Introdução à Análise de Complexidade de Algoritmos

Disciplina: Estruturas de Dados

Curso Superior de Tecnologia em
Análise e Desenvolvimento de Sistemas

Prof. Dr. Paulo César Rodacki Gomes

paulo.rodacki@blumenau.ifc.edu.br

IFC - Instituto Federal Catarinense

Campus Blumenau

Rua Bernardino José Oliveira, 81 - Badenfurt
CEP: 89070-270 - Blumenau/SC

Agosto 2014

Sumário

Lista de Figuras

Lista de Algoritmos

Prefácio

O presente trabalho apresente uma introdução aos conceitos fundamentais de projeto e análise de algoritmos e foi concebido na forma de texto para ensino da disciplina de análise de algoritmos, podendo constituir uma apostila ou parte de um livro-texto sobre o assunto. Seu conteúdo é dirigido a alunos de graduação em Ciência da Computação, Sistemas de Informação ou Engenharias. O documento está organizado em 4 capítulos:

- Capítulo 1: Algoritmos. Onde é feita uma introdução ao assunto, fornecendo as bases para os demais capítulos.
- Capítulo 2: Ordens assintóticas de complexidade. Apresenta e discute as notações O, Θ e Ω e suas principais propriedades.
- Capítulo 3: Recursividade. Que apresenta o princípio de divisão e conquista para projeto de algoritmos e métodos de análise de algoritmos recursivos.
- Capítulo 4: Considerações finais.

Ao longo do texto, quadros com fundo cinza discutem informações complementares ou apresentam tópicos mais avançados, incluindo assuntos para pesquisa. Ao final de cada um dos três capítulos iniciais, são propostos exercícios. Devido à extensão do tema, este documento aborda apenas tópicos iniciais sobre complexidade de algoritmos. Portanto, sua publicação ainda exige maior aprofundamento, com discussão de conceitos mais avançados, tais como problemas NP-completos.

Paulo César Rodacki Gomes

Capítulo 1

Algoritmos

A palavra "algoritmo", denota qualquer método especial para resolver um certo tipo de problema [17]. Informalmente podemos definir um algoritmo como um procedimento computacional bem definido que pega um valor, ou um conjunto de valores, como entrada e produz um valor, ou conjunto de valores, como saída. Portanto, um algoritmo é uma seqüência de passos computacionais não ambíguos que transforma a entrada em saída [9].

Um exemplo de problema para o qual existem vários algoritmos é o problema de ordenação de uma sequência de números em uma ordem não-decrescente. Formalmente, o problema de ordenação pode ser definido da sequinte forma:

Entrada: uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$.

Saída: uma permutação (ou reordenamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Dada uma sequência de entrada como $\langle 25, 37, 12, 48, 57, 92, 33, 86 \rangle$, por exemplo, um algortimo de ordenação deve retornar uma sequência de saída igual a $\langle 12, 25, 33, 37, 48, 57, 86, 92 \rangle$. A sequência de entrada é chamada de uma instância do problema de ordenação. Uma instânca de um problema corresponte então a todo o conjunto de entrada necessário para resolver o problema.

Um algoritmo é dito correto quando, para qualquer instância de entrada, produz uma saída correta, se forem fornecidos tempo e memória suficiente para sua execução [28]. Neste caso, dizemos que um algoritmo correto resolve um dado problema computacional. Um algoritmo incorreto pode produzir uma saída incorreta ou pode até não terminar sua execução.

O fato de um algoritmo teoricamente resolver determinado problema nem sempre significa que seja aceitável na prática, pois o tempo e a memória consumidos podem ser de extrema importância. Muitas vezes existem vários algoritmos diferentes para resolver um mesmo problema, e uma questão importante na Ciência da Computação é determinar qual o melhor algoritmo para resolver o problema. Nesse caso, uma regra geral é escolher o algoritmo que é mais fácil de entender, de implementar e de documentar [1]. Entretanto, normalmente o

desempenho do algoritmo é um fator importante, então é necessário escolher um algoritmo que execute com rapidez e use os recursos computacionais disponíveis eficientemente. Por este motivo, é necessário considerar o problema de avaliar o tempo de execução de uma algoritmo e quais ações podem ser tomadas para solucionar problemas com mais rapidez.

1.1 A escolha de um algoritmo

A escolha de um algoritmo para resolver determinado problema computacional passa por uma série de critérios. Por exemplo, caso seja necessário implementar determinado software para ser utilizado somente uma vez, com um conjunto pequeno de dados, então deve-se escolher o algoritmo mais simles e fácil de implementar conhecido, para ter o código escrito e depurado o mais rápido possível. Porém, se o software vai ser usado várias vezes (e possivelmente mantido) por várias pessoas por um período de tempo mais longo, então outras questões se tornam importantes.

A primeira questão a ser considerada é a **simplicidade**. Um algoritmo mais simples é desejável por vários motivos. Em primeiro lugar, é mais fácil de implementar, e existe menor probabilidade de se cometer erros ao implementar algoritmos mais simples, em relação a algoritmos mais complicados.

Outra questão é a **clareza** na escrita do código fonte do algoritmo e o cuidado na documentação para que ele possa ser mantido posteriormente por outras pessoas. Um algoritmo simples sempre é mais fácil de ser descrito. Se ele for implementado de forma clara e com boa documentação, modificações posteriores no código fonte original podem ser feitas de forma mais fácil e segura por uma outra pessoa pois o autor original muitas vezes não está mais disponível para realizar essa tarefa.

Finalmente, chegamos à questão central de nosso estudo, a **eficiência** do algoritmo. Quando o algoritmo vai ser empregado várias vezes, sua eficiência torna-se importante. Geralmente a eficiência é associada ao tempo que um algoritmo leva para resolver uma instância de um problema, porém, outros recursos também podem ser considerados, tais como: **(i)** a quantidade de memória utilizada pelo algoritmo, **(ii)** tráfego gerado na rede quando o algoritmo é distribuído e **(iii)** quantidade de dados que deve ser lida e/ou escrita em disco, quando for o caso [9] [22].

1.2 Análise de algoritmos

A avaliação da eficiência de um algoritmo é chamada de **análise de um algoritmo**, e consiste fundamentalmente em estimar a quantidade de recursos requeridos pelo algoritmo, em especial o **tempo** necessário para solucionar um problema. Em geral, ao se analizar vários algoritmos que solucionam um determinado problema, o mais eficiente é identificado. Além disso, esta análise pode indicar mais de um candidato e apontar vários algoritmos inferiores que devem ser descartados.

Para se fazer a análise de algoritmos é necessário definir um modelo da tecnologia de implementação que será usada, incluindo um modelo para os recursos dessa tecnologia e os custos envolvidos. No presente documento, vamos utilizar o modelo computacional RAM onde operações simples tais como $+$, $-$, etc. consomem exatamente uma unidade de tempo para serem executadas. Laços e chamadas a sub-rotinas não são considerados operações simples, mas são compostos por agrupamentos de operações simples. Cada operação de acesso a memória também consome exatamente uma unidade de tempo[25]. Além disso, as instruções são executadas seqüencialmente, sem operações concorrentes.

Além da análise de algoritmos propriamente dita, o tempo de execução de um algoritmo pode ser avaliado através do processo de *benchmarking*. Quando temos dois ou mais algoritmos que resolvem o mesmo problema, podemos gerar várias instâncias típicas do problema, aceitando-as como um conjunto representativo, e executar os algoritmos medindo na prática o tempo que cada um leva para resolver o conjunto de instâncias. No caso do problema de ordenação, por exemplo, pode-se testar vários algoritmos quanto sua eficiência (e corretude) incluindo seqüências de números já ordenados, seqüências vazias e assim por diante.

A avaliação de eficiência de algoritmos através de *benchmarking* aparentemente é uma boa opção, porém ela traz algumas desvantagens. Em primeiro lugar, todos os algoritmos a serem avaliados precisam ser implementados a priori, isso pode ser uma dificuldade, visto que pode-se gastar muito tempo para implementar todos os algoritmos para um determinado problema. Outra desvantagem é o fato de que, mesmo com algoritmos corretos, certos problemas são intratáveis computacionalmente, fazendo com que o tempo para o algoritmo produzir uma resposta seja impraticável. Por isso, veremos nas próximas seções conceitos e métodos para fazer a análise "teórica" de algoritmos.

O comportamento de um algoritmo para resolver determinado problema pode ser diferente para cada possível entrada. Tomando como exemplo o problema de ordenação, um algoritmo para este problema leva tempos diferentes para ordenar duas sequências com a mesma quantidade de números, dependendo de quão próximas elas já estão de estarem ordenadas. Entretanto, para analisar um algoritmo, em geral é mais importante verificar que o seu tempo de execução cresce à medida que cresce o tamanho da entrada. Portanto, a seguir definiremos com maiores detalhes o "tamanho da entrada" e o "tempo de execução".

1.2.1 Tamanho de entrada

O tamanho de entrada de um problema depende da natureza do problema. Para o problema de ordenação, uma boa medida do tamanho da entrada é a quantidade de números da sequência a ser ordenada. Para um algoritmo que resolve sistemas de n equações lineares com n incógnitas, o tamanho da entrada pode ser expresso por n. Algumas vezes é conveniente descrever o tamanho da entrada com mais de um parâmetro. Por exemplo, em algoritmos pra resolver problemas utilizando grafos, normalmente o tamanho da entrada é dado pela quantidade de vértices somada à quantidade de arestas do grafo.

1.2.2 Tempo de execução

O tempo de execução de um algoritmo para uma determinada entrada é função do número de operações primitivas ou "passos" executados. É conveniente definirmos a noção de "passo de execução" de forma mais independente possível de máquina. Podemos inicialmente dizer que cada linha de código (ou pseudocódigo) leva um tempo constante para ser executada. Uma linha pode levar um tempo diferente de outra linha, então assumimos que i-ésima linha de código leva um tempo c_i constante. Isto reflete de forma razoavelmente fiel o modelo computacional que adotamos. Ao analisar algoritmos, podemos iniciar com expressões confusas para o tempo de execução, mas nosso objetivo é chegar a uma expressão simples, que facilite a comparação entre diferentes algoritmos para resolver um problema.

Definimos a função $T(n)$ para representar o tempo requerido para um algoritmo solucionar um problema de tamanho n. Podemos chamar $T(n)$ de tempo de execução do algoritmo. Por exemplo, um determinado algoritmo pode ter tempo de execução expresso por $T(n) = cn$, onde c é uma constante. Mesmo sem sabermos o tamanho de determinada entrada ou o tempo exato de execução do algoritmo, a expressão nos indica que o tempo de execução cresce linearmente à medida que cresce o tamanho da entrada do problema. Pode-se dizer que este algoritmo executa em *tempo linear* ou que é um *algoritmo linear* para resolver o problema.

Como já foi mencionado anteriormente, diferentes instâncias de um problema do mesmo tamanho podem acarretar em tempos de execução diferentes. Freqüentemente a análise de algoritmos é feita para o pior caso de entrada, ou seja, o tempo máximo de execução de uma entrada, dentre todas as possíveis entradas de tamanho n. A seguir, veremos dois exemplos de análise de algoritmos de ordenação.

1.3 Análise do algoritmo Selection Sort

Como um primeiro exemplo de anáise de algoritmo, vamos considerar o algoritmo de ordenação por seleção, ou *SelectionSort* [19]. A idéia geral do algoritmo é bastante simples: dado um vetor A (ou array) com n números, em cada iteração o menor elemento da parte embaralhada vetor ocupa a primeira posição da parte ordenada, no início do vetor. Assim, na primeira iteração, o menor elemento vai ocupar a primeira posição do vetor. Na próxima iteração o segundo menor valor ocupa a segunda posição do vetor, e assim por diante. O índice i comanda o laço principal, e o índice j varre a parte não-ordenada do vetor procurando pelo menor elemento. Quando ele é encontrado, seu índice é armazenado na variável *menor*, e ao final da varredura o menor valor é trocado com o i-ésimo valor.

