Capítulo 2

Conceitos básicos

Este capítulo tem o objetivo de familiarizá-lo com a estrutura que usaremos em todo o livro para refletir sobre o projeto e a análise de algoritmos. Ele é autônomo, mas inclui diversas referências ao material que será apresentado nos Capítulos 3 e 4. (E também contém diversos somatórios, que o Apêndice A mostra como resolver.)

Começaremos examinando o problema do algoritmo de ordenação por inserção para resolver o problema de ordenação apresentado no Capítulo 1. Definiremos um "pseudocódigo" que deverá ser familiar aos leitores que tenham estudado programação de computadores, e o empregaremos com a finalidade de mostrar como serão especificados nossos algoritmos. Tendo especificado o algoritmo, demonstraremos então que ele efetua a ordenação corretamente e analisaremos seu tempo de execução. A análise introduzirá uma notação centrada no modo como o tempo aumenta com o número de itens a serem ordenados. Seguindo nossa discussão da ordenação por inserção, introduziremos a abordagem de dividir e conquistar para o projeto de algoritmos e a utilizaremos com a finalidade de desenvolver um algoritmo chamado ordenação por intercalação. Terminaremos com uma análise do tempo de execução da ordenação por intercalação.

2.1 Ordenação por inserção

Nosso primeiro algoritmo, o de ordenação por inserção, resolve o ***problema de ordenação*** introduzido no Capítulo 1:

Entrada: Uma seqüência de n números $\langle a_1, a_2, ..., a_n \rangle$.

Saída: Uma permutação (reordenação) $\langle a'_1, a'_2, ..., a'_n \rangle$ da seqüência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Os números que desejamos ordenar também são conhecidos como ***chaves***.

Neste livro, descreveremos tipicamente algoritmos como programas escritos em um ***pseudocódigo*** muito semelhante em vários aspectos a C, Pascal ou Java. Se já conhece qualquer dessas linguagens, você deverá ter pouca dificuldade para ler nossos algoritmos. O que separa o pseudocódigo do código "real" é que, no pseudocódigo, empregamos qualquer método expressivo para especificar de forma mais clara e concisa um dado algoritmo. Às vezes, o método mais claro é a linguagem comum; assim, não se surpreenda se encontrar uma frase ou sentença em nosso idioma (ou em inglês) embutida no interior de uma seção de código "real". Outra diferen-

ça entre o pseudocódigo e o código real é que o pseudocódigo em geral não se relaciona com questões de engenharia de software. As questões de abstração de dados, modularidade e tratamento de erros são freqüentemente ignoradas, com a finalidade de transmitir a essência do algoritmo de modo mais conciso.

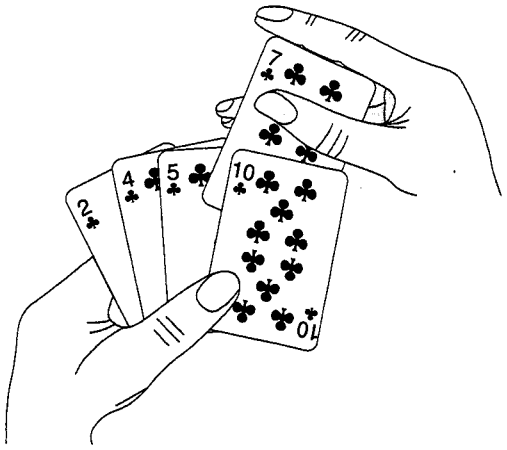

FIGURA 2.1 Ordenando cartas com o uso da ordenação por inserção

Começaremos com a ***ordenação por inserção***, um algoritmo eficiente para ordenar um número pequeno de elementos. A ordenação por inserção funciona da maneira como muitas pessoas ordenam as cartas em um jogo de bridge ou pôquer. Iniciaremos com a mão esquerda vazia e as cartas viradas com a face para baixo na mesa. Em seguida, removeremos uma carta de cada vez da mesa, inserindo-a na posição correta na mão esquerda. Para encontrar a posição correta de uma carta, vamos compará-la a cada uma das cartas que já estão na mão, da direita para a esquerda, como ilustra a Figura 2.1. Em cada instante, as cartas seguras na mão esquerda são ordenadas; essas cartas eram originalmente as cartas superiores da pilha na mesa.

