Capítulo 8

Ordenação em tempo linear

Apresentamos até agora diversos algoritmos que podem ordenar n números no tempo $O(n \lg n)$. A ordenação por intercalação e o heapsort alcançam esse limite superior no pior caso; quicksort o alcança na média. Além disso, para cada um desses algoritmos, podemos produzir uma seqüência de n números de entrada que faz o algoritmo ser executado no tempo $\Omega(n \lg n)$.

Esses algoritmos compartilham uma propriedade interessante: *a seqüência ordenada que eles determinam* se *baseia apenas em comparações entre os elementos de entrada*. Chamamos esses algoritmos de ordenação de ***ordenações por comparação***. Todos os algoritmos de ordenação apresentados até agora são portanto ordenações por comparação.

Na Seção 8.1, provaremos que qualquer ordenação por comparação deve efetuar $\Omega(n \lg n)$ comparações no pior caso para ordenar n elementos. Desse modo, a ordenação por intercalação e heapsort são assintoticamente ótimas, e não existe nenhuma ordenação por comparação que seja mais rápida por mais de um fator constante.

As Seções 8.2, 8.3 e 8.4 examinam três algoritmos de ordenação – ordenação por contagem, radix sort (ordenação da raiz) e bucket sort (ordenação por balde) – que são executados em tempo linear. É desnecessário dizer que esses algoritmos utilizam outras operações diferentes de comparações para determinar a seqüência ordenada. Em conseqüência disso, o limite inferior $\Omega(n \lg n)$ não se aplica a eles.

8.1 Limites inferiores para ordenação

Em uma ordenação por comparação, usamos apenas comparações entre elementos para obter informações de ordem sobre uma seqüência de entrada $\langle a_1, a_2, ..., a_n \rangle$. Ou seja, dados dois elementos a_i e a_j, executamos um dos testes $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ ou $a_i > a_j$, para determinar sua ordem relativa. Podemos inspecionar os valores dos elementos ou obter informações de ordem sobre eles de qualquer outro modo.

Nesta seção, vamos supor sem perda de generalidade que todos os elementos de entrada são distintos. Dada essa hipótese, as comparações da forma $a_i = a_j$ são inúteis; assim, podemos supor que não é feita nenhuma comparação dessa forma. Também observamos que as comparações $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ e $a_i < a_j$ são todas equivalentes, em virtude de produzirem informações idênticas sobre a ordem relativa de a_i e a_j. Por essa razão, vamos supor que todas as comparações têm a forma $a_i \leq a_j$.

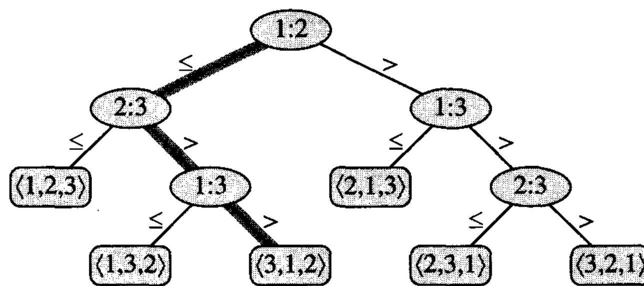

FIGURA 8.1 A árvore de decisão para ordenação por inserção, operando sobre três elementos. Um nó interno anotado por $i{:}j$ indica uma comparação entre a_i e a_j. Uma folha anotada pela permutação $\langle\pi(1), \pi(2), \ldots, \pi(n)\rangle$ indica a ordenação $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. O caminho sombreado indica as decisões tomadas durante a ordenação da seqüência de entrada $\langle a_1 = 6, a_2 = 8, a_3 = 5\rangle$; a permutação $\langle 3, 1, 2\rangle$ na folha indica que a seqüência ordenada é $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. Existem $3! = 6$ permutações possíveis dos elementos de entrada; assim, a árvore de decisão deve ter no mínimo 6 folhas

O modelo de árvore de decisão

As ordenações por comparação podem ser vistas de modo abstrato em termos de ***árvores de decisão***. Uma árvore de decisão é uma árvore binária completa que representa as comparações executadas por um algoritmo de ordenação quando ele opera sobre uma entrada de um tamanho dado. Controle, movimentação de dados e todos os outros aspectos do algoritmo são ignorados. A Figura 8.1 mostra a árvore de decisão correspondente ao algoritmo de ordenação por inserção da Seção 2.1, operando sobre uma seqüência de entrada de três elementos.

Em uma árvore de decisão, cada nó interno é anotado por $i{:}j$ para algum i e j no intervalo $1 \leq i, j \leq n$, onde n é o número de elementos na seqüência de entrada. Cada folha é anotada por uma permutação $\langle\pi(1), \pi(2), \ldots, \pi(n)\rangle$. (Consulte a Seção C.1 para adquirir experiência em permutações.) A execução do algoritmo de ordenação corresponde a traçar um caminho desde a raiz da árvore de decisão até uma folha. Em cada nó interno, é feita uma comparação $a_i \leq a_j$. A subárvore da esquerda determina então comparações subseqüentes para $a_i \leq a_j$, e a subárvore da direita determina comparações subseqüentes para $a_i > a_j$. Quando chegamos a uma folha, o algoritmo de ordenação estabelece a ordem $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Como qualquer algoritmo de ordenação correto deve ser capaz de produzir cada permutação de sua entrada, uma condição necessária para uma ordenação por comparação ser correta é que cada uma das $n!$ permutações sobre n elementos deve aparecer como uma das folhas da árvore de decisão, e que cada uma dessas folhas deve ser acessível a partir da raiz por um caminho correspondente a uma execução real da ordenação por comparação. (Iremos nos referir a tais folhas como "acessíveis".) Desse modo, consideraremos apenas árvores de decisão em que cada permutação aparece como uma folha acessível.