A figura 1.1 ilustra este procedimento tomando como exemplo de entrada um vetor $A = \langle 25, 37, 12, 48, 57, 92, 33, 86 \rangle$. Os quadrados representam os elementos do vetor A, com os seus índices logo acima. Os quadrados cinza representam a parte do vetor já ordenada. Cada figura (de a até h) mostra a iteração do

algoritmo para cada valor do índice i.

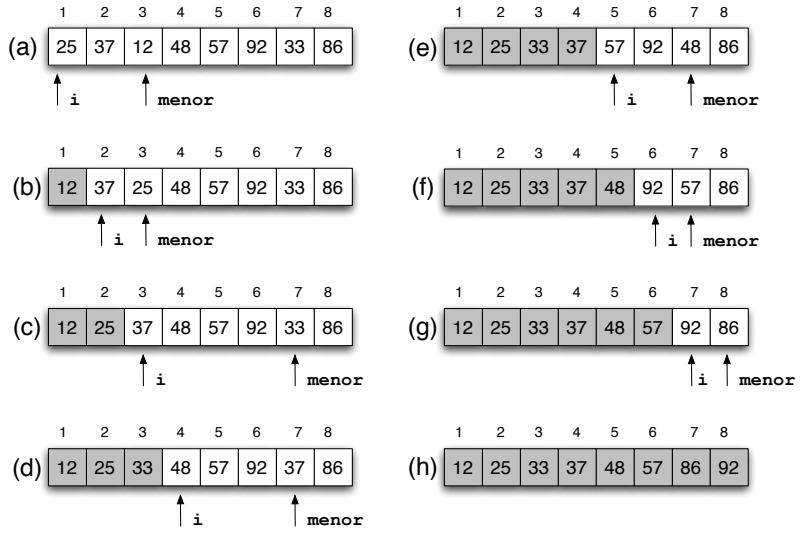

Figura 1.1: Exemplo de execução do algoritmo SelectionSort

Para analisar o *SelectionSort* precisamos olhar cuidadosamente o pseudo-código do algoritmo no quadro 1.1. Cada linha k de código consome um determinado tempo constante, chamado de "custo" e representado por c_k.

```
1 Algoritmo: SelectionSort(A)
2 para i ← 1 até n − 1 faça
3     menor ← i;
4     para j ← i + 1 até n faça
5         se A[j] < A[menor] então
6             menor ← j;
7     temp ← A[menor];
8     A[menor] ← A[i];
9     A[i] ← temp;
```

Algoritmo 1.1: SelectionSort

O cálculo do tempo de execução do algoritmo precisa verificar quantas vezes a k-ésima linha de custo c_k é executada para ordenar um vetor com n elementos. Para isso, definimos o termo t_i, que representa a quantidade de vezes que o laço da linha 4 é executado para cada $i = 1, 2, \ldots, (n-1)$. Assim, montamos a tabela 1.1 relacionando o custo de cada linha de pseudo-código com a quantidade de vezes que a linha é executada. Quando o laço "para–até–faça" encerra normalmente, o teste é executado uma vez a mais do que o corpo do laço.

O laço das linhas 4 a 6, é sempre executado j vezes, com$i$ variando de 1 a $(n-1)$ e j variando de $i+1$ até n. Portanto, a linha 4 é executada $(n-i)$ vezes para cada valor de i, e as linhas 5 e 6 são executadas $(n-i)-1$ vezes.

Tabela 1.1: Custo do SelectionSort

linha	custo	n.^{o} de execuções
2	c_2	n
3	c_3	n-1
4	c_4	$\sum_{i=1}^{n-1} t_i$
5	c_5	$\sum_{i=1}^{n-1}(t_i - 1)$
6	c_6	$\sum_{i=1}^{n-1}(t_i - 1)$
7	c_7	n-1
8	c_8	n-1
9	c_9	n-1

O tempo de execução do algoritmo é igual à soma dos custos (tempos) de execução de cada comando. Um comando que leva c_k passos para executar e é executado n vezes vai contribuir com um tempo igual a $c_k n$ ao tempo total de execução. Para calcular o tempo de execução do *SelectionSort*, somamos os produtos dos custos com as quantidades de vezes que são executados, obtendo a seguinte expressão:

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_4 \textstyle\sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1}(t_i - 1) + c_6 \sum_{i=1}^{n-1}(t_i - 1) \\ &\quad + c_7(n-1) + c_8(n-1) + c_9(n-1). \end{aligned} \tag{1.1}$$

Como o laço da linha 4 é executado $(n - i)$ vezes para cada valor de i, com i variando de 1 até $(n - 1)$, temos:

$$\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1}(n - i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

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

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

Para as linhas 5 e 6 temos:

$$\sum_{i=1}^{n-1}(t_i - 1) = \sum_{i=1}^{n-1} t_i - \sum_{i=1}^{n-1} 1$$

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

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

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

Substituindo estes termos na equação 1.1, encontramos:

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_4\left[\tfrac{n(n-1)}{2}\right] + c_5\left[\tfrac{(n-2)(n-1)}{2}\right] + c_6\left[\tfrac{(n-2)(n-1)}{2}\right] \\ &\quad + c_7(n-1) + c_8(n-1) + c_9(n-1) \therefore \\ T(n) &= \left(\tfrac{c_4}{2} + \tfrac{c_5}{2} + \tfrac{c_6}{2}\right) n^2 + \left(c_2 + c_3 - \tfrac{c_4}{2} - \tfrac{3c_5}{2} - \tfrac{3c_6}{2} + c_7 + c_8 + c_9\right) n \\ &\quad - \left(c_3 - c_5 - c_6 + c_7 + c_8 + c_9\right). \end{aligned} \tag{1.2}$$

Este tempo de execução pode ser expresso como $T(n) = an^2 + bn + c$, para constantes a, b e c que dependem dos custos (ou tempos) c_i de execução de cada comando do algoritmo. Portanto, verificamos que o tempo de execução do algoritmo *SelectionSort* é uma função quadrática do valor n.

Algoritmos *online*
Existem classes de algoritmos que não se enquadram nos termos descritos neste capítulo. Um exemplo disso são os algoritmos *online*. Na análise e projeto de algoritmos tradicionais, que podemos chamar de algoritmos *offline*, assume-se que o algoritmo possui total conhecimento dos dados de entrada. Entratanto em muitas aplicações práticas isso não acontece. Nos problemas *online* a entrada é parcialmente conhecida e dados de entrada relevantes podem surgir durante a execução do algoritmo.

Problemas *online* surgem em várias áreas de aplicação, tais como alocação de recursos em sistemas operacionais [18] [24], compressão de dados [27] [15], computação distribuída [5], mineração de dados [30], planejamento e controle de produção[6] [8], robótica [10] [26] [14] [11] e trabalho colaborativo [13] [12].

1.4 Comparação de tempos de execução

Na seção anterior, verificamos que o tempo de execução do algoritmo *Selection-Sort* é uma função quadrática da quantidade de elementos a serem ordenados. Mas o que realmente significa um algoritmo ter tempo de execução *quadrático*?

Para ter noção do que significa isso, imaginemos que exista um determinado algoritmo A para resolver o problema de ordenação, e que o tempo de execução

deste algoritmo é dado por $T_A(n) = 100n$ (portanto, uma função *linear* de n) e um outro algoritmo B, com tempo de execução dado por $T_B(n) = 2n^2$. Supondo que ambos os tempos de execução referem-se à quantidade de tempo em milissegundos que um determinado computador leva para executar os algoritmos para uma instância de problema de tamanho n, poderíamos montar o gráfico da figura 1.2 correlacionando o tempo de execução com o tamanho do problema para os dois algoritmos.

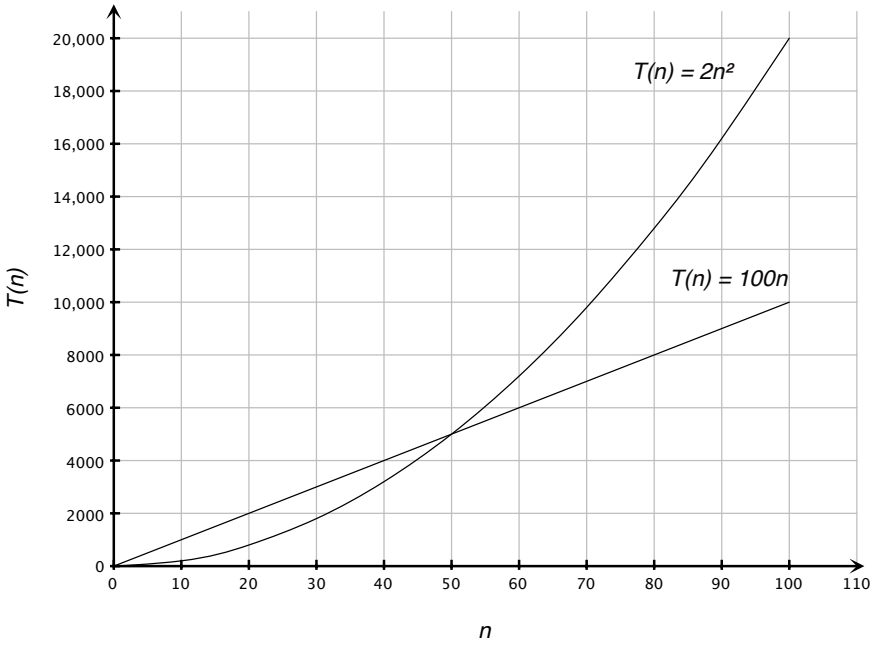

Figura 1.2: Tempo de execução de algoritmo quadrático x algoritmo linear

Neste gráfico podemos ver que, para instâncias do problema menores que 50, o algoritmo B produz uma resposta em menos tempo que o algoritmo A. Com uma entrada de tamanho 50, os dois algoritmos apresentam praticamente o mesmo desempenho, porém, quando a entrada começa a ficar maior, o algortmo B tende a ser mais lento que o algoritmo A. Além disso, a partir de $n = 50$, quanto maior o valor de n, maior a diferença de desempenho dos dois algoritmos e portanto maior a vantagem de se usar o algoritmo A ao invés de B para resolver o mesmo problema. Para entradas de tamanho 100, o algoritmo A é duas vezes mais rápido que o algoritmo B. Para entradas de tamanho 1000, é 20 vezes mais rápido.

A função do tempo de execução de um algoritmo pode também nos indicar o tamanho máximo de problema que podemos resolver com determinado algoritmo e determinado computador. À medida que a velocidade e o poder de processamento dos computadores cresce, obtemos uma melhora maior com algoritmos cuja função de tempo de execução cresce mais lentamente do que com

aqueles cuja função cresce mais rapidamente. Por exemplo, com os algoritmos da figura 1.2, suponha que dispomos de 100 segundos de tempo de processamento. Se se o computador utilizado ficasse 10 vezes mais rápido, poderíamos resolver problemas que antes levariam 1000 segundos. Com o algoritmo A, essa melhoria permite que se resolva problemas 10 vezes maiores, mas com o algoritmo B, seria possível resolver problemas apenas 3 vezes maiores. A tabela 1.2 correlaciona os tamanhos máximos de problemas que poderiam ser resolvidos com os algoritmos A e B (denotados por $max\langle n_A \rangle$ e $max\langle n_B \rangle$, respectivamente), em função do tempo disponível em segundos.

Tabela 1.2: Tamanho máximo do problema em função do tempo

Tempo (seg.)	$max\langle n_A \rangle$	$max\langle n_B \rangle$
1	10	22
10	100	70
100	1000	223
1000	10000	707

É importante ressaltar que os valores na tabela 1.2 são meramente ilustrativos, pois na prática é difícil saber o tempo exato de execução de um algoritmo sem implementá-lo e executá-lo. Além disso, conforme ja foi comentado, para alguns algoritmos o tempo de execução pode ser diferente para diferentes entradas do mesmo tamanho.

Na análise que fizemos do *SelectionSort*, assumimos algumas simplificações importantes. Em primeiro lugar, ignoramos o custo real de execução de cada comando, usando constantes arbitrárias c_i para representar estes custos. Depois disso, observamos que essas constantes representavam um detalhamento desnecessário, pois se não sabemos o valor real das constantes, poderíamos reduzir a função 1.2 a $T(n) = an^2 + bn + c$, para constantes a, b e c que dependem dos custos c_i.

Na verdade, ignoramos tanto os custos reais quanto os custos abstratos c_i. Porém, como foi visto acima, mesmo ignorando estes custos, podemos tirar conclusões importantes acerta do tempo de execução de algoritmos e comparar o desempenho de diferentes algoritmos para um mesmo problema através a análise da ordem ou taxa de crescimento da função do tempo de execução. Este assunto será abordado em maior profundidade no próximo capítulo.

1.5 Exercícios propostos

1. Dada a tabela abaixo, para cada função $f(n)$ e tempo t, determine o maior tamanho n do problema que pode ser resolvido no tempo t, assumindo que o problema leva $f(n)$ microssegundos para ser resolvido.

$f(n)$	$t = 1$seg	1 min	1 hora	1 dia	1 mês	1 ano	1 século
$\log n$							
$\sqrt{n}$							
n							
$n \log n$							
n^2							
n^3							
2^n							
$n!$							

2. Utilizando a figura 1.1 como exemplo, mostre a execução do algoritmo *SelectionSort* para a seqüência $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

3. Seja uma matriz A de elementos inteiros de dimensões $n \times n$. Os elementos de cada linha e de cada coluna estão ordenados em ordem não-decrescente. Dado um valor x, escreva um algoritmo para determinar se existem $i, j \in 1, 2, \ldots, n$ tais que $x = a_{i,j}$. Faça uma estimativa de seu tempo de execução $T(n)$ em função de n.

Capítulo 2

Ordens Assintóticas de Complexidade

No capítulo anterior, vimos conceitos relativos à eficiência de algoritmos e calculamos o tempo de execução do algoritmo *SelectionSort*. Dada uma entrada de valor n, verificamos que o algoritmo produz uma solução em um tempo que é uma função quadrática do tamanho do problema. Vimos também a comparação do tempo de execução entre um algoritmo com tempo quadrático e outro com tempo linear, e concluimos que, mesmo sem saber o tempo exato (real) de execução de determinado algoritmo, podemos obter informações importantes sobre o comportamento do algoritmo quando o tamanho do problema tende a crescer para um número suficientemente grande. Neste caso estamos avaliando a eficiência assintótica de algoritmos.

No presente capítulo, vamos estudar conceitos fundamentais necessários para a análise **assintótica** de algoritmos. Ou seja, conceitos que nos permitam avaliar a tentência de crescimento de tempo de execução de algoritmos, à medida que o tamanho da entrada do problema tende a um limite. Tal tendência reflete a quantidade de trabalho requerido pelo algoritmo para resolver um problema, essa quantidade é chamada de **complexidade do algoritmo** [29], a análise desta tendência também é chamada **análise de complexidade do algoritmo** [28].

As ordens assintóticas de complexidade seguem notações chamadas assintóticas, que são definidas em termos de funções cujos domínios estão contidos no conjunto dos números naturais $\mathbb{N} = \{0, 1, 2, \ldots\}$

2.1 Notação O

A notação O define um limite assintótico superior para determinada função. Para definir a notação O, postulamos as seguintes afirmações: seja $T(n)$ o tempo de execução de algum algoritmo, medido em função do tamanho de entrada n, assim, $T(n)$ pode ser expresso por uma função $f(n)$, e assumimos que:

1. O argumento n é um inteiro não negativo (ou seja, está contido em $\mathbb{N}$), e
2. $f(n)$ é não–negativa para todos os argumentos n.

Agora, seja $g(n)$ uma função definida no domínio dos números naturais $\mathbb{N}$. Dizemos que "$f(n)$ é $O(g(n))$" se o valor de $f(n)$ é no máximo uma constante vezes $g(n)$, exceto possivelmente para alguns valores pequenos de n. Mais formalmente, dizemos que "$f(n)$ é $O(g(n))$" se existe um inteiro n_0 e uma constante $c > 0$ tal que para todos os inteiros $n \geq n_0$, temos $f(n) \leq cg(n)$. Ou então:

$$\begin{aligned} O(g(n)) \quad = \quad & \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que} \\ & 0 \leq f(n) \leq cg(n) \text{ para todos } n \geq n_0\}. \end{aligned}$$

Isto implica no fato de $O(g(n))$ representar um *conjunto* de funções com as propriedade definidas acima. A figura 2.1 mostra graficamente esta situação: com valores de n maiores que n_0, a função $f(n)$ sempre é menor que $g(n)$ multiplicado por uma determinada constante c. O valor n_0 representa um valor mínimo para que $g(n)$ seja um limite superior assintótico para $f(n)$.

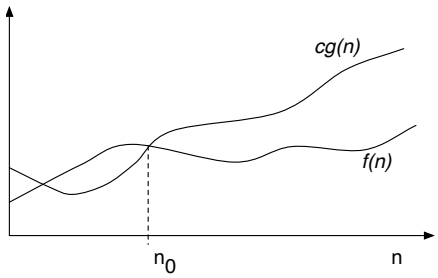

Figura 2.1: Notação O, onde $f(n) = O(g(n))$

Exemplo: suponha que um determinado algoritmo possui os seguintes tempos de execução $T(0) = 1$, $T(1) = 4$, $T(2) = 9$ e, no caso geral, $T(n) = (n + 1)^2$. Podemos afirmar que $T(n)$ é $O(n^2)$, ou que $T(n)$ é *quadrático*, porque se $c = 4$ e $n_0 = 1$, então $T(n) = (n + 1)^2 \leq 4n^2$ para $n \geq 1$. Para provar essa afirmação, expandimos $(n + 1)^2$, resultando em $n^2 + 2n + 1$. Se $n \geq 1$, então $n \leq n^2$ e $1 \leq n^2$. Portanto: $n^2 + 2n + 1 \leq n^2 + 2n^2 + ^2 = 4n^2$, provando que $T(n) \leq cf(n)$ para $c = 4$ e $f(n) = n^2$.

Em princípio, pode parecer estranho que, embora $(n + 1)^2$ seja maior que n^2, nós afirmamos que $(n + 1)^2$ é $O(n^2)$. De fato, podemos dizer ainda que $(n + 1)^2$ possui limite assintótico superior definido por qualquer fração de n^2, por exemplo, $O(n^2/200)$. Para provar, basta fazer $n_0 = 1$ e $c = 800$. Então, se $n \geq 1$, temos que $(n + 1)^2 \leq 800(n^2/200) = 4n^2$.

Estas observações são regidas pelos seguintes princípios gerais:

1. **Fatores constantes não importam:** para qualquer constante $d > 0$ e qualquer função $f(n)$, $f(n)$ é $O(df(n))$. Prova: seja $n_0 = 0$ e $c = 1/d$,

então $f(n) \leq c(df(n))$, pois $cd = 1$. da mesma forma, se sabemos que $f(n)$ é $O(g(n))$, então sabemos que $f(n)$ é $O(dg(n))$ para qualquer $d > 0$, mesmo se d possuir um valor muito pequeno. Isto se justifica porque sabemos que $f(n) \leq c_1 g(n)$ para uma constante c_1 e para todos $n \geq n_0$. Se escolhermos $c = c_1/d$, podemos verificar que $f(n) \leq c(dg(n))$ para $n \geq n_0$.

2. **Termos de menor ordem podem ser desprezados:** suponha que $f(n)$ é um polinômio na forma $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$ onde o coeficiente do termo de maior ordem, a_k, é positivo. Então, podemos desprezar todos os termos exceto o de maior grau (o de maior expoente k) e, pela regra anterior, ignorar a constante a_k, substituindo-a por 1. Portanto, podemos concluir que $f(n)$ é $O(n^2)$. Para provar este princípio, seja $n_0 = 1$ e c igual à soma de todos os coeficientes positivos a_i com $0 \leq a_i \leq k$. Se um coeficiente a_j é negativo, então certamente $a_j n^j \leq 0$ (pois $n \geq 0$). Se a_j é positivo, então $a_j n^j \leq a_j n^k$ para todo $j < k$, desde que $n \geq 1$. Portanto, $f(n)$ não é maior que n^k multiplicado pela soma dos coeficientes positivos, ou seja cn^k.

Exemplo: considerando o polinômio $f(n) = 4n^5 + 3n^4 - 10n^3 + n - 15$, como o termo de maior grau é n^5, podemos afirmar que $f(n)$ é $O(n^5)$. Para provar esta afirmação, seja $n_0 = 1$ e c a soma dos coeficientes positivos. Estes coeficientes são os dos termos de grau 5, 4 e 1, então $c = 4 + 3 + 1 = 8$. Assim, para $n \geq 1$, temos $4n^5 + 3n^4 - 10n^3 + n - 15 \leq 4n^5 + 3n^5 + n^5 = 8n^5$. Portanto $f(n) \leq 8n^5$, ou $f(n) \leq cn^5$ provando que $f(n)$ é $O(n^5)$.

A definição de $O(g(n))$ requer que todas as funções contidas no conjunto $O(g(n))$ sejam assintoticamente não-negativas, pois $0 \leq f(n) \leq cg(n)$ para valores de n suficientemente grandes. Conseqüentemente a própria função $g(n)$ deve ser assintoticamente não-negativa, caso contrário $O(g(n))$ é um conjunto vazio [20].

A notação O é usada para indicar limites superiores para o tempo de execução de algoritmos, porém, muitas vezes somente o limite superior não é suficiente. Por exemplo, apesar eventulmente verificarmos que o tempo de execução de algum algoritmo é quadrático (ou seja $O(n^2)$), podemos dizer também que ele é $O(2^n)$. Ou seja, estamos afirmando que ele não requer mais do que tempo exponencial para resolver o problema. Esta afirmação claramente é "crua" demais para se avaliar o tempo de execução, pois apesar de correta, ela é tecnicamente fraca porque na realidade o algoritmo executa em tempos muito menores que este [22]. Se quisermos obter uma estimativa mais realista do tempo de execução de um algoritmo, precisamos limites mais justos para estas medidas de tempo conforme será visto a seguir.

2.2 Notação Θ

No capítulo anterior, vimos que o algoritmo *SelectionSort* para o problema de ordenação é quadrático, sendo seu tempo de execução assintoticamente proporcional ao quadrado do tamanho de uma instância do problema. Com isso, estamos afirmando que o tempo de execução de *SelectionSort* é $T(n) = \Theta(n^2)$.

Vamos então definir o significado desta notação. Para uma dada função $g(n)$, denotamos por $\Theta(g(n))$ o *conjunto de funções*

$$\begin{aligned}\Theta(g(n)) \quad = \quad & \{f(n) : \text{existem constantes positivas } c_1,\ c_2 \text{ e } n_0 \text{ tais que} \\ & 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todos } n \geq n_0\}.\end{aligned}$$

Uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existirem constantes positivas c_1 e c_2 tais que $f(n)$ possa ficar compreendida entre $c_1 g(n)$ e $c_2 g(n)$ para valores suficientemente grandes de n. É importante salientar $\Theta(g(n))$ é um conjunto, portanto quando escrevemos "$f(n) = \Theta(g(n))$", estamos indicando que $f(n)$ é membro de $\Theta(g(n))$, ou "$f(n) \in \Theta(g(n))$".

A figura 2.2 ilustra intuitivamente as funções $f(n)$ e $g(n)$, com $f(n) = \Theta(g(n))$. Para todos os valores de $n \geq n_0$, $f(n)$ é maior ou igual a $c_1 g(n)$ e menor ou igual a $c_2 g(n)$. Assim, podemos dizer que $g(n)$ é limite assintoticamente *justo*, ou assintoticamente *exato* para $f(n)$.

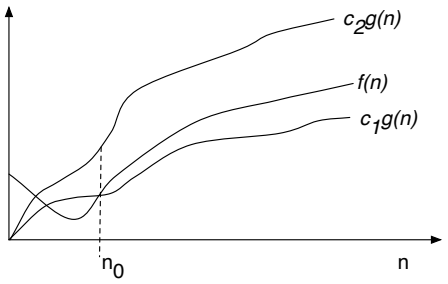

Figura 2.2: Notação Θ, onde $f(n) = \Theta(g(n))$

Exemplo: considerando o polinômio $f(n) = \frac{1}{2}n^2 - 3n$, afirmamos que ele é $\Theta(n^2)$. Para provar esta afirmação, pela definição de Θ, devemos determinar constantes positivas c_1, c_2 e n_0 tais que $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ para todo $n \geq n_0$. Dividindo esta expressão por n^2, obtemos $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$. Podemos satisfazer a inequação da direita ($\frac{1}{2} - \frac{3}{n} \leq c_2$) para qualquer valor $n \geq 1$ escolhendo $c_2 = 1/2$. Da mesma forma, podemos satisfazer a inequação da esquerda ($c_1 \leq \frac{1}{2} - \frac{3}{n}$) para qualquer valor $n \geq 7$ escolhendo $c_1 = 1/14$. Portanto, escolhento $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, provamos que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Exemplo: analogamente ao exemplo anterior, podemos usar a definição formal de Θ para verificar que $15n^3 \neq \Theta(n^2)$. Para efeito de contradição, suponha que existam c_2 e n_0 tais que $15n^3 \leq c_2 n^2$ para todo $n \geq n_0$. Então temos $n \leq c_2/15$, que não pode ser satisfeito para valores de n arbitrariamente grandes, visto que c_2 é uma constante.

2.3 Notação Ω

Na seção 2.1, vimos que a notação O define limites assintóticos superiores para uma dada função. Da mesma forma, podemos estabelecer limites assintóticos inferiores pela definição da notação Ω a seguir:

$$\begin{aligned}\Omega(g(n)) \quad = \quad & \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que} \\ & 0 \leq cg(n) \leq f(n) \text{ para todos } n \geq n_0\}.\end{aligned}$$

Esta definição atesta que se o tempo de execução de um determinado algoritmo é $\Omega(g(n))$, então, não importando qual instância particular de problema de tamanho n for escolhida como entrada dentre um conjunto de instâncias deste tamanho, o tempo de execução para este conjunto de instâncias é no mínimo uma constante multiplicada por $g(n)$, para n suficientemente grande.

A figura 2.3 ilustra intuitivamente o comportamento de uma função $f(n)$ com $f(n) = \Omega(g(n))$. Pode-se ver que para todos os valores de $n \geq n_0$, $f(n)$ é maior ou igual a $cg(n)$.

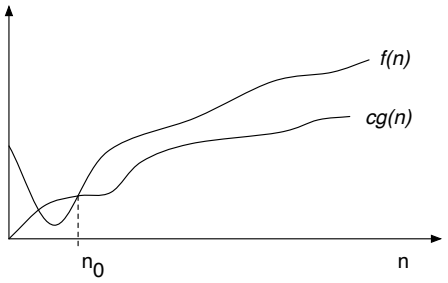

Figura 2.3: Notação Ω, onde $f(n) = \Omega(g(n))$

As definições assintóticas apresentadas até agora permitem que se prove o seguinte teorema:

Teorema: para qualquer par de funções $f(n)$ e $g(n)$, $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Para demonstrar deste teorema, vamos verificar que $f(n) = an^2 + bn + c = \Theta(n^2)$. Dadas quaisquer constantes a, b e c com $a > 0$, implica em podermos provar que $an^2 + bn + c = O(n^2)$ e $an^2 + bn + c = \Omega(n^2)$. Este teorema freqüentemente é usado para provar limites assintoticamente exatos a partir de limites superiores e inferiores [9].

2.4 Análise do algoritmo Bubble Sort

Para exemplificar a notação Ω, vamos examinar o algoritmo *BubbleSort*. Este algoritmo é um dos mais simples para resolver o problema de ordenação, e funciona da seguinte forma: tomando como entrada um vetor com n elementos, o algoritmo inicia comparando o primeiro e o segundo elementos do vetor. Se o primeiro elemento for maior que o segundo, eles são trocados. A seguir, o algoritmo repete este procedimento com o próximo par de valores no vetor (segundo e terceiro elementos), até chegar ao último par. Ao final desse processo, o maior elemento estará colocado na última posição do vetor. Todo o procedimento é repetido até a penúltima posição do vetor, e assim por diante, até que todo o vetor esteja ordenado.

A figura 2.4 ilustra este procedimento tomando como exemplo de entrada uma vetor $A = \langle 25, 37, 12, 48, 57, 92, 33, 86 \rangle$. Os quadrados cinza escuro mostram a parte do vetor que já se encontra ordenada. Na primeira "passada", o algoritmo move o maior valor para a última posição do vetor. Na segunda passada o segundo maior valor vai para a penúltima posição, e assim por diante. Neste exemplo é importante notar que, depois da quarta passada do algoritmo, o vetor encontra-se ordenado e a partir daí nenhuma troca de valores será feita.

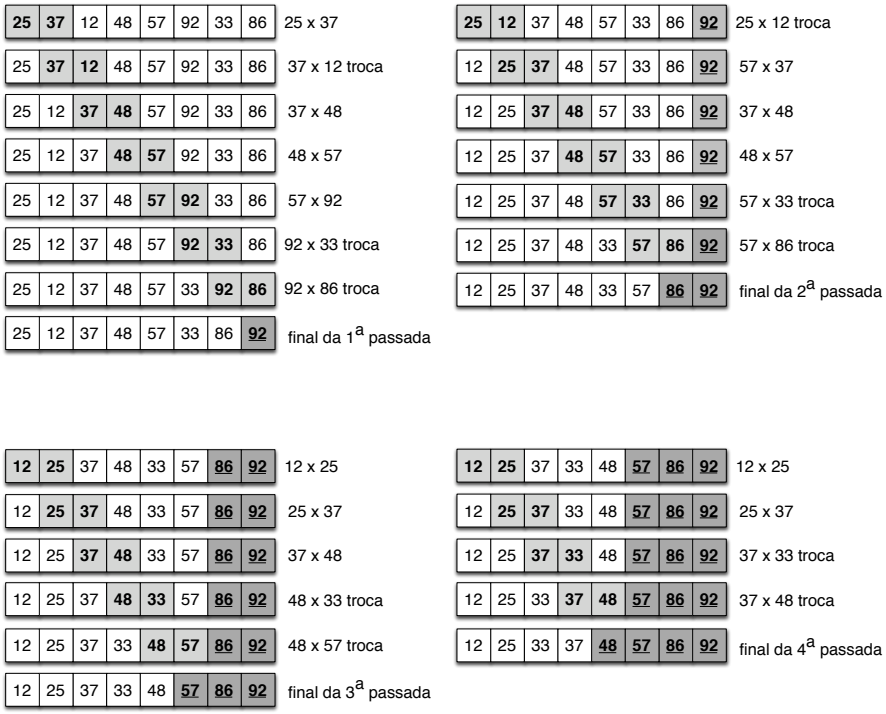

Figura 2.4: Exemplo de execução do algoritmo BubbleSort

O pseudo código do *BubbleSort* para ordenar um vetor A com n elementos é mostrado no quadro 2.1. A variável *booleana* "troca" sinaliza quando houve

uma varredura no vetor sem ocorrer nenhuma troca de valores. Neste caso o vetor já está ordenado e o algoritmo pode encerrar sua execução.

```
1  Algoritmo: BubbleSort(A)
2  int i, j;
3  para i ← n até 1 faça
4      boolean troca ← falso;
5      para j ← 1 até (i-1) faça
6          se A[j] > A[j + 1] então
7              int temp ← A[j];
8              A[j] ← A[j + 1];
9              A[j + 1] ← temp;
10             troca ← verdadeiro;
11     se troca ≠ verdadeiro então retorna
```

Algoritmo 2.1: BubbleSort

Se analisarmos o pseudo-código do *BubbleSort* da mesma forma que fizemos com o algoritmo *SelectionSort* na seção 1.3, montamos a seguinte tabela com custo e a quantidade de vezes que cada linha é executada.

Tabela 2.1: Custo do BubbleSort

linha	custo	n.^{o} de execuções
2	c_2	1
3	c_3	n+1
4	c_4	n
5	c_5	$\sum_{i=1}^{n} t_i$
6, 7, 8, 9, 10	$c_6, c_7, c_8, c_9, c_{10}$	$\sum_{i=1}^{n}(t_i - 1)$
11	c_{11}	n

Multiplicando o custo de cada linha com a quantidade de vezes que ela é executada, chegaremos à seguinte função para o tempo de execução do algoritmo:

$$T(n) = c_2 + c_3(n+1) + c_4 n + c_5 \sum_{i=1}^{n} t_i + (c_6 + c_7 + c_8 + c_9 + c_{10}) \sum_{i=i}^{n} (t_i - 1) + c_{11} n.$$

Ao contrário do que acontece com o *SelectionSort*, entradas do mesmo tamanho podem produzir tempos de execução diferentes para o algoritmo *BubbleSort*. O pior caso para este algoritmo ocorre quando os valores da entrada estão em ordem decrescente, e o algoritmo executa em tempo quadrático (deixamos essa verificação como exercício para o leitor). Porém, quando os valores da entrada

já estão ordenados, o algoritmo encerra sua execução logo na primeira passada completa pelo vetor. As linhas 3, 4 e 11 são executadas somente uma vez, a linha 5 é executada n vezes, a linha 6 é executada $n-1$ vezes e as linhas 7 a 10 não são executadas, resultando em:

$$T(n) = c_2 + c_3 + c_4 + c_5 n + c_6(n-1) + c_{11} \therefore$$

$$T(n) = (c_5 + c_6)n + (c_2 + c_3 + c_4 + -c_6 + c_{11}).$$

Como os termos c_k são constantes, este resultado pode ser simplificado em $T(n) = an + b$, onde a e b são constantes dependentes de c_k. Isto nos mostra que, apesar do algoritmo rodar em tempo quadrático no pior caso, ele também pode rodar em tempo linear dependendo dos dados de entrada. Se o tempo de execução do *BubbleSort* é linear no melhor caso e quadrático no pior, podemos dizer que para este algorimo, o tempo de execução fica entre $\Omega(n)$ e $O(n^2)$.

O que significa a taxa de crescimento de uma função?

No presente capítulo fizemos a análise do algoritmo *BubbleSort*, concluíndo que sua ordem de complexidade no pior caso é quadrática. Vimos também que fatores constantes podem ser desprezados na análise assintótica das funções que descrevem ordens de complexidade de algoritmos. Para comparação de desempenho de diferentes algoritmos que resolvem um mesmo problema, é importante analisar a taxa de crescimento das funções quando o tamanho da entrada tende a crescer. Mas, na prática, o que isso tudo significa?

O quadro abaixo exemplifica o crescimento de funções comuns na análise de algoritmos em função do crescimento do tamanho do problema dado por n [25]. É mostrado o tempo necessário para cada algoritmo executar ξ operações, sendo que cada operação demora hipoteticamente um nanossegundo (10^{-9}segundos).

n	$\xi = \log n$	n	$n \log n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 anos
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 s	$10^{15} anos$
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 dias	
10^2	0.007 μs	0.10 μs	0.644 μs	10 μs	8×10^{13} anos	
10^3	0.010 μs	1 μs	9.966 μs	1 ms		
10^4	0.013 μs	10 μs	130 μs	100 ms		
10^5	0.017 μs	0.1 ms	1.67 ms	10 s		
10^6	0.020 μs	1 ms	19.93 ms	16.7 min		
10^7	0.023 μs	0.01 s	0.23 s	1.16 dias		
10^8	0.027 μs	0.10 s	2.66 s	115.7 dias		
10^9	0.030 μs	1 s	29.90 s	31.7 anos		

Observando este quadro, podemos chegar às seguintes conclusões: (i) todos os algoritmos levam praticamente o mesmo tempo com $n = 10$; (ii) o uso do algo-

ritmo com tempo de execução $n!$ se torna impraticável antes de $n = 20$; (iii) o algoritmo com tempo de execução igual a 2^n possui uma abrangência maior de utilização, mas também se torna impraticável para $n > 40$; (iv) o algoritmo quadrático (com tempo igual a n^2) é rápido até $n = 100$, mas seu desempenho logo se deteriora daí pra frente, tornando-se impraticável com $n > 10^6$; (v) os algoritmos com tempo n e $n \log n$ continuam com bom desempenho em problemas com entrada de até um bilhão de ítens; e (vi) na prática provavelmente não encontraremos um problema real onde um algoritmo $\Theta(\log n)$ seja lento demais.