Nosso pseudocódigo para ordenação por inserção é apresentado como um procedimento chamado INSERTION-SORT, que toma como parâmetro um arranjo $A[1 .. n]$ contendo uma seqüência de comprimento n que deverá ser ordenada. (No código, o número n de elementos em A é denotado por *comprimento*$[A]$.) Os números da entrada são ***ordenados no local***: os números são reorganizados dentro do arranjo A, com no máximo um número constante deles armazenado fora do arranjo em qualquer instante. O arranjo de entrada A conterá a seqüência de saída ordenada quando INSERTION-SORT terminar.

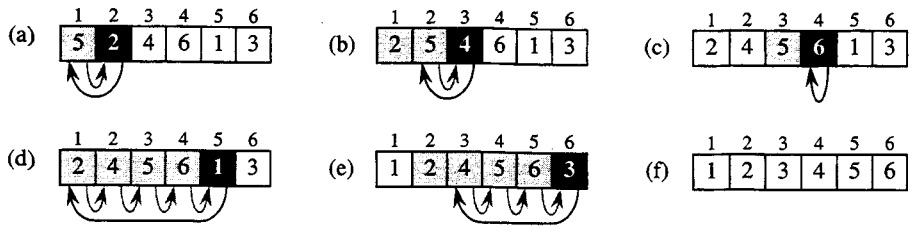

FIGURA 2.2 A operação de INSERTION-SORT sobre o arranjo $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Os índices do arranjo aparecem acima dos retângulos e os valores armazenados nas posições do arranjo aparecem dentro dos retângulos. **(a)**–**(e)** As iterações do loop **for** das linhas 1 a 8. Em cada iteração, o retângulo preto contém a chave obtida de $A[j]$, que é comparada aos valores contidos nos retângulos sombreados à sua esquerda, no teste da linha 5. Setas sombreadas mostram os valores do arranjo deslocados uma posição à direita na linha 6, e setas pretas indicam para onde a chave é deslocada na linha 8. **(f)** O arranjo ordenado final

```
INSERTION-SORT(A)
1 for j ← 2 to comprimento[A]
2     do chave ← A[j]
3        ▷ Inserir A[j] na seqüência ordenada A[1..j − 1].
4        i ← j − 1
5        while i > 0 e A[i] > chave
6           do A[i + 1] ← A[i]
7              i ← i − 1
8        A[i + 1] ← chave
```

Loops invariantes e a correção da ordenação por inserção

A Figura 2.2 mostra como esse algoritmo funciona para $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. O índice j indica a "carta atual" sendo inserida na mão. No início de cada iteração do loop **for** "externo", indexado por j, o subarranjo que consiste nos elementos $A[1 .. j-1]$ constitui a mão atualmente ordenada, e os elementos $A[j + 1 .. n]$ correspondem à pilha de cartas ainda na mesa. Na verdade, os elementos $A[1 .. j-1]$ são os elementos que estavam *originalmente* nas posições de 1 a $j - 1$, mas agora em seqüência ordenada. Enunciamos formalmente essas propriedades de $A[1 .. j-1]$ como um ***loop invariante***:

> No começo de cada iteração do loop ***for*** das linhas 1 a 8, o subarranjo $A[1 .. j-1]$ consiste nos elementos contidos originalmente em $A[1 .. j-1]$, mas em seqüência ordenada.

Usamos loops invariantes para nos ajudar a entender por que um algoritmo é correto. Devemos mostrar três detalhes sobre um loop invariante:

Inicialização: Ele é verdadeiro antes da primeira iteração do loop.

Manutenção: Se for verdadeiro antes de uma iteração do loop, ele permanecerá verdadeiro antes da próxima iteração.

Término: Quando o loop termina, o invariante nos fornece uma propriedade útil que ajuda a mostrar que o algoritmo é correto.

Quando as duas primeiras propriedades são válidas, o loop invariante é verdadeiro antes de toda iteração do loop. Note a semelhança em relação à indução matemática; nesta última, para provar que uma propriedade é válida, você demonstra um caso básico e uma etapa indutiva. Aqui, mostrar que o invariante é válido antes da primeira iteração é equivalente ao caso básico, e mostrar que o invariante é válido de uma iteração para outra equivale à etapa indutiva.

A terceira propriedade talvez seja a mais importante, pois estamos usando o loop invariante para mostrar a correção. Ela também difere do uso habitual da indução matemática, em que a etapa indutiva é usada indefinidamente; aqui, paramos a "indução" quando o loop termina.

Vamos ver como essas propriedades são válidas para ordenação por inserção:

Inicialização: Começamos mostrando que o loop invariante é válido antes da primeira iteração do loop, quando $j = 2$.[1] Então, o subarranjo $A[1 .. j-1]$ consiste apenas no único elemento $A[1]$, que é de fato o elemento original em $A[1]$. Além disso, esse subarranjo é ordenado (de forma trivial, é claro), e isso mostra que o loop invariante é válido antes da primeira iteração do loop.

---

[1] Quando o loop é um loop **for**, o momento em que verificamos o loop invariante imediatamente antes da primeira iteração ocorre logo após a atribuição inicial à variável do contador de loop e imediatamente antes do primeiro teste no cabeçalho do loop. No caso de INSERTION-SORT, esse instante ocorre após a atribuição de 2 à variável j, mas antes do primeiro teste para verificar se $j \leq comprimento[A]$.

Manutenção: Em seguida, examinamos a segunda propriedade: a demonstração de que cada iteração mantém o loop invariante. Informalmente, o corpo do loop **for** exterior funciona deslocando-se $A[j-1], A[j-2], A[j-3]$ e daí por diante uma posição à direita, até ser encontrada a posição adequada para $A[j]$ (linhas 4 a 7), e nesse ponto o valor de $A[j]$ é inserido (linha 8). Um tratamento mais formal da segunda propriedade nos obrigaria a estabelecer e mostrar um loop invariante para o loop **while** "interno". Porém, nesse momento, preferimos não nos prender a tal formalismo, e assim contamos com nossa análise informal para mostrar que a segunda propriedade é válida para o loop exterior.

Término: Finalmente, examinamos o que ocorre quando o loop termina. No caso da ordenação por inserção, o loop **for** externo termina quando j excede n, isto é, quando $j = n + 1$. Substituindo j por $n + 1$ no enunciado do loop invariante, temos que o subarranjo $A[1 .. n]$ consiste nos elementos originalmente contidos em $A[1 .. n]$, mas em seqüência ordenada. Contudo, o subarranjo $A[1 .. n]$ é o arranjo inteiro! Desse modo, o arranjo inteiro é ordenado, o que significa que o algoritmo é correto.

Empregaremos esse método de loops invariantes para mostrar a correção mais adiante neste capítulo e também em outros capítulos.

Convenções de pseudocódigo

Utilizaremos as convenções a seguir em nosso pseudocódigo.

1. O recuo (ou endentação) indica uma estrutura de blocos. Por exemplo, o corpo do loop **for** que começa na linha 1 consiste nas linhas 2 a 8, e o corpo do loop **while*** que começa na linha 5 contém as linhas 6 e 7, mas não a linha 8. Nosso estilo de recuo também se aplica a instruções **if-then-else**. O uso de recuo em lugar de indicadores convencionais de estrutura de blocos, como instruções **begin** e **end**, reduz bastante a desordem ao mesmo tempo que preserva, ou até mesmo aumenta, a clareza.[2]
2. As construções de loops **while**, **for** e **repeat** e as construções condicionais **if**, **then** e **else** têm interpretações semelhantes às que apresentam em Pascal.[3] Porém, existe uma diferença sutil com respeito a loops **for**: em Pascal, o valor da variável do contador de loop é indefinido na saída do loop mas, neste livro, o contador do loop retém seu valor após a saída do loop. Desse modo, logo depois de um loop **for**, o valor do contador de loop é o valor que primeiro excedeu o limite do loop **for**. Usamos essa propriedade em nosso argumento de correção para a ordenação por inserção. O cabeçalho do loop **for** na linha 1 é **for** $j \leftarrow 2$ **to** *comprimento*$[A]$, e assim, quando esse loop termina, $j =$ *comprimento*$[A] + 1$ (ou, de forma equivalente, $j = n + 1$, pois $n =$ *comprimento*$[A]$).
3. O símbolo "▷" indica que o restante da linha é um comentário.
4. Uma atribuição múltipla da forma $i \leftarrow j \leftarrow e$ atribui às variáveis i e j o valor da expressão e; ela deve ser tratada como equivalente à atribuição $j \leftarrow e$ seguida pela atribuição $i \leftarrow j$.
5. Variáveis (como i, j e *chave*) são locais para o procedimento dado. Não usaremos variáveis globais sem indicação explícita.

