo fatorial
  - ❒ Relação de recorrência
  - ❒ Árvore de recursão
- Comparar as notações ó, ômega e teta
- Avaliar o custo de um algoritmo usando as notações ó, ômega e teta
- Demonstrar teoremas referentes às notações ó, ômega e teta
- Explicar por que o principal interesse em análise de algoritmos se concentra no pior caso
- Apresentar exemplos de algoritmos com custos limitados por funções comuns em análise de algoritmos
- Descrever regras práticas usadas em análise temporal de algoritmos para cada tipo de instrução
- Explicar quando a análise de custo espacial de um algoritmo se faz necessária
- Escrever a relação de recorrência associada a um algoritmo recursivo
- Descrever a abordagem de conjectura e prova para resolução de relações de recorrência
- Construir uma árvore de recursão para uma dada relação de recorrência

OBJETIVOS

EFICIÊNCIA (OU DESEMPENHO) de um programa é medida em termos do espaço de armazenamento e do tempo que ele utiliza para realizar uma tarefa. Mais precisamente, um programa é eficiente quando ele é executado utilizando o menor espaço de memória possível no menor intervalo de tempo.

A eficiência de um programa depende de inúmeros fatores, dentre os quais:

[1] O hardware no qual o programa é executado, o que inclui memórias, CPU etc.

[2] O sistema operacional sob a supervisão do qual o programa é executado.

[3] A linguagem de programação utilizada para codificar o programa-fonte.

[4] As estruturas de dados utilizadas pelo programa (i.e., como os dados manipulados pelo programa são organizados).

[5] A complexidade de cada algoritmo que o programa implementa.

[6] A competência do programador que codificou o programa.

Se um programa deve ser construído para atender uma determinada plataforma que abranja os fatores de **[1]** a **[3]** acima, não há muito que se possa fazer para melhorar sua eficiência. Neste livro, se está interessado em atender os fatores de **[4]** a **[6]** enumerados acima. Obviamente, como o tema principal deste livro são estruturas de dados, aqui serão apresentadas diversas maneiras nas quais dados podem ser organizados.

Muitos problemas com os quais um programador se depara frequentemente possuem vários algoritmos funcionalmente equivalentes. Dois algoritmos são funcionalmente equivalentes quando, pelo menos em princípio, obtêm os mesmos resultados para um mesmo problema (v. **Seção 5.3**). Mas o fato de dois algoritmos serem funcionalmente equivalentes não quer dizer que eles sejam equivalentes em termos de eficiência. Ao contrário, existem algoritmos que são funcionalmente equivalentes e que, ao mesmo tempo, divergem bastante em termos de eficiência. Este livro apresentará diversos exemplos de tais algoritmos.

Este capítulo é dedicado ao estudo da inerente complexidade de algoritmos, que é um assunto da maior relevância em programação e está associada a como um algoritmo se comporta quando o tamanho dos dados de entrada cresce arbitrariamente. Esse assunto não deve ser confundido com eficiência de programas, que, conforme foi visto, depende de hardware, sistema operacional etc. Quer dizer, embora, neste livro, muitos algoritmos sejam apresentados em formato de programas ou trechos de programa em C, análise de algoritmos deve ser praticada antes mesmo de um algoritmo ser codificado (i.e., transformado em programa).

6.1 Complexidade de Algoritmos

Complexidade de algoritmo é a taxa de crescimento dos recursos que um algoritmo requer com relação ao tamanho dos dados que ele processa. Tipicamente, os recursos considerados são o tempo e o espaço de armazenamento despendidos pelo algoritmo para processar os dados de entrada. Resumidamente, complexidade de algoritmo é a taxa com a qual o armazenamento utilizado ou tempo de execução cresce em função do tamanho dos dados de entrada. A complexidade de um algoritmo é também denominada seu **custo de processamento**. A complexidade de um algoritmo medida em termos de tempo de processamento é denominada **custo temporal**, enquanto a complexidade medida em termos de espaço adicional ocupado pelos dados que um algoritmo processa é denominada **custo espacial**.

O custo temporal de um algoritmo pode ser expresso em termos do número de operações usadas pelo algoritmo quando o problema apresenta um determinado tamanho. As operações usadas para medir custos podem ser operações elementares, tais como soma, multiplicação e comparação de inteiros.

A **análise de algoritmos** permite avaliar a eficiência de um algoritmo de modo independente de hardware e software; ou seja, um algoritmo sob análise sequer precisa ser implementado.

Análise de custo espacial tipicamente ocorre quando o algoritmo usa espaço adicional que, de algum modo, é proporcional ao tamanho dos dados de entrada. Por exemplo, muitos algoritmos de ordenação de arrays executam suas tarefas usando os próprios arrays recebidos como entrada (i.e., ordenação in loco). Por outro lado, alguns algoritmos de ordenação usam espaço adicional proporcional ao tamanho do array a ser ordenado. Tanto num caso quanto no outro, um algoritmo pode usar algumas poucas variáveis auxiliares. Essas variáveis auxiliares não são relevantes para a análise de custo espacial de um algoritmo. No caso de ordenação, apenas algoritmos que não são in loco precisam ser analisados em termos de uso de espaço.

Normalmente, o custo espacial de um algoritmo é menos significativo do que seu custo temporal porque muitos algoritmos que processam as estruturas de dados usadas aqui não usam espaço adicional (v. **Seção 6.8**).

Levar em consideração custo temporal e espacial de um algoritmo é deveras importante em programação. Afinal, é necessário saber se o algoritmo levará microssegundos, horas, dias ou centenas de anos para apresentar uma solução (v. **Tabela 6–3**). Também é preciso saber qual é a quantidade de memória que um algoritmo requer para produzir uma solução. Enfim, análise de algoritmos serve para classificar algoritmos como adequados ou não adequados. Assim considera-se um algoritmo eficiente quando não se conhece nenhum outro funcionalmente equivalente a ele e que possua menor custo.

6.2 Análise Assintótica

Em Matemática, **análise assintótica** é uma técnica de aproximação de funções. Um notável exemplo de uso dessa técnica é a famosa **fórmula de Stirling** para aproximação de fatorial:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Em programação, princípios de análise assintótica são utilizados para avaliação de custos de algoritmos em termos de tempo e espaço gastos para realização de suas respectivas tarefas. Essa avaliação baseia-se na ideia de que à medida que o tamanho dos dados de entrada de um problema cresce, o custo computacional do algoritmo que o resolve estabiliza-se numa função de proporcionalidade simples relacionada com alguma função matemática conhecida (p. ex., função linear, quadrática, logarítmica etc.).

Como foi visto na introdução deste capítulo, o tempo gasto na execução de um programa depende de diversos fatores, de modo que não é possível descrever o custo temporal de execução de um algoritmo em unidades de tempo usuais (p. ex., segundos). Em vez disso, fala-se em proporcionalidade (p. ex., *o tempo de execução de tal algoritmo é proporcional a* n^2, sendo n o tamanho dos dados de entrada). Nesse caso, a constante de proporcionalidade não é estipulada, visto que ela depende dos diversos fatores já mencionados.

O **tamanho de entrada** de um algoritmo é um número inteiro positivo que determina o tamanho dos dados de entrada que o algoritmo processa. Tipicamente o número de instruções que o algoritmo executa é proporcional a esse valor (v. abaixo). O tamanho de entrada de um algoritmo pode ser o número de itens que ele processa cada vez que é executado. Por exemplo, o tamanho da entrada de um algoritmo que ordena um array é exatamente o tamanho (i.e., o número de elementos) desse array. O tamanho de entrada de um algoritmo também pode ser determinado pelo único valor que ele recebe como entrada. Por exemplo, o tamanho de entrada de um algoritmo que calcula o fatorial de um número n é exatamente n.

O custo de um algoritmo é medido como uma função do tamanho dos dados que ele processa. Isto é, se o tamanho dos dados (p. ex., o número de elementos de um array) for n, seu custo é associado a alguma função

f(n), cujo domínio é o conjunto dos números naturais e cuja imagem é o conjunto dos números reais não negativos. Em análise de algoritmos, a descrição precisa dessa função não é importante. O que realmente importa é determinar a taxa (ou ordem) de crescimento de tal função com relação a *n*. Por exemplo, quando o valor de *n* é duplicado, o que ocorre com *f(n)*? Será que o valor de *f(n)* também duplica ou será que ele quadruplica? Em última instância, a taxa de crescimento de um algoritmo prevê seu comportamento para entradas arbitrariamente grandes.

Análise assintótica serve para comparar algoritmos funcionalmente equivalentes que concorrem entre si (como os algoritmos de busca e ordenação que serão apreciados no **Volume 2**) e também para prever o desempenho de um dado algoritmo mesmo que ele não tenha competidor. Por exemplo, todos os algoritmos conhecidos para resolução do problema das torres de Hanói (v. **Seção 4.8.1**) possuem os mesmos custos e ela dita que eles são impraticáveis para entradas relativamente grandes (p. ex., quando o número de discos é igual a 20). A resolução de problemas dessa natureza utilizando algoritmos excede a capacidade de qualquer plataforma computacional.

Existem algoritmos que, apesar de curtos, são muito difíceis de analisar, mas a maioria dos algoritmos apresentado neste livro e em outros textos sobre Estruturas de Dados são relativamente fáceis de analisar. Portanto não se deixe intimidar pelas aparências do formalismo que será apresentado a seguir.

6.3 Notações Ó, Ômega e Teta

Uma notação matemática, denominada **notação ó grande** (ou, simplesmente, **notação ó**), é usada para expressar custos de algoritmos utilizando análise assintótica. Essa notação usa funções cujo domínio é o conjunto dos números naturais e a imagem é o conjunto dos números reais não negativos. A variável utilizada nessas funções é, tipicamente, denominada *n* e representa um tamanho de entrada. A notação ó usa essas funções para estabelecer uma ordem de magnitude para o crescimento temporal e espacial de algoritmos.

Definição 6.1 Uma função $T(n)$ é $O(f(n))$ se e somente se existem constantes c_0 e n_0, sendo $c_0 > 0$ e $n_0 \geq 0$ e tais que $T(n) \leq c_0 f(n)$ para todo $n \geq n_0$.

Em linguagem mais simples, a **Definição 6.1** pode ser parafraseada como: se, para valores suficientemente grandes de *n*, os valores de *T* forem menores do que aqueles de um múltiplo de *f* (i.e., $c_0 f$), então *T* é da ordem de, no máximo, *f* [i.e., $T(n)$ é $O(f(n))$].

A **Definição 6.1** pode ainda ser interpretada da seguinte maneira: $T(n)$ é $O(f(n))$ se e somente se $T(n)$ torna-se, a partir de um determinado ponto (i.e., quando $n \geq n_0$), proporcional a $f(n)$. A partir desse ponto, *n* é suficientemente grande para fazer com que $T(n)$ estabilize-se numa taxa de crescimento assintótico determinada por $f(n)$. Informalmente, quando se diz que $T(n)$ é $O(f(n))$, sabe-se que $f(n)$ é um limite superior para a taxa de crescimento de $T(n)$.

A **Figura 6–1** ilustra a relação entre duas funções *reais* $T(n)$ e $f(n)$ quando $T(n)$ é $O(f(n))$. Nessa figura, a porção tracejada do gráfico de $T(n)$ é aquela que satisfaz $T(n) < c_0 f(n)$.

Na comunidade de programação, a sentença matemática $T(n)$ é $O(f(n))$ pode ser lida de diversas maneiras. Escolha a sua adequadamente:

- ❒ *T(n) é da ordem de f(n)* — embora bastante utilizada, essa afirmação é pobre em precisão semântica
- ❒ *T(n) é limitada por f(n)* — essa afirmação é pobre porque não especifica se o limite provido por *f(n)* é superior ou inferior
- ❒ *T(n) é ó de f(n)* — essa afirmação não é muito precisa, mas não é completamente má, pois, apesar de também existir notação ó pequeno, essa notação raramente é usada em análise de algoritmos
- ❒ *T(n) é ó grande de f(n)* — essa afirmação é perfeita
- ❒ *f(n) é limite superior de T(n)* — afirmação é perfeita

- *f(n) é a cota assintótica superior de T(n)* — afirmação é perfeita

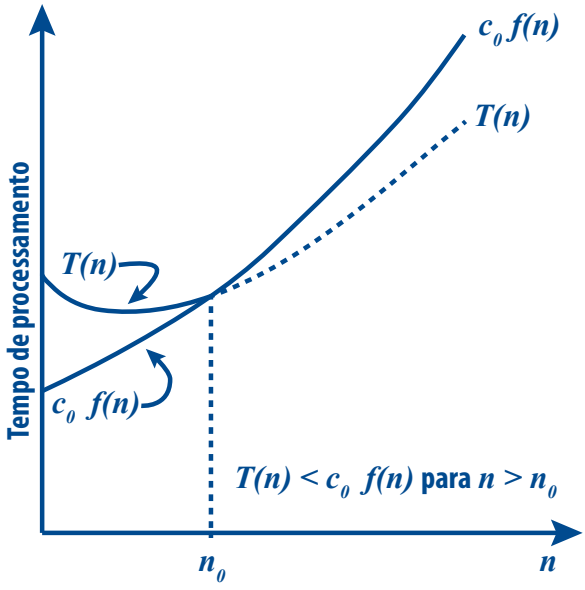


Figura 6–1: Interpretação Gráfica de Custo Ó

Frequentemente, a notação ó grande é usada de modo incorreto. A seguir, alguns exemplos de uso abusivo da notação ó grande:

- $T(n) \leq O(f(n))$. O uso da notação ó grande está incorreto porque o significado do símbolo $\leq$ já está implícito na **Definição 6.1**.
- $T(n) \geq O(f(n))$. O uso do símbolo $\geq$ é ainda pior do que o símbolo $\leq$ porque não faz jus à **Definição 6.1**.
- $T(n) = O(f(n))$. No sentido usual de igualdade, esta expressão está incorreta, pois a **Definição 6.1** não diz respeito a igualdade. Quer dizer, a notação $T(n) = O(f(n))$ é considerada abusiva, mas, mesmo assim, é largamente utilizada.

Por meio de simbolismo matemático, a notação ó grande pode ser usada corretamente das seguintes maneiras:

- $T(n)$ é $O(f(n))$.
- $T(n) \in O(f(n))$. O uso do símbolo $\in$ é perfeitamente aceitável, pois $O(f(n))$ define um conjunto de funções.

Exemplo 6.1 Mostre que se $T(n) = 5n^2 + 3n + 2$, então $T(n)$ é $O(n^4)$.

Solução: Para demonstrar essa afirmação, deve-se propor um valor para a constante c_0 da **Definição 6.1** e tentar encontrar um valor de n_0 tal que, para valores de n maiores do que n_0, a desigualdade $5n^2 + 3n + 2 \leq c_0 n^4$ torne-se válida. Note que o valor de c_0, não pode ser menor do que ou igual a zero, pois, como n é um número natural, se c_0 for um número negativo ou zero, essa desigualdade jamais será satisfeita. Portanto pode-se começar com o menor valor possível de c_0, que é *1*. A procura pelo valor n_0 que torna a desigualdade válida pode ser realizada usando o artifício mostrado a seguir:

n	$T(n) = 5n^2 + 3n + 2$	$c_0 f(n) = c_0 n^4 = n^4$	Satisfaz a desigualdade?
1	*10*	*1*	Não
2	*28*	*16*	Não
3	*56*	*81*	Sim

Evidentemente, o fato de se ter mostrado que $5n^2 + 3n + 2 \leq n^4$ quando $n = 3$ não significa dizer que essa desigualdade seja verdadeira quando $n > 3$. Uma prova rigorosa de que esse fato realmente é verdadeiro pode ser obtida por meio de **indução matemática finita** (v. **Apêndice B**).

Deve-se observar que $O(f(n))$ descreve a coleção de todas as funções para as quais existem as constantes c_0 e n_0 que tornam válida a desigualdade $T(n) \leq c_0 f(n)$. Por exemplo, do mesmo modo que se afirmou que $T(n) = 5n^2 + 3n + 2$ é $O(n^4)$, se poderia ter afirmado que $T(n)$ é $O(n^3)$, $T(n)$ é $O(n^{15})$, $T(n)$ é $O(2n)$ etc. Esses fatos revelam uma debilidade da notação ó grande: quando se diz que $T(n)$ é $O(f(n))$ nada é informado com relação à proximidade da função $f(n)$ com relação à função $T(n)$.

Exemplo 6.2 Mostre que a função $T(n) = 5n^2 + 3n + 2$ é $O(n^2)$.

Solução: Para demonstrar essa afirmação, considere $c_0 = 6$ e $n_0 = 4$ e mostre que $5n^2 + 3n + 2 \leq 6n^2$, para $n \geq 4$ (verifique isso).

É importante salientar que a análise assintótica de algoritmos é muito pouco informativa com respeito ao tempo real com que um dado algoritmo será executado. Por exemplo, dois algoritmos funcionalmente equivalentes podem ter custo temporal $O(1)$, mas isso não significa que, para uma dada entrada, eles serão executados no mesmo intervalo de tempo. Não se deve esquecer que análise assintótica é válida para comparar algoritmos apenas quando o tamanho da entrada se torna suficientemente grande.