Portanto, mesmo ignorando fatores constantes, podemos ter uma boa idéia se determinado algoritmo vai executar em um tempo plausível sendo dado o tamanho de entrada de uma instância do problema.

2.5 Combinando notações assintóticas

A natureza da notação assintótica nos permite escrever "$n = O(n^2)$", por exemplo. Essa expressão pode parecer estranha a princípio, mas se nos lembrarmos que este tipo de notação define um *conjunto de funções*, a expressão acima indica que a função linear $f(n) = n$ pertence a um conjunto de funções que possuem cn^2 como um limite assintótico superior para determinada constante $c > 0$ e para valores de n suficientemente grandes. De fato, o sinal de igualdade em $n = O(n^2)$ na verdade significa *pertinência*, ou seja $n \in O(n^2)$ [20].

Estas observações mostram que existem técnicas gerais para combinar a notação assintótica em expressões que remetem a expressões relacionais da álgebra (como a iguldade mostrada acima) e levam a propriedades para a combinação de notação assintótica. Assim, uma série de regras podem ser inferidas. A primeira delas é a regra da soma.

2.5.1 Soma

Devido ao fato exposto acima, também é possível escrever $3n^2 + n - 2 = 3n^2 + \Theta(n)$. Esta equação pode ser interpretada da seguinte maneira: quando a notação assintótica aparece em alguma equação (ou inequação), ela denota alguma função desconhecida. Por exemplo: a equação $3n^2 + n - 2 = 3n^2 + \Theta(n)$ significa que $3n^2 + n - 2 = 3n^2 + f(n)$, e que $f(n)$ é alguma função pertencente a $\Theta(n)$. Neste caso em particular, $f(n) = n - 2$, que realmente é $\Theta(n)$.

Em alguns casos, a notação assintótica pode aparecer no lado esquerdo da equação. Por exemplo: $4n^3 + \Theta(n^2) = \Theta(n^3)$. esta equação significa que, não importando qual função desconhecida seja escolhida para o lado esquerdo da igualdade, sempre é possivel escolher uma outra função para o lado direito de forma que a igualdade seja válida. Ou seja, nesta equação, para qualquer função $f(n) \in \Theta(n^2)$, existe uma função $g(n) \in \Theta(n^3)$ tal que $4n^3 + f(n) = g(n)$ para qualquer n.

As expressões acima também podem ser interpretadas da seguinte maneira: suponha que estamos analisando um algoritmo composto por duas partes. O tempo de execução de uma delas é $O(n^2)$ e da outra é $O(n^3)$. Esses tempos poderiam ser somados para se obter o tempo de execução do algoritmo todo.

Formalmente, supondo que se saiba que determinado tempo de execução $T_1(n)$ é $O(g(n))$, e outro tempo de execução T_2 é $O(f(n))$. Além disso, suponha que $f(n)$ não cresce mais rapidamente que $g(n)$, ou seja, $f(n)$ é $g(n)$. Então podemos concluir que $T_1(n) + T_2(n)$ é $O(g(n))$.

Para provar esta afirmação, sabemos que existem constantes n_1, n_2, n_3, c_1, c_2 e c_3 tais que:

a) Se $n \geq n_1$, então $T_1(n) \leq c_1 g(n)$;

b) Se $n \geq n_2$, então $T_2(n) \leq c_2 f(n)$;

c) Se $n \geq n_3$, então $f(n) \leq c_3 g(n)$.

Seja n_0 o maior valor entre n_1, n_2 e n_3, tal que as inequações (a), (b) e (c) sejam satisfeitas para todo $n \geq n_0$. Então temos $T_1(n)+T_2(n) \leq c_1 g(n)+c_2 f(n)$. utilizando a inequação (c), podemos escrever $T_1(n)+T_2(n) \leq c_1 g(n)+c_2 c_3 g(n)$.

Portanto, temos que $T_1(n) + T_2(n) \leq cg(n)$, onde $c = c_1 + c_2 c_3$, e assim concluimos que $T_1(n) + T_2(n)$ é $O(g(n))$.

2.5.2 Transitividade

A propriedade de transitividade para notação assintótica é semelhante à dos números inteiros e reais. Uma relação é dita transitiva se obedecer à seguinte lei: "se $A \leq B$ e $B \leq C$, então $A \leq C$". Por exemplo, se $4 \leq 7$ e $7 \leq 11$, então sabemos que $4 \leq 11$.

A notação assintótica O, por exemplo, também possui transitividade: se $f(n)$ é $O(g(n))$ e $g(n)$ é $O(h(n))$, então implica em $f(n)$ ser $O(h(n))$.

Para provar esta propriedade, suponha que $f(n)$ é $O(g(n))$. Então, pela definição de O, existem constantes n_1 e c_1 tais que $f(n) \leq c_1 g(n)$ para todo $n \geq n_1$. Analogamente, se $g(n)$ é $O(h(n))$, então existem constantes n_2 e c_2 tais que $g(n) \leq c_2 h(n)$ para todo $n \geq n_2$. Seja n_0 o maior valor dentre n_1 e n_2, e seja $c = c_1 c_2$. Então, para todo $n \geq n_0$, sabemos que $f(n) \leq c_1 g(n)$ e $g(n) \leq c_2 h(n)$. Portanto $f(n) \leq c_1 c_2 h(n)$, provando que $f(n)$ é $O(h(n))$.

Esta propriedade também se aplica às outras notações assintóticas. Resumidamente podemos listar as seguintes regras quanto à transitividade:

- se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, então $f(n) = \Theta(h(n))$;
- se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$;
- se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$, então $f(n) = \Omega(h(n))$.

2.5.3 Reflexividade

A propriedade de reflexividade afirma que determinada função $f(n)$ é $O(f(n))$. Como prova esta propriedade, basta usar a definição da notação O. Se $f(n)$ é $O(f(n))$, então existem constantes c e n tais que $f(n) \leq cf(n)$ para todo $n \geq n_0$. Seja $c = 1$ e n_0, então $f(n) \leq f(n)$ para todo valor $n \geq 0$, atendendo a defininição de O e provando a propriedade de reflexividade. Generalizando, temos que:

- $f(n) = \Theta(f(n))$;
- $f(n) = O(f(n))$;
- $f(n) = \Omega(f(n))$.

2.5.4 Simetria

Por fim, ainda temos as relações de simetria e simetria transposta, que intuitivamente podem ser observadas nas figuras 2.1, 2.2 e 2.3 e são definidas a seguir:

- Simetria: $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$;
- Simetria transposta: $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$.

Ordens de complexidade polinomiais e não-polinomiais (NP)
O grau de um polinômio é o maior expoente encontrado entre seus termos. Por exemplo, o grau do polinômio $f(n) = 4n^5 + 3n^4 - 10n^3 + n - 15$ é 5 porque $4n^5$ é o termo de maior ordem. Funções *exponenciais* possuem a forma a^n para $a > 1$. Qualquer função exponencial cresce mais rápido que qualquer função polinomial. Ou seja, é possível demonstrar que para qualquer polinômio $p(n)$ que $p(n)$ é $O(a^n)$. Da mesma forma, nenhuma função exponencial a^n com $a > 1$ é $O(p(n))$ para qualquer polinômio $p(n)$ [16].

Os algoritmos analisados neste documento são algoritmos da classe P, com tempo polinomial. Isto é, para entradas de tamanho n, seu tempo de execução no pior caso é $O(n^k)$ para alguma constante k. Existe uma classe de problemas, chamados NP, para os quais não se conhecem soluções polinomiais. Um problema é dito não-deterministicamente polinomial (NP) se uma solução hipotétiica do problema puder ser verificada em tempo polinomial. Caso contrário, um problema não polinomial é chamado de NP completo. Caso consigamos verificar que determinado algoritmo é NP-completo, então ele é intratável. Neste caso, é melhor procurar desenvolver um algoritmo aproximativo do que tentar buscar a solução exata [25].

No cpróximo capítulo, estudaremos a análise de uma importante classe de algoritmos, os algoritmos *recursivos*.

2.6 Exercícios propostos

1. Considere as 2 funções f_1: n^2 e f_2: n^3
   Para cada i e j iguais a 1 e 2, determine se $f_i(n)$ é $O(f_j(n))$. Encontre valores para n_0 e c que provem essa relação, ou estabeleça valores para derivar uma contradição que prove que $f_i(n)$ não é $O(f_j(n))$.

2. Encontre valores para n_0 e c que provem que as relações abaixo são verdadeiras.

   a) n^2 é $O(0.001n^3)$

   b) $25n^4 - 19n^3 + 13n^2 - 106n + 77$ é $O(n^4)$

   c) 2^{n+10} é $O(2^n)$

   d) n^{10} é $O(3^n)$

3. Considere o problema de **busca**:
   Entrada: seqüência de n números $A = \langle a_1, a_2, \ldots, a_n \rangle$ e um valor v.
   Saída: um valor i tal que $v = A[i]$ ou o valor especial $NULO$ caso v não apareça em A.
   Escreva o pseudo código de um algoritmo para resolver este problema. Determine O e Ω do seu algoritmo.

4. Considere novamente o problema de **busca**. Observe que se a seqüência A estiver ordenada, podemos comparar v com o elemento no meio da seqüência e eliminar metade da seqüência para fazer uma nova comparação. A **busca binária** é um algoritmo que executa esse tipo de procedimento. Escreva o peseudo código de um algoritmo iterativo que execute busca binária. Prove que o algoritmo é $\Theta(n)$.

Capítulo 3

Recursividade

Até agora vimos exemplos de análise de dois algoritmos iterativos (*Selection-Sort* e *BubbleSort*). Porém, muitos problemas em Ciência da Computação são modelados e resolvidos através de definições e algoritmos recursivos. O presente capítulo discute recorrência no contexto de algoritmos recursivos e apresenta as bases para análise de complexidade deste tipo de algoritmo.

3.1 Definições recursivas

Numa definição recursiva, classes de objetos ou fatos são definidos em termos de objetos ou fatos da mesma classe. A definição deve ter um significado. Por exemplo, a definição "um cavalo é um cavalo da mesma cor" não faz sentido e portanto não é uma definição válida. Além disso, a definição recursiva também não pode ser paradoxal. Por exemplo, "uma chave é uma chave se e somente se não for uma chave". Este é outro exemplo de definição recursiva inválida. Ao invés disso, a definição recursiva deve conter as seguintes regras:

a) Uma ou mais regras-base (ou regras básicas) nas quais objetos simples ou elementares são definidos, e;

b) Uma ou mais regras indutivas, pelas quais objetos maiores são definidos em termos de objetos menores pertencentes à coleção de objetos definidos.

Exemplo: o fatorial de um número inteiro positivo n é denotado por $n!$ e seu valor é igual a $n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$. Por exemplo, $4! = 4 \times 3 \times 2 \times 1 = 24$. O caso especial $0!$ é definido com o valor 1. O fatorial pode ser definido recursivamente da seguinte forma:

$$n! = \begin{cases} 1, & \text{se } n \in \{0,1\} \text{ (\textit{regra-base})}; \\ n(n-1)!, & \text{se } n > 1 \text{ (\textit{regra indutiva})}. \end{cases}$$

Exemplo: em estruturas de dados temos muitos exemplos de definições recursivas para estruturas comuns. A figura 3.1 mostra esquematicamente a definição recursiva de árvores binárias, onde uma árvore binária é: (i) uma árvore vazia (*regra-base*) ou (ii) um nó raiz seguido de duas sub-árvores, a sub-árvore da esquerda – *sae* – e a sub-árvore da direita – *sad* (*regra indutiva*).

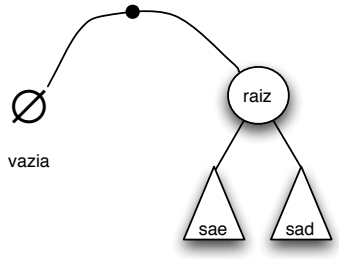

Figura 3.1: Definição recursiva de árvores binárias

A seguir, vamos discutir a noção de recursividade no projeto e análise de complexidade de algoritmos.