---

[2] Em linguagens de programação reais, em geral não é aconselhável usar o recuo sozinho para indicar a estrutura de blocos, pois os níveis de recuo são difíceis de descobrir quando o código se estende por várias páginas.

[3] A maioria das linguagens estruturadas em blocos tem construções equivalentes, embora a sintaxe exata possa diferir da sintaxe de Pascal.

* Manteremos na edição brasileira os nomes das instruções e dos comandos de programação (destacados em negrito) em inglês, bem como os títulos dos algoritmos, conforme a edição original americana, a fim de facilitar o processo de conversão para uma linguagem de programação qualquer, caso necessário. Por exemplo, usaremos **while** em vez de **enquanto**. (N.T.)

6. Elementos de arranjos são acessados especificando-se o nome do arranjo seguido pelo índice entre colchetes. Por exemplo, $A[i]$ indica o i-ésimo elemento do arranjo A. A notação ".." é usada para indicar um intervalo de valores dentro de um arranjo. Desse modo, $A[1\ ..\ j]$ indica o subarranjo de A que consiste nos j elementos $A[1], A[2], ..., A[j]$.

7. Dados compostos estão organizados tipicamente em ***objetos***, os quais são constituídos por ***atributos*** ou ***campos***. Um determinado campo é acessado usando-se o nome do campo seguido pelo nome de seu objeto entre colchetes. Por exemplo, tratamos um arranjo como um objeto com o atributo *comprimento* indicando quantos elementos ele contém. Para especificar o número de elementos em um arranjo A, escrevemos $comprimento[A]$. Embora sejam utilizados colchetes para indexação de arranjos e atributos de objetos, normalmente ficará claro a partir do contexto qual a interpretação pretendida.

   Uma variável que representa um arranjo ou um objeto é tratada como um ponteiro para os dados que representam o arranjo ou objeto. Para todos os campos f de um objeto x, a definição de $y \leftarrow x$ causa $f[y] = f[x]$. Além disso, se definirmos agora $f[x] \leftarrow 3$, então daí em diante não apenas $f[x] = 3$, mas também $f[y] = 3$. Em outras palavras, x e y apontarão para ("serão") o mesmo objeto após a atribuição $y \leftarrow x$.

   Às vezes, um ponteiro não fará referência a nenhum objeto. Nesse caso, daremos a ele o valor especial NIL.

8. Parâmetros são passados a um procedimento ***por valor***: o procedimento chamado recebe sua própria cópia dos parâmetros e, se ele atribuir um valor a um parâmetro, a mudança *não* será vista pela rotina de chamada. Quando objetos são passados, o ponteiro para os dados que representam o objeto é copiado, mas os campos do objeto não o são. Por exemplo, se x é um parâmetro de um procedimento chamado, a atribuição $x \leftarrow y$ dentro do procedimento chamado não será visível para o procedimento de chamada. Contudo, a atribuição $f[x] \leftarrow 3$ será visível.

9. Os operadores booleanos "e" e "ou" são operadores de ***curto-circuito***. Isto é, quando avaliamos a expressão "x e y", avaliamos primeiro x. Se x for avaliado como FALSE, então a expressão inteira não poderá ser avaliada como TRUE, e assim não avaliaremos y. Se, por outro lado, x for avaliado como TRUE, teremos de avaliar y para determinar o valor da expressão inteira. De forma semelhante, na expressão "x ou y", avaliamos a expressão y somente se x for avaliado como FALSE. Os operadores de curto-circuito nos permitem escrever expressões booleanas como "x ... NIL e $f[x] = y$" sem nos preocuparmos com o que acontece ao tentarmos avaliar $f[x]$ quando x é NIL.

Exercícios

2.1-1
Usando a Figura 2.2 como modelo, ilustre a operação de INSERTION-SORT no arranjo $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2
Reescreva o procedimento INSERTION-SORT para ordenar em ordem não crescente, em vez da ordem não decrescente.

2.1-3
Considere o ***problema de pesquisa***:

Entrada: Uma seqüência de n números $A = \langle a_1, a_2, ..., a_n \rangle$ e um valor v.

Saída: Um índice i tal que $v = A[i]$ ou o valor especial NIL, se v não aparecer em A.