Exemplo 6.3 Mostre que n^2 não é $O(n)$.

Solução: A prova será feita por **redução ao absurdo** (ou por **contradição**). Suponha que n^2 seja $O(n)$. Nesse caso, existem constantes $c_0 > 0$ e $n_0 \geq 0$ tais que $n^2 \leq c_0 n$ quando $n > n_0$.

Seja n um número inteiro positivo tal que $n > c_0$ e $n > n_0$. Então, tem-se que:

$n > c_0 \Rightarrow n \cdot n > c_0 \cdot n \Rightarrow n^2 > c_0 n$

Assim existe um número inteiro $n > n_0$ tal que $n^2 > c_0 n$, o que contradiz a hipótese. Essa contradição mostra que n^2 não pode ser $O(n)$.

Exemplo 6.4 Apresente uma estimativa ó grande para a função matemática $n!$, definida como: $n! = 1.2.3. \ldots .n$.

Solução: Cada termo do produto na definição de $n!$ é menor do que ou igual a n. Logo:

$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq n \cdot n \cdot n \cdot \ldots \cdot n = n^n$

Portanto $n!$ é $O(n^n)$, considerando $c_0 = 1$ e $n_0 = 1$.

Exemplo 6.5 Mostre que $n \log n$ é $O(n^2)$.

Solução: Quando $n > 1$, $n \log n < n \cdot n = n^2$. Logo, considerando $c_0 = 1$ e $n_0 = 1$, tem-se que $n \log n$ é $O(n^2)$.

Não custa lembrar que estimativas que usam notação ó apresentam apenas limites superiores. Por exemplo, não é correto dizer que um algoritmo é ineficiente porque ele apresenta custo $O(n^5)$ porque um algoritmo com custo $O(n)$ também satisfaz essa estimativa. Assim uma notação que apresente limites inferior e superior se faz necessária.

Definição 6.2 $T(n)$ é $\Omega(f(n))$ se e somente se existem constantes c_0 e n_0, sendo $c_0 > 0$, $n_0 \geq 0$ e tais que $T(n) \geq c_0 f(n)$ para todo $n \geq n_0$. A sentença $T(n)$ é $\Omega(f(n))$ é lida como: *$T(n)$ é ômega (grande) de $f(n)$* ou *$f(n)$ é limite inferior de $T(n)$*.

Trocando em miúdos, a **Definição 6.2** pode ser interpretada como: se, para valores suficientemente grandes de n, os valores de T são maiores que aqueles de um múltiplo de f (i.e., $c_0 f$), então T é da ordem de, no mínimo, f [i.e., $T(n)$ é $\Omega(f(n))$].

A **Figura 6–2** ilustra a relação entre duas funções reais $T(n)$ e $f(n)$ quando $T(n)$ é $\Omega(f(n))$. Nessa figura, a porção tracejada do gráfico de $T(n)$ é aquela que satisfaz $T(n) > c_0 f(n)$.

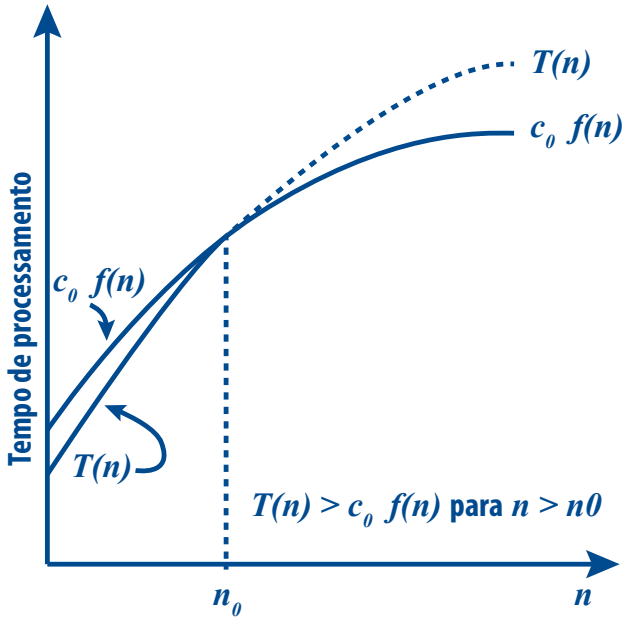


FIGURA 6–2: INTERPRETAÇÃO GRÁFICA DE CUSTO ÔMEGA

Exemplo 6.6 Mostre que a função $T(n) = 5n^2 + 3n + 2$ é $\Omega(n^2)$.

Solução: Considere $n_0 = 0$ e $c_0 = 1$ e mostre que $5n^2 + 3n + 2 \geq n^2$, para $n \geq 0$. Então, de acordo com a **Definição 6.2**, $5n^2 + 3n + 2$ é $\Omega(n^2)$.

Exemplo 6.7 Mostre que a função $T(n) = 5n^2 + 3n + 2$ é $\Omega(n)$.

Solução: Novamente, considerando-se $n_0 = 0$ e $c_0 = 1$, pode-se demonstrar que $5n^2 + 3n + 2 \geq n$, para $n \geq 0$. Assim, em consonância com a **Definição 6.2**, $5n^2 + 3n + 2$ é $\Omega(n)$.

Como os dois últimos exemplos ilustram, a notação ômega padece do mesmo problema da notação ó. Quer dizer, quando se afirma, por exemplo, que $T(n)$ é $\Omega(n^5)$, pode-se facilmente provar que $T(n)$ também é $\Omega(n^3)$, $\Omega(n^2)$, $\Omega(n)$ e assim por diante. Novamente, na prática, a função mais útil é aquela mais próxima de $T(n)$ [i.e., a função que representa o maior limite inferior de $T(n)$]. Por isso, a notação teta se faz necessária.

Definição 6.3 $T(n)$ é $\theta(f(n))$ se e somente se $T(n)$ é $O(f(n))$ e $T(n)$ é $\Omega(f(n))$.

A sentença $T(n)$ é $\theta(f(n))$ é lida como: *$T(n)$ é teta grande de $f(n)$* ou *$f(n)$ é limite estrito de $T(n)$*. A **Definição 6.3** pode ser parafraseada como: se, para valores suficientemente grandes de n, os valores de T são limitados inferiormente e superiormente por valores múltiplos de f (i.e., se os valores de T estão entre $c_0 f$ e $c_1 f$), então T é da ordem de f [i.e., $T(n)$ é $\theta(f(n))$].

A notação teta é usada para estimar a ordem de crescimento temporal ou espacial de um algoritmo e é mais precisa do que as notações ó e ômega, pois estima um limite inferior e outro superior. Apesar disso, muitas vezes, por negligência, a notação teta não é mencionada explicitamente em análise de algoritmos. Nessa situações, quando se faz referência à notação ó, a verdadeira intenção é referir-se à notação teta.

A **Figura 6–3** ilustra a relação entre duas funções reais $T(n)$ e $f(n)$ quando $T(n)$ é $\theta(f(n))$. Nessa figura, a porção tracejada do gráfico de $T(n)$ é aquela que satisfaz as relações $T(n) > c_0 f(n)$ [i.e., $T(n)$ é $\Omega(f(n))$] e $T(n) < c_1 f(n)$ [i.e., $T(n)$ é $O(f(n))$].

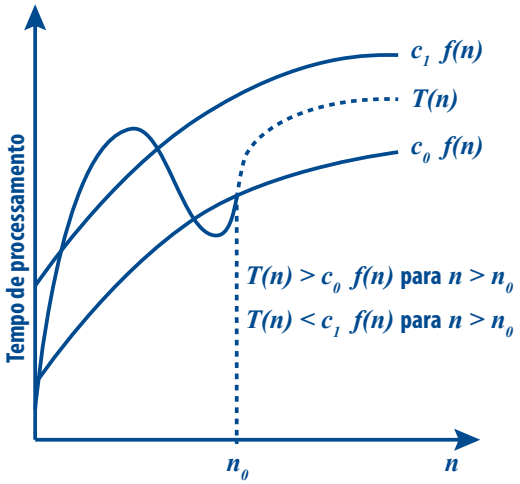
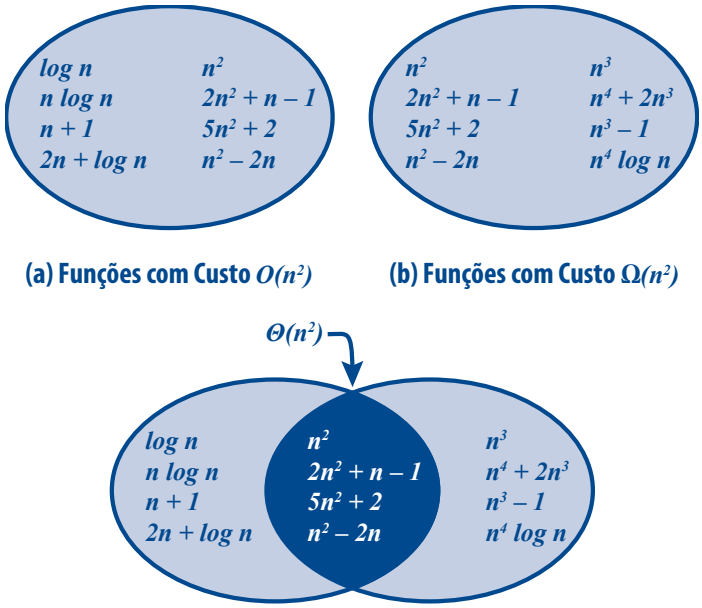


Figura 6–3: Interpretação Gráfica de Custo Teta

O uso de diagramas de Venn pode facilitar o entendimento das três notações as relações que elas apresentam entre si, como mostra a **Figura 6–4**.



(a) Funções com Custo $O(n^2)$ **(b) Funções com Custo $\Omega(n^2)$**

(c) Funções com Custo $\Theta(n^2) = O(n^2) \cap \Omega(n^2)$

Figura 6–4: Relação entre Notações Ó, Ômega e Teta Usando Diagramas de Venn

Exemplo 6.8 Mostre que a função $T(n) = 5n^2 + 3n + 2$ é $\theta(n^2)$.

Solução: De acordo com o **Exemplo 6.2**, $T(n)$ é $O(n^2)$ e de acordo com o **Exemplo 6.6**, $T(n)$ é $\Omega(n^2)$. Portanto, de acordo com a **Definição 6.3**, $T(n)$ é $\theta(n^2)$.

Existem ainda as notações ó pequeno, ômega pequeno e teta pequeno, mas essas notações raramente são usadas em análise de algoritmos. Assim, em nome da simplicidade linguística, neste livro, *notação ó* é o mesmo que *notação ó grande*, *notação teta* é o mesmo que *notação teta grande* e *notação ômega* é o mesmo que *notação ômega grande*.

Resumindo, a notação ó define uma função que é limite *superior* de outra, mas nada é informado com respeito à proximidade entre as duas funções. Assim, como foi visto, é correto dizer que, se $T(n) = 5n^2 + 3n + 2$, então $T(n)$ é $O(n^3)$, $T(n)$ é $O(n^{15})$, $T(n)$ é $O(2 \cdot n)$ etc. Desse modo, a notação ó não é muito útil. De modo semelhante, a notação ômega refere-se ao limite *inferior* de uma função e também não é muito informativa. Por exemplo, considerando-se a mesma função $T(n)$ acima, pode-se dizer que $T(n)$ é $\Omega(n)$, $T(n)$ é $\Omega(\log n)$, $T(n)$ é $\Omega(n^{1/2})$ etc. Por outro lado, a notação teta refere-se ao limite *estrito* de uma função e, portanto, não sofre com a frouxidão que aflige as notações ó e ômega. Por exemplo, pode-se dizer que, se $T(n) = 5n^2 + 3n + 2$, então $T(n)$ é $\theta(n^2)$ mas não é correto afirmar que $T(n)$ é $\theta(n^3)$.

As notações ó, ômega e teta são úteis porque, frequentemente, a função $T(n)$, que representa o custo de um algoritmo é complicada, mas encontrar uma função $f(n)$ tal que $T(n)$ seja $O(f(n))$, $\Omega(f(n))$ ou $\theta(f(n))$ é relativamente mais simples.

Exemplo 6.9 Mostre que $T(n) = 2n^4 + 5n^3 + 3$ é $\theta(n^4)$.

Solução: Para $n > 0$, tem-se que $2n^4 \leq 2n^4 + 5n^3 + 3$. Portanto, considerando-se $n_0 = 0$ e $c_0 = 2$, tem-se que $T(n)$ é $\Omega(n^4)$.

Agora, para $n > 1$, $2n^4 + 5n^3 + 3 \leq 2n^4 + 5n^4 + 3n^4 = 10n^4$. Logo, considerando-se $n_0 = 1$ e $c_1 = 10$, tem-se que $T(n)$ é $O(n^4)$. Como $T(n)$ é $\Omega(n^4)$ e $T(n)$ é $O(n^4)$, então $T(n)$ é $\theta(n^4)$.

Teorema 6.1 $f(n)$ é $\theta(g(n))$ se e somente se $g(n)$ é $\theta(f(n))$.

O **Teorema 6.1** quer dizer que se $f(n)$ é $\theta(g(n))$, então $f(n)$ e $g(n)$ são assintoticamente equivalentes; ou seja, tanto $f(n)$ é limite estrito de $g(n)$ quanto $g(n)$ é limite estrito de $f(n)$.

Corolário 6.1 $f(n)$ é $\theta(g(n))$ se e somente se $f(n)$ é $O(g(n))$ e $g(n)$ é $O(f(n))$.

Exemplo 6.10 Mostre que n^4 é $O(2n^4 + 5n^3 + 3)$.

Solução: No **Exemplo 6.9**, mostrou-se que $2n^4 + 5n^3 + 3$ é $\theta(n^4)$. Logo, de acordo com o **Corolário 6.1**, n^4 é $O(2n^4 + 5n^3 + 3)$.

Corolário 6.2 $f(n)$ é $\theta(g(n))$ se e somente se $f(n)$ é $\Omega(g(n))$ e $g(n)$ é $\Omega(f(n))$.

Exemplo 6.11 Mostre que n^4 é $\Omega(2n^4 + 5n^3 + 3)$.

Solução: No **Exemplo 6.9**, mostrou-se que $2n^4 + 5n^3 + 3$ é $\theta(n^4)$. Logo, de acordo com o **Corolário 6.2**, n^4 é $\Omega(2n^4 + 5n^3 + 3)$.

Exemplo 6.12 Mostre que n^2 não é $\theta(n)$.

Solução: De acordo com o **Corolário 6.1**, se n^2 é $\theta(n)$, então n^2 é $O(n)$. Mas, como foi provado no **Exemplo 6.3**, n^2 não é $O(n)$. Logo n^2 não pode ser $\theta(n)$.

Teorema 6.2 Suponha que $a_0, a_1, \ldots, a_k$ sejam números reais, com $a_k \neq 0$. Então, tem-se que: $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ é $\theta(n^k)$.

Exemplo 6.13 Mostre que $T(n) = 2n^4 + 5n^3 + 3$ é $\theta(n^4)$.

Solução: Esse é o mesmo **Exemplo 6.9**, mas, agora, a solução desse problema pode ser obtida imediatamente por meio do **Teorema 6.2**.

Exemplo 6.14 Mostre que o seguinte somatório é $\theta(n^2)$:

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

Solução: No **Apêndice B**, mostra-se que:

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

Mas, com um pouco de manipulação algébrica, obtém-se:

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

Logo, de acordo com o **Teorema 6.2**, tem-se que o referido somatório é $\theta(n^2)$.

Alguns julgam útil utilizar o seguinte artifício para memorizar os significados das notações ó, ômega e teta:

- A letra grega ômega (Ω) possui dois traços horizontais na parte *inferior*. Portanto a notação ômega grande define um limite *inferior*.
- A letra grega teta (θ) possui um traço horizontal em sua parte *central*. Assim a notação teta grande define um limite *central* (i.e., entre dois limites).
- A letra romana ó (O) possui traço na parte *superior* em alguns manuscritos. Logo a notação ó grande define um limite *superior*. (Esse último argumento é um tanto forçado, mas esse é apenas um artifício de memorização.)

A **Tabela 6–1** compara as três notações apresentadas neste capítulo.

Notação	Expressão	Interpretação
Ó	$T(n)$ **é** $O(f(n))$	**O custo temporal ou espacial representado pela função** $T(n)$ **é, *no máximo*, proporcional a** $f(n)$**, quando** n **se torna suficientemente grande. Em outras palavras, o tempo (ou espaço adicional) de execução não cresce mais rapidamente do que** $f(n)$**.**
Ômega	$T(n)$ **é** $\Omega(f(n))$	**O custo temporal ou espacial representado pela função** $T(n)$ **é, *no mínimo*, proporcional a** $f(n)$**, quando** n **se torna suficientemente grande. Em outras palavras, o tempo (ou espaço adicional) de execução não cresce mais lentamente do que** $f(n)$**.**
Teta	$T(n)$ **é** $\theta(f(n))$	**O custo temporal ou espacial representado pela função** $T(n)$ **é *proporcional* a** $f(n)$**, quando** n **se torna suficientemente grande. Em outras palavras, o tempo (ou espaço adicional) de execução cresce à mesma taxa que** $f(n)$**.**

Tabela 6–1: Comparação entre as Notações Ó, Ômega e Teta

6.4 Casos Melhor, Pior e Mediano

O desempenho de um algoritmo varia não apenas de acordo com a quantidade de dados que ele processa, como também varia de acordo com a *qualidade* desses dados. Suponha, por exemplo, que o problema em questão seja a ordenação de um array de inteiros em ordem crescente. Conforme será visto no **Volume 2**, existem diversos algoritmos diferentes, mas funcionalmente equivalentes, capazes de realizar essa tarefa. É intuitivo supor nesse exemplo que, quanto maior for o número de elementos do array (i.e., o tamanho dos dados de entrada), maior será o tempo gasto por qualquer algoritmo de ordenação para processar o array (i.e., ordená-lo). De fato, essa intuição corresponde à realidade, mas a taxa de crescimento entre esses algoritmos de ordenação varia drasticamente. Por outro lado, a relação de ordem original entre os elementos do array a ser ordenado (i.e., a qualidade

dos dados de entrada) também influencia o desempenho de um algoritmo de ordenação. Agora a intuição pode indicar que quanto mais previamente ordenado for um array a ser ordenado, melhor será o desempenho do algoritmo que irá ordená-lo. Porém essa intuição falha, pois existem algoritmos de ordenação que apresentam melhor desempenho quando os dados a ser ordenados estão completamente fora de ordem.

Em análise de algoritmos, a qualidade dos dados é classificada em três casos: melhor, pior e médio:

- **Melhor caso** é uma propriedade dos dados de entrada relativa a um algoritmo específico que resulta no *melhor* custo temporal ou espacial do algoritmo. Isto é, a melhor configuração dos dados para um algoritmo é aquela que faz com ele apresente seu melhor desempenho. Por exemplo, o melhor caso para o algoritmo de ordenação *Bubble Sort* (v. **Seção 3.12.2**) ocorre quando os dados a ser ordenados já estão quase ordenados. Isso pode parecer óbvio, mas, por incrível que possa parecer, nem sempre esse é o melhor caso para outros algoritmos de ordenação. Normalmente, é relativamente trivial determinar qual é o melhor caso de um algoritmo.
- **Pior caso** é uma propriedade dos dados de entrada que resultam no *pior* custo do algoritmo que os processam. Em outras palavras, pior caso de um algoritmo corresponde a um conjunto de dados de entrada para os quais o algoritmo exibe seu pior desempenho. Tal caso possui propriedades que impedem o algoritmo de funcionar eficientemente. Por exemplo, o pior caso para o algoritmo de ordenação *Bubble Sort* (v. **Seção 3.12.2**) ocorre quando os dados a ser ordenados estão totalmente fora de ordem. O pior caso de entrada para um algoritmo é normalmente fácil de ser identificado.
- **Caso médio** (ou **mediano)** é uma propriedade dos dados de entrada obtida por meio do cálculo de uma média de custo do algoritmo quando ele processa todos os conjuntos possíveis de dados. Ou seja, o caso médio de entrada é associado ao número médio de operações necessárias para resolver um problema calculado sobre todas as possíveis entradas desse problema que tenham um determinado tamanho. Na prática, muitas vezes, é difícil determinar com precisão o que constitui um caso médio de entrada e, assim, torna-se complicado analisar o comportamento de um algoritmo sob essa circunstância. Uma simplificação frequentemente utilizada para cálculo de custo de caso médio consiste em considerar equiprováveis todas as entradas de dados. Então, determina-se o número de operações executadas para cada entrada, somam-se esses custos e divide-se pelo número total de entradas. Mas, esse raciocínio só se aplica quando a probabilidade de ocorrência de todas as entradas é a mesma, o que nem sempre é o caso. Assim, tipicamente, determinar o significado do caso médio de entrada não é uma tarefa trivial[1].

Exemplo 6.15 A função escrita em C apresentada a seguir encontra o número de ocorrências de um determinado valor num array de inteiros. Quais são os casos pior, melhor e médio dos dados para essa função?

```
int Ocorrencias(const int ar[], int tam, int num)
{
   int i, ocorr = 0;

   for (i = 0; i < tam; ++i)
      if (ar[i] == num)
         ++ocorr;

   return ocorr;
}
```

[1] Do ponto de vista formal de Estatística, o custo médio de um algoritmo é seu custo médio esperado, também conhecido como esperança matemática ou expectância. Esse custo representa aquele que é esperado quando o algoritmo é aplicado muitas vezes sobre os mesmos dados de entrada. Se todas essas aplicações tiverem a mesma probabilidade, o custo esperado será a média aritmética (v. **Exemplo 6.15**).

Solução: (a) Neste problema, qualquer que seja o caso de entrada considerado, todos os elementos do array precisam ser examinados. Portanto os três casos são equivalentes e o custo temporal do algoritmo é $\theta(n)$ nos três casos (v. **Seção 6.11.2**).

Exemplo 6.16 Qual é (a) o pior caso, (b) o melhor caso e (c) o caso mediano de um algoritmo que verifica se um determinado valor inteiro se encontra armazenado num array de elementos inteiros.

Solução: Este exemplo é parecido com o **Exemplo 6.15**, mas agora o que se deseja saber apenas é se o valor em questão ocorre ou não num array, de modo que nem sempre todos os elementos do array precisam ser examinados. Assim o pior caso desse algoritmo ocorre quando o valor procurado é o último elemento do array ou ele não se encontra no array [o custo temporal desse caso é $\theta(n)$ — v. **Seção 6.11.2**]. Por outro lado, o melhor caso acontece quando o valor procurado é o primeiro elemento do array [nesse caso, o custo temporal é $\theta(1)$ — v. **Seção 6.11.1**]. Intuitivamente, o caso mediano ocorre quando o algoritmo examina, em média, metade dos elementos do array [aqui, o custo temporal é novamente $\theta(n)$][2].

É importante salientar que aquilo que constitui pior caso de entrada para um algoritmo pode ser considerado melhor caso para outro algoritmo funcionalmente equivalente e vice-versa. Também é comum ter-se um algoritmo para o qual não haja distinção entre casos (i.e., o custo temporal ou espacial do algoritmo independe da configuração dos dados de entrada, como mostra o **Exemplo 6.15**).

Muitas vezes, avaliar um algoritmo considerando seu desempenho no melhor caso não é muito frutífero, pois a maioria dos algoritmos funcionalmente equivalentes apresentam desempenhos equivalentes quando o melhor caso é levado em consideração. Assim, nesse caso, não se deve esperar que algum algoritmo prevaleça com relação a outro, de modo que, raramente, considerar o melhor caso serve para comparar algoritmos. Apesar disso, existem circunstâncias nas quais o melhor caso de um algoritmo é levado em consideração em sua análise. Por outro lado, levar o pior caso em consideração na análise de um algoritmo garante sempre que o algoritmo não terá desempenho pior do que aquele determinado durante sua análise.

6.5 Funções Comuns em Análise de Algoritmos

Existem muitas classes de complexidade algorítmica que aparecem com frequência em análise de algoritmos e estruturas de dados. Algoritmos podem ser classificados de acordo com seus custos temporais ou espaciais, e a **Tabela 6–2** apresenta classes de crescimento de funções comumente encontradas em análise de algoritmos.

Custo	Denominação	Comentários
$\theta(1)$	Constante	❒ **Um algoritmo com custo constante executa um número fixo de operações, de modo que o tempo gasto pelo algoritmo é independente do tamanho dos dados de entrada** ❒ **Exemplo: acesso a um elemento de array** (v. **Seção 3.3**)

Tabela 6–2: Classes de Crescimento de Funções Frequentes em Programação

[2] A análise formal do caso mediano é um tanto complicada e será apresentada no **Volume 2**.

Custo	Denominação	Comentários
$\theta(\log n)$	Logarítmico	❒ Um algoritmo dessa categoria reduz continuamente o tamanho do problema ou divide um conjunto de dados ao meio, essa metade é dividida novamente ao meio, e assim por diante ❒ Quando n dobra, o custo aumenta um valor constante independente de n, pois: $\log 2 \cdot n = \log 2 + \log n$ ❒ A base do logaritmo é irrelevante porque, como mostra o Apêndice B, todos os logaritmos com base constante diferem entre si por um valor constante ❒ **Exemplos:** busca binária (v. Seção 6.11.4) e exponenciação por quadratura (v. Seção 6.11.8)
$\theta(n)$	Linear	❒ O custo do algoritmo é diretamente proporcional ao tamanho dos dados ❒ **Exemplo:** acesso sequencial a cada elemento de um array unidimensional (v. Seção 3.1)
$\theta(n \log n)$	Linear logarítmico	❒ Um algoritmo dessa categoria invoca outro algoritmo com custo logarítmico n vezes ❒ **Exemplo:** busca binária num array bidimensional contendo n listas, cada uma das quais com n elementos ordenados
$\theta(n^2)$	Quadrático	❒ Tipicamente, um algoritmo desse tipo possui um laço de repetição aninhado em outro ❒ **Exemplo:** acesso sequencial a cada elemento de um array bidimensional contendo n linhas e n colunas (v. Seção 3.6)
$\theta(n^3)$	Cúbico	❒ Tipicamente, um algoritmo desse tipo possui um laço de repetição aninhado em outro laço que, por sua vez, também é aninhado em outro laço. Tal algoritmo é impraticável quando n é muito grande. ❒ **Exemplo:** multiplicação de matrizes representadas como arrays bidimensionais
$\theta(n^c)$	Polinomial	❒ Quando $c > 3$, um algoritmo dessa natureza pode ser impraticável ❒ **Exemplo:** não há exemplo no escopo deste livro
$\theta(c^n)$	Exponencial	❒ Um algoritmo desse tipo tem utilidade apenas para entradas muito pequenas, mas existem muitos problemas para os quais, até o momento, só existem algoritmos dessa natureza ❒ **Exemplo:** um algoritmo que gera todos os subconjuntos de um conjunto ou resolve o problema das torres de Hanói (v. Seção 6.11.6) tem custo temporal $\theta(2^n)$
$\theta(n!)$	Fatorial	❒ Um algoritmo dessa natureza é útil apenas para entradas muito pequenas ❒ **Exemplo:** um algoritmo que gera todas as permutações de um conjunto

Tabela 6–2: Classes de Crescimento de Funções Frequentes em Programação

A **Figura 6–5** mostra a aparência gráfica de algumas funções que aparecem na **Tabela 6–2** (supondo, evidentemente, que essas funções fossem reais).

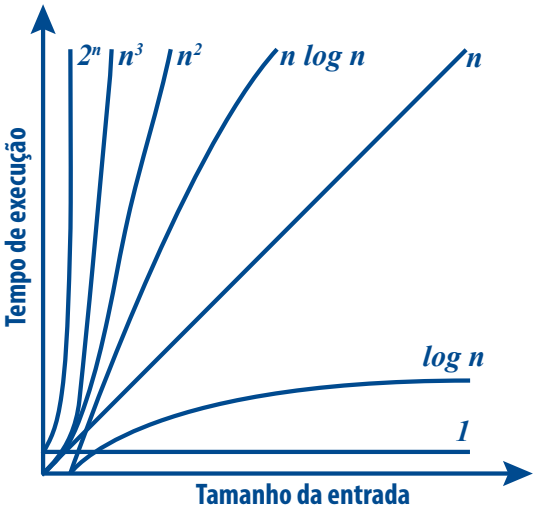


Figura 6–5: Gráficos de Funções Comuns em Análise de Algoritmos

A classificação de algoritmos apresentada na **Tabela 6–2** é comum, mas não é completa. Por exemplo, existem algoritmos com custo temporal igual a $\theta(1.5^n)$ (v. **Seção 6.11.7**) e algoritmos com custo temporal igual a $\theta(n^{1.5})$[3].

Um fato que costuma intrigar iniciantes é a presença de funções logarítmicas em análise de algoritmos. Algoritmos com custos temporais $\theta(\log n)$ incluem um laço de repetição no corpo do qual, a cada iteração, uma porção dos dados de entrada (frequentemente, a metade) é descartada. Um exemplo clássico de algoritmo com tal comportamento é aquele que efetua busca binária (v. **Seção 6.11.4**).

A **Tabela 6–3** mostra o tempo aproximado para execução de $f(n)$ operações quando $f(n)$ representa algumas funções comuns em análise de algoritmos. Aqui, supõe-se que cada operação leva 1 nanossegundo (i.e., 10^{-9} de um segundo) para ser concluída.

$f(n)$	$n = 10$	$n = 1.000$	$n = 100.000$	$n = 10.000.000$
$\log_2 n$	$3{,}3 \times 10^{-9}$ s	10^{-8} s	$1{,}7 \times 10^{-8}$ s	$2{,}3 \times 10^{-8}$ s
n	10^{-8} s	10^{-6} s	0,0001 s	0,01 s
$n \log_2 n$	$3{,}3 \times 10^{-8}$ s	10^{-5} s	0,0017 s	0,23 s
n^2	10^{-7} s	0,001 s	10 s	27,8 minutos
n^3	10^{-6} s	1 s	11,6 dias	31.710 anos
2^n	10^{-6} s	$\approx 10^{284}$ anos	$\approx 10^{30.084}$ anos	$\approx 10^{3.010.277}$ anos

Tabela 6–3: Tempo Estimado para Execução de Operações Hipotéticas

Podem-se usar as estimativas de tempo apresentadas na **Tabela 6–3** para prever se um determinado algoritmo com custo temporal conhecido apresentará uma solução em tempo razoável para um problema de tamanho específico quando for convertido em programa e executado num computador. Algoritmos com custo temporal polinomial podem ser beneficiados com a melhora de processamento dos computadores, mas essa melhora em poder de processamento ajuda pouco na superação das dificuldades encontradas por algoritmos com custo exponencial ou fatorial.

A **Tabela 6–3** sugere que algoritmos com alta taxa de crescimento [p.ex., com custo $\theta(2^n)$] têm aplicabilidade prática apenas para valores relativamente muito pequenos. Por exemplo, mesmo que a suposta celeridade de

[3] Um algoritmo que apresenta esse custo temporal é o algoritmo de ordenação de Shell, que não é explorado nesta obra.

execução de instruções seja multiplicada por mil, o tempo gasto por um algoritmo com custo temporal $\theta(2^n)$ para processar um tamanho de entrada maior do que 1.000 continuará sendo um número impronunciável de anos. Por outro lado, considerando essa hipótese, o tempo gasto por um algoritmo com custo $\theta(n^3)$ levaria *apenas* 31 anos para completar sua tarefa de processar uma entrada de tamanho igual a 10.000.000.

Esses exemplos demonstram a importância da análise de custos de algoritmos e que ela não pode ser negligenciada com base no fato de a tecnologia computacional ter evoluído constantemente e muito rapidamente. Em resumo, eficiência de um computador é de pouca valia se os programas que ele executa usam algoritmos ineficientes.

Exemplo 6.17 Um algoritmo consome 0,5 ms quando o tamanho dos dados de entrada que ele processa é 100. Qual é o tamanho do problema que pode ser resolvido em 1 min quando o custo temporal desse algoritmo é: (a) linear, (b) quadrático e (c) cúbico.

Solução: Sabe-se que $t_1 = 0{,}5\,ms = 5.10^{-4}\,s$, $n_1 = 100$ e $t_2 = 1\,min = 60\,s$. Além disso, tem-se que o custo temporal do algoritmo é dado por: $T(n) = c \cdot f(n)$, em que c é uma constante.

(a) Sendo o custo temporal linear, tem-se que: $T(n) = c \cdot n$. Assim as seguintes equações são válidas:

[1] $5.10^{-4} = c.100$

e

[2] $60 = c \cdot n$

Resolvendo o sistema formado pelas equações **[1]** e **[2]**, tem-se que o tamanho do problema que pode ser resolvido em 1 min é: $n = 1{,}2.10^7$.

Os demais itens são resolvidos de modo semelhante e são deixados como exercícios para o leitor. As respostas são: (b) $n = 3464$ e (c) $n = 229$.

6.6 Resultados Teóricos Importantes

Tipicamente, algoritmos são constituídos de várias instruções, de modo que determinar o custo de um algoritmo requer combinar os custos das instruções que o constituem. A seguir, serão apresentados alguns teoremas importantes referentes à análise assintótica de algoritmos. A importância desses resultados reside no fato de eles terem aplicação imediata na avaliação dessas combinações de instruções.

Teorema 6.3 Se $T(n)$ é $c_1\, f(n) + c_2$, então $T(n)$ é $\theta(f(n))$.

O **Teorema 6.3** quer dizer que constantes multiplicativas e aditivas podem ser ignoradas em análise de algoritmos, pois elas são irrelevantes. Isso ocorre porque, à medida que o valor de n cresce, essas constantes contribuem cada vez menos para o custo de processamento.

Para entender a razão pela qual constantes podem ser desprezadas, considere a **Tabela 6–4**. Essa tabela sugere que, à medida que o valor assumido pela variável n cresce, a importância dos coeficientes *500* e *1000* se torna cada vez menor.

n	$T(n) = 500n^2 + 1000$	Contribuição da constante *500*	Contribuição da constante *1000*
1	*1500*	*33%*	*67%*
10	*51000*	*1%*	*2%*
100	*5001000*	*0,01%*	*0,02%*

Tabela 6–4: Por Que Constantes Podem Ser Desprezadas na Notação Ó?

Exemplo 6.18 Mostre que a função $T(n) = 500n^2 + 1000$ é $\theta(n^2)$.

Solução: É fácil mostrar que essa função é $\theta(500n^2 + 1000)$. Mas, de acordo com o **Teorema 6.3**, $\theta(500n^2 + 1000)$ é $\theta(n^2)$.

Teorema 6.4 Se $T_1(n)$ é $\theta(f(n))$ e $T_2(n)$ é $\theta(g(n))$, então $T_1(n) + T_2(n)$ é $\theta(max(f(n), g(n)))$.

O **Teorema 6.4** é denominado **regra da soma** e sua consequência prática é que se pode avaliar o custo de execução de uma sequência de instruções, estimando-se primeiro o custo de execução de cada instrução e, então, considerando-se o custo de toda a sequência de instruções como o maior desses custos.

Exemplo 6.19 Um determinado algoritmo é dividido em dois trechos. Um trecho desse algoritmo tem custo temporal $T_1(n)$ dado por $\theta(n^3)$ ao passo que o outro trecho tem custo temporal $T_2(n)$ dado por $\theta(n^2)$. Qual é o custo temporal desse algoritmo?

Solução: O custo temporal do algoritmo é $\theta(n^3) + \theta(n^2)$, que, de acordo com o **Teorema 6.4**, é $\theta(max(n^3, n^2)) = \theta(n^3)$, visto que $n^3 \geq n^2$, $\forall\ n \geq 0$.

Teorema 6.5 Se $T_1(n)$ é $\theta(f(n))$ e $T_2(n)$ é $\theta(g(n))$, então $T_1(n) \cdot T_2(n)$ é $\theta(f(n) \cdot g(n))$.

O **Teorema 6.5** é denominado **regra do produto** e uma aplicação prática importante desse teorema é a análise temporal de laços repetição aninhados.

Exemplo 6.20 Mostre que $T(n) = 5n \log (n!)$ é $\theta(n^2 \cdot \log n)$.

Solução: Sejam $T_1(n) = 5n$ e $T_2(n) = \log (n!)$. Então, tem-se que $T(n) = T_1(n)\ T_2(n)$. Mas, $T_1(n)$ é $\theta(n)$, de acordo com o **Teorema 6.3**, e $T_2(n)$ é $\theta(\log n^n) = \theta(n \cdot \log n)$. Esse último resultado é obtido utilizando um raciocínio semelhante àquele do **Exemplo 6.4**. Portanto, em consonância com o **Teorema 6.5**, $T(n)$ é $\theta(n \cdot n \log n) = \theta(n^2 \cdot \log n)$.

Teorema 6.6 n^k é $O(c^n)$, para $k > 0$ e $c > 1$.

O **Teorema 6.6** informa que nenhuma função polinomial é maior do que uma função exponencial por maior que seja o grau do polinômio (v. exemplo abaixo).

Exemplo 6.21 Mostre que n^{1000} é $O(1.0001^n)$.

Solução: Como $1.0001 > 1$, de acordo como o **Teorema 6.6**, n^{1000} é $O(1.0001^n)$.

Teorema 6.7 Se $T(n)$ é $\theta(f(n))$ e $f(n)$ é $\theta(g(n))$, então $T(n)$ é $\theta(g(n))$.

O **Teorema 6.7** é denominado **regra de transitividade** e pode ser parafraseado como: $\theta(\theta(g(n)))$ é $\theta(g(n))$.

Exemplo 6.22 Suponha que $T(n)$ é $\theta(f(n))$ e $f(n)$ é n^2. Avalie a complexidade de $T(n)$.

Solução: De acordo com o **Teorema 6.7**, $T(n)$ é $\theta(n^2)$.

Quando a taxa de crescimento de um algoritmo é constante (i.e., ela independe do tamanho da entrada), seu custo é $\theta(1)$. Isso significa que $\theta(c)$ é o mesmo que $\theta(1)$ para qualquer que seja a constante c. Por exemplo, o acesso a um elemento de um array requer apenas a avaliação de uma expressão que independe do tamanho do array (v. **Seção 3.3**). Portanto o custo de uma instrução dessa natureza é $\theta(1)$.

6.7 Regras Práticas para Análise Temporal de Algoritmos

Apesar de não haver nenhum conjunto completo de regras para análise de algoritmos, as regras práticas que serão apresentadas adiante ajudam bastante nessa tarefa.

Quando se inspeciona algum algoritmo com o objetivo de se determinar seu custo temporal, o foco deve ser naquelas instruções que dependem do tamanho da entrada, pois todas aquelas que não apresentam essa dependência têm custo temporal $\theta(1)$. Quando todas as instruções de um algoritmo têm custo temporal $\theta(1)$, o algoritmo inteiro também tem custo temporal global $\theta(1)$. Além disso, quando um algoritmo possui instruções que *não* têm custo temporal $\theta(1)$, as instruções avaliadas como $\theta(1)$ podem ser ignoradas, de acordo com o **Teorema 6.4**.

6.7.1 Análise Temporal de Instruções Simples

Normalmente, o custo temporal de uma instrução na maioria das linguagens de programação pode ser considerado $\theta(1)$, mas existem exceções, como, por exemplo, atribuições entre estruturas e instruções que envolvem chamadas de algumas funções que não podem ser consideradas $\theta(1)$ (v. adiante).

6.7.2 Análise Temporal de Sequências de Instruções

O custo temporal de uma sequência de instruções é determinado pela regra da soma (**Teorema 6.4**). Isto é, considera-se o maior custo temporal dentre todos os custos temporais das instruções que fazem parte da sequência.

6.7.3 Análise Temporal de Estruturas de Controle

Desvios incondicionais são estruturas de controle que não possuem corpos; i.e., essas estruturas de controle não possuem outras instruções subordinadas a elas. Todas as demais estruturas de controle possuem corpos.

Usualmente, estruturas de controle que possuem corpos são avaliadas de dentro para fora (principalmente laços de repetição). Ou seja, estima-se primeiro o custo temporal do corpo da estrutura e depois estima-se o custo temporal da estrutura como um todo.

Desvios Condicionais

O custo temporal de uma instrução condicional *se-então-senão* (**if-else** em C) é determinado pelo custo de avaliação da expressão condicional [normalmente, $\theta(1)$] mais o maior custo temporal dentre as instruções que compõem as partes *se-então* (**if**) e *senão* (**else**). Por exemplo, se o custo de avaliação da expressão condicional for $\theta(1)$, o custo temporal das instruções da parte *se-então* (**if**) for $\theta(n)$ e o custo temporal das instruções da parte *senão* (**else**) for $\theta(n^2)$, então o custo temporal da instrução condicional será $\theta(1) + \theta(n^2)$, que, pela regra da soma (**Teorema 6.4**), é $\theta(n^2)$. Evidentemente, se uma instrução condicional não tiver parte *senão* (**else**), consideram-se apenas os custos temporais da avaliação da condição e da execução das instruções que fazem parte do corpo da instrução condicional.

Estruturas de seleção (**switch-case** em C) são avaliadas de modo semelhante a instruções condicionais *se-então-senão*, já que estruturas de seleção são desvios condicionais múltiplos.

Exemplo 6.23 Suponha que, no trecho de programa a seguir, a função `F()` tenha custo temporal $\theta(n^2)$ e o custo temporal da função `G(n)` seja $\theta(n \log n)$. Mostre que esse trecho de programa tem custo $\theta(n^2)$.

```
if (x < 0)
   F();
else
   G();
```

Solução: O custo dessa instrução **if-else** é a soma do custo de avaliação de sua expressão condicional, que é $\theta(1)$, com o maior custo entre as partes **if** [que tem custo $\theta(n^2)$] e **else** [que tem custo $O(n \log n)$]. Agora, no **Exemplo 6.5**, mostrou-se que $n \log n$ é $\theta(n^2)$. Logo o custo dessa instrução é $\theta(1) + \theta(n^2)$, que, de acordo com o **Teorema 6.4**, é $\theta(n^2)$.

Desvios Incondicionais e Instruções de Retorno

Qualquer desvios incondicional ou instrução **return** tem custo temporal $\theta(1)$. A exceção para essa regra ocorre quando uma expressão que acompanha uma instrução **return** tem custo temporal que não pode ser considerado constante, como por exemplo:

```
return realloc(p, tamanho);
```

Nesse exemplo, o custo da instrução **return** corresponderá ao custo da chamada de **realloc()**, que não é $\theta(1)$ (v. **Seção 9.2**).

Laços de Repetição

O custo temporal de um laço de repetição é a soma sobre o número de vezes que o laço é executado do custo temporal da sequência de instruções que compõe o corpo do laço mais o custo temporal de avaliação da condição de parada do laço. Usualmente, essa estimativa é o produto do número de vezes que o laço é executado pelo custo temporal da instrução de maior custo temporal dentro do corpo do laço, mas, algumas vezes, o número de repetições do laço não é fácil de determinar.

Frequentemente, o custo temporal de avaliação da condição de parada do laço é $\theta(1)$, mas existem situações nas quais é exatamente esse custo que determina o custo de todo o laço (v. **Exemplo 6.25**).

Exemplo 6.24 Mostre que o laço **for** a seguir tem custo temporal $\theta(n)$.

```
for (i = 0; i < n; ++i)
   soma += ar[i];
```

Solução: A instrução **for** acima efetua as seguintes operações:

1. Uma atribuição inicial: `i = 0`
2. $n + 1$ avaliações da expressão: `i < n`
3. n incrementos de `i`: `++i`
4. n atribuições com soma na instrução: `soma += ar[i]`

Portanto o número de instruções elementares com custo temporal $\theta(1)$ é:

$1 + n + 1 + n + n = 3n + 2$

Logo $T(n) = 3n + 2$ e, de acordo como o **Teorema 6.2**, $T(n)$ é $\theta(n)$.

6.7.4 Análise Temporal de Chamadas de Funções

Chamadas de funções são avaliadas de acordo com a próprio custo temporal da função.

A maioria das chamadas das funções da biblioteca padrão de C têm custo temporal $\theta(1)$, mas há notáveis exceções, dentre as quais:

- **qsort()** (v. **Seção 11.5**) tem custo temporal $\theta(n \log n)$, em que n é o tamanho do array sendo ordenado. Se o array estiver quase ordenado (pior caso), o custo da função **qsort()** pode ser $\theta(n^2)$ (dependendo de como ela foi implementada).
- **bsearch()** (v. **Seção 11.5**) tem custo temporal $\theta(\log n)$, em que n é o tamanho do array no qual é efetuada a busca.
- A maioria das operações sobre strings implementadas como funções declaradas no cabeçalho `<string.h>` (v. **Seção 3.7**) tem custo temporal $\theta(n)$, em que n é o número de caracteres do string sendo processado.

- No pior casos, **realloc()** (v. **Seção 9.2**) tem custos temporal e espacial $\theta(n)$, sendo n o número de bytes do espaço sendo redimensionado.

Exemplo 6.25 A função a seguir provavelmente constitui a implementação mais comum da função **strcpy()** da biblioteca padrão de C. Determine seu custo temporal.

```
char *strcpy(char *s1, const char *s2)
{
   char *retorno = s1;
   while (*s1++ = *s2++)
      ;
   return retorno;
}
```

Solução: Embora não apareça nenhum valor n que indique o tamanho dos dados processados pela função **strcpy()**, claramente o tempo gasto por essa função depende do tamanho do string copiado. Mais precisamente, a expressão do laço **while** é executada $n + 1$ vezes, em que n é o comprimento do string. Como esse laço é a instrução dominante da função e constantes aditivas não são importantes em análise de algoritmos, o custo temporal da função é $\theta(n)$, em que n é o comprimento do string.

6.8 Análise de Custo Espacial de Algoritmos

Análise espacial de um algoritmo só se faz necessária quando ele usa, explicitamente, espaço adicional que depende do tamanho do problema ou se o algoritmo é recursivo e, assim, implicitamente, usa espaço adicional. Em qualquer outra situação, o custo de um algoritmo em termos de espaço é $\theta(1)$, mesmo que o algoritmo use muitas variáveis auxiliares.

Qualquer algoritmo recursivo usa espaço adicional proporcional ao tamanho dos dados que ele recebe como entrada. Mais precisamente, o custo espacial de um algoritmo recursivo depende do número de chamadas recursivas que ele executa, pois, a cada chamada recursiva, ocorre a criação de um registro de ativação (v. **Seção 4.3**).

Exemplo 6.26 A função `ExibeArquivoInvNaTelaRec()` apresentada a seguir exibe, recursivamente, um arquivo de texto invertido na tela. Mostre que o algoritmo implementado por essa função a seguir tem custo espacial $\theta(n)$, em que n é o número de bytes do arquivo lidos.

```
void ExibeArquivoInvNaTelaRec(FILE *stream)
{
   int c;
   c = fgetc(stream);
   if ( !feof(stream) && !ferror(stream) )
      ExibeArquivoInvNaTelaRec(stream);
   putchar(c);
}
```

Solução: A função em questão foi discutida na **Seção 4.8.7**. Essa função possui apenas uma chamada recursiva que é levada a efeito enquanto não ocorre erro de leitura nem o final do arquivo lido é atingido. Portanto o número de registros de ativação criados por essas chamadas recursivas corresponde exatamente ao número n de bytes lidos no arquivo. Logo o espaço adicional utilizado pela função é $c.n$, em que c é uma constante que representa o tamanho de cada registro de ativação criado. Isso mostra que o custo espacial dessa função é $\theta(cn)$, que é o mesmo que $\theta(n)$, de acordo com o **Teorema 6.3**.

A análise espacial de algoritmos iterativos é relativamente fácil, mas nem sempre a análise espacial de um algoritmo recursivo é tão fácil quanto aquela apresentada no último exemplo.

6.9 Algoritmos Recursivos e Relações de Recorrência

A análise temporal de algoritmos recursivos é bem mais complicada do que o que foi visto acima para algoritmos não recursivos. Analisar um algoritmo recursivo requer, em primeiro lugar, associá-lo a uma **relação de recorrência** que define condições matemáticas que seu custo temporal deve satisfazer. Mais precisamente, a função *T(n)* que descreve o custo temporal do algoritmo deve ser escrita como equações que precisam ser atendidas. Em seguida, essas equações devem ser resolvidas para que o custo do algoritmo seja determinado. Esse último passo raramente é trivial. Por isso, o leitor deve considerar esta e a próxima seção apenas como um estudo introdutório do assunto.

Exemplo 6.27 Considere a função recursiva a seguir que calcula o fatorial de um número natural. Determine o custo temporal dessa função[4].

```
int Fatorial(int n)
{
   if (n == 0)
      return 1;
   else
      return n*Fatorial(n - 1);
}
```

Solução: Para não haver confusão de terminologia com a denominação *função* em matemática, doravante, a função `Fatorial()` será referida como *algoritmo*.

Uma rápida inspeção do algoritmo acima indica que seu custo temporal está associada ao número de vezes que a operação de multiplicação é efetuada para cada valor de n. Mas, para um dado valor de n, o número de multiplicações efetuadas é o número de multiplicações efetuadas pela chamada `Fatorial(n - 1)` mais 1, levando em consideração que o valor retornado por `Fatorial(n - 1)` é multiplicado por `n`. Portanto, se o número de multiplicações para um dado valor de n for representado por T_n, tem-se que:

$$T_n = T_{n-1} + 1$$

Essa última equação é denominada **equação de recorrência** porque o valor da função para n é determinado pelo valor da função para um valor menor de n (no caso em questão, $n - 1$). Agora, uma equação de recorrência apenas não é suficiente para definir uma função única. Quer dizer, é necessário ter uma ou mais **condições iniciais**, que são valores conhecidos da função que não dependem de n. No caso do algoritmo sob escrutínio, sabe-se que, quando n é igual a 0, não ocorre nenhuma multiplicação. Logo a única condição inicial da relação de recorrência pode ser escrita como:

$$T_0 = 0$$

Resumindo, o algoritmo em questão está associado à seguinte relação de recorrência:

$$T_n = \begin{cases} 0 & se\ n = 0 \\ T_{n-1} + 1 & se\ n > 0 \end{cases}$$

Encontrar uma **solução** para essa relação de recorrência significa encontrar um valor para T_n que dependa apenas de n (i.e., que não dependa de T_{n-1}). Existem várias técnicas de resolução de relações de recorrência. A técnica a ser utilizada aqui é relativamente simples, mas sua utilidade é bem limitada. Essa técnica consiste em aplicar a relação de recorrência para alguns valores de n e tentar

[4] **NB:** Essa função não está corretamente implementada, pois ela não termina quando `n < 0`. Ela foi propositadamente implementada assim apenas para facilitar sua análise.

encontrar um padrão de resultados que permita propor uma solução (**conjectura**) para essa relação de recorrência. Em seguida, tenta-se provar por meio de **indução matemática** (v. **Apêndice B**) que essa conjectura é verdadeira.

Aplicando-se a relação de recorrência sob escrutínio sobre alguns valores de n, obtêm-se os seguintes resultados:

$T_1 = T_{1-1} + 1 = T_0 + 1 = 0 + 1 = 1$
$T_2 = T_{2-1} + 1 = T_1 + 1 = 1 + 1 = 2$
$T_3 = T_{3-1} + 1 = T_2 + 1 = 2 + 1 = 3$
$T_4 = T_{4-1} + 1 = T_3 + 1 = 3 + 1 = 4$

Observando os resultados obtidos acima, suspeita-se que a solução para a relação de recorrência é:

$T_n = n$

Assim tenta-se provar utilizando indução matemática que essa conjectura realmente é a solução procurada. Essa prova será apresentada a seguir. (Se você desconhece prova por indução, consulte o **Apêndice B** antes de prosseguir.)

Base da indução. Para $n = 0$, tem-se que $T_0 = 0$, o que corresponde à primeira equação da relação de recorrência.

Hipótese indutiva. Suponha que a conjectura é válida para $n = k$ (i.e., $T_k = k$).

Passo indutivo. Tem-se que $T_{k+1} = T_{k+1-1} + 1 = T_k + 1 = k + 1$, em que a primeira igualdade é decorrente da segunda equação da relação de recorrência e a última igualdade é consequência da hipótese indutiva.

A técnica de **tentativa e erro** para resolução de relações de recorrência apresentada acima também é denominada de **método de substituição**. Como já foi dito, a análise de algoritmos recursivos raramente é trivial, pois nem sempre a relação de recorrência associada a um algoritmo recursivo é tão fácil de resolver como no exemplo apresentado nesta seção.

6.10 Árvores de Recursão

6.10.1 Conceitos

Uma **árvore de recursão** é uma representação gráfica de uma relação de recorrência[5]. O uso de interpretações algorítmicas para relações de recorrência facilita o entendimento de árvores de recursão. Tal interpretação enxerga uma equação de recorrência como um algoritmo recursivo que resolve um problema de tamanho n. A **Tabela 6–5** mostra três relações de recorrência e suas respectivas interpretações algorítmicas, sem incluir os casos-base dessas relações.

Equação de Recorrência	Interpretação
$T(n) = 2 \cdot T(n/2) + n$	Para resolver um problema de tamanho n, deve-se resolver dois problemas de tamanho $n/2$ e executar n unidades adicionais de processamento
$T(n) = 3 \cdot T(n - 2) + 1$	Para resolver um problema de tamanho n, deve-se resolver três problemas de tamanho $n - 2$ e executar uma unidade adicional de processamento
$T(n) = T(n/4) + n^2$	Para resolver um problema de tamanho n, deve-se resolver um problema de tamanho $n/4$ e executar n^2 unidades adicionais de processamento

Tabela 6–5: Interpretações Algorítmicas de Relações de Recorrência

[5] Se você tiver dificuldade para entender intuitivamente o que é uma árvore, leia as primeiras duas paginas do **Capítulo 12**.

Na seção corrente, **tamanho de um problema** é o mesmo que tamanho da entrada de um problema.

A **Figura 6–6** mostra os quatro níveis iniciais de uma árvore que representa a relação de recorrência $T(n) = 2 \cdot T(n/2) + n$. Cada nível de uma árvore de recursão representa um nível de recursão da relação de recorrência que a árvore representa e possui cinco partes:

[1] A primeira parte à esquerda do gráfico representa o **número de problemas**

[2] A segunda parte à esquerda do gráfico é o **tamanho do problema**

[3] A **representação gráfica** da árvore ocupa a parte central acompanhada da respectiva relação de recorrência para facilidade de referência

[4] A primeira parte à direita do gráfico representa o **custo temporal de resolução de cada problema** daquele nível

[5] A última parte à direita do gráfico é o **custo temporal de resolução de todos os problemas** do respectivo nível

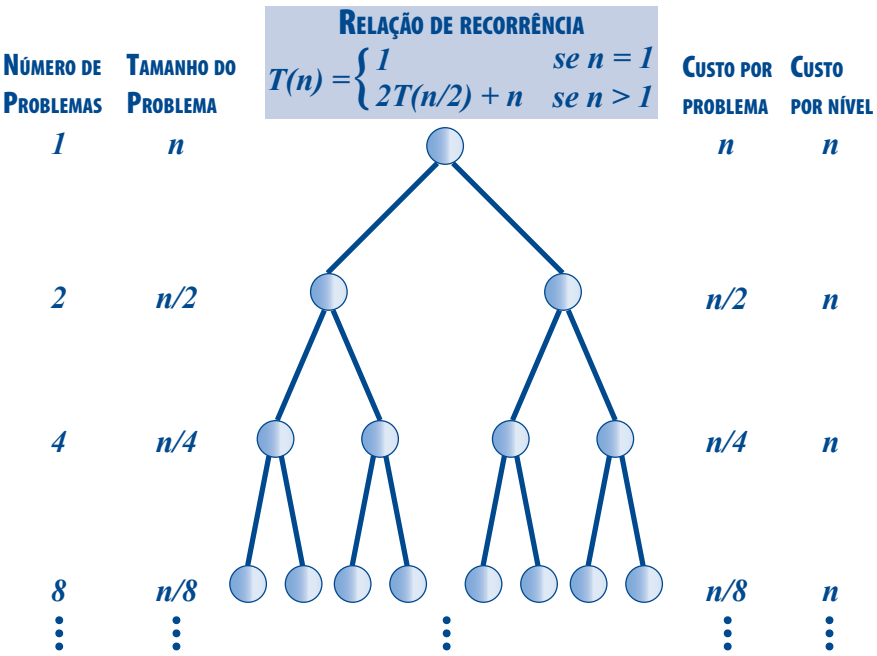


Figura 6–6: Quatro Níveis de uma Árvore de Recursão

Numa árvore de recursão, cada custo é expresso em número de supostas operações na interpretação algorítmica da respectiva relação de recorrência.

A árvore de recursão da **Figura 6–6** indica que, no primeiro nível (i.e., **nível 0**) de recursão, há apenas um problema de tamanho n a ser resolvido. Por isso, o custo temporal total desse nível corresponde a n operações. O próximo nível representa a divisão do problema original em dois subproblemas, cada um dos quais com tamanho $n/2$ representando o termo $2 \cdot T(n/2)$ da relação de recorrência em questão. Depois que esses dois subproblemas são resolvidos, retorna-se ao primeiro nível da árvore e executam-se mais n operações adicionais para concluir o termo não recursivo da relação de recorrência. Para calcular o custo de um determinado nível, somam-se os custos de todos os subproblemas desse nível. Quando todos os subproblemas de um nível têm o mesmo tamanho, isso equivale a multiplicar o número de subproblemas pelo custo de cada um deles.

Existem representações gráficas alternativas para árvores de recursão e a **Figura 6–7** mostra uma forma alternativa simplificada da árvore de recursão da **Figura 6–6**.

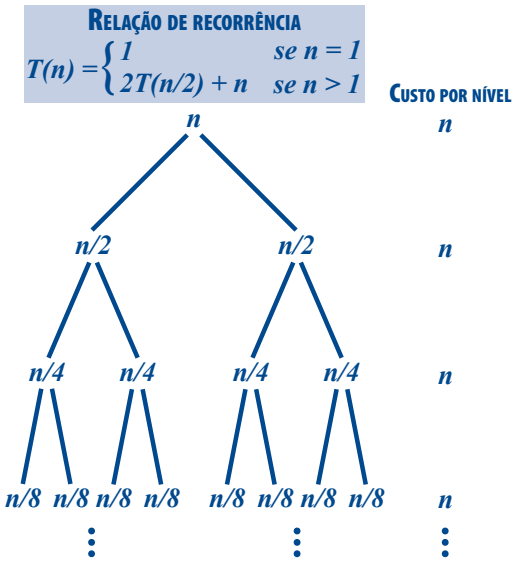


Figura 6–7: Árvore de Recursão Alternativa

É importante salientar que uma árvore de recursão não provê diretamente a solução para a relação de recorrência à qual está associada. Entretanto, ela pode muito útil para ajudar a encontrar uma conjectura que poderá então ser provada usando indução matemática (v. **Seção 6.9**).

6.10.2 Exemplos

Observação: Em todos os exemplos apresentados nesta seção, assume-se que o tamanho da entrada do problema (n) é uma potência de dois (i.e., $n = 2^k$, para algum inteiro $k > 1$).

Equação de Recorrência 1: T(n) = 2·T(n/2) + n

Cada vez que se acrescenta um novo nível a uma árvore de recursão, apresenta-se o resultado de mais uma iteração da respectiva relação de recorrência. Quando se conhece o custo de cada nível, podem-se somar os custos de todos os níveis para obter o custo total. Portanto é necessário determinar quantos níveis uma árvore de recursão possui.

Na **Figura 6–8**, que revela a árvore de recursão completa da relação de recorrência sob discussão, vê-se que, no nível i, há 2^i subproblemas de tamanho $n/2^i$. Como resolver um problema de tamanho 2^i tem custo 2^i, o custo por nível é $(2^i)\cdot(n/2^i) = n$. Para calcular o número de níveis da árvore, verifica-se que, em cada nível, o problema é reduzido à metade. Como a árvore começa com o problema inteiro e, a cada nível, o tamanho do problema é reduzido à metade, o número de vezes que o problema original é reduzido é $\log n$ e, portanto, a altura da árvore é $\log n + 1$, como mostra a **Figura 6–8**.

O cálculo do custo total no último nível de uma árvore de recursão é diferente daquele dos demais níveis. O custo do nível inferior de uma árvore de recursão é calculado tomando-se como referência o caso-base da relação de recorrência, enquanto, nos demais níveis, o custo de cada um deles é determinado pela respectiva equação recursiva. No nível inferior da árvore da **Figura 6–8**, o número de nós é $2^{\log n} = n$ e o custo de cada um deles é $T(1) = 1$. Portanto o custo do último nível dessa árvore é n.

O custo total de uma árvore de recursão é a soma dos custos dos níveis da árvore. No exemplo da **Figura 6–8**, o custo total é fácil de calcular pois, como todos os níveis têm o mesmo custo, basta multiplicar a altura da árvore pelo custo de cada nível. Assim o custo total dessa árvore é $n\cdot(\log n + 1)$. Portanto pode-se conjecturar que $T(n)$ é $\theta(n \log n)$ e usar o método de substituição junto com indução matemática (v. **Seção 6.9**) para provar essa conjectura, como se vê no **Exemplo 6.28**.

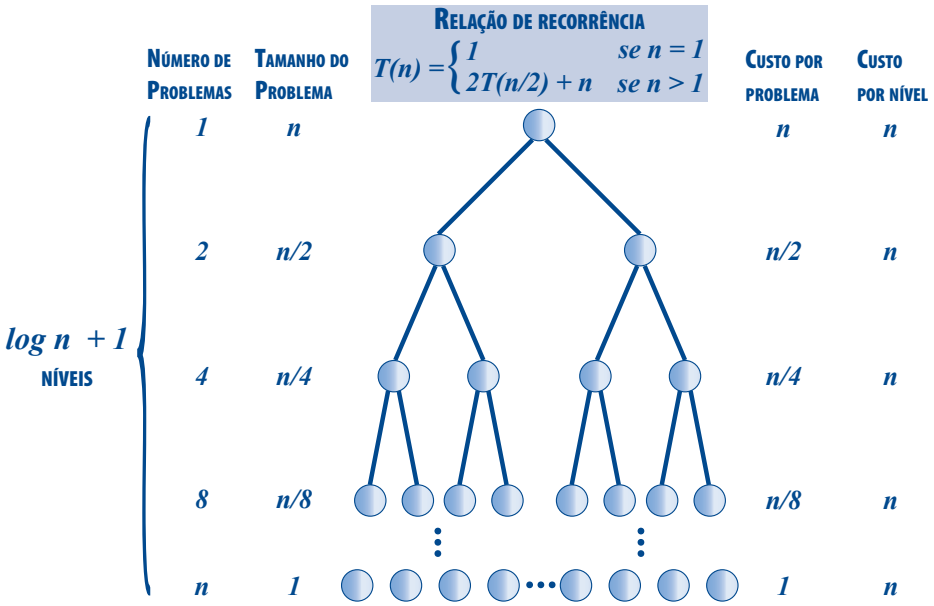


Figura 6–8: Árvore de Recursão da Equação de Recorrência 1

Exemplo 6.28 Mostre que a solução para a relação da relação de recorrência:

$$T(n) = \begin{cases} 1 & se\ n = 1 \\ 2T(n/2) + n & se\ n > 1 \end{cases}$$

é $O(n \log n)$.

Solução: Note que o enunciado do problema não afirma que a forma fechada da referida relação de recorrência é $T(n) = n \log n$. Ele apenas afirma que, se $f(n) = n \log n$, então $T(n)$ é $O(f(n))$ e para tal não é necessário encontrar uma forma fechada dessa relação de recorrência. Será utilizada indução forte (v. **Apêndice B**) para mostrar que $\exists\ c_0, n_0 \mid T(n) = c_0\, n \log n, \forall\ n \geq n_0$.

Base da indução. Como $T(n)$ é $\theta(1)$ para valores de n suficientemente pequenos, é sempre possível encontrar um valor de c_0 que satisfaça a base da indução.

Hipótese indutiva. Suponha que $T(n) \leq c_0\, k \log k, \forall\ k < n$. [Deve-se mostrar que $T(n) \leq c_0\, n \log n$.]

		Justificativa
Passo indutivo	$T(n) = 2 \cdot T(n/2) + n$	Por definição de $T(n)$
	$\leq 2 \cdot c_0\,(n/2) \log (n/2) + n$	Hipótese indutiva, pois $n/2 < n$
	$\leq c_0\, n \log n - c_0\, n + n$	Manipulação algébrica simples
	$\leq c_0\, n \log n$	Substituindo $-c_0\, n + n$ por 0, o lado direito é acrescido quando $-c_0\, n + n \leq 0$; ou seja, quando $c_0 \geq 1$

Equação de Recorrência 2: T(n) = T(n/2) + n

Considere a seguinte relação de recorrência:

$$T(n) = \begin{cases} 1 & se\ n = 1 \\ T(n/2) + n & se\ n > 1 \end{cases}$$

Apesar de essa relação de recorrência ser bem parecida com a relação de recorrência do exemplo anterior, sua árvore de recursão, apresentada na **Figura 6–9**, é bem diferente. Note nessa figura que, em cada nível, o tamanho de cada subproblema e o custo de cada um deles são os mesmos do exemplo anterior. Além disso, o número de

níveis nas árvores de recursão dos dois exemplos é o mesmo. No entanto os custos por nível são diferentes nos dois casos, de modo que, no exemplo corrente, não se pode mais multiplicar a altura da árvore pelo custo de cada nível. No caso presente, o custo associado à árvore de recursão da **Figura 6–9** é dado por:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1 = n\left(1 + \frac{1}{2} + \frac{1}{4} \ldots + \left(\frac{1}{2}\right)^{\log n}\right) = n \cdot \sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^{i}$$

Esse último somatório representa uma série geométrica cujo maior termo é *1*. É fácil provar por indução que o resultado desse somatório é $\theta(1)$, de sorte que conjectura-se que $T(n)$ é $\theta(n)$.

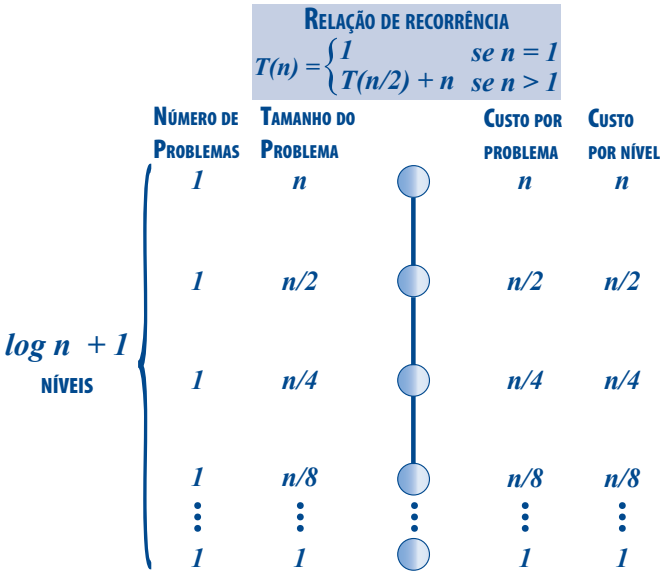


Figura 6–9: Árvore de Recursão da Equação de Recorrência 2

Exercício: Para completar o exemplo que se acabou de discutir, mostre que a solução da relação de recorrência em questão é de fato $\theta(n)$. **Sugestão**: use como referência o **Exemplo 6.28**.

Equação de Recorrência 3: 4·T(n) = T(n/2) + n

Como último exemplo, considere a relação de recorrência e árvore de recursão exibidas na **Figura 6–10**.

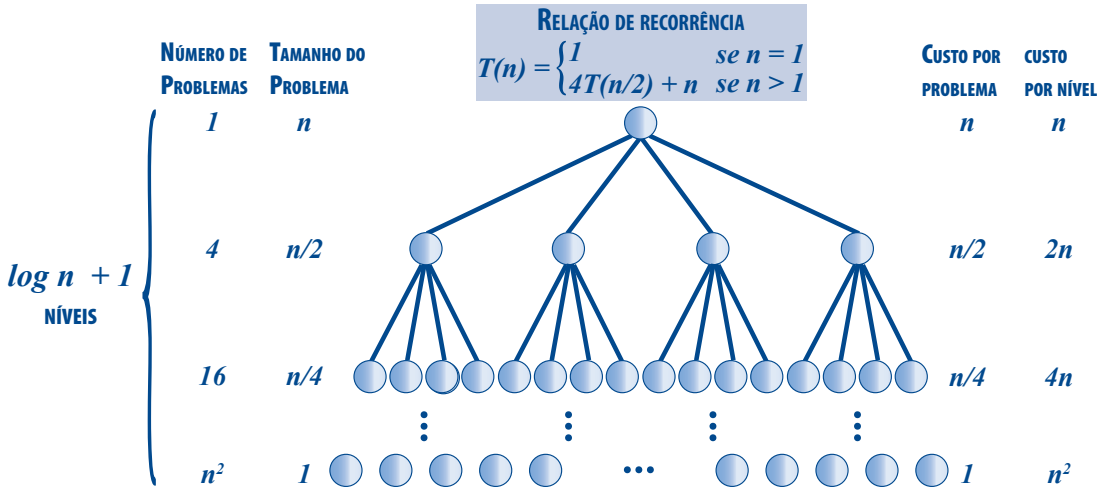


Figura 6–10: Árvore de Recursão da Equação de Recorrência 3

Na árvore de recursão da **Figura 6–10**, no nível i, cada nó corresponde a um problema de tamanho $n/2^i$ e tem custo $n/2^i$, de modo que o custo no nível i é $4i \cdot (n/2^i) = 2^i \cdot n$. A mesma fórmula também é aplicável ao nível mais baixo da árvore, pois, nesse nível, há $4^{\log n} = 2^{\log n} \cdot n$ nós e o custo de cada um dos quais é 1. Assim o custo total dessa árvore pode ser calculado por meio do seguinte somatório:

$$T(n) = n \cdot \sum_{i=0}^{\log n} 2^i = n \cdot \frac{1 - 2^{\log n + 1}}{1 - 2} = 2n^2 - n$$

Logo $T(n)$ é $\theta(n^2)$.

6.10.3 Análise

É interessante notar as semelhanças e diferenças entre os três exemplos apresentados na **Seção 6.10.2**. Em todos esses exemplos, as alturas das árvores de recursão são iguais a $\log n + 1$ e, além disso, o tamanho de cada subproblema de um nível é a metade do tamanho de cada subproblema do nível precedente. Contudo, nos três exemplos, a quantidade de processamento em cada nível é diferente, como mostra a **Tabela 6–6**.

Equação de Recorrência	Quantidade de Processamento	Custo Total
$T(n) = 2 \cdot T(n/2) + n$	É a mesma em cada nível	$\theta(n \log n)$
$T(n) = T(n/2) + n$	Decresce geometricamente à medida que se desce na árvore de recursão, de modo que o nível com o maior custo é o nível da raiz	$\theta(n)$
$T(n) = 4 \cdot T(n/2) + n$	O número de nós por nível cresce geometricamente mais rapidamente do que o tamanho do problema e o nível com o maior custo é o nível inferior	$\theta(n^2)$

Tabela 6–6: Diferenças Algorítmicas de Relações de Recorrência

É importante observar que no segundo e no terceiro exemplos, o custo total é determinado por uma série geométrica, que é limitada superior e inferiormente por uma constante multiplicada pelo custo do nível que possui o maior custo. No segundo exemplo, o nível da raiz é aquele que possui o maior custo, que é n, e, por isso, o custo total nesse caso é $\theta(n)$. Por outro lado, no terceiro exemplo, o nível mais baixo possui o maior custo, que é n^2; assim o custo total é $\theta(n^2)$.

Se você entendeu bem as semelhanças e diferenças entre os três exemplos apresentados aqui, você será capaz de entender a maioria das relações de recorrência que aparecem em análise de algoritmos. O **Teorema 6.8** generaliza os resultados obtidos nesta seção[6].

Teorema 6.8 Suponha que se tenha uma relação de recorrência da forma:

$$T(n) = \begin{cases} \text{não negativo} & \text{se } n = 1 \\ a \cdot T(n/2) + n & \text{se } n > 1 \end{cases}$$

Sendo a uma constante inteira não negativa, tem-se que:

(i) Se $a < 2$, $T(n)$ é $\theta(n)$.

(ii) Se $a = 2$, $T(n)$ é $\theta(n \log n)$.

(iii) Se $a > 2$, $T(n)$ é $\theta(n^{\log a})$.

É importante enfatizar que, nos exemplos apresentados na **Seção 6.10.2**, por simplicidade, assume-se que o tamanho do problema é uma potência de 2; ou seja, $n = 2^k$, sendo k um inteiro não negativo. Se n não for uma potência de 2, o problema pode ser considerado parte de um problema maior com tamanho 2^{k+1} elementos, sendo $2^k < n < 2^{k+1}$. Isto é, 2^{k+1} é a menor potência de 2 que é maior do que n. Informalmente, o resultado obtido nesses dois casos é mesmo, pois 2^{k+1} é menor do que $2n$, de modo que: $T(2 \cdot n) = \theta(2 \cdot n \log 2 \cdot n) = \theta(n \cdot \log n)$.

[6] Este teorema é uma formulação mais fraca do famoso **Teorema Mestre**, mas ele é suficiente para os propósitos deste livro.

6.11 Exemplos de Programação

6.11.1 Um Algoritmo com Custo Temporal θ(1)

Problema: Determine o custo temporal da seguinte função em C que retorna o valor do primeiro elemento de um array:

```
int PrimeiroElemento(int ar[], int n)
{
   return ar[0];
}
```

Solução: Qualquer operação de acesso a um elemento de array tem custo temporal $\theta(1)$, pois a operação não depende do tamanho do array. Portanto o custo temporal dessa função é $\theta(1)$.

6.11.2 Um Algoritmo com Custo Temporal θ(n)

Problema: Determine o custo temporal da seguinte função em C que exibe na tela o valor de cada elemento de um array de elementos do tipo **int**.

```
void ExibeArray(int ar[], int n)
{
   for (int i = 0; i < n; ++i)
      printf("%d\t", ar[i]);
}
```

Solução: Cada uma das três expressões que constituem o laço **for** tem custo temporal $\theta(1)$. A primeira expressão é avaliada uma única vez, a segunda expressão é avaliada $n + 1$ vezes e a terceira expressão é avaliada n vezes. O corpo do laço contém uma única instrução com custo temporal $\theta(1)$ que é executada n vezes. Portanto o custo temporal do algoritmo implementado pela função `ExibeArray()` é dada por: $\theta(1) + \theta(n + 1) + \theta(n) + \theta(n)$, que é $\theta(n)$.

6.11.3 Um Algoritmo com Custo Temporal θ(n²)

Problema: Mostre que o algoritmo seguido pela função `BubbleSort()` apresentada na **Seção 3.12.2** para ordenação de arrays pelo método da bolha, no pior caso, tem custo temporal $\theta(n^2)$. A referida função é reproduzida a seguir sem os comentários que a acompanham para facilitar sua análise.

```
                void BubbleSort(int ar[], int n)
                {
                   int i, aux,
                       ordenado = 0;

[1]                while (!ordenado) {                  θ(?)
[2]                   ordenado = 1;                     θ(1)
[3]                   for (i = 0; i < n - 1; i++)       θ(?)
[4]                      if (ar[i] > ar[i+1]){          θ(1)
[5]                         ordenado = 0;               θ(1)
[6]                         aux = ar[i];                θ(1)
[7]                         ar[i] = ar[i+1];            θ(1)
[8]                         ar[i+1] = aux;              θ(1)
                         }
[9]                   --n;                              θ(1)
                   }
                }
```

Solução: As linhas de interesse na função acima são aquelas numeradas de **[1]** a **[8]**.

- As instruções nas linhas **[2]** e de **[4]** a **[9]** são todas $\theta(1)$; ou seja, todas elas têm custos temporais constantes (i.e., independentes do tamanho da entrada).
- Pela regra da soma, cada uma das instruções de **[5]** a **[8]**, que compõem o corpo do **if**, tem custo temporal $\theta(1)$. A condição de teste da instrução **if** também tem custo temporal $\theta(1)$. Portanto, como um todo, a instrução **if** tem custo temporal $\theta(1)$.
- O corpo do laço **for** contém apenas a instrução **if**, cujo custo temporal é $\theta(1)$. No pior caso, o corpo desse laço é executado $n - 1$ vezes na primeira passagem, $n - 2$ vezes na segunda passagem e assim por diante, até que, na última passagem, ele é executado apenas uma vez[7]. Assim o número de vezes que o corpo do laço **for** é executado é dado por:

$$\underbrace{(n-1)}_{\text{1ª passagem}} + \underbrace{(n-2)}_{\text{2ª passagem}} + \ldots + \underbrace{2}_{\text{Penúltima passagem}} + \underbrace{1}_{\text{Última passagem}} = \frac{(n-1)\cdot n}{2}$$

- Como o corpo do laço **for** contém a instrução que é executada o maior número de vezes, esse valor determina o custo temporal da função **BubbleSort()**, que, no pior caso, é $\theta(n^2)$.

6.11.4 Um Algoritmo com Custo Temporal θ(log n)

Problema: A função `BuscaBinaria()` apresentada a seguir, efetua **busca binária** num array de inteiros considerado ordenado em ordem crescente. Mostre que o algoritmo seguido por essa função tem custo temporal $\theta(\log n)$.

```
int BuscaBinaria( const int ar[], int n, int valor )
{
   int inf, /* Limite inferior */
       sup, /* Limite superior */
       meio; /* Meio do intervalo */

      /* Limites inferior e superior iniciais */
[1]   inf = 0;                                          θ(1)
[2]   sup = n - 1;                                      θ(1)

      /* Efetua a busca binária */
[3]   while (inf <= sup) {                              θ(?)
         /* Calcula o meio do intervalo */
[4]      meio = inf + (sup - inf)/2;                    θ(1)

         /* Verifica se o valor encontra-se no meio */
[5]      if (ar[meio] == valor)                         θ(1)
[6]         return meio; /* Encontrado */               θ(1)

         /* Ajusta o intervalo de busca */
[7]      if (valor < ar[meio])                          θ(1)
[8]         sup = meio - 1;                             θ(1)
[9]      else                                           θ(1)
[10]        inf = meio + 1;                             θ(1)
   }

[11]  return -1; /* Elemento não foi encontrado */      θ(1)
}
```

[7] A expressão condicional do laço **for** é avaliada uma vez a mais em cada passagem, enquanto o incremento é avaliado o mesmo número de vezes do corpo do laço em cada passagem. A avaliação de qualquer dessas expressões tem custo temporal $\theta(1)$, de modo que as avaliações desses expressões não afetam a análise apresentada aqui.

Solução: As linhas de interesse para análise são numeradas de **[3]** a **[10]**, pois essas linhas envolvem um laço de repetição **while** que depende do tamanho do array, embora isso possa não ser aparente à primeira vista. As demais instruções da função têm $\theta(1)$ e não influenciarão na análise. Portanto a análise de custo temporal fica restrita à análise do laço **while**. Para simplificar essa análise, supõe-se que o tamanho do array no qual a busca será efetuada é múltiplo de 2[8].

Inicialmente, o intervalo de busca consiste em todo o array. Então, a cada iteração do laço **while**, esse intervalo é reduzido à metade até que seu tamanho seja reduzido a *1*. Portanto, supondo que o valor de n é uma potência de *2* (v. **Seção 6.10**), o número de vezes que o corpo do laço é executado é o número de vezes que o tamanho da entrada n pode ser reduzido a *1*, o que pode ser expresso matematicamente pela seguinte equação:

$$n/2^i = 1$$

em que i é o número de vezes que o laço é executado e n é o tamanho da entrada (i.e., o número de elementos do array).

Essa última equação pode ser resolvida assim:

$$n/2^i = 1 \Rightarrow 2^i = n \Rightarrow i = log_2\ n$$

A última igualdade é decorrência direta da definição de logaritmo (v. **Apêndice B**). Como a base de um logaritmo não é importante em análise assintótica, tem-se que a custo temporal do algoritmo implementado pela função é $\theta(log\ n)$. O raciocínio empregado na análise do algoritmo de busca binária é ilustrado na **Figura 6–11**.

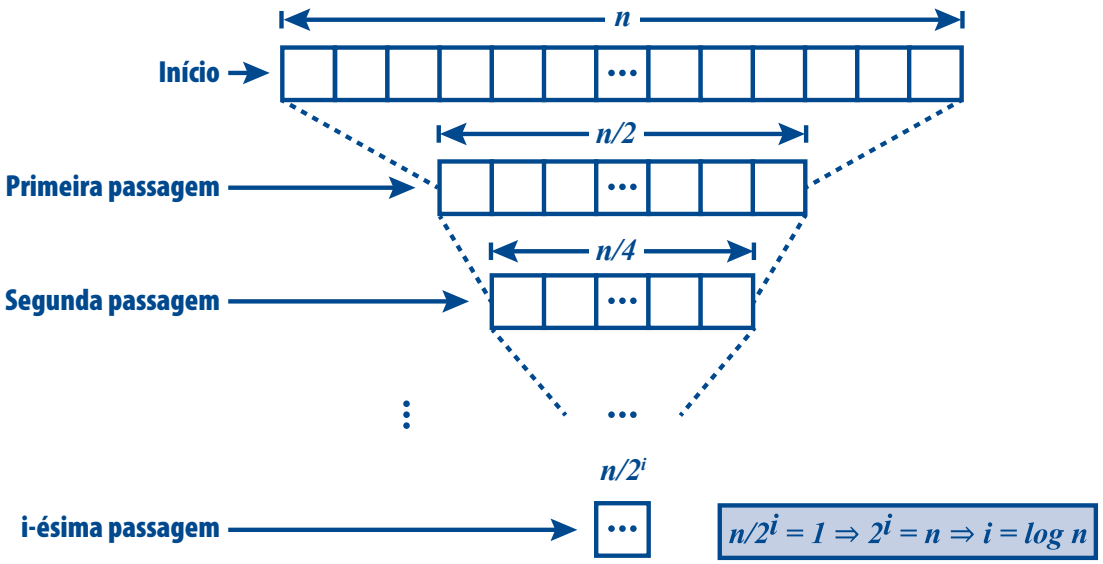


Figura 6–11: Custo Temporal de Busca Binária

6.11.5 Um Algoritmo com Custo Temporal θ(n log n)

Problema: A função `Rodadas()`, apresentada a seguir, não realiza nenhuma operação útil, de modo que seu papel aqui é meramente didático. Mostre que o algoritmo seguido por essa função tem custo temporal $\theta(n\ log\ n)$.

[8] No **Capítulo 3** do **Volume 2**, será apresentada uma análise bem precisa para o custo do algoritmo de busca binária.

```
                void Rodadas(int n)
                {
                   int i, j;

[1]                for (i = 1; i <= n; i++) {          θ(?)
[2]                   printf("\n\nRodada %d: ", i);   θ(1)

[3]                   j = n;                           θ(1)

[4]                   while (j > 1) {                  θ(?)
[5]                      j = j/2;                      θ(1)
[6]                      printf("%d\t", j);            θ(1)
                   }
                }
}
```

Solução: Como é recomendado, a análise deve começar pelo corpo da instrução **for**, que é a única instrução dessa função. Esse corpo possui três instruções, sendo que as duas primeiras apresentam custo temporal $\theta(1)$. A terceira instrução em questão é um laço **while** e, como obviamente não existe custo menor do que $\theta(1)$, de acordo o **Teorema 6.4**, o custo temporal do corpo da instrução **for** corresponderá ao custo temporal dessa instrução **while**. Utilizando o mesmo raciocínio empregado na avaliação da função `BuscaBinaria()` do exemplo da **Seção 6.11.4**, conclui-se que avaliação do laço **while** sob escrutínio é $\theta(\log n)$. Agora, sem considerar a avaliação do seu corpo, o laço **for** dessa função tem custo temporal $\theta(n)$. Portanto o custo temporal do laço **for** e seu corpo combinados é $\theta(n \log n)$.

6.11.6 Quando Será o Fim do Mundo?

Preâmbulo: O problema das torres de Hanói, discutido na **Seção 4.8.1**, foi popularizado pelo matemático francês Édouard Lucas em 1883. Recordando o que foi visto naquela seção, o objetivo do quebra-cabeças é deslocar um número n de discos de uma haste para outra com as seguintes restrições: **(1)** apenas um disco pode ser deslocado de cada vez; **(2)** apenas um disco que se encontre no topo em alguma das hastes pode ser movido e **(3)** um disco maior não pode repousar sobre um disco menor. A **Figura 6–12** mostra a condição inicial do quebra-cabeças quando n é igual a três discos.

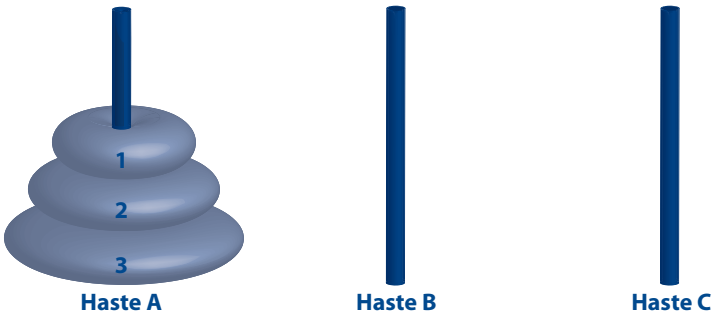


FIGURA 6–12: PROBLEMA DAS TORRES DE HANÓI REVISITADO

Segundo Édouard Lucas, esse quebra-cabeças seria uma versão simplificada da mítica Torre de Brahma (não é a cerveja, é uma divindade hindu), supostamente existente num templo da Índia. Segundo a lenda, quando criou o universo, Brahma colocou no referido templo três hastes verticais de diamante e, numa delas, 64 anéis de ouro de tamanhos diferentes, empilhados em ordem de tamanho do menor para o maior. Então, aos sacerdotes do templo caberia a tarefa de transferir essa pilha de discos para uma das duas outras hastes preservando a mesma ordem original. Para

isso, teriam de transferir um disco de cada vez e poderiam utilizar a outra haste como auxiliar, mas nunca poderiam um disco maior sobre um disco menor. Quando todos os 64 discos fossem transferidos, o templo seria destruído e o mundo se acabaria.

Problema: Supondo que os sacerdotes indianos sejam suficientemente eficientes para mover um disco a cada segundo e acreditando na lenda, quando será o fim do mundo?

Solução: Seja T_n o custo temporal necessário para resolver o problema para n discos. Então T_n deve ser descrito pela seguinte relação de recorrência:

$$T_n = \begin{cases} 0, \text{ se } n = 0 \\ 2T_{n-1} + 1, \text{ se } n > 0 \end{cases}$$

A primeira linha dessa relação de recorrência é decorrente do fato de não ser necessário fazer nada quando o número de discos é igual a zero. A segunda linha é obtida utilizando-se o seguinte raciocínio:

1. Para mover n discos ($n > 0$) da haste de origem (haste **A** na **Figura 6–12**) para a haste de destino (haste **C** na **Figura 6–12**) é necessário, em primeiro lugar, mover os $n - 1$ discos que se encontram sobre o disco maior para a haste auxiliar (haste **B** na **Figura 6–12**). O custo temporal de realização dessa tarefa é, por definição, T_{n-1}.
2. O próximo passo consiste em mover o disco maior (disco **3** na **Figura 6–12**) para a haste de destino. Novamente, por definição, esse passo tem custo temporal $T_1 = 1$.
3. Finalmente, devem-se mover os $n - 1$ discos que foram movidos no **Passo 1** para a haste de destino. Por definição, esse passo tem custo temporal T_{n-1}.

Portanto o custo temporal total para mover n discos ($n > 1$) da haste de origem para a haste de destino é:

$T_{n-1} + 1 + T_{n-1} = 2 \cdot T_{n-1} + 1$

A resolução de qualquer relação de recorrência requer experiência e algum insight (consulte o **Apêndice B**). Nesse caso específico, a relação de recorrência que descreve o custo temporal para o problema das torres de Hanói pode ser resolvida seguindo os passos apresentados a seguir.

Em primeiro lugar, note que a relação de recorrência pode ser reescrita como:

$T_0 = 0$

$T_n = 2T_{n-1} + 1, \text{ se } n > 0$

Agora, somando-se 1 a ambos os lados das duas equações, obtém-se:

$T_0 + 1 = 1$

$T_n + 1 = 2T_{n-1} + 2 = 2(T_{n-1} + 1)$

Considere $A_n = T_n + 1$. Então, essas últimas equações podem ser reescritas em termos de A_n como:

$A_0 = 1$

$A_n = 2A_{n-1}$

Neste ponto, pode-se usar uma técnica bastante comum em resolução de relações de recorrência dessa natureza que é popularmente conhecida como *adivinhar e testar* (ou, menos pejorativamente, *conjecturar e provar*). Isto é, atribuindo-se valores para n na última relação de recorrência, pode-se obter algum insight sobre qual o resultado será:

$A_1 = 2A_0$

$A_2 = 2A_1 = 2^2A_0$

$A_3 = 2A_2 = 2^3A_0$

$A_4 = 2A_3 = 2^4A_0$

Observando-se o padrão seguido pela relação de recorrência parece uma boa conjectura supor que $A_n = 2^n A_0$ é a solução para a relação de recorrência. Resta tentar provar que essa conjectura é verdadeira utilizando indução matemática (v. **Apêndice B**).

Claramente, a conjectura é válida para $n = 0$, pois:

$A_0 = 2_0 A_0 = A_0$

Agora, supondo que a conjectura é válida para $n = k$, resta mostrar que ela também é válida para $n = k + 1$; i.e., que $A_{k+1} = 2^{k+1} A_0$ o que é efetuado a seguir.

$A_{k+1} = 2A^{k-1} + 1 = 2A^k$ (de acordo com a segunda equação da relação de recorrência)

Ora, mas de acordo com a hipótese indutiva, tem-se que $A^k = 2^k A_0$. Logo:

$A_{k+1} = 2 \cdot A^k = 2 \cdot 2^k A_0 = 2^{k+1} \cdot A_0$

Assim a conjectura está provada (e deixa de ser *conjectura*). ■

Finalmente, pode-se responder à questão do fim do mundo: com toda eficiência dos sacerdotes indianos, eles levarão 2^{64} segundos para cumprir a tarefa. Esse intervalo de tempo corresponde a cerca de 585 bilhões de anos (a Terra existe há *apenas* cerca de 4,6 bilhões de anos!). Portanto, mesmo que acredite na lenda, pode dormir sossegado pois a tarefa ainda levará um *tempinho* para ser concluída.

Pode-se ainda afirmar que o custo temporal do algoritmo implementado na **Seção 4.8.1** é $\theta(2^n)$, visto que esse custo é proporcional ao número de movimentos efetuados.

6.11.7 Fibonacci + Recursão = Ineficiência 2

Problema: Na **Seção 4.8.2** mostrou-se informalmente o quão ineficiente era a função `Fib()` que calcula números de Fibonacci recursivamente. (a) Mostre que o custo temporal dessa função é $\Omega(1.5^n)$. (b) Mostre que o custo espacial dessa função é $\theta(n)$.

Solução: (a) A função `Fib()` é reproduzida a seguir para facilidade de referência.

```
int Fib(int n)
{
   if (n < 2)
      return n; /* Dois primeiros termos */

   return Fib(n - 1) + Fib(n - 2); /* Caso recursivo */
}
```

A relação de recorrência associada à essa função pode ser escrita como:

$$T_n = \begin{cases} c_1 & \text{se } n < 2 \\ T_{n-1} + T_{n-2} + c_2 & \text{se } n \geq 2 \end{cases}$$

Nessa relação de recorrência, c_1 e c_2 são constantes desconhecidas. A constante c_1 corresponde ao tempo gasto pela função `Fib()` quando `n` é menor do que `2`. Esse valor é constante porque, nesse caso, tudo que a função faz é avaliar a expressão `n < 2` e retornar o valor de `n`. A constante c_2 corresponde ao tempo gasto pela função `Fib()` para somar os resultados retornados pelas duas chamadas recursivas efetuadas quando `n ≥ 2`:

Agora, essa relação de recorrência não pode ser resolvida de acordo com o que foi apresentado acima. Mais precisamente, essa é, reconhecidamente, uma relação de recorrência difícil de resolver porque não é possível encontrar nenhum padrão que possa ser usado como conjectura a ser provada por meio de indução. Ou seja, essa relação de recorrência deve ser reduzida a uma equação característica (v. **Apêndice B**), que, muitas vezes, não é difícil de resolver, desde que ela seja homogênea. Infelizmente, esse não é o caso devido à presença da constante c_2.

A boa notícia é que o que se requer aqui não é a solução dessa relação de recorrência, mas apenas uma estimativa em termos de notação ó.

Os números de Fibonacci também podem também ser representados por uma relação de recorrência:

$$F_n = \begin{cases} 1 & se\ n < 2 \\ F_{n-1} + F_{n-2} & se\ n \geq 2 \end{cases}$$

Essa última relação de recorrência, apesar de parecida com aquela anterior, é bem mais fácil de resolver porque ela resulta numa equação característica homogênea (v. **Apêndice B**). A solução dessa última relação de recorrência (novamente, v. **Apêndice B**) é dada por:

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Ora, examinando-se as duas relações de recorrência apresentadas acima, conclui-se por indução matemática (v. **Apêndice B**) que $T_n \geq F_n$. Pode-se ainda provar por indução matemática que, para $n > 4$, $F_n \geq (3/2)^n$. Portanto pode-se concluir que $T_n \geq (3/2)^n = 1.5^n$, para $n > 4$. Logo T_n é $\Omega(1.5^n)$, o que significa que o custo temporal da função `Fib()` cresce exponencialmente.

(b) O custo espacial de uma função é proporcional ao número de registros de ativação alocados ao mesmo tempo na pilha de execução.

Agora, examinando-se o diagrama de recursão apresentado na **Seção 4.8.2** para a chamada `Fib(6)`, nota-se que há 24 chamadas adicionais decorrentes da chamada inicial da função `Fib()`. Também, de acordo como o que foi exposto na **Seção 4.3**, cada uma dessas chamadas dá origem a um registro de ativação. Portanto a chamada `Fib(6)` causará a criação de 25 registros de ativação. Contudo, não é o caso que todos eles estarão alocados na pilha de ativação ao mesmo tempo. Por exemplo, considere novamente `Fib(6)`. Quando essa chamada é efetuada, a função retorna `Fib(5) + Fib(4)`. Ocorre, porém, que a chamada `Fib(4)` só será efetivada quando a chamada `Fib(5)` retornar. Mas, quando `Fib(5)` é chamada, a função retorna `Fib(4) + Fib(3)` e essa última chamada só é realizada após o retorno de `Fib(4)` e assim por diante. Desse modo, o maior número de registros de ativação simultaneamente alocados na pilha de execução na chamada de `Fib(6)` é decorrente das chamadas que se encontram no interior do espaço azulado da **Figura 6–13**.

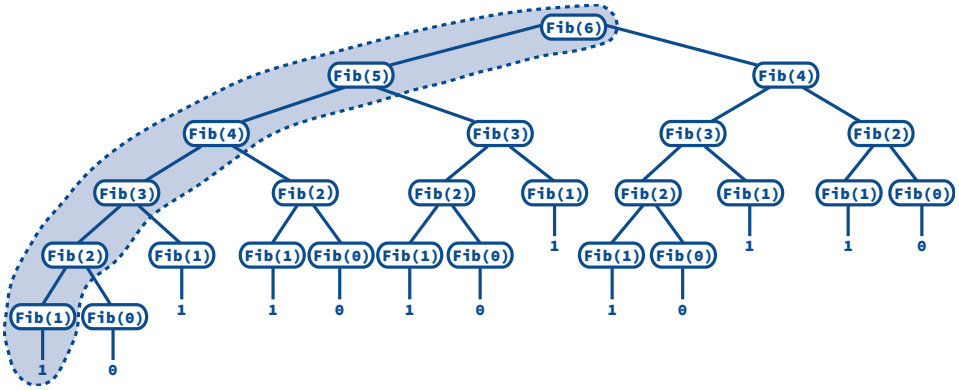


Figura 6–13: Registros de Ativação Alocados Simultaneamente na Chamada Fib(6)

De acordo com o que foi exposto, pode-se concluir, informalmente, que o número de registros de ativação alocados ao mesmo tempo para a função `Fib()` é, no pior caso, igual ao valor do parâmetro `n` dessa função. Portanto o custo espacial dessa função é $\theta(n)$.

6.11.8 Exponenciação por Quadratura 2

Problema: Mostre que a função `ExponenciacaoQuad()` vista na **Seção 4.8.5** possui custo temporal igual a $\theta(\log n)$, em que n é o expoente.

Solução: A função `ExponenciacaoQuad()` é reapresentada a seguir para facilidade de referência:

```
double ExponenciacaoQuad(double x, int n)
{
   if (n < 0)
      return ExponenciacaoQuad(1/x, -n);

   if (n == 0)
      return 1;

   if (n == 1)
      return x;

   if (n%2 == 0)
      return ExponenciacaoQuad(x*x, n/2);

   return x * ExponenciacaoQuad(x*x, (n-1)/2);
}
```

A função `ExponenciacaoQuad()` é chamada sucessivamente com os expoentes $n/2$, $n/2^2$, $n/2^4$, ..., $n/2^k$, até que esse valor seja 1. Assim existem k chamadas recursivas, de modo que: $n/2^k = 1 \Rightarrow 2^k = n \Rightarrow k = log_2 n$. Portanto o custo dessa função é $\theta(\log n)$.

6.12 Exercícios de Revisão

Observação: Quando for necessário, assuma que a base logarítmica utilizada é 2. Em outras palavras, suponha, quando necessário, que $\log n$ é o mesmo que $log_2 n$.

Complexidade de Algoritmos (Seção 6.1)

1. Que fatores influenciam a eficiência de um programa?
2. Como é medida a eficiência (ou desempenho) de um programa?
3. Dois algoritmos funcionalmente equivalentes apresentam necessariamente a mesma complexidade?
4. Apresente dois algoritmos funcionalmente equivalentes que apresentam o mesmo custo temporal.
5. O que é complexidade de algoritmo (ou complexidade computacional)?
6. O que é custo de processamento de um algoritmo?
7. (a) O que é custo temporal? (b) O que é custo espacial?
8. Em que medida se baseia a análise de custo temporal de um algoritmo?
9. Por que se dá mais atenção ao custo temporal do que ao custo espacial de um algoritmo?
10. No trecho de programa em C abaixo, suponha que os valores das expressões `a > b` e `b > c` sejam independentes e que, em média, `a` seja maior do que `b` *75%* das vezes, e `b` seja maior do que `c` *25%* das vezes. Quantas vezes se espera que as funções `F()` e `G()` sejam executadas?

```
for (i = 1; i <= 10000; ++i)
   if (a > b)
      v[i] = F(i);
   else if (b > c)
      v[i] = G(i);
```

Análise Assintótica (Seção 6.2)

11. O que é análise assintótica?

12. Qual é a importância da análise assintótica na avaliação de custos de algoritmos?

13. O número de operações executadas por um algoritmo A_1 é $10n \log n$ ao passo que o número de operações executadas por um algoritmo A_2 é $2n^2$. Encontre um valor n_0 tal que A_1 tem melhor desempenho do que A_2 para $n > n_0$.

14. Suponha que você tenha disponíveis dois algoritmos funcionalmente equivalentes que resolvem um determinado problema cujo tamanho é n. O primeiro algoritmo usa $n \log n$ operações para resolver o problema, enquanto o segundo algoritmo usa $n^{3/2}$ operações. À medida que n cresce, qual é o algoritmo que usa o menor número de operações?

Notações Ó, Ômega e Teta (Seção 6.3)

15. Defina e explique as notações:

(a) Ó

(b) Ômega

(c) Teta

16. Compare as notações ó, ômega e teta.

17. Por que se diz que a notação ômega é mais precisa do que a notação ó?

18. Suponha que A e B sejam algoritmos funcionalmente equivalentes, A tenha custo temporal $O(n)$ e B tenha custo temporal $O(n^2)$. É correto afirmar que A é *melhor* do que B?

19. Suponha que A e B são algoritmos funcionalmente equivalentes, A é $\theta(n)$ e B é $\theta(n^2)$. É correto afirmar que A é assintoticamente *melhor* do que B?

20. Interprete as seguintes afirmações:

(a) $f(n)$ é $O(n)$

(b) $f(n)$ é $\Omega(n)$

(a) $f(n)$ é $\theta(n)$

21. Encontre duas funções $f(n)$ e $g(n)$ tais que nem $f(n)$ é $O(g(n))$ nem $g(n)$ é $O(f(n))$.

22. A seguinte função encontra o menor valor de um array. Qual é seu custo temporal em termos de notação teta?

```
int MenorEmAray(int ar[], int tam)
{
   int menor = ar[0];

   for (int i = 1; i < tam; ++i)
      if (ar[i] < menor)
         menor = ar[i];

   return menor;
}
```

23. Mostre que se $T(n)$ é $O(f(n))$, então $cT(n)$ também é $O(f(n))$, para qualquer constante $c > 0$.

24. (a) Mostre que se $T_1(n)$ é $O(f(n))$ e $T_2(n)$ é $O(g(n))$, então $T_1(n) + T_2(n)$ é $O(f(n) + g(n))$. (b) Esse resultado contradiz o **Teorema 6.4** (regra da soma)?