3.2 Algoritmos recursivos

A idéia de recursividade é também comum em algoritmos, muitos algoritmos úteis na computação são recursivos. Em termos de codificação deste tipo de algoritmo, significa que uma instância de execução de um procedimento (ou função, ou método) recursivo chama outra instância do próprio procedimento. Por exemplo, a definição recursiva da função fatorial pode levar à implementação recursiva[1] mostrada no quadro 3.1. A recursão acontece na linha 5, quando o procedimento chama uma outra instância de si próprio para calcular o fatorial de um número menor que o da instância atual.

```
1 Algoritmo: Fatorial(n)
2 se (n=0) ou (n=1) então
3 |  retorna 1;
4 senão
5 |  retorna n× Fatorial(n-1);
```

Algoritmo 3.1: Implementação recursiva do fatorial

Em geral, este tipo de algoritmo resolve o problema através de uma abordagem de "divisão e conquista", composta por em três passos básicos: **divisão**, **conquista** e **combinação** [23]. Neste paradigma, o problema é *dividido* sucessivamente em sub-problemas menores até chegar em sub-problemas elementares de fácil solução, neste ponto o problema é resolvido, ou *conquistadado*. A partir daí, os problemas menores solucionados são *combinados* gerando soluções de problemas maiores até que se chegue à solução do problema original. No caso do fatorial, a divisão acontece na chamada recursiva do cálculo do fatorial de um número menor, os casos elementars são para $n = 0$ e $n = 1$, e a conquista vem na multiplicação do resultado da chamada recursiva com o "n" atual.

[1] É possível também implementar a função fatorial de forma iterativa, porém a implementação recursiva resulta em um código mais aderente à definição recursiva da função.

Outro exemplo de algoritmo baseado em estratégia de divisão e conquista é o método de ordenação *MergeSort*. Neste método, seqüências de n números são divididas recursivamente em seqüências de tamenho $n/2$. O problema é "conquistado", ou resolvido, quando se chega a seqüências de apenas um número que, por definição, já estão ordenadas, constituindo portanto soluções de sub-problemas elementares.

A partir daí as soluções elementares são combinadas no procedimento chamado *merge*. Este procedimento combina dois vetores ordenados. Seu funcionamento pode ser exemplificado pela combinação de cartas de baralho. Suponha que tenhamos duas pilhas de cartas ordenadas e queremos combiná-las para ter apenas uma pilha. Para isso, basta olhar as primeiras cartas das duas pilhas e escolher a menor delas. Esta carta menor deve ser retirada e colocada na pilha resultante com a face que contém o número virada para baixo. Depois disso, o procedimento deve ser repetido com as duas cartas que estão no topo das duas pilhas originais. A figura 3.2 demonstra o funcionamento do *MergeSort* para uma seqüência de oito números.

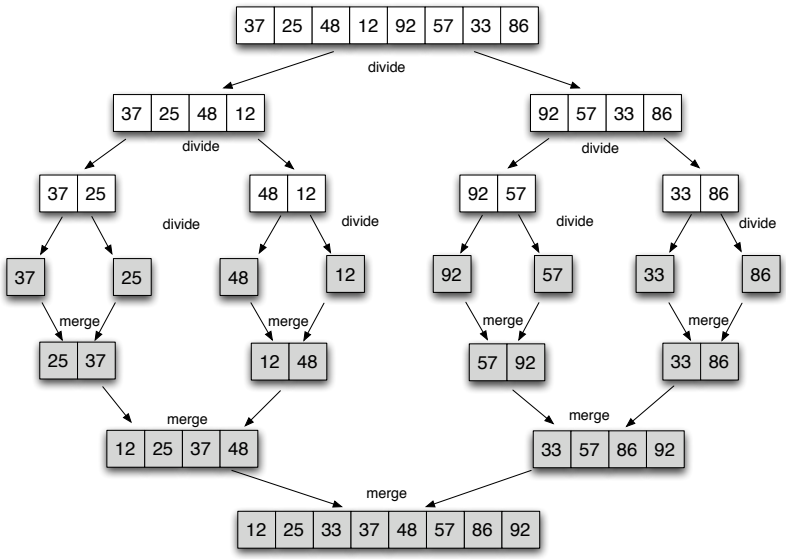

Figura 3.2: Exemplo de execução do algoritmo MergeSort

Conforme pode ser visto no pseudo-código do quadro 3.2, o algoritmo *MergeSort* recebe o vetor a ser ordenado (A) e os íncices do primeiro e do último elemento. Se $ini < fim$, então calcula-se o índice do meio[2] do vetor (ou do sub-vetor) e chama-se recursivamente *MergeSort* para as duas metades do vetor. Caso contráro, tem-se um caso elementar, com vetor contendo zero ou um elemento apenas. Após isso, chama-se o procedimento Merge para combinar dois sub-vetores ordenados.

[2]A expressão $\lfloor x \rfloor$ denota o maior inteiro menor ou igual a x.

```
1  Algoritmo: Mergesort(A, ini, fim)

2  se ini < fim então
3  |  meio ← ⌊(ini + fim)/2⌋;
4  |  MergeSort(A, ini, meio);
5  |  MergeSort(A, meio+1, fim);
6  |  Merge(A, ini, meio, fim);
```

Algoritmo 3.2: MergeSort

O procedimento Merge, cujo pseudocódigo é mostrado no quadro 3.3, combina duas regiões ordenadas do vetor A. Podemos imaginar as duas regiões como as duas pilhas de cartas originais ordenadas, sendo combinadas em uma pilha única. A chamada *Merge(A, ini, meio, fim)* assume que os subvetores $A[ini \ldots meio]$ e $A[(meio + 1) \ldots fim]$ estão ordenados e $ini \leq meio < fim$. O procedimento Merge copia os valores dos dois subvetores para vetores auxiliares ESQ e DIR e os devolve combinados em ordem para o vetor A.

Para evitar que se verifique em cada passo se chegamos ao final de um dos dois vetores auxiliares, é usado um valor "sentinela" igual a ∞. Dessa forma, quando o valor ∞ for alcançado, todos os valores do outro vetor auxiliar possuem valores menores, e são transferidos para o vetor A.

```
1  Algoritmo: Merge(A, ini, meio, fim)

2  n1 ← meio − ini + 1;
3  n2 ← fim − meio;
4  crie vetores auxiliares ESQ[1 ... (n1 + 1)] e DIR[1 ... (n2 + 1)];
5  para i ← 1 até n1 faça
6  |  ESQ[i] ← A[ini + i − 1];
7  para j ← 1 até n2 faça
8  |  DIR[i] ← A[meio + j];
9  ESQ[n1 + 1] ← ∞;
10 DIR[n2 + 1] ← ∞;
11 i ← 1;
12 j ← 1;
13 para k ← ini até fim faça
14 |  se ESQ[i] ≤ DIR[j] então
15 |  |  A[k] ← ESQ[i];
16 |  |  i ← i + 1;
17 |  senão
18 |  |  A[k] ← DIR[j];
19 |  |  j ← j + 1;
```

Algoritmo 3.3: Procedimento Merge

A figura 3.3 ilustra a última execução do procedimento Merge no exemplo

da figura 3.2. Esta execução corresponde à chamada $Merge(A, 1, 4, 8)$. Os quadrados escuros representam os valores finais em suas posições corretas nos vetores.

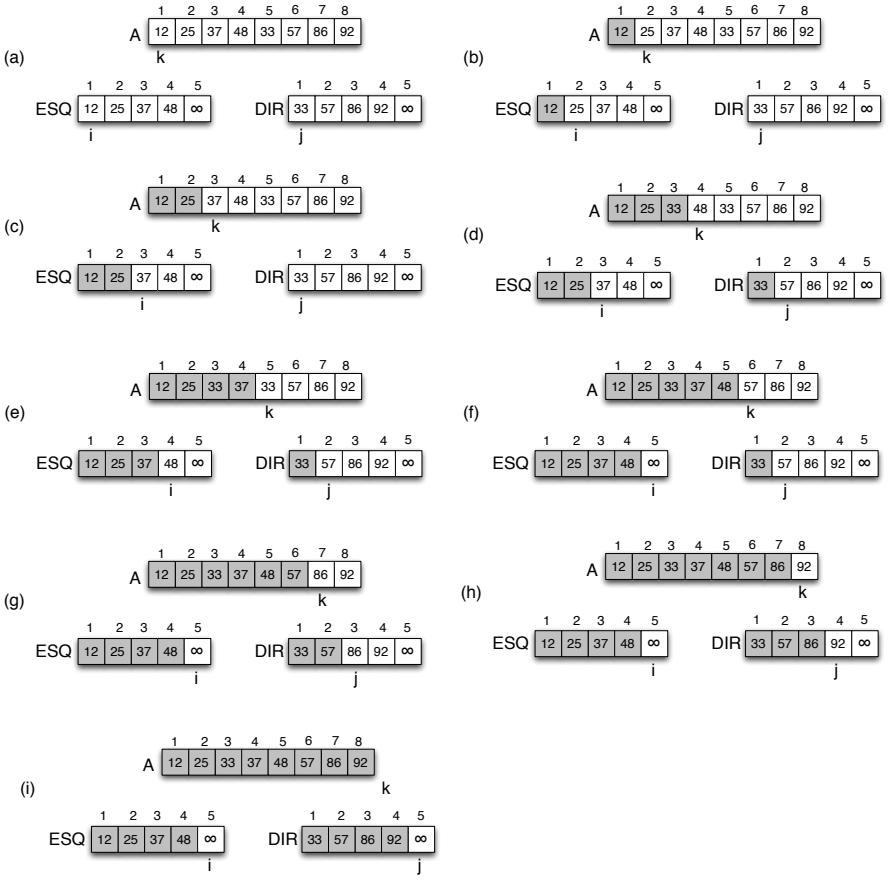

Figura 3.3: Exemplo de execução do procedimento Merge

Observando o pseudocódigo do quadro 3.3, podemos verificar as linhas 2 a 4 e 9 a 12 executam em tempo constante. Os laços "*Para*" das linhas 5 a 8 levam tempo $\Theta(n_1 + n_2) = \Theta(n)$ e o laço "*Para*" das linhas 13 a 19 executa n passos cada um com tempo constante, levando então tempo $\Theta(n)$. Portanto, concluimos que o procedimento Merge roda em tempo $\Theta(n)$, onde $n = fim - ini + 1$.

3.3 Análise de algoritmos recursivos

Para analisar um algoritmo rescursivo, é necessário descrever seu tempo de execução através de uma equação de recorrência, que descreve o tempo de execução de uma problema de tamanho n em função de instâncias de tamanho menor do mesmo problema. A equação de recorrência pode então ser resolvida com auxílio de conceitos matemáticos [21].

A construção da equação de recorrência é feita a partir dos três passos básicos dos algoritmos recursivos (dividir, conquistar e combinar). Portanto, seja $T(n)$ o tempo de execução de um problema de tamanho n. Se o problema é suficientemente pequeno, por exemplo, com $n \leq c$, para determinada constante c, a solução elementar leva tempo constante, que é $\Theta(1)$. Supondo que a divisão do problema leva a a sub-problemas, cada qual de tamanho $1/b$ do tamanho original (por exemplo, para o *MergeSort*, a e b são iguais a 2). Seja $D(n)$ o tempo necessário para dividir o problema e $C(n)$ o tempo para combinar as soluções de sub-problemas na solução do problema original, temos a seguinte equação de recorrência:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n \leq c; \\ aT(n/b) + D(n) + C(n), & \text{caso contrário.} \end{cases} \tag{3.1}$$

Para demonstrar a utilização da equação de recorrência, vamos aplicá-la ao algoritmo *MergeSort* apresentado na seção anterior. O algoritmo funciona para problemas de tamanho ímpar, mas para simplificação da presente análise, vamos considerar que o tamanho do problema é potência de 2.