Escreva o pseudocódigo para ***pesquisa linear***, que faça a varredura da seqüência, procurando por v. Usando um loop invariante, prove que seu algoritmo é correto. Certifique-se de que seu loop invariante satisfaz às três propriedades necessárias.

2.1-4

Considere o problema de somar dois inteiros binários de n bits, armazenados em dois arranjos de n elementos A e B. A soma dos dois inteiros deve ser armazenada em forma binária em um arranjo de $(n + 1)$ elementos C. Enuncie o problema de modo formal e escreva o pseudocódigo para somar os dois inteiros.

2.2 Análise de algoritmos

Analisar um algoritmo significa prever os recursos de que o algoritmo necessitará. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, mas com freqüência é o tempo de computação que desejamos medir. Em geral, pela análise de vários algoritmos candidatos para um problema, pode-se identificar facilmente um algoritmo mais eficiente. Essa análise pode indicar mais de um candidato viável, mas vários algoritmos de qualidade inferior em geral são descartados no processo.

Antes de podermos analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo dos recursos dessa tecnologia e seus custos. Na maior parte deste livro, faremos a suposição de um modelo de computação genérico com um único processador, a ***RAM (random-access machine*** – máquina de acesso aleatório), como nossa tecnologia de implementação e entenderemos que nossos algoritmos serão implementados sob a forma de programas de computador. No modelo de RAM, as instruções são executadas uma após outra, sem operações concorrentes (ou simultâneas). Porém, em capítulos posteriores teremos oportunidade de investigar modelos de hardware digital.

No sentido estrito, devemos definir com precisão as instruções do modelo de RAM e seus custos. Porém, isso seria tedioso e daria pouco percepção do projeto e da análise de algoritmos. Também devemos ter cuidado para não abusar do modelo de RAM. Por exemplo, e se uma RAM tivesse uma instrução de ordenação? Então, poderíamos ordenar com apenas uma instrução. Tal RAM seria irreal, pois os computadores reais não têm tais instruções. Portanto, nosso guia é o modo como os computadores reais são projetados. O modelo de RAM contém instruções comumente encontradas em computadores reais: instruções aritméticas (soma, subtração, multiplicação, divisão, resto, piso, teto), de movimentação de dados (carregar, armazenar, copiar) e de controle (desvio condicional e incondicional, chamada e retorno de sub-rotinas). Cada uma dessas instruções demora um período constante.

Os tipos de dados no modelo de RAM são inteiros e de ponto flutuante. Embora normalmente não nos preocupemos com a precisão neste livro, em algumas aplicações a precisão é crucial. Também supomos um limite sobre o tamanho de cada palavra de dados. Por exemplo, ao trabalharmos com entradas de tamanho n, em geral supomos que os inteiros são representados por $c \lg n$ bits para alguma constante $c \geq 1$. Exigimos $c \geq 1$ para que cada palavra possa conter o valor de n, permitindo-nos indexar os elementos de entradas individuais, e limitamos c a uma constante para que o tamanho da palavra não cresça arbitrariamente. (Se o tamanho da palavra pudesse crescer arbitrariamente, seria possível armazenar enormes quantidades de dados em uma única palavra e operar sobre toda ela em tempo constante – claramente um cenário impraticável.)

Computadores reais contêm instruções não listadas anteriormente, e tais instruções representam uma área cinza no modelo de RAM. Por exemplo, a exponenciação é uma instrução de tempo constante? No caso geral, não; são necessárias várias instruções para calcular x^y quando x e y são números reais. Porém, em situações restritas, a exponenciação é uma operação de tempo constante. Muitos computadores têm uma instrução "deslocar à esquerda" que desloca em tempo constante os bits de um inteiro k posições à esquerda. Na maioria dos computadores, deslocar os bits de um inteiro uma posição à esquerda é equivalente a efetuar a multiplicação por 2.

Deslocar os bits k posições à esquerda é equivalente a multiplicar por 2^k. Portanto, tais computadores podem calcular 2^k em uma única instrução de tempo constante, deslocando o inteiro 1 k posições à esquerda, desde que k não seja maior que o número de bits em uma palavra de computador. Procuraremos evitar essas áreas cinza no modelo de RAM, mas trataremos a computação de 2^k como uma operação de tempo constante quando k for um inteiro positivo suficientemente pequeno.