25. Mostre que se $T_1(n)$ é $O(f(n))$ e $T_2(n)$ é $O(g(n))$, então $T_1(n) - T_2(n)$ *não é* necessariamente $O(f(n) - g(n))$.

26. Mostre que n é $O(n \log n)$.

27. Mostre que $(n^2 + 1)/(n + 1)$ é $O(n)$.

28. Mostre que n^2 é $O(n^3)$, mas n^3 não é $O(n^2)$.

29. Mostre que, se $T(n)$ é $O(n)$, então $T(n)$ é $O(n^2)$.

30. A seguinte função encontra o maior e o menor valores de um array. Usando a notação teta, qual é seu custo temporal?

```
void MenorMaiorEmAray(int ar[], int tam, int *menor, int *maior)
{
   *menor = ar[0];
   *maior = ar[0];

   for (int i = 1; i < tam; ++i)
      if (ar[i] < *menor)
         *menor = ar[i];

   for (int i = 1; i < tam; ++i)
      if (ar[i] > *maior)
         *maior = ar[i];
}
```

31. Mostre que o somatório a seguir é $\theta(n^2)$.

$$\sum_{i=1}^{n} i$$

32. Suponha que dois algoritmos funcionalmente equivalentes possuem tempos de execução dados por: $T_1(n) = 250n$ e $T_2(n) = n^2$. (a) Em termos de notação teta, qual é o custo temporal de cada um desses algoritmos? (b) Qual é o melhor custo temporal em termos de análise assintótica? (c) Para que valores de n é melhor usar o algoritmo com o pior custo temporal?

33. Mostre que $cf(n)$ é $\theta(f(n))$.

34. Mostre que c é $\theta(1)$, para qualquer constante c.

35. Mostre que (a) 2^n é $O(n!)$, mas (b) $n!$ não é $O(2^n)$.

36. Mostre que 2^{n+c} é $\theta(2^n)$, para qualquer constante c.

37. Mostre que $\log(n!)$ é $O(n \log n)$.

38. Mostre que $f(n)$ é $O(g(n))$ se e somente se $g(n)$ é $\Omega(f(n))$.

39. Mostre que $n \log n$ é $\Omega(n)$.

40. Mostre que n^2 é $\Omega(n \log n)$.

41. Mostre que $f(n)$ é $O(f(n))$.

42. Mostre que $2n^6 + n^5 - n^2$ é $\theta(n^6)$.

43. Mostre que n^5 não é $\theta(n^4)$.

44. Mostre que $2 + 4 + 6 + \ldots + 2n$ é $\theta(n^2)$.

45. Mostre que $1^2 + 2^2 + \ldots + n^2$ é $\theta(n^3)$.

Casos Melhor, Pior e Mediano (Seção 6.4)

46. (a) O que significa melhor caso de um algoritmo? (b) E pior caso? (c) E caso mediano?

47. Por que o principal interesse em análise de algoritmos se concentra no pior caso?

48. O pior caso de um algoritmo é necessariamente pior do que seu melhor caso?

49. É possível que os casos pior, melhor e mediano de um algoritmo sejam os mesmos?

50. Qual é o custo temporal de (a) pior caso, (b) melhor caso e (c) caso médio de um algoritmo que determina o menor valor armazenado num array de elementos do tipo **int**.

Funções Comuns em Análise de Algoritmos (Seção 6.5)

51. O que significa afirmar que $T(n)$ é $O(1)$?

52. Como são denominadas as seguintes classes de funções em análise de algoritmos?

(a) $\theta(1)$

(b) $\theta(\log n)$
(c) $\theta(n)$
(d) $\theta(n \log n)$
(e) $\theta(n^2)$
(f) $\theta(n^c)$, $c > 2$
(g) $\theta(c^n)$, $c > 1$

53. Apresente exemplos de algoritmos que apresentam as seguintes taxas de crescimento?
(a) $\theta(1)$
(b) $\theta(\log n)$
(c) $\theta(n)$
(d) $\theta(n \log n)$
(e) $\theta(n^2)$
(f) $\theta(2^n)$

54. (a) Ordene as seguintes funções por ordem crescente de crescimento: n, $n^{1.5}$, n^2, $n \log n$, $n \log^2 n$, $n \log n^2$, 2^n, n^3 e $\sqrt{n}$. (b) Quais dessas funções possuem a mesma taxa de crescimento?

55. Como os maiores valores da **Tabela 6–3** (i.e., $10^{30.084}$ e $10^{3.010.277}$) podem ser obtidos?

56. Um algoritmo consome $0{,}5\,\mu s$ quando o tamanho da entrada processada é 100. Quanto tempo esse algoritmo levará para processar uma entrada de tamanho 500 quando o custo temporal do algoritmo é:
(a) Linear
(b) Quadrático
(c) Cúbico

57. Qual é o maior valor de n para o qual é possível resolver em um segundo um problema que requer $f(n)$ operações, sendo que cada operação é executada em 10^{-9} segundos, quando a função $f(n)$ é dada por:
(a) n
(b) $\log n$
(c) $n \log n$
(d) n^2
(e) 2^n

58. Como muda o custo temporal de um algoritmo quando o tamanho de sua entrada passa de n para $2 \cdot n$, quando seu custo temporal é:
(a) $\theta(n)$
(b) $\theta(\log n)$
(c) $\theta(n^2)$

Resultados Teóricos Importantes (Seção 6.6)

59. Por que constantes multiplicativas e aditivas podem ser ignoradas em análise de algoritmos?

60. (a) Qual é a importância prática do **Teorema 6.4** (**Regra da Soma**)? (b) Qual é a importância prática do **Teorema 6.5** (**Regra do Produto**)?

61. (a) Por que é estranho em análise de algoritmos afirmar que um algoritmo tem custo $\theta(2n^3 + n - 1)$. (b) Qual seria o modo mais comum de apresentar o custo temporal desse algoritmo?

62. Por que se diz que $\theta(c)$ é o mesmo que $\theta(1)$ para qualquer constante c?

63. Considere a função $T(n) = 5n^2 + 2n + 3$. Construa uma tabela semelhante à **Tabela 6–4** que mostre que os termos $2n$ e 3 da função $T(n)$ tornam-se cada vez mais irrelevantes à medida que n cresce.

Regras Práticas para Análise Temporal de Algoritmos (Seção 6.7)

64. Que instruções de um algoritmo possuem custo temporal $\theta(1)$?

65. Como se avalia um desvio condicional (**if-else**)?

66. (a) Como se avalia um laço de repetição sem aninho? (b) Como se avalia um laço de repetição aninhado?

67. Como se avalia uma chamada de função num programa escrito em C?

68. Suponha que `i`, `j`, `k`, `n` e `soma` sejam variáveis do tipo **int**. Para cada um dos seguintes trechos programas a seguir apresente seu custo temporal usando a notação teta.

(i)
```
soma = 0;

for(i = 0; i < n; ++i)
   ++soma;
```

(ii)
```
soma = 0;

for(i = 0; i < n; ++i)
   for(j = 0; j < n; ++j)
      ++soma;
```

(iii)
```
soma = 0;

for(i = 0; i < n; ++i)
   for(j = 0; j < n*n; ++j)
      ++soma;
```

(iv)
```
soma = 0;

for(i = 0; i < n; ++i)
   for(j = 0; j < i; ++j)
      ++soma;
```

(v)
```
soma = 0;

for(i = 0; i < n; ++i)
   for(j = 0; j < i*i; ++j)
      for(k = 0; k < j; ++k)
         ++soma;
```

(vi)
```
soma = 0;

for(i = 1; i < n; ++i)
   for(j = 1; j < i*i; ++j)
      if(j % i == 0)
         for(k = 0; k < j; ++k)
            ++soma;
```

(vii)
```
for ( i = n; i <= 1; )
   i = i/2;
```

(viii)
```
for(i = 1; i <= n; i *= 2)
   ;
```

(ix)
```
soma = 0;

for(i = 1; i*i <= n; i++)
   soma++;
```

(x)
```
soma = 0;

for(i = 1; i <= n; i++)
   for(j = 1; j <= n; j *= 2)
      soma++;
```

69. Utilizando a notação teta, apresente uma estimativa para o número de operações de soma e multiplicação efetuadas no trecho de programa a seguir:

```
x = 0;

for (int i = 1; i <= 5; ++i)
   for (int j = 1; i <= 3; ++j)
      x = x + i*j;
```

70. Apresente uma estimativa em termos de notação teta para o número de operações de soma efetuadas no seguinte fragmento de programa:

```
x = 0;

for (int i = 0; i < n; ++i)
   for (int j = 0; i < n; ++j)
      x = x + i + j;
```

71. A função a seguir representa uma implementação comum da função **strcat()** da biblioteca padrão de C. Mostre que o custo temporal dessa função é $\theta(max(n, m))$, onde n e m são os tamanhos dos strings representados pelos parâmetros `s1` e `s2`.

```
char *ConcatenaStrings(char *s1, const char *s2)
{
   char *inicio = s1;

   while (*s1++)
      ;

   s1--;

   while (*s1++ = *s2++)
      ;

   return inicio;
}
```

72. O seguinte trecho de programa soma duas matrizes $n \times n$ representadas por dois arrays bidimensionais. Determine, em notação teta, o custo temporal desse trecho de programa.

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      c[i][j] = a[i][j] + b[i][j];
```

73. O seguinte trecho de programa foi escrito com o objetivo de multiplicar duas matrizes $n \times n$ representadas por dois arrays bidimensionais. Determine, em notação teta, o custo temporal desse trecho de programa.

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      for (k = a[i][j] = 0; k < n; k++)
         c[i][j] += a[i][k] * b[k][j];
```

74. O seguinte trecho de programa calcula a transposta de uma matriz $n \times n$ representada por um array bidimensional. Determine o custo temporal desse trecho de programa.

```
for (i = 0; i < n - 1; i++)
   for (j = i + 1; j < n; j++) {
      aux = a[i][j];
      a[i][j] = a[j][i];
      a[j][i] = aux;
   }
```

75. O método de ordenação denominado ordenação por seleção é implementado pela função `OrdenaPorSelecao()` apresentada a seguir. Qual é o custo temporal dessa função em termos de notação teta (a) no melhor caso e (b) no pior caso?

```
void OrdenaPorSelecao(int ar[], int n)
{
   int i, j, m, aux;

   for (i= 0; i < n; i++) {
      m = i;

      for (j = i + 1; j <= n - 1; j++)
         if (ar[j] < ar[m])
            m = j;

      if (ar[i] != ar[m]) {
         aux = ar[i];
         ar[i] = ar[m];
         ar[m] = aux;
      }
   }
}
```

76. Uma função `G()` que calcula a soma dos valores retornados por uma outra função `F()` é implementada como:

```
int G(const int *ar, int n)
{
   int soma = 0;

   for (int i = 0; i < n; ++i)
      soma += F(ar[i]);

   return soma;
}
```

Se a função `F()` tem custo temporal $\theta(n)$, qual é o custo temporal da função `G()`?

77. Qual é o custo temporal da função `F()` definida a seguir?

```
int F(int n)
{
   int i = 1;

   while (i < n) {
      int j = n;

      while(j > 0)
         j = j/2;

      i = 2*i;
   }

   return i;
}
```

78. Qual é o custo temporal do seguinte trecho de programa?

```
int i, j, x;

for (i = 1; i <= n; 1++)
   for (j = 1; j <= 2; j++)
      x = x + 1;
```

79. Um dado algoritmo resolve um determinado problema com tempo de execução $T_1(n) = n^3$. Um segundo algoritmo resolve o mesmo problema com tempo de execução $T_2(n) = 2n + 500$. (a) Qual é o custo temporal de cada algoritmo em termos de notação teta? (b) De acordo com a análise assintótica de algoritmos, qual é o algoritmo mais eficiente? (c) Sob que condições o algoritmo considerado menos eficiente de acordo com a notação teta é, de fato, o algoritmo mais eficiente.

Análise de Custo Espacial de Algoritmo (Seção 6.8)

80. Um algoritmo utiliza 100 variáveis locais para resolver um determinado problema. Qual é seu custo espacial?

81. Por que a análise de custo espacial de um algoritmo recursivo sempre se faz necessária?

82. Em que se baseia a análise de custo espacial de um algoritmo recursivo?

83. Utilize um raciocínio similar àquele empregado no **Exemplo 6.26** para mostrar que a função `ExibeArquivoNaTelaRec()`, discutida na **Seção 4.8.7** e reapresentada a seguir, tem custo espacial $\theta(n)$, em que n é o número de bytes lidos no arquivo.

```
void ExibeArquivoNaTelaRec(FILE *stream)
{
   if ( !feof(stream) && !ferror(stream) ) {
      putchar(fgetc(stream));
      ExibeArquivoNaTelaRec(stream);
   }
}
```

84. Mostre que o custo espacial da função recursiva `Fatorial()` definida na **Seção 6.9** é $\theta(n)$.

Algoritmos Recursivos e Relações de Recorrência (Seção 6.9)

85. O que é uma relação de recorrência?

86. Qual é o vínculo existente entre algoritmos recursivos e relação de recorrência?

87. Uma função para cálculo de exponencial é definida como:

```
double MinhaPow(double base, int exp)
{
   if (!exp)
      return 1.0;
   else if (!(exp%2)) /* exp é par */
      return MinhaPow(base*base, exp/2);
   else
      return MinhaPow(base*base, exp/2)*base;
}
```

Quantas operações de multiplicação serão efetuadas quando a função acima for chamada como `MinhaPow(5.0, 12)`?

88. Considere a função `F()` definida como:

```
int F(int x, int n)
{
   if (n == 1)
      return x;
   else
      return x + F(x, n/2);
}
```

Supondo que essa função seja sempre chamada com valores positivos de `n`, qual é o custo temporal dela?

Árvores de Recursão (Seção 6.10)

89. (a) O que é uma árvore de recursão? (b) Que informações uma árvore de recursão contém?

90. O que é interpretação algorítmica de uma relação de recorrência?

91. (a) Apresente graficamente a árvore recursão para a relação de recorrência a seguir e calcule uma estimativa em termos de notação teta para ela. Assuma que n é uma potência de 3.

$$T(n) = \begin{cases} 1 & \text{se } n < 3 \\ 3T(n/3) + n & \text{se } n \geq 3 \end{cases}$$

92. (a) Por que, nos exemplos da **Seção 6.10**, assume-se que n é uma potência de 2? (b) Por que, na questão **91**, assume-se que n é uma potência de 3?

93. Desenhe árvores de recursão para as seguintes relações de recorrência, supondo que $T(1) = 1$:

(a) $T(n) = 3T(n/2) + n$. Suponha que $n = 2^k$, para $k > 1$.

(b) $T(n) = T(n/4) + 1$. Suponha que $n = 4^k$, para $k > 1$.

94. Apresente os custos temporais para as relações de recorrência da questão **93.**

Exemplos de Programação (Seção 6.11)

95. Em que consiste a abordagem denominada *conjecturar e provar* para resolução de relações de recorrência?

96. Que argumentação é usada para mostrar que o custo espacial da função `Fib()` é $\theta(n)$?

6.13 Exercícios de Programação

EP6.1 (a) Implemente uma função em C que calcula valores da seguinte relação de recorrência:

$$T_n = \begin{cases} 1 & \text{se } n = 1 \\ T_{n-1} + 2n - 1 & \text{se } n > 1 \end{cases}$$

(b) Implemente um programa em C que chame a função solicitada no item (a) várias vezes e apresente o resultado. (c) Examine atentamente os resultados desse programa e utilize-os para criar uma conjectura sobre a solução para a referida relação de recorrência. (d) Utilize indução matemática para provar que a conjectura criada no item (c) realmente é a solução para a referida relação de recorrência. (e) O que ocorre quando a função implementada em (a) recebe um parâmetro menor do que 1?

EP6.2 Crie uma função em C que implemente um algoritmo de força bruta que, dado um conjunto de pontos no plano cartesiano, determina o par de pontos mais próximos entre si. Cada ponto é do tipo `tPonto`, definido como:

```
typedef struct {
           double x, y;
        } tPonto;
```

O protótipo da função solicitada deve ser:

```
void PontosMaisProximos( tPonto *p1, tPonto *p2,
                         const tPonto pontos[], int n )
```

Nesse protótipo, `p1` e `p2` são os pontos desejados, `pontos[]` é um array que armazena o referido conjunto de pontos e `n` é o tamanho desse array.

[**Sugestão:** De acordo com a Geometria Analítica, a distância entre dois pontos (x_i, y_i) e (x_j, y_j) no plano cartesiano é dada por:

$$d_{i,j} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$