Um limite inferior para o pior caso

O comprimento do caminho mais longo da raiz de uma árvore de decisão até qualquer de suas folhas acessíveis representa o número de comparações do pior caso que o algoritmo de ordenação correspondente executa. Conseqüentemente, o número de comparações do pior caso para um dado algoritmo de ordenação por comparação é igual à altura de sua árvore de decisão. Um limite inferior sobre as alturas de todas as árvores de decisão em que cada permutação aparece como uma folha acessível é portanto um limite inferior sobre o tempo de execução de qualquer algoritmo de ordenação por comparação. O teorema a seguir estabelece esse limite inferior.

Teorema 8.1
Qualquer algoritmo de ordenação por comparação exige $\Omega(n \lg n)$ comparações no pior caso.

Prova Da discussão precedente, basta determinar a altura de uma árvore de decisão em que cada permutação aparece como uma folha acessível. Considere uma árvore de decisão de altura h com l folhas acessíveis correspondente a uma ordenação por comparação sobre n elementos. Como cada uma das $n!$ permutações da entrada aparece como alguma folha, temos $n! \leq l$. Tendo em vista que uma árvore binária de altura h não tem mais de 2^h folhas, temos

$$n! \leq l \leq 2^h ,$$

que, usando-se logaritmos, implica

$$\begin{aligned} h &\geq \lg(n!) && \text{(pois a função lg é monotonicamente crescente)} \\ &= \Omega(n \lg n) && \text{(pela equação (3.18))} . \end{aligned}$$

■

Corolário 8.2
O heapsort e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

Prova Os $O(n \lg n)$ limites superiores sobre os tempos de execução para heapsort e ordenação por intercalação correspondem ao limite inferior $\Omega(n \lg n)$ do pior caso do Teorema 8.1. ■

Exercícios

8.1-1
Qual é a menor profundidade possível de uma folha em uma árvore de decisão para uma ordenação por comparação?

8.1-2
Obtenha limites assintoticamente restritos sobre $\lg(n!)$ sem usar a aproximação de Stirling. Em vez disso, avalie o somatório $\sum_{k=1}^{n} \lg k$, empregando técnicas da Seção A.2.

8.1-3
Mostre que não existe nenhuma ordenação por comparação cujo tempo de execução seja linear para pelo menos metade das $n!$ entradas de comprimento n. E no caso de uma fração $1/n$ das entradas de comprimento n? E no caso de uma fração $1/2^n$?

8.1-4
Você recebeu uma seqüência de n elementos para ordenar. A seqüência de entrada consiste em n/k subseqüências, cada uma contendo k elementos. Os elementos em uma dada subseqüência são todos menores que os elementos na subseqüência seguinte e maiores que os elementos na subseqüência precedente. Desse modo, tudo que é necessário para ordenar a seqüência inteira de comprimento n é ordenar os k elementos em cada uma das n/k subseqüências. Mostre um limite inferior $\Omega(n \lg k)$ sobre o número de comparações necessárias para resolver essa variação do problema de ordenação. (*Sugestão*: Não é rigoroso simplesmente combinar os limites inferiores para as subseqüências individuais.)

8.2 Ordenação por contagem

A ***ordenação por contagem*** pressupõe que cada um dos n elementos de entrada é um inteiro no intervalo de 1 a k, para algum inteiro k. Quando $k = O(n)$, a ordenação é executada no tempo $\Theta(n)$.

A idéia básica da ordenação por contagem é determinar, para cada elemento de entrada x, o número de elementos menores que x. Essa informação pode ser usada para inserir o elemento x

diretamente em sua posição no arranjo de saída. Por exemplo, se há 17 elementos menores que x, então x é colocado na posição de saída 18. Esse esquema deve ser ligeiramente modificado para manipular a situação na qual vários elementos têm o mesmo valor, pois não queremos inserir todos eles na mesma posição.

No código para ordenação por contagem, partimos da suposição de que a entrada é um arranjo $A[1 \,..\, n]$ e, portanto, *comprimento*$[A] = n$. Exigimos dois outros arranjos: o arranjo $B[1 \,..\, n]$ contém a saída ordenada, e o arranjo $C[0 \,..\, k]$ fornece um espaço de armazenamento de trabalho temporário.

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2      do C[i] ← 0
 3  for j ← 1 to comprimento[A]
 4      do C[A[j]] ← C[A[j]] + 1
 5  ▷ Agora C[i] contém o número de elementos iguais a i.
 6  for i ← 1 to k
 7      do C[i] ← C[i] + C[i – 1]
 8  ▷ Agora C[i] contém o número de elementos menores que ou iguais a i.
 9  for j ← comprimento[A] downto 1
10      do B[C[A[j]]] ← A[j]
11         C[A[j]] ← C[A[j]] – 1
```

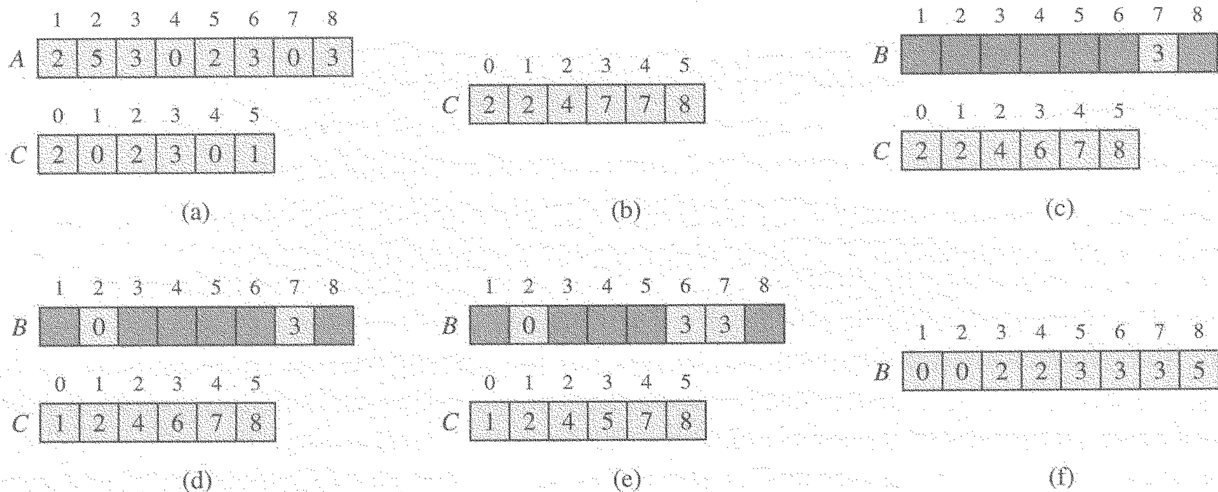

FIGURA 8.2 A operação de COUNTING-SORT sobre um arranjo de entrada $A[1 \,..\, 8]$, onde cada elemento de A é um inteiro não negativo não maior que $k = 5$. **(a)** O arranjo A e o arranjo auxiliar C após a linha 4. **(b)** O arranjo C após a linha 7. **(c)–(e)** O arranjo de saída B e o arranjo auxiliar C após uma, duas e três iterações do loop nas linhas 9 a 11, respectivamente. Apenas os elementos levemente sombreados do arranjo B foram preenchidos. **(f)** O arranjo de saída final ordenado B

A ordenação por contagem é ilustrada na Figura 8.2. Após a inicialização no loop **for** das linhas 1 e 2, inspecionamos cada elemento de entrada no loop **for** das linhas 3 e 4. Se o valor de um elemento de entrada é i, incrementamos $C[i]$. Desse modo, depois da linha 4, $C[i]$ contém um número de elementos de entrada igual a i para cada inteiro $i = 0, 1, ..., k$. Nas linhas 6 e 7 determinamos, para cada $i = 0, 1, ..., k$, quantos elementos de entrada são menores que ou iguais a i; mantendo uma soma atualizada do arranjo C.

Finalmente, no loop **for** das linhas 9 a 11, colocamos cada elemento $A[j]$ em sua posição ordenada correta no arranjo de saída B. Se todos os n elementos forem distintos, então, quando entrarmos pela primeira vez na linha 9, para cada $A[j]$, o valor $C[A[j]]$ será a posição final correta

de $A[j]$ no arranjo de saída, pois existem $C[A[j]]$ elementos menores que ou iguais a $A[j]$. Como os elementos podem não ser distintos, decrementamos $C[A[j]]$ toda vez que inserimos um valor $A[j]$ no arranjo B; isso faz com que o próximo elemento de entrada com um valor igual a $A[j]$, se existir algum, vá para a posição imediatamente anterior a $A[j]$ no arranjo de saída.

Quanto tempo a ordenação por contagem exige? O loop **for** das linhas 1 e 2 demora o tempo $\Theta(k)$, o loop **for** das linhas 3 e 4 demora o tempo $\Theta(n)$, o loop **for** das linhas 6 e 7 demora o tempo $\Theta(k)$ e o loop **for** das linhas 9 a 11 demora o tempo $\Theta(n)$. Portanto, o tempo total é $\Theta(k + n)$. Na prática, normalmente usamos a ordenação por contagem quando temos $k = O(n)$, em cujo caso o tempo de execução é $\Theta(n)$.

A ordenação por contagem supera o limite inferior de $\Omega(n \lg n)$ demonstrado na Seção 8.1, porque não é uma ordenação por comparação. De fato, nenhuma comparação entre elementos de entrada ocorre em qualquer lugar no código. Em vez disso, a ordenação por contagem utiliza os valores reais dos elementos para efetuar a indexação em um arranjo. O limite inferior $\Omega(n \lg n)$ para ordenação não se aplica quando nos afastamos do modelo de ordenação por comparação.

Uma propriedade importante da ordenação por contagem é o fato de ela ser ***estável***: números com o mesmo valor aparecem no arranjo de saída na mesma ordem em que se encontram no arranjo de entrada. Ou seja, os vínculos entre dois números são rompidos pela regra de que qualquer número que aparecer primeiro no arranjo de entrada aparecerá primeiro no arranjo de saída. Normalmente, a propriedade de estabilidade só é importante quando dados satélite são transportados juntamente com o elemento que está sendo ordenado. A estabilidade da ordenação por contagem é importante por outra razão: a ordenação por contagem é usada freqüentemente como uma sub-rotina em radix sort. Como veremos na próxima seção, a estabilidade da ordenação por contagem é crucial para a correção da radix sort.

Exercícios

8.2-1

Usando a Figura 8.2 como modelo, ilustre a operação de COUNTING-SORT sobre o arranjo $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2

Prove que COUNTING-SORT é estável.

8.2-3

Suponha que o cabeçalho do loop **for** na linha 9 do procedimento COUNTING-SORT seja reescrito:

9 **for** $j \leftarrow 1$ **to** *comprimento*[A]

Mostre que o algoritmo ainda funciona corretamente. O algoritmo modificado é estável?

8.2-4

Descreva um algoritmo que, dados n inteiros no intervalo de 0 a k, realiza o pré-processamento de sua entrada e depois responde a qualquer consulta sobre quantos dos n inteiros recaem em um intervalo $[a \mathinner{..} b]$ no tempo $O(1)$. Seu algoritmo deve utilizar o tempo de pré-processamento $\Theta(n + k)$.

8.3 Radix sort

A ***radix sort*** (ou ***ordenação da raiz***) é o algoritmo usado pelas máquinas de ordenação de cartões que agora são encontradas apenas nos museus de informática. Os cartões estão organizados em 80 colunas, e em cada coluna pode ser feita uma perfuração em uma de 12 posições. O

ordenador pode ser "programado" mecanicamente para examinar uma determinada coluna de cada cartão em uma pilha e distribuir o cartão em uma de 12 caixas, dependendo de qual foi o local perfurado. Um operador pode então juntar os cartões caixa por caixa, de modo que os cartões com a primeira posição perfurada fiquem sobre os cartões com a segunda posição perfurada e assim por diante.

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

FIGURA 8.3 A operação de radix sort sobre uma lista de sete números de 3 dígitos. A primeira coluna é a entrada. As colunas restantes mostram a lista após ordenações sucessivas sobre posições de dígitos cada vez mais significativas. As setas verticais indicam a posição do dígito sobre o qual é feita a ordenação para produzir cada lista a partir da anterior

No caso de dígitos decimais, apenas 10 posições são utilizadas em cada coluna. (As outras duas posições são usadas para codificação de caracteres não numéricos.) Assim, um número de d dígitos ocuparia um campo de d colunas. Tendo em vista que o ordenador de cartões pode examinar apenas uma coluna de cada vez, o problema de ordenar n cartões em um número de d dígitos requer um algoritmo de ordenação.

Intuitivamente, poderíamos ordenar números sobre seu dígito *mais significativo*, ordenar cada uma das caixas resultantes recursivamente, e então combinar as pilhas em ordem. Infelizmente, como os cartões em 9 das 10 caixas devem ser postos de lado para se ordenar cada uma das caixas, esse procedimento gera muitas pilhas intermediárias de cartões que devem ser controladas. (Ver Exercício 8.3-5.)

A radix sort resolve o problema da ordenação de cartões de modo contra-intuitivo ordenando primeiro sobre o dígito *menos significativo*. Os cartões são então combinados em uma única pilha, com os cartões na caixa 0 precedendo os cartões na caixa 1, que precedem os cartões na caixa 2 e assim por diante. Então, a pilha inteira é ordenada novamente sobre o segundo dígito menos significativo e recombinada de maneira semelhante. O processo continua até os cartões terem sido ordenados sobre todos os d dígitos. É interessante observar que, nesse ponto, os cartões estão completamente ordenados sobre o número de d dígitos. Desse modo, apenas d passagens pela pilha são necessárias para se fazer a ordenação. A Figura 8.3 mostra como a radix sort opera sobre uma "pilha" (ou um "deck") de sete números de 3 dígitos.

É essencial que as ordenações de dígitos nesse algoritmo sejam estáveis. A ordenação executada por um ordenador de cartões é estável, mas o operador tem de ser cauteloso para não alterar a ordem dos cartões à medida que eles são retirados de uma caixa, ainda que todos os cartões em uma caixa tenham o mesmo dígito na coluna escolhida.

Em um computador típico, que é uma máquina seqüencial de acesso aleatório, a radix sort é usada às vezes para ordenar registros de informações chaveados por vários campos. Por exemplo, talvez fosse desejável ordenar datas por três chaves: ano, mês e dia. Poderíamos executar um algoritmo de ordenação com uma função de comparação que, dadas duas datas, comparasse anos e, se houvesse uma ligação, comparasse meses e, se ocorresse outra ligação, comparasse dias. Como outra alternativa, poderíamos ordenar as informações três vezes com uma ordenação estável: primeiro sobre o dia, em seguida sobre o mês, e finalmente sobre o ano.

O código para radix sort é direto. O procedimento a seguir supõe que cada elemento no arranjo de n elementos A tem d dígitos, onde o dígito 1 é o dígito de mais baixa ordem e o dígito d é o dígito de mais alta ordem.

RADIX-SORT(A, d)
1 **for** $i \leftarrow 1$ **to** d
2 **do** usar uma ordenação estável para ordenar o arranjo A sobre o dígito i

Lema 8.3
Dados n números de d dígitos em que cada dígito pode assumir até k valores possíveis, RADIX-SORT ordena corretamente esses números no tempo $\Theta(d(n + k))$.

Prova A correção de radix sort se segue por indução sobre a coluna que está sendo ordenada (ver Exercício 8.3-3). A análise do tempo de execução depende da ordenação estável usada como algoritmo de ordenação intermediária. Quando cada dígito está no intervalo de 0 a $k - 1$ (de modo que possa assumir até k valores possíveis) e k não é muito grande, a ordenação por contagem é a escolha óbvia. Cada passagem sobre n números de d dígitos leva então o tempo $\Theta(n + k)$. Há d passagens; assim, o tempo total para radix sort é $\Theta(d(n + k))$. ■

Quando d é constante e $k = O(n)$, radix sort é executada em tempo linear. Mais geralmente, temos alguma flexibilidade em como quebrar cada chave em dígitos.

Lema 8.4
Dados n números de b bits e qualquer inteiro positivo $r \leq b$, RADIX-SORT ordena corretamente esses números no tempo $\Theta((b/r)(n + 2^r))$.

Prova Para um valor $r \leq b$, visualizamos cada chave como tendo $d = \lceil b/r \rceil$ dígitos de r bits cada. Cada dígito é um inteiro no intervalo 0 a $2^r - 1$, de forma que podemos usar a ordenação por contagem com $k = 2^r - 1$. (Por exemplo, podemos visualizar uma palavra de 32 bits como tendo 4 dígitos de 8 bits, de forma que $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ e $d = b/r = 4$.) Cada passagem da ordenação por contagem leva o tempo $\Theta(n + k) = \Theta(n + 2^r)$ e há d passagens, dando um tempo de execução total igual a $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

Para valores de n e b dados, desejamos escolher o valor de r, com $r \leq b$, que minimiza a expressão $(b/r)(n + 2^r)$. Se $b < \lfloor \lg n \rfloor$, para qualquer valor de $r \leq b$, temos $(n + 2^r) = \Theta(n)$. Desse modo, a escolha de $r = b$ produz um tempo de execução $(b/b)(n + 2^b) = \Theta(n)$, que é assintoticamente ótimo. Se $b \geq \lfloor \lg n \rfloor$, então a escolha de $r = \lfloor \lg n \rfloor$ fornece o melhor tempo dentro de um fator constante, que podemos ver como a seguir. A escolha de $r = \lfloor \lg n \rfloor$ produz um tempo de execução $\Theta(bn/\lg n)$. À medida que aumentamos r acima de $\lfloor \lg n \rfloor$, o termo 2^r no numerador aumenta mais rápido que o termo r no denominador, e assim o aumento de r acima de $\lfloor \lg n \rfloor$ resulta em um tempo de execução $\Omega(bn/\lg n)$. Se, em vez disso, diminuirmos r abaixo de $\lfloor \lg n \rfloor$, então o termo b/r aumentará, e o termo $n + 2^r$ permanecerá em $\Theta(n)$.

É preferível radix sort a um algoritmo de ordenação baseado em comparação, como quicksort? Se $b = O(\lg n)$, como é freqüentemente o caso, e escolhemos $r \approx \lg n$, então o tempo de execução de radix sort é $\Theta(n)$, que parece ser melhor que o tempo do caso médio de quicksort, $\Theta(n \lg n)$. Porém, os fatores constantes ocultos na notação Θ são diferentes. Embora radix sort possa fazer menos passagens que quicksort sobre as n chaves, cada passagem de radix sort pode tomar um tempo significativamente maior. Determinar o algoritmo de ordenação preferível depende das características das implementações, da máquina subjacente (por exemplo, quicksort utiliza com freqüência caches de hardware de modo mais eficaz que radix sort) e dos dados de entrada. Além disso, a versão de radix sort que utiliza a ordenação por contagem como ordenação estável intermediária não efetua a ordenação local, o que é feito por muitas das ordenações por comparação no tempo $\Theta(n \lg n)$. Desse modo, quando o espaço de armazenamento da memória primária é importante, um algoritmo local como quicksort pode ser preferível.

Exercícios

8.3-1

Usando a Figura 8.3 como modelo, ilustre a operação de RADIX-SORT sobre a seguinte lista de palavras em inglês: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3-2

Quais dos seguintes algoritmos de ordenação são estáveis: ordenação por inserção, ordenação por intercalação, heapsort e quicksort? Forneça um esquema simples que torne estável qualquer algoritmo de ordenação. Quanto tempo e espaço adicional seu esquema requer?

8.3-3

Use a indução para provar que radix sort funciona. Onde sua prova necessita da hipótese de que a ordenação intermediária é estável?

8.3-4

Mostre como ordenar n inteiros no intervalo de 0 a $n^2 - 1$ no tempo $O(n)$.

8.3-5 ★

No primeiro algoritmo de ordenação de cartões desta seção, exatamente quantas passagens de ordenação são necessárias para ordenar números decimais de d dígitos no pior caso? Quantas pilhas de cartões um operador precisaria controlar no pior caso?

8.4 Bucket sort

A ***bucket sort*** (ou ***ordenação por balde***) funciona em tempo linear quando a entrada é gerada a partir de uma distribuição uniforme. Como a ordenação por contagem, a bucket sort é rápida porque pressupõe algo sobre a entrada. Enquanto a ordenação por contagem presume que a entrada consiste em inteiros em um intervalo pequeno, bucket sort presume que a entrada é gerada por um processo aleatório que distribui elementos uniformemente sobre o intervalo $[0, 1)$. (Consulte a Seção C.2 para ver uma definição de distribuição uniforme.)

A idéia de bucket sort é dividir o intervalo $[0, 1)$ em n subintervalos de igual tamanho, ou ***baldes***, e depois distribuir os n números de entrada entre os baldes. Tendo em vista que as entradas são uniformemente distribuídas sobre $[0, 1)$, não esperamos que muitos números caiam em cada balde. Para produzir a saída, simplesmente ordenamos os números em cada balde, e depois percorremos os baldes em ordem, listando os elementos contidos em cada um.

Nosso código para bucket sort pressupõe que a entrada é um arranjo de n elementos A, e que cada elemento $A[i]$ no arranjo satisfaz a $0 \leq A[i] < 1$. O código exige um arranjo auxiliar $B[0 .. n-1]$ de listas ligadas (baldes) e pressupõe que existe um mecanismo para manter tais listas. (A Seção 10.2 descreve como implementar operações básicas sobre listas ligadas.)

BUCKET-SORT(A)
1 $n \leftarrow$ *comprimento*$[A]$
2 **for** $i \leftarrow 1$ **to** n
3 **do** inserir $A[i]$ na lista $B[\lfloor nA[i] \rfloor]$
4 **for** $i \leftarrow 0$ **to** $n - 1$
5 **do** ordenar lista $B[i]$ com ordenação por inserção
6 concatenar as listas $B[0], B[1], ..., B[n-1]$ juntas em ordem

A Figura 8.4 mostra a operação de bucket sort sobre um arranjo de entrada de 10 números.

Para ver que esse algoritmo funciona, considere dois elementos $A[i]$ e $A[j]$. Suponha sem perda de generalidade que $A[i] \leq A[j]$. Tendo em vista que $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, o elemento $A[i]$ é inserido no mesmo balde que $A[j]$ ou em um balde com índice mais baixo. Se $A[i]$ e $A[j]$ são inseri-

dos no mesmo balde, então o loop **for** das linhas 4 e 5 os coloca na ordem adequada. Se $A[i]$ e $A[j]$ são inseridos em baldes diferentes, a linha 6 os coloca na ordem adequada. Portanto, a bucket sort funciona corretamente.

Para analisar o tempo de execução, observe que todas as linhas exceto a linha 5 demoram o tempo $O(n)$ no pior caso. Resta equilibrar o tempo total ocupado pelas n chamadas à ordenação por inserção na linha 5.

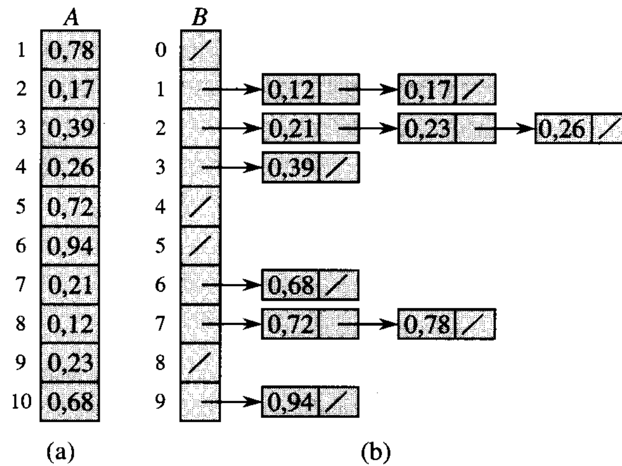

FIGURA 8.4 A operação de BUCKET-SORT. **(a)** O arranjo de entrada $A[1 \,..\, 10]$. **(b)** O arranjo $B[0 \,..\, 9]$ de listas ordenadas (baldes) depois da linha 5 do algoritmo. O balde i contém valores no intervalo $[i/10, (i + 1)/10)$. A saída ordenada consiste em uma concatenação em ordem das listas $B[0], B[1], ..., B[9]$

Para analisar o custo das chamadas para ordenação por inserção, seja ni a variável aleatória que denota o número de elementos inseridos no balde $B[i]$. Tendo em vista que a ordenação por inserção funciona em tempo quadrático (consulte a Seção 2.2), o tempo de execução de bucket sort é

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \,.$$

Tomando as expectativas de ambos os lados e usando a linearidade de expectativa, temos

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \qquad \text{(por linearidade de expectativa)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) \qquad \text{(pela equação (C.21))} \,. \qquad (8.1) \end{aligned}$$

Afirmamos que

$$\mathrm{E}[n_i^2] = 2 - 1/n \qquad (8.2)$$

para $i = 0, 1, ..., n - 1$. Não surpreende que cada balde i tenha o mesmo valor de $\mathrm{E}[n_i^2]$, pois cada valor no arranjo de entrada A tem igual probabilidade de cair em qualquer balde. Para provar a equação (8.2), definimos variáveis indicadoras aleatórias

$X_{ij} = \text{I}\ \{A[j] \text{ recai no balde } i\}$

para $i = 0, 1, ..., n - 1$ e $j = 1, 2, ..., n$. Desse modo,

$$n_i = \sum_{j=1}^{n} X_{ij} .$$

Para calcular $\text{E}[n_i^2]$, expandimos o quadrado e reagrupamos os termos:

$$\begin{aligned}
\text{E}[n_i^2] &= \text{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \text{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij} X_{ik}\right] \\
&= \text{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1 \le j \le n} \sum_{\substack{1 \le j \le n \\ k \ne j}} X_{ij} X_{ik}\right] \\
&= \sum_{j=1}^{n} \text{E}[X_{ij}^2] + \sum_{1 \le j \le n} \sum_{\substack{1 \le j \le n \\ k \ne j}} \text{E}[X_{ij} X_{ik}] , \qquad (8.3)
\end{aligned}$$

onde a última linha se segue por linearidade de expectativa. Avaliamos os dois somatórios separadamente. A variável indicadora aleatória X_{ij} é 1 com probabilidade $1/n$ e 0 em caso contrário e, portanto,

$$\begin{aligned}
\text{E}[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\
&= \frac{1}{n} .
\end{aligned}$$

Quando $k \ne j$, as variáveis X_{ij} e X_{ik} são independentes e, por conseguinte,

$$\begin{aligned}
\text{E}[X_{ij} X_{ik}] &= \text{E}[X_{ij}]\ \text{E}[X_{ik}] \\
&= \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n^2} .
\end{aligned}$$

Substituindo esses dois valores esperados na equação (8.3), obtemos

$$\begin{aligned}
\text{E}[X_{ij}^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le j \le n \\ k \ne j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}
\end{aligned}$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n},$$

o que prova a equação (8.2).

Usando esse valor esperado na equação (8.1), concluímos que o tempo esperado para bucket sort é $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$. Desse modo, o algoritmo de bucket sort inteiro funciona no tempo esperado linear.

Mesmo que a entrada não seja obtida a partir de uma distribuição uniforme, bucket sort ainda pode ser executada em tempo linear. Como a entrada tem a propriedade de que a soma dos quadrados dos tamanhos de baldes é linear no número total de elementos, a equação (8.1) nos diz que bucket sort funcionará em tempo linear.

Exercícios

8.4-1

Usando a Figura 8.4 como modelo, ilustre a operação de BUCKET-SORT no arranjo $A = \langle 0{,}79, 0{,}13, 0{,}16, 0{,}64, 0{,}39, 0{,}20, 0{,}89, 0{,}53, 0{,}71, 0{,}42 \rangle$.

8.4-2

Qual é o tempo de execução do pior caso para o algoritmo de bucket sort? Que alteração simples no algoritmo preserva seu tempo de execução esperado linear e torna seu tempo de execução no pior caso igual a $O(n \lg n)$?

8.4-3

Seja X uma variável aleatória que é igual ao número de caras em dois lançamentos de uma moeda comum. Qual é $E[X^2]$? Qual é $E^2[X]$?

8.4-4 ★

Temos n pontos no círculo unitário, $p_i = (x_i, y_i)$, tais que $0 < x_i^2 + y_i^2 \leq 1$ para $i = 1, 2, ..., n$. Suponha que os pontos estejam distribuídos uniformemente; ou seja, a probabilidade de se encontrar um ponto em qualquer região do círculo é proporcional à área dessa região. Projete um algoritmo de tempo esperado $\Theta(n)$ para ordenar os n pontos por suas distâncias $d_i = \sqrt{x_i^2 + y_i^2}$ a partir da origem. (*Sugestão*: Projete os tamanhos de baldes em BUCKET-SORT para refletir a distribuição uniforme dos pontos no círculo unitário.)

8.4-5 ★

Uma ***função de distribuição de probabilidades*** $P(x)$ para uma variável aleatória X é definida por $P(x) = \Pr\{X \leq x\}$. Suponha que uma lista de n variáveis aleatórias $X_1, X_2, ..., X_n$ seja obtida a partir de uma função de distribuição de probabilidades contínuas P que possa ser calculada no tempo $O(1)$. Mostre como ordenar esses números em tempo esperado linear.

Problemas

8-1 Limites inferiores do caso médio na ordenação por comparação

Neste problema, provamos um limite inferior $\Omega(n \lg n)$ sobre o tempo de execução esperado de qualquer ordenação por comparação determinística ou aleatória sobre n elementos de entrada distintos. Começamos examinando uma ordenação por comparação determinística A com árvore de decisão T_A. Supomos que toda permutação de entradas de A é igualmente provável.

a. Suponha que cada folha de T_A seja identificada com a probabilidade de ser alcançada dada uma entrada aleatória. Prove que exatamente $n!$ folhas são identificadas com $1/n!$ e que as restantes são identificadas com 0.

b. Seja $D(T)$ um valor que denota o comprimento do caminho externo de uma árvore de decisão T; isto é, $D(T)$ é a soma das profundidades de todas as folhas de T. Seja T uma árvore de decisão com $k > 1$ folhas, e sejam RT e LT as subárvores direita e esquerda de T. Mostre que $D(T) = D(LT) + D(RT) + k$.

c. Seja $d(k)$ o valor mínimo de $D(T)$ sobre todas as árvores de decisão T com $k > 1$ folhas. Mostre que $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$. (*Sugestão*: Considere uma árvore de decisão T com k folhas que alcance o mínimo. Seja i_0 o número de folhas em LT e $k - i_0$ o número de folhas em RT.)

d. Prove que, para um dado valor de $k > 1$ e i no intervalo $1 \le i \le k - 1$, a função $i \lg i + (k - i) \lg(k - i)$ é minimizada em $i = k/2$. Conclua que $d(k) = \Omega(k \lg k)$.

e. Prove que $D(T_A) = \Omega(n! \lg(n!))$ e conclua que o tempo esperado para ordenar n elementos é $\Omega(n \lg n)$.

Agora, considere uma ordenação por comparação *aleatória* B. Podemos estender o modelo de árvore de decisão para tratar a aleatoriedade, incorporando dois tipos de nós: os nós de comparação comum e os nós "aleatórios". Um nó aleatório modela uma opção aleatória da forma RANDOM $(1, r)$ feita pelo algoritmo B; o nó tem r filhos, cada um dos quais tem igual probabilidade de ser escolhido durante uma execução do algoritmo.

f. Mostre que, para qualquer ordenação por comparação aleatória B, existe uma ordenação por comparação determinística A que não faz mais comparações sobre a média que B.

8-2 Ordenação local em tempo linear

Vamos supor que temos um arranjo de n registros de dados para ordenar e que a chave de cada registro tem o valor 0 ou 1. Um algoritmo para ordenar tal conjunto de registros poderia ter algum subconjunto das três características desejáveis a seguir:

1. O algoritmo é executado no tempo $O(n)$.
2. O algoritmo é estável.
3. O algoritmo ordena localmente, sem utilizar mais que uma quantidade constante de espaço de armazenamento além do arranjo original.

a. Dê um algoritmo que satisfaça aos critérios 1 e 2 anteriores.

b. Dê um algoritmo que satisfaça aos critérios 1 e 3 anteriores.

c. Dê um algoritmo que satisfaça aos critérios 2 e 3 anteriores.

d. Algum dos seus algoritmos de ordenação das partes (a)-(c) pode ser usado para ordenar n registros com chaves de b bits usando radix sort no tempo $O(bn)$? Explique como ou por que não.

e. Suponha que os n registros tenham chaves no intervalo de 1 a k. Mostre como modificar a ordenação por contagem de tal forma que os registros possam ser ordenados localmente no tempo $O(n + k)$. Você pode usar o espaço de armazenamento $O(k)$ fora do arranjo de entrada. Seu algoritmo é estável? (*Sugestão*: Como você faria isso para $k = 3$?)

8-3 Ordenação de itens de comprimento variável

a. Você tem um arranjo de inteiros, no qual diferentes inteiros podem ter números de dígitos distintos, mas o número total de dígitos sobre *todos* os inteiros no arranjo é n. Mostre como ordenar o arranjo no tempo $O(n)$.

b. Você tem um arranjo de cadeias, no qual diferentes cadeias podem ter números de caracteres distintos, mas o número total de caracteres em todas as cadeias é n. Mostre como ordenar as cadeias no tempo $O(n)$.

(Observe que a ordem desejada aqui é a ordem alfabética padrão; por exemplo, `a < ab < b`.)

8-4 Jarros de água

Vamos supor que você tem n jarros de água vermelhos e n jarros azuis, todos de diferentes formas e tamanhos. Todos os jarros vermelhos contêm quantidades diferentes de água, como também os jarros azuis. Além disso, para todo jarro vermelho, existe um jarro azul que contém a mesma quantidade de água e vice-versa.

Sua tarefa é encontrar um agrupamento dos jarros em pares de jarros vermelhos e azuis que contêm a mesma quantidade de água. Para isso, você pode executar a seguinte operação: escolher um par de jarros no qual um é vermelho e um é azul, encher o jarro vermelho com água, e depois despejar a água no jarro azul. Essa operação lhe informará se o jarro vermelho ou o jarro azul pode conter mais água, ou se eles têm o mesmo volume. Suponha que tal comparação demore uma unidade de tempo. Seu objetivo é encontrar um algoritmo que faça um número mínimo de comparações para determinar o agrupamento. Lembre-se de que você não pode comparar diretamente dois jarros vermelhos ou dois jarros azuis.

a. Descreva um algoritmo determinístico que use $\Theta(n^2)$ comparações para agