2 DANDO A PARTIDA

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 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 de ordenação por inserção, demonstraremos que ele efetua a ordenação corretamente e analisaremos seu tempo de execução. A análise introduzirá uma notação que focaliza o modo como o tempo aumenta com o número de itens a ordenar. Seguindo nossa discussão da ordenação por inserção, introduziremos a abordagem de divisão e conquista para o projeto de algoritmos e a utilizaremos para desenvolver um algoritmo denominado 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*** apresentado no Capítulo 1:

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

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

Os números que desejamos ordenar também são conhecidos como ***chaves***. Embora conceitualmente estejamos ordenando uma sequência, a entrada é dada na forma de um arranjo com n elementos.

Neste livro, descreveremos tipicamente algoritmos como programas escritos em um ***pseudocódigo*** semelhante em vários aspectos a C, C++, Java, Python ou Pascal. Se você já conhece qualquer dessas linguagens, deverá ter pouca dificuldade para ler nossos algoritmos. O que distingue o pseudocódigo do código "real" é que, no pseudocódigo, empregamos qualquer método expressivo que seja mais claro e conciso para especificar 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 em uma seção de código "real". Outra diferença entre o pseudocódigo e o código real é que o pseudocódigo em geral não se preocupa com questões de engenharia de software. As questões de abstração de dados, modularidade e tratamento de erros são frequentemente ignoradas, de modo a transmitir a essência do algoritmo de modo mais conciso.

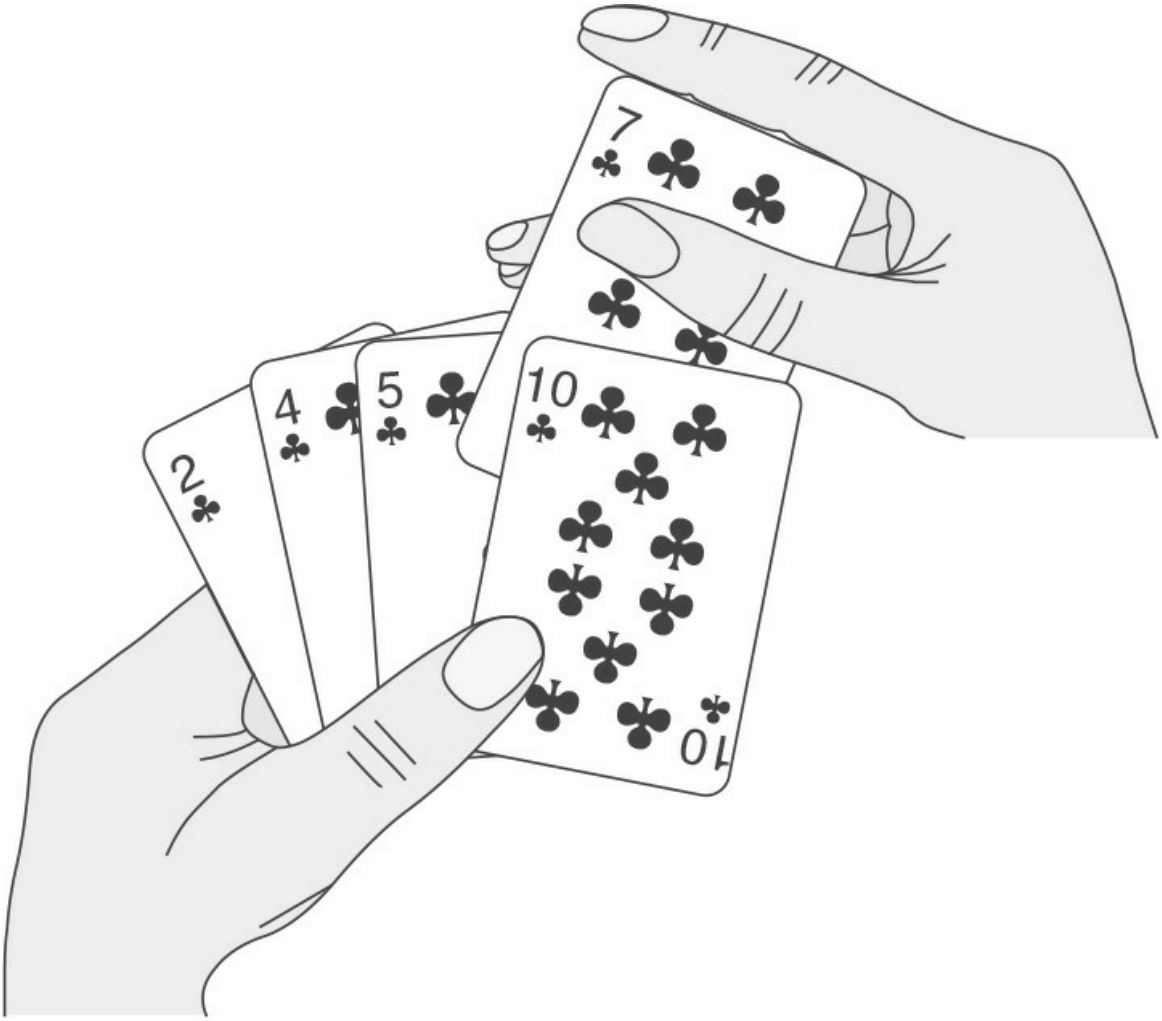

Figura 2.1 Ordenando cartas com o uso da ordenação por inserção.

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 baralho. Iniciamos com a mão esquerda vazia e as cartas viradas para baixo, na mesa. Em seguida, retiramos uma carta de cada vez da mesa e a inserimos na posição correta na mão esquerda. Para encontrar a posição correta para uma carta, nós a comparamos com cada uma das cartas que já estão na mão, da direita para a esquerda, como ilustra a Figura 2.1. Em todas as vezes, as cartas que seguramos na mão esquerda são ordenadas, e essas cartas eram as que estavam na parte superior da pilha sobre a mesa.

Nosso pseudocódigo para ordenação por inserção é apresentado como um procedimento denominado Insertion-Sort, que toma como parâmetro um arranjo $A[1 .. n]$ contendo uma sequência de comprimento n que deverá ser ordenada. (No código, o número n de elementos em A é denotado por $A \cdot comprimento$.) O algoritmo ordena os números da entrada **no lugar:** reorganiza os números 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 sequência de saída ordenada quando Insertion-Sort terminar.

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

Invariantes de laço 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" que está sendo inserida na mão. No início de cada iteração do laço **for**, indexado por j, o subarranjo que consiste nos elementos $A[1 .. j - 1]$ constitui a mão ordenada atualmente e o subconjunto remanescente $A[j + 1 .. n]$ corresponde à pilha de cartas que ainda está sobre a mesa. Na verdade, os elementos $A[1 .. j - 1]$ são os que estavam *originalmente* nas posições 1 a $j - 1$, mas agora em sequência ordenada. Afirmamos essas propriedades de $A[1 .. j - 1]$ formalmente como um de *invariante de laço*:

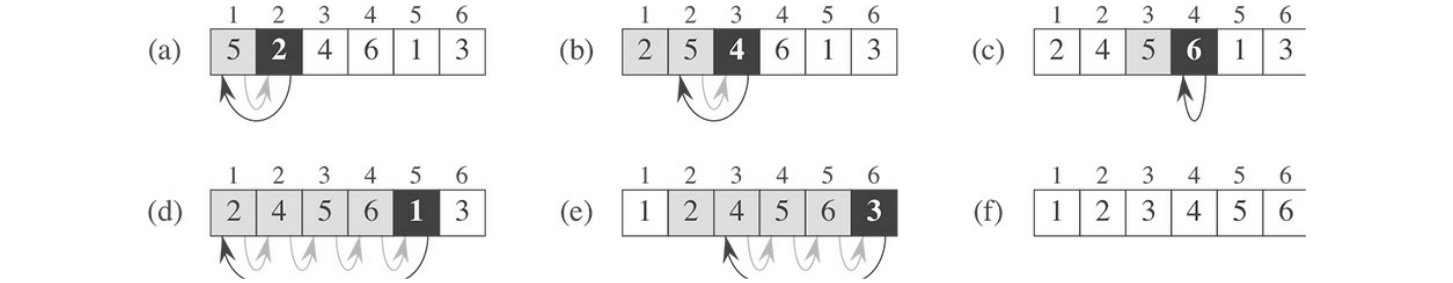

Figura 2.2 A operação de INSERTION-SORT sobre o arranjo $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Os índices do arranjo aparecem acima dos retângulos, e os valores armazenados nas posições do arranjo aparecem dentro dos retângulos. **(a)**–**(e)** Iterações do laço **for** das linhas 1 a 8. Em cada iteração,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

No início de cada iteração para o laço **for** das linhas 1–8, o subarranjo $A[1 .. j - 1]$ consiste nos elementos que estavam originalmente em $A[1 .. j - 1]$, porém em sequência ordenada.

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

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

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

Término: Quando o laço 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 invariante de laço é verdadeiro antes de toda iteração do laço. (É claro que temos a liberdade de usar fatos confirmados além do invariante de laço em si para provar que ele permanece verdadeiro antes de cada iteração.) Observe a semelhança com a indução; nesta, para provar que uma

propriedade é válida, provamos uma base e um passo de indução. Aqui, mostrar que o invariante é válido antes da primeira iteração equivale à base, e mostrar que o invariante é válido de uma iteração para outra equivale ao passo.

A terceira propriedade talvez seja a mais importante, visto que estamos usando o invariante de laço para mostrar a correção. Normalmente, usamos o invariante de laço juntamente com a condição que provocou o término do laço. O modo de utilização da propriedade de término é diferente do modo de utilização da indução: nesta, a etapa indutiva é aplicada indefinidamente; aqui, paramos a "indução" quando o laço termina.

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

Inicialização: Começamos mostrando que o invariante de laço é válido antes da primeira iteração do laço, 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 (trivialmente, é claro), e isso mostra que o invariante de laço é válido antes da primeira iteração do laço.

Manutenção: Em seguida, abordamos a segunda propriedade: mostrar que cada iteração mantém o invariante de laço. Informalmente, o corpo do laço **for** funciona deslocando $A[j - 1]$, $A[j - 2]$, $A[j - 3]$, e assim por diante, uma posição para a direita até encontrar a posição adequada para $A[j]$ (linhas 4 a 7); nesse ponto ele insere o valor de $A[j]$ (linha 8). Então, o subarranjo $A[1 .. j]$ consiste nos elementos presentes originalmente em $A[1 .. j]$, mas em sequência ordenada. Portanto, incrementar j para a próxima iteração do laço **for** preserva o invariante de laço.

Um tratamento mais formal da segunda propriedade nos obrigaria a estabelecer e mostrar um invariante para o laço **while** das linhas 5–7. 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 laço externo.

Término: Finalmente, examinamos o que ocorre quando o laço termina. A condição que provoca o término do laço **for** é que $j > A \cdot comprimento = n$. Como cada iteração do laço aumenta j de 1, devemos ter $j = n + 1$ nesse instante. Substituindo j por $n + 1$ no enunciado do invariante de laço, temos que o subarranjo $A[1 .. n]$ consiste nos elementos originalmente contidos em $A[1 .. n]$, mas em sequência ordenada. Observando que o subarranjo $A[1 .. n]$ é o arranjo inteiro, concluímos que o arranjo inteiro está ordenado. Portanto o algoritmo está correto.

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

Convenções de pseudocódigo

Utilizaremos as convenções a seguir em nosso pseudocódigo.

- O recuo indica estrutura de bloco. Por exemplo, o corpo do laço for que começa na linha 1 consiste nas linhas 2 a 8, e o corpo do laço **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-else**.[2] O uso de recuo em vez de indicadores convencionais de estrutura de bloco, como instruções **begin** e **end**, reduz bastante a confusão, ao mesmo tempo que preserva ou até mesmo aumenta a clareza.[3]
- As interpretações das construções de laço **while**, **for** e **repeat-until** e das construções condicionais **if-else** são semelhantes às das linguagens C, C++, Java, Python e Pascal.[4] Neste livro, o contador do laço mantém seu valor após sair do laço, ao contrário de algumas situações que surgem em C++, Java e Pascal. Desse modo, logo depois de um laço **for**, o valor do contador de laço é o valor que primeiro excedeu o limite do laço **for**. Usamos essa propriedade em nosso argumento de correção para a ordenação por inserção. O cabeçalho do laço **for** na linha 1 é **for** $j = 2$ **to** $A \cdot comprimento$ e, assim, quando esse laço termina, $j = A \cdot comprimento + 1$ (ou, o que é equivalente, $j = n + 1$, visto que $n = A \cdot comprimento$).

  Usamos a palavra-chave **to** quando um laço **for** incrementa seu contador do laço a cada iteração, e usamos a palavra-chave **downto** quando um laço **for** decrementa seu contador de laço. Quando o contador do laço mudar

por uma quantidade maior do que 1, essa quantidade virá após a palavra-chave opcional **by**.

- O símbolo "//" indica que o restante da linha é um comentário.
- Uma atribuição múltipla da forma $i = j = e$ atribui às variáveis i e j o valor da expressão e; ela deve ser tratada como equivalente à atribuição $j = e$ seguida pela atribuição $i = j$.
- Variáveis (como i, j e *chave*) são locais para o procedimento dado. Não usaremos variáveis globais sem indicação explícita.
- 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 uma faixa de valores dentro de um arranjo. Desse modo, $A[1 \,..\, j]$ indica o subarranjo de A que consiste nos j elementos $A[1]$, $A[2]$, ..., $A[j]$.
- Dados compostos estão organizados tipicamente em ***objetos***, compostos por ***atributos***. Acessamos um determinado atributo usando a sintaxe encontrada em muitas linguagens de programação orientadas a objetos: o nome do objeto, seguido por um ponto, seguido pelo nome do atributo. 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 $A \cdot comprimento$.

Uma variável que representa um arranjo ou objeto é tratada como um ponteiro para os dados que representam o arranjo ou objeto. Para todos os atributos f de um objeto x, definir $y = x$ causa $y \cdot f = x \cdot f$. Além disso, se definirmos agora $x \cdot f = 3$, daí em diante não apenas $x \cdot f = 3$, mas também $y \cdot f = 3$. Em outras palavras, x e y apontarão para o mesmo objeto após a atribuição $y = x$.

A notação que usamos para atributos pode ser utilizada "em cascata". Por exemplo, suponha que o atributo f seja, em si, um ponteiro para algum tipo de objeto que tem um atributo g. Então, a notação $x \cdot f \cdot g$ estará implicitamente entre parênteses como $(x \cdot f) \cdot g$. Em outras palavras, se tivéssemos atribuído $y = x \cdot f$, então $x \cdot f \cdot g$ é o mesmo que $y \cdot g$.

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

- Parâmetros são passados para um procedimento ***por valor*** : o procedimento chamado recebe sua própria cópia dos parâmetros e, se tal procedimento atribuir um valor a um parâmetro, a mudança *não* será vista pelo procedimento de chamada. Quando objetos são passados, o ponteiro para os dados que representam o objeto é copiado, mas os atributos do objeto, não. Por exemplo, se x é um parâmetro de um procedimento chamado, a atribuição $x = y$ dentro do procedimento chamado não será visível para o procedimento de chamada.
  Contudo, a atribuição $x \cdot f = 3$ será visível. De maneira semelhante, arranjos são passados por apontador; assim, um apontador para o arranjo é passado, em vez do arranjo inteiro, e as mudanças nos elementos individuais do arranjo são visíveis para o procedimento de chamada.
- Uma instrução **return** transfere imediatamente o controle de volta ao ponto de chamada no procedimento de chamada. A maioria das instruções **return** também toma um valor para passar de volta ao chamador. Nosso pseudocódigo é diferente de muitas linguagens de programação, visto que permite que vários valores sejam devolvidos em uma única instrução **return.**
- Os operadores booleanos "e" e "ou" são operadores com ***curto-circuito***. Isto é, quando avaliamos a expressão "x e y", avaliamos primeiro x. Se x for avaliado como FALSE, a expressão inteira não poderá ser avaliada como TRUE, e assim não avaliamos y. Se, por outro lado, x for avaliado como TRUE, teremos de avaliar y para determinar o valor da expressão inteira. De modo 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 "$x \neq$ NIL e $x \cdot f = y$" sem nos preocuparmos com o que acontece ao
  tentarmos avaliar $x \cdot f$ quando x é NIL.
- A palavra-chave **error** indica que ocorreu um erro porque as condições para que o procedimento fosse chamado estavam erradas. O procedimento de chamada é responsável pelo tratamento do erro, portanto não especificamos a ação que deve ser executada.

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 busca:

Entrada: Uma sequência 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 ***busca linear***, que faça a varredura da sequência, procurando por v. Usando um invariante de laço, prove que seu algoritmo é correto. Certifique-se de que seu invariante de laço satisfaz as 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 formalmente 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 necessita. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, porém mais frequentemente é o tempo de computação que desejamos medir. Em geral, pela análise de vários algoritmos candidatos para um problema, pode-se identificar facilmente um que seja o mais eficiente. Essa análise pode indicar mais de um candidato viável, porém, em geral, podemos descartar vários algoritmos de qualidade inferior no processo.

Antes de podermos analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo para os recursos dessa tecnologia e seus custos. Na maior parte deste livro, consideraremos um modelo de computação genérico de máquina de acesso aleatório (***random-access machine***, ***RAM***) com um único processador como nossa tecnologia de implementação e entenderemos que nossos algoritmos serão implementados como programas de computador. No modelo de RAM, as instruções são executadas uma após outra, sem operações concorrentes.

No sentido estrito, deveríamos definir com precisão as instruções do modelo de RAM e seus custos. Porém, isso seria tedioso e nos daria pouca percepção do projeto e da análise de algoritmos. Não obstante, 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, visto que 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 (como 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 uma quantidade de tempo constante.

Os tipos de dados no modelo de RAM são inteiros e de ponto flutuante (para armazenar números reais). Embora normalmente não nos preocupemos com a precisão neste livro, em algumas aplicações a precisão é crucial. Também consideramos um limite para o tamanho de cada palavra de dados. Por exemplo, ao trabalharmos com entradas de tamanho n, em geral consideramos 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, o que nos permite indexar os elementos individuais da entrada, e c terá de ser obrigatoriamente uma constante para que o tamanho da palavra não cresça arbitrariamente. (Se o tamanho da palavra pudesse crescer arbitrariamente, seria possível armazenar enorme quantidade de dados em uma única palavra e executar operações com tudo isso em tempo constante — claramente um cenário irreal.)

Computadores reais contêm instruções que não citamos, e tais instruções representam uma área cinzenta 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 para a esquerda" (*shift left*) que desloca em tempo constante os bits de um inteiro k posições para a esquerda. Na maioria dos computadores, deslocar os bits de um inteiro uma posição para a esquerda equivale a multiplicar por 2; assim, deslocar os bits k posições para a esquerda equivale 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 para a esquerda, desde que k não seja maior que o número de bits em uma palavra de computador. Procuraremos evitar essas áreas cinzentas no modelo de RAM, mas trataremos o cálculo de 2_k como uma operação de tempo constante quando k for um inteiro positivo suficientemente pequeno.

No modelo de RAM, não tentamos modelar a hierarquia da memória que é comum em computadores contemporâneos. Isto é, não modelamos caches ou memória virtual. Vários modelos computacionais tentam levar em conta os efeitos da hierarquia de memória, que às vezes são significativos em programas reais em máquinas reais. Alguns problemas neste livro examinam os efeitos da hierarquia de memória mas, em sua maioria, as análises neste livro não os considerarão.

Os modelos que incluem a hierarquia de memória são bem mais complexos que o modelo de RAM, portanto 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 determinado algoritmo, ainda estaremos diante de muitas opções na hora de decidir como expressar nossa análise. Gostaríamos de dispor de um meio de expressão que seja simples de escrever e manipular, que mostre as características importantes de requisitos de recursos de um algoritmo e que suprima os detalhes tediosos.

Análise da ordenação por inserção

O tempo despendido pelo procedimento Insertion-Sort depende da entrada: ordenar mil números demora mais que ordenar três números. Além disso, Insertion-Sort pode demorar quantidades de tempo diferentes para ordenar duas sequências de entrada do mesmo tamanho, dependendo do quanto elas já estejam ordenadas. Em geral, o tempo gasto por um algoritmo cresce com o tamanho da entrada; assim, é tradicional descrever o tempo de execução de um programa em 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 para ***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 n do arranjo 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 vez 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 determinada entrada é o número de operações primitivas ou "passos" executados. É conveniente definir a noção de passo de modo que ela seja tão independente de máquina quanto possível. Por enquanto, vamos adotar a visão a seguir. Uma quantidade de tempo constante é exigida para executar cada linha do nosso pseudo código. Uma linha pode demorar uma quantidade de tempo diferente de outra linha, mas consideraremos que cada execução da i-ésima linha leva um tempo c_i, onde c_i é uma constante. Esse ponto de vista está de acordo com o modelo de RAM e também reflete o modo como o pseudo-código seria implementado na maioria dos computadores reais.[5]

Na discussão a seguir, nossa expressão para o tempo de execução de Insertion-Sort evoluirá de uma fórmula confusa que utiliza todos os custos de instrução c_i até uma notação muito mais simples, que também é mais concisa e mais fácil de manipular. Essa notação mais simples também facilitará a tarefa de determinar 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 $n = A \cdot comprimento$, seja t_j o número de vezes que o teste do laço **while** na linha 5 é executado para aquele valor de j. Quando um laço **for** ou **while** termina da maneira usual (isto é, devido ao teste no cabeçalho do laço), o teste é executado uma vez mais do que o corpo do laço. Consideramos que comentários não são instruções executáveis e, portanto, não demandam nenhum tempo.

	INSERTION-SORT(A)	*custo*	*vezes*
1	**for** $j = 2$ **to** $A \cdot comprimento$	c_1	n
2	$chave = A[j]$	c_2	$n-1$
3	//Inserir $A[j]$ na sequência ordenada $A[1..j-1]$.	0	$n-1$
4	$i = j-1$	c_4	$n-1$
5	**while** $i > 0$ e $A[i] > chave$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] = chave$	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_{in} para o tempo de execução total.[6] Para calcular $T(n)$, o tempo de execução de Insertion-Sort de uma entrada de n valores, somamos os produtos das *colunas custo* e *vezes*, obtendo

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

Mesmo para entradas de 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. Então, para cada $j = 2, 3, ..., n$, descobrimos 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

$$T(n) = c_{1n} + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Podemos expressar esse tempo de execução 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

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

(o Apêndice A apresenta modos de resolver esses somatórios), descobrimos que, no pior caso, o tempo de execução de Insertion-Sort é

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

Podemos expressar esse tempo de execução do pior caso 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 determinada entrada, embora em capítulos posteriores veremos alguns algoritmos "aleatorizados" 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, examinamos 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 em determinar apenas o ***tempo de execução do pior caso***; ou seja, o tempo de execução mais longo para *qualquer* entrada de tamanho n. Apresentamos três razões para essa orientação.

- O tempo de execução do pior caso de um algoritmo estabelece um limite superior para o tempo de execução para qualquer entrada. Conhecê-lo nos dá uma garantia de que o algoritmo nunca demorará mais do que esse tempo. Não precisamos fazer nenhuma suposição sobre o tempo de execução esperando que ele nunca seja muito pior.
- Para alguns algoritmos, o pior caso ocorre com bastante frequência. Por exemplo, na pesquisa de um banco de dados em busca de determinada informação, o pior caso do algoritmo de busca frequentemente ocorre quando a informação não está presente no banco de dados. Em algumas aplicações, a busca de informações ausentes pode ser frequente.
- Muitas vezes, o "caso médio" é quase tão ruim quanto o pior caso. Suponha que escolhemos n números aleatoriamente e aplicamos ordenação por inserção. Quanto tempo transcorrerá até que o algoritmo determine o lugar no subarranjo $A[1 .. j - 1]$ em que deve ser inserido o elemento $A[j]$? Em média, metade dos elementos em $A[1 .. j - 1]$ é menor que $A[j]$ e metade dos elementos é maior. Portanto, em média, verificamos metade do

subarranjo $A[1 .. j - 1]$ e, portanto, $t_j = j/2$. Resulta que o tempo de execução obtido para o caso médio é uma função quadrática do tamanho da entrada, exatamente o que ocorre com o tempo de execução do pior caso.

Em alguns casos particulares, estaremos interessados no tempo de execução do ***caso médio*** de um algoritmo; veremos, neste livro, a técnica da ***análise probabilística*** aplicada a vários algoritmos. O escopo da análise do caso médio é limitado porque pode não ser evidente o que constitui uma entrada "média" para determinado problema. Muitas vezes consideraremos que todas as entradas de um dado tamanho são igualmente prováveis. Na prática, é possível que essa suposição seja violada, mas, às vezes, podemos utilizar um ***algoritmo aleatorizado***, que efetua escolhas ao acaso, para permitir uma análise probabilística e produzir um tempo ***esperado*** de execução. Estudaremos algoritmos randomizados com mais detalhes no Capítulo 5 e em vários outros capítulos subsequentes.

Ordem de crescimento

Usamos algumas abstrações simplificadoras para facilitar nossa análise do procedimento Insertion-Sort. Primeiro, ignoramos o custo real de cada instrução, usando as constantes c_i para representar esses custos. Então, observamos que até mesmo essas constantes nos dão mais detalhes do que realmente necessitamos: expressamos o tempo de execução do pior caso como $an^2 + bn + c$ para algumas constantes a, b e c que dependem dos custos de instrução c_i. Desse modo, ignoramos não apenas os custos reais de instrução, mas também os custos abstratos c_i.

Agora, faremos mais uma abstração simplificadora. É a ***taxa de crescimento***, ou ***ordem de crescimento***, do tempo de execução que realmente nos interessa. Portanto, consideramos apenas o termo inicial de uma fórmula (por exemplo, an^2), já que os termos de ordem mais baixa são relativamente insignificantes para grandes valores de n. Também ignoramos o coeficiente constante do termo inicial, visto que fatores constantes são menos significativos que a taxa de crescimento na determinação da eficiência computacional para grandes entradas. No caso da ordenação por inserção, quando ignoramos os termos de ordem mais baixa e o coeficiente constante do termo inicial, resta apenas o fator de n^2 do termo inicial. Afirmamos que a ordenação por inserção tem um tempo de execução do pior caso igual a $\Theta(n^2)$ (lido como "teta de n ao quadrado"). Neste capítulo usaremos informalmente a notação Θ e a definiremos com precisão no Capítulo 3.

Em geral, consideramos que um algoritmo é mais eficiente que outro se seu tempo de execução do pior caso apresentar uma ordem de crescimento mais baixa. Devido a fatores constantes e termos de ordem mais baixa, um algoritmo cujo tempo de execução tenha uma ordem de crescimento mais alta pode demorar menos tempo para pequenas entradas do que um algoritmo cuja ordem de crescimento seja mais baixa. Porém, para entradas suficientemente grandes, um algoritmo $\Theta(n^2)$, por exemplo, será executado mais rapidamente no pior caso que um algoritmo $\Theta(n^3)$.

Exercícios

2.2-1 Expresse a função $n^3/1000 - 100n^2 - 100n + 3$ em termos da notação Θ.

2.2-2 Considere a ordenação de n números armazenados no arranjo A, localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em $A[1]$. Em seguida, determine o segundo menor elemento de A e permute-o com $A[2]$. Continue dessa maneira para os primeiros $n - 1$ elementos de A. Escreva o pseudocódigo para esse algoritmo, conhecido como **ordenação por seleção**. Qual invariante de laço esse algoritmo mantém? Por que ele só precisa ser executado para os primeiros $n - 1$ elementos, e não para todos os n elementos? Forneça os tempos de execução do melhor caso e do pior caso da ordenação por seleção em notação Θ.

2.2-3 Considere mais uma vez a busca linear (veja Exercício 2.1-3). Quantos elementos da sequência de entrada precisam ser verificados em média, considerando que o elemento que está sendo procurado tenha a mesma probabilidade de ser qualquer elemento no arranjo? E no pior caso? Quais são os tempos de execução do