No modelo de RAM, não tentaremos modelar a hierarquia da memória que é comum em computadores contemporâneos. Isto é, não modelaremos caches ou memória virtual (que é implementada com maior freqüência com paginação por demanda). Vários modelos computacionais tentam levar em conta os efeitos da hierarquia de memória, que às vezes são significativos em programas reais de máquinas reais. Alguns problemas neste livro examinam os efeitos da hierarquia de memória mas, em sua maioria, as análises neste livro não irão considerá-los.

Os modelos que incluem a hierarquia de memória são bem mais complexos que o modelo de RAM, de forma que pode ser difícil utilizá-los. Além disso, as análises do modelo de RAM em geral permitem previsões excelentes do desempenho em máquinas reais.

Até mesmo a análise de um algoritmo simples no modelo de RAM pode ser um desafio. As ferramentas matemáticas exigidas podem incluir análise combinatória, teoria das probabilidades, destreza em álgebra e a capacidade de identificar os termos mais significativos em uma fórmula. Tendo em vista que o comportamento de um algoritmo pode ser diferente para cada entrada possível, precisamos de um meio para resumir esse comportamento em fórmulas simples, de fácil compreensão.

Embora normalmente selecionemos apenas um único modelo de máquina para analisar um determinado algoritmo, ainda estaremos diante de muitas opções na hora de decidir como expressar nossa análise. Um objetivo imediato é encontrar um meio de expressão que seja simples de escrever e manipular, que mostre as características importantes de requisitos de recurso de um algoritmo e que suprima os detalhes tediosos.

Análise da ordenação por inserção

O tempo despendido pelo procedimento INSERTION-SORT depende da entrada: a ordenação de mil números demora mais que a ordenação de três números. Além disso, INSERTION-SORT pode demorar períodos diferentes para ordenar duas seqüências de entrada do mesmo tamanho, dependendo do quanto elas já estejam ordenadas. Em geral, o tempo de duração de um algoritmo cresce com o tamanho da entrada; assim, é tradicional descrever o tempo de execução de um programa como uma função do tamanho de sua entrada. Para isso, precisamos definir os termos "tempo de execução" e "tamanho da entrada" com mais cuidado.

A melhor noção de ***tamanho da entrada*** depende do problema que está sendo estudado. No caso de muitos problemas, como a ordenação ou o cálculo de transformações discretas de Fourier, a medida mais natural é o *número de itens na entrada* – por exemplo, o tamanho do arranjo n para ordenação. Para muitos outros problemas, como a multiplicação de dois inteiros, a melhor medida do tamanho da entrada é o *número total de bits* necessários para representar a entrada em notação binária comum. Às vezes, é mais apropriado descrever o tamanho da entrada com dois números em lugar de um. Por exemplo, se a entrada para um algoritmo é um grafo, o tamanho da entrada pode ser descrito pelos números de vértices e arestas no grafo. Indicaremos qual medida de tamanho da entrada está sendo usada com cada problema que estudarmos.

O ***tempo de execução*** de um algoritmo em uma determinada entrada é o número de operações primitivas ou "etapas" executadas. É conveniente definir a noção de etapa (ou passo) de forma que ela seja tão independente da máquina quanto possível. Por enquanto, vamos adotar a visão a seguir. Um período constante de tempo é exigido para executar cada linha do nosso pseudocódigo. Uma única linha pode demorar um período diferente de outra linha, mas vamos considerar que cada execução da i-ésima linha leva um tempo c_i, onde c_i é uma constante. Esse pon-

to de vista está de acordo com o modelo de RAM, e também reflete o modo como o pseudocódigo seria implementado na maioria dos computadores reais.[4]

Na discussão a seguir, nossa expressão para o tempo de execução de INSERTION-SORT evoluirá desde uma fórmula confusa que utiliza todos os custos da instrução c_i até uma notação mais simples, mais concisa e mais facilmente manipulada. Essa notação mais simples também facilitará a tarefa de descobrir se um algoritmo é mais eficiente que outro.

Começaremos apresentando o procedimento INSERTION-SORT com o "custo" de tempo de cada instrução e o número de vezes que cada instrução é executada. Para cada $j = 2, 3, ..., n$, onde $n = comprimento[A]$, seja t_j o número de vezes que o teste do loop **while** na linha 5 é executado para esse valor de j. Quando um loop **for** ou **while** termina da maneira usual (isto é, devido ao teste no cabeçalho loop), o teste é executado uma vez além do corpo do loop. Supomos que comentários não são instruções executáveis e, portanto, não demandam nenhum tempo.

INSERTION-SORT(A)	*custo*	*vezes*
1 **for** $j \leftarrow 2$ **to** *comprimento*$[A]$	c_1	n
2 **do** *chave* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Inserir $A[j]$ na seqüência ordenada $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] >$ *chave*	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow$ *chave*	c_8	$n-1$

O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução executada; uma instrução que demanda c_i passos para ser executada e é executada n vezes, contribuirá com $c_i n$ para o tempo de execução total.[5] Para calcular $T(n)$, o tempo de execução de INSERTION-SORT, somamos os produtos das *colunas custo* e *vezes*, obtendo

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) - c_8(n-1).$$

Mesmo para entradas de um dado tamanho, o tempo de execução de um algoritmo pode depender de *qual* entrada desse tamanho é dada. Por exemplo, em INSERTION-SORT, o melhor caso ocorre se o arranjo já está ordenado. Para cada $j = 2, 3, ..., n$, descobrimos então que $A[i] \leq$ *chave* na linha 5 quando i tem seu valor inicial $j-1$. Portanto, $t_j = 1$ para $j = 2, 3, ..., n$, e o tempo de execução do melhor caso é

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

---

[4]Há algumas sutilezas aqui. As etapas computacionais que especificamos em linguagem comum freqüentemente são variantes de um procedimento que exige mais que apenas uma quantidade constante de tempo. Por exemplo, mais adiante neste livro, poderíamos dizer "ordene os pontos pela coordenada x" que, como veremos, demora mais que uma quantidade constante de tempo. Além disso, observe que uma instrução que chama uma sub-rotina demora um tempo constante, embora a sub-rotina, uma vez invocada, possa durar mais. Ou seja, separamos o processo de ***chamar*** a sub-rotina – passar parâmetros a ela etc. – do processo de ***executar*** a sub-rotina.

[5]Essa característica não se mantém necessariamente para um recurso como a memória. Uma instrução que referencia m palavras de memória e é executada n vezes não consome necessariamente mn palavras de memória no total.

Esse tempo de execução pode ser expresso como $an + b$ para *constantes* a e b que dependem dos custos de instrução c_i; assim, ele é uma ***função linear*** de n.

Se o arranjo estiver ordenado em ordem inversa – ou seja, em ordem decrescente –, resulta o pior caso. Devemos comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado inteiro, $A[1..j-1]$, e então $t_j = j$ para 2, 3, ..., n. Observando que

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

e

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

(veremos no Apêndice A como resolver esses somatórios), descobrimos que, no pior caso, o tempo de execução de INSERTION-SORT é

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Esse tempo de execução no pior caso pode ser expresso como $an^2 + bn + c$ para constantes a, b e c que, mais uma vez, dependem dos custos de instrução c_i; portanto, ele é uma ***função quadrática*** de n.

Em geral, como na ordenação por inserção, o tempo de execução de um algoritmo é fixo para uma determinada entrada, embora em capítulos posteriores devamos ver alguns algoritmos "aleatórios" interessantes, cujo comportamento pode variar até mesmo para uma entrada fixa.

Análise do pior caso e do caso médio

Em nossa análise da ordenação por inserção, observamos tanto o melhor caso, no qual o arranjo de entrada já estava ordenado, quanto o pior caso, no qual o arranjo de entrada estava ordenado em ordem inversa. Porém, no restante deste livro, em geral nos concentraremos apenas na descoberta do ***tempo de execução do pior caso***; ou seja, o tempo de execução mais longo para *qualquer* entrada de tamanho n. Apresentaremos três razões para essa orientação.

- O tempo de execução do pior caso de um algoritmo é um limite superior sobre o tempo de execução para qualquer entrada. Conhecê-lo nos dá uma garantia de que o algoritmo nunca irá demorar mais tempo. Não precisamos fazer nenhuma suposição baseada em fatos sobre o tempo de execução, e temos a esperança de que ele nunca seja muito pior.
- Para alguns algoritmos, o pior caso ocorre com bastante freqüência. Por exemplo, na pesquisa de um banco de dados em busca de um determinado fragmento de informação, o pior caso do algoritmo de pesquisa ocorrerá freqüentemente quando a informação não estiver presente no banco de dados. Em algumas aplicações de pesquisa, a busca de informações ausentes pode ser freqüente.