A **divisão** do problema acontece na linha 3 do algoritmo do quadro 3.2 e consiste apenas no cálculo do índice do meio do sub-vetor, levando portanto tempo constante, ou seja, $D(n) = \Theta(1)$. A **conquista** do problema é feita pela resolução de dois sub-problemas, cada um com tamanho $n/2$, o que acrescenta $2T(n/2)$ ao tempo de execução. A **combinação** de dois sub-problemas em um problema de tamanho n é feita pelo procedimento Merge e leva tempo $C(n) = \Theta(n)$ conforme foi visto na seção anterior. De acordo com as propriedades das combinações de notações assintóticas (seção 2.5), temos que $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$. Substituindo estes termos na equação geral de recorrência, temos a seguinte equação de recorrência para o tempo de execução do algoritmo *MergeSort* no pior caso:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1; \\ 2T(n/2) + \Theta(n), & \text{se } n > 1. \end{cases} \tag{3.2}$$

Após a montagem da equação de recorrência, é preciso resolvê-la para se obter a complexidade do algoritmo recursivo correspondente. Para isso existem três métodos: o método da árvore de recorrência, o método de substituição e o método baseado no teorema geral. A seguir, veremos em maiores detalhes cada um destes métodos.

3.3.1 Método da árvore de recorrência

O método da árvore de recorrência também é chamado de método iterativo pois promove-se a iteração da recorrência para construção da árvore. A idéia central deste método é expandir a árvore e expressar a soma de seus termos em função de n e de certas condições iniciais. Este método é particularmente adequado para análise de complexidade de algoritmos de divisão e conquista.

Para demonstrar o método, vamos tentar resolver a equação de recorrência 3.2, do algoritmo *mergeSort*. Seja a constante c o tempo necessário para resolver problemas de tamanho 1. Então esta equação pode ser reescrita da seguinte forma:

$$T(n) = \begin{cases} c, & \text{se } n = 1; \\ 2T(n/2) + cn, & \text{se } n > 1. \end{cases} \tag{3.3}$$

O desenvolvimento desta equação leva à árvore de recorrência exibida na figura 3.4. Numa árvore de recorrência, cada nó representa o custo de um único sub-problema dentro do conjunto de chamadas recursivas de um determinado procedimento. O custo de cada nível é obtido a partir da soma dos custos de todos os nós do nível, e o custo total é calculado através da soma dos custos de todos os níveis da árvore.

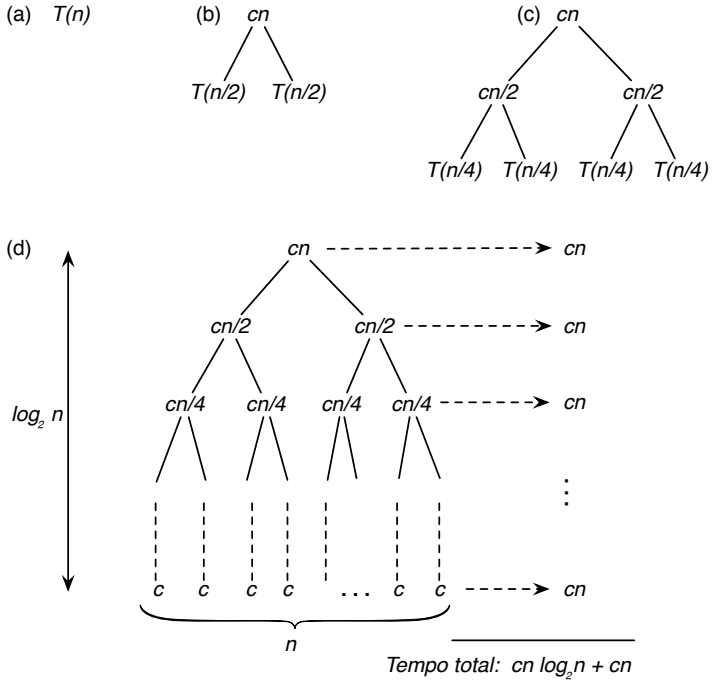

Figura 3.4: Árvore de recorrência para $T(n) = 2T(n/2) + cn$

Inicialmente, a figura 3.4a mostra o tempo total $T(n)$. Em b, temos o início da expansão da árvore, onde a raiz cn representa o tempo da solução no nível mais alto da árvore, e as duas sub-árvores representam as duas recursões, ou seja, os tempos de resolução dos dois sub-problemas. O custo de cada um deles é $cn/2$. Continuando a expansão da árvore, chega-se a n problemas de tamanho 1, com tempo de execução constante igual a c, resultando na árvore mostrada na figura 3.4d.

Adicionando os custos em cada nível da árvore temos um custo cn no primeiro nível. No próximo nível temos $c(n/2) + c(n/2) = cn$. A seguir, temos $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, e assim por diante. No último nível, temos n nós com custo c cada, resultando em cn. O número total de níveis na árvore é $\log_2 n + 1$. Para comprovar isso, basta verificar que quando $n = 1$ a ávore só possui um nível. Como $\log_2 1 = 0$, então o número correto de níveis é dado por $\log_2 n + 1$.

Para computar o custo total representado pela equação 3.3, basta somar os custos de cada nível da árvore. Temos $\log_2 n + 1$ níveis com custo cn cada um. Portanto o custo total é $cn(\log_2 n + 1) = cn \log_2 n + cn$. Desprezando o termo de menor ordem e promovendo a mudança de base do logaritmo, temos que a ordem de complexidade do *MergeSort* é $\Theta(n \log n)$.

Um exemplo menos simples seria o da resolução da equação de recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$, onde vamos tentar encontrar o limite superior para o tempo de execução do algoritmo. Sem perda de generalidade, podemos iniciar criando a árvore de recorrência para a equação $T(n) = 3T(n/4) + cn^2$, sendo o coeficiente constante $c > 0$.

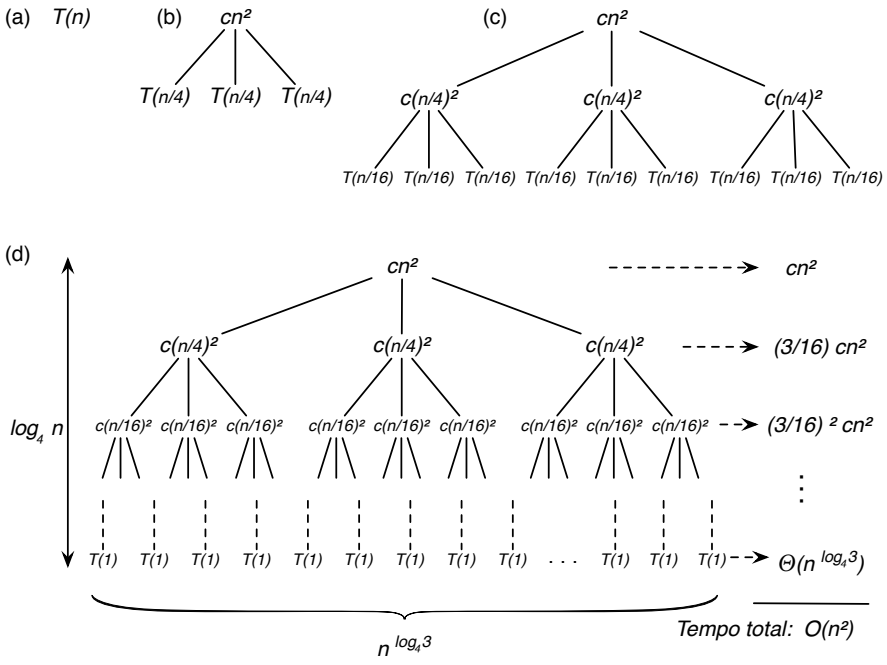

Figura 3.5: Árvore de recorrência para $T(n) = 3T(n/4) + cn^2$

A árvore de recorrência é mostrada na figura 3.5. Para facilitar a análise, é conveniente assumirmos que n é potência de 4. Em 3.5a é mostrado $T(n)$, que sofre uma primeira expansão em 3.5b. O termo cn^2 na raiz da árvore representa o custo da primeira recursão, e suas três sub-árvores representam os custos dos

três sub-problemas de tamanho $n/4$. Na figura 3.5c cada uma destas sub-árvores é expandida. Os nós de custo $T(n/4)$ são expandidos em subárvores com custo $c(n/4)^2$ na raiz e três sub-árvores filhas com custos $T(n/16)$. A expansão da árvore continua conforme descrito na equação de recorrência até que todas as suas partes constituintes sejam estruturadas conforme a figura 3.5d.

A cada nova recorrência, o tamanho do problema diminui. Portanto eventualmente chega-se a uma condição de contorno. Como assumimos que n é potência de 4, as folhas da árvore representam sub-problemas de tamanho $n = 1$. Para cálculo do custo total da árvore é necessário saber a sua profundidade. O tamanho do sub-problema para um nó na profundidade i é igual a $n/4^i$. Sub-problemas de tamanho $n = 1$ estão na profundidade máxima da árvore i_{max}. Então, neste caso $n/4^{i_{max}} = 1$, ou equivalentemente $i_{max} = \log_4 n$. Assim, a árvore possui $\log_4 n + 1$ níveis (0, 1, 2, ..., $\log_4 n$).

Em seguida, é necessário calcular o custo de cada nível da árvore. Um determinado nível possui o triplo do número de nós do nível acima. Então, o i-ésimo nível possui 3^i nós. O custo de cada nó do primeiro ao penúltimo nível é igual a $c(n/4^i)^2$, com $i = 0, 1, 2, \ldots, (\log_4 n - 1)$. Multiplicando o custo de cada nó pela quantidade de nós de cada nível, temos o custo total de cada nível da árvore dado por $3^i c(n/4^i)^2 = (3/16)^i cn^2$.

No nível das folhas, com $i_{max} = \log_4 n$, temos um total de $3^{i_{max}}$ nós. Substituindo o valor de i_{max}, temos $3^{i_{max}} = 3^{\log_4 n} = n^{\log_4 3}$ nós no último nível da árvore[3]. Se cada nó contribui com um custo $T(1) = \Theta(1^2) = \Theta(1)$, então temos um custo total $n^{\log_4 3} T(1) = \Theta(n^{\log_4 3})$ neste nível.

Finalmente, para determinar o custo total da árvore, basta somar os custos de todos os níveis:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \therefore$$

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}). \tag{3.4}$$

A equação acima contém uma série geométrica definida como:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1},$$

o que nos leva a reescrever a equação 3.4 da seguinte forma:

$$T(n) = \frac{(3/16)^{log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}).$$

[3]Lembramos que $a^{log_b c} = c^{log_b a}$.

Esta equação é um tanto difícil de resolvel. Porém, podemos tirar proveito do fato de estarmos fazendo uma análise assintótica da complexidade da equação de recorrência. Assim, podemos considerar o tamanho do problema tendendo a infinito e a série geométrica infinita decrescente (com $|x| < 1$) como um limite superior, podendo ser expressa como:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x},$$

então, temos:

$$T(n) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \therefore$$

$$T(n) < \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \therefore$$

$$T(n) < \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \therefore$$

$$T(n) = O(n^2).$$

O método de resolução pela expansão da árvore de recorrência constitui uma forma intuitiva de avaliação de complexidade de algoritmos recursivos do tipo divisão e conquista. Entretanto, em algus casos a resolução por este método é demasiadamente intrincada ou até mesmo impossível. A seguir veremos o método por substituição, que pode ser uma alternativa nesses casos.

3.3.2 Método de resolução por substituição

O método de resolução por substituição consiste em inicialmente estimar uma solução hipotética para o problema, e depois usar indução matemática para determinar as constantes envolvidas e provar a corretude da solução.

Como exemplo, vamos utilizar este método para provar que a solução da equação de recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ encontrada na seção anterior é correta. Sendo $T(n) = O(n^2)$ um limite superior para complexidade desta recorrência, queremos provar que $T(n) \leq dn^2$ para alguma constante $d > 0$. Usando a mesma constante $c > 0$ da seção anterior, temos:

$$T(n) \leq 3T(\lfloor n/4 \rfloor) + cn^2 \therefore T(n) \leq 3d\lfloor n/4 \rfloor^2 + cn^2$$

$$T(n) \leq 3d(n/4)^2 + cn^2 \therefore T(n) = \frac{3}{16} dn^2 + cn^2 \therefore T(n) \leq dn^2.$$

Onde $T(n) \leq dn^2$ é satisfeito com $d \geq (16/13)c$.

O método por substituição é relativamente mais simples e direto do que o método por expansão da árvore de recorrência, resultando em provas bastante suscintas das equações de recorrência. Entretanto, esbarra na dificuldade de se fazer uma boa estimativa inicial da solução. A árvore de recorrência pode ser uma ferramenta para se fazer estas estimativas, provando-se a corretude da solução por indução.

3.3.3 Método de resolução pelo teorema geral

O método de resolução pelo teorema geral é uma espécie de "receita de bolo" para resoução de equações de recorrência do tipo $T(n) = aT(n/b) + f(n)$, onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva. O problema deve ser enquadrado em um dentre três possíveis casos, feito isso, o teorema geral provê uma solução trivial. Vejamos então o teorema:

Teorema Geral
Sejam as constante a e b onde $a \geq 1$ e $b > 1$. Seja $f(n)$ uma função e seja $T(n)$ definida no domínio dos inteiros não negativos pela equação de recorrência $T(n) = aT(n/b) + f(n)$, onde n/b pode ser interpretado tanto por $\lfloor n/b \rfloor$ quanto por $\lceil n/b \rceil$. Então $T(n)$ pode ser limitada assintoticamente da seguinte forma:

1. se $f(n) = O(n^{\log_b a - \varepsilon})$ para uma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;
2. se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;
3. se $f(n) = \Omega(n^{\log_b a + \varepsilon})$, para uma constante $\varepsilon > 0$ e se $af(n/b) \leq cf(n)$ para uma constante $c < 1$ e para todos n suficientemente grandes, então $T(n) = \Theta(f(n))$.

No três casos enumerados no teorema acima, $f(n)$ é comparada com a função $n^{log_b a}$. A solução da recorrência é dada pela maior das duas funções. No caso 1 a função $n^{log_b a}$ é maior que $f(n)$. No caso 2 são iguais, e no caso 3 a função $f(n)$ é maior. No caso 2, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$. Outro aspecto importante é que no primeiro caso não basta $f(n)$ ser menor que $n^{\log_b a}$, ela tem que ser polinomialmente menor. Ou seja, $f(n)$ precisa ser assintoticamente menor que $n^{\log_b a}$ em um fator n^{ε}. Da mesma forma, no caso 3, $f(n)$ precisa ser polinomialmente maior que $n^{\log_b a}$ para satisfazer a condição $af(n/b) \leq cf(n)$.

O teorema geral não cobre todos os possíveis casos para $f(n)$. Por exemplo, quando $f(n)$ é menor que $n^{log_b a}$, mas não é polinomialmente menor, existe uma lacuna no caso 1 e o teorema não pode ser empregado. Situações similares podem ocorrer nos casos 2 e 3.

Para exemplificar o emprego do teorema, tomemos a equação de recorrência $T(n) = 9T(n/3) + n$. Para esta equação, temos $a = 9$, $b = 3$ e $f(n) = n$. Então $n^{log_b a} = n^{log_3 9} = n^2 = \Theta(n^2)$. Visto que, para a constante $\varepsilon = 1$, temos $f(n) = O(n^{\log_3 9 - \varepsilon}) = O(n^{2-\varepsilon}) = O(n^{2-1}) = O(n)$. Portanto, aplicando o caso 1 do teorema geral, concluímos que a solução da equação de recorrência é $T(n) = \Theta(n^2)$.

Análise de competitividade
Devido às suas características, o modelo computacional RAM não é adequado para algoritmos *online*, e a análise de complexidade vista no presente documento não pode ser aplicada neste caso. O desempenho de tais algoritmos deve ser analisado através da **análise de competitividade** [2]. A idéia de competitividade consiste em comparar a saída gerada por algoritmo *online* co a saída gerada por um algoritmo *offline*. O algoritmo é determinístico e onisciente no sentido de que posui o conhecimento completo de entrada podendo assim computar uma solução ótima. Quanto mais próximo da solução ótima o algoritmo *onine* chegar, maior seu grau de competitividade [13] .

Formalmente, um algoritmo *online* A é apresentado como um conjunto de requisições $s = s_1, s_2, \ldots, s_m$. As requisições s_t, com $t \in [1 \ldots m]$, devem ser atendidas em sua ordem de ocorrência. Mais especificamente, ao atender a requisição s_t, o algoritmo A desconhece qualquer requisição t', onde $t' > t$. Atender uma requisição sempre implica em custo. O objetivo do algoritmo é minimizar o custo total decorrente da seqüência completa de requisições [3].

Na análise de competitividade, o algoritmo A é comparado com um algoritmo *offline* ótimo. Dada uma seqüência de requisições s, seja $C_A(s)$ o custo relativo a A e $C_{OPT}(s)$ o custo relativo a um algoritmo *offline* ótimo OPT. O algoritmo A é chamado ***c*-competitivo** se existe uma constante a tal que $C_A(s) \leq c.C_{OPT}(s) + a$, para todas as seqüências de requisições s. O fator c é chamado de **taxa de competitividade** de A [4]. A análise de competitividade tem recebido forte interesse de pesquisa nos últimos 15 anos, e consitui uma importante técnica para avaliação de desempenho no pior caso para algoritmos *online* [7].

Este capítulo apresentou recursividade no contexto projeto e análise de algoritmos, mostrando a montagem de equações de recorrência utilizadas para análise de complexidade deste tipo de algoritmo através de três métodos diferentes. A seguir são propostos exercícios relativos a estes tópicos.

3.4 Exercícios propostos

1. Utilizando a figura 3.2 como exemplo, mostre a execução do algoritmo *MergeSort* para a seqüência $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2. Verifique se as afirmações a seguir são verdadeiras ou falsas. Justifique sua resposta.

   a) Nenhum algoritmo baseado no princípio de divisão e conquista tem tempo de execução de ordem exponencial. Isto ocorre porque estes algoritmos são recursivos e recursividade evita a explosão combinatória.

   b) Suponha que em determinado algoritmo do tipo divisão e conquista cada etapa de divisão substitui o problema por 16 sub-problemas, cada um com 25% do tamanho do problema corrente. Além disso, o esforço para obter a solução do problema corrente, baseado nas soluções dos 16 sub-problemas menores é constante, independente do tamanho do problema. Neste caso, podemos dizer que a complexidade do algoritmo é $O(n^3)$.

   c) Nenhum algoritmo baseado no princípio de divisão e conquista tem tempo de execução de ordem logarítmica. Isto ocorre porque estes algoritmos são recursivos e recursividade implica sempre num esforço pelo menos linear para a construção das chamadas recursivas.

3. Utilize a árvore de recorrência para determinar um limite superior assintótico para a recorrência $T(n) = 3T(n/2) + n$. Utilize o método de substituição para verificar a resposta.

4. Idem para a recorrência $T(n) = 4T(n/2) + cn$, onde c é uma constante.

5. Utilize o método de substituição para provar que $T(n) = \Theta(n \log n)$ é uma solução correta para a equação de recorrência $T(n) = 2T(n/2) + cn$, do algoritmo *MergeSort*.

6. Utilize o teorema geral para encontrar limites assintóticos Θ para as seguintes equações de recorrência:

   a) $T(n) = T(2n/3) + 1$;

   b) $T(n) = 3T(n/4) + n \log n$;

   c) $T(n) = 4T(n/2) + n$;

   d) $T(n) = 4T(n/2) + n^2$;

   e) $T(n) = 4T(n/2) + n^3$.

7. O algoritmo de busca binária em vetores ordenados possui a seguinte equação de recorrência: $T(n) = T(n/2) + \Theta(1)$. Aplique o teorema geral para provar que a solução desta recorrência é $T(n) = \Theta(\log n)$.

8. Escreva um algoritmo recursivo para o problema de **busca binária**. Construa sua equação de recorrência e prove a complexidade do algoritmo.

Capítulo 4

Considerações finais

O presente trabalho apresentou uma introdução a análise de complexidade de algoritmos. O assunto foi dividido em três capítulos abordando as bases para a análise de complexidade e é dirigido a alunos de graduaçao em Ciência da Computação, Sistemas de Informação ou Engenharias. Em cada capítulo foram apresentados conceitos fundamentais, exemplo e exercícios, juntamente com informações complementares (nos quadros com fundo cinza).

Devido à extensão do tema, este documento aborda apenas tópicos iniciais sobre complexidade de algoritmos. Portanto, sua publicação ainda exige maior aprofundamento, com discussão de conceitos mais avançados, tais como problemas NP-completos.

Referências Bibliográficas

[1] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of computer science*. Computer Science Press, Inc., New York, NY, USA, 1992.

[2] Miklos Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 35th Annual Symp. on Foundations of Computer Science*, pages 401–411, 2003.

[3] Susanne Albers. Competitive online algorithms. *Optima: Mathematical Programming Society Newsletter*, 54:1–7, 1996.

[4] Suzanne Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1–2):3–26, July 2003.

[5] Yossi Azar. On-line load balancing. In *Theoretical Computer Science*, pages 218–225. Springer, 1992.

[6] Sanjoy K. Baruah and Mary Ellen Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Transactions on Computers*, 47:1027–1033, 1996.

[7] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis.* Cambridge: Cambridge University Press. xviii, 414 p., 1998.

[8] Facultad Ciencias, Exactas Naturales, Informe T, Leen Stougie, Esteban Feuerstein, Esteban Feuerstein, Marcelo Mydlarsz, and Marcelo Mydlarsz. On-line multi-threaded scheduling, 1999.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[10] Xiaotie Deng, Tiko Kameda, and Christos Papadimitriou. How to learn an unknown environment. i: the rectilinear case. *J. ACM*, 45(2):215–245, 1998.

[11] Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. *ACM Trans. Algorithms*, 2(3):380–402, 2006.

[12] Bruno Feijó, Paulo César Rodacki Gomes, Joao Bento, Sérgio Scheer, and Renato Cerqueira. Distributed agents supporting event-driven design processes. In John S. Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design 98*, chapter 10. Kluwer Academic Publishers, 1998.

[13] Bruno Feijó, Paulo César Rodacki Gomes, Sérgio Scheer, and João Bento. Online algorithms supporting emergence in distributed cad systems. *Advances in Engineering Software*, 32(10), 2001.

[14] Sandor Fekete, Rolf Klein, and Andreas Nuechter. Searching with an autonomous robot. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 449–450, New York, NY, USA, 2004. ACM.

[15] Peter Fenwick. Burrows–wheeler compression: Principles and reflections. *Theor. Comput. Sci.*, 387(3):200–219, 2007.

[16] Juris Hartmanis. On computational complexity and the nature of computer science. *ACM Comput. Surv.*, 27(1):7–16, 1995.

[17] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. W. H. Freeman & Co., New York, NY, USA, 1978.

[18] Tracy Kimbrel. Online paging and file caching with expiration times. *Theor. Comput. Sci.*, 268(1):119–131, 2001.

[19] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, 1998.

[20] Michael C. Loui. Computational complexity theory. *ACM Comput. Surv.*, 28(1):47–49, 1996.

[21] George S. Lueker. Some techniques for solving recurrences. *ACM Comput. Surv.*, 12(4):419–436, 1980.

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

[23] Edward M. Reingold. Basic techniques for design and analysis of algorithms. *ACM Comput. Surv.*, 28(1):19–21, 1996.

[24] Steven S. Seiden. Randomized online multi-threaded paging. *Nordic J. of Computing*, 6(2):148–161, 1999.

[25] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2008.

[26] Cyrill Stachniss, Óscar Martínez Mozos, and Wolfram Burgard. Efficient exploration of unknown indoor environments using a team of mobile robots. *Annals of Mathematics and Artificial Intelligence*, 52(2-4):205–227, 2008.

[27] Hsien-Wen Tseng and Chin-Chen Chang. Error resilient locally adaptive data compression. *J. Syst. Softw.*, 79(8):1156–1160, 2006.

[28] Toscani L. V. and Veloso P. *Complexidade de Algoritmos*. Série Livros Didáticos - SBC. Editora Sagra Luzzato, 2002.

[29] Bruce Weide. A survey of analysis techniques for discrete algorithms. *ACM Comput. Surv.*, 9(4):291–313, 1977.

[30] Yunfei Yin. A proximate dynamics model for data mining. *Expert Syst. Appl.*, 36(6):9819–9833, 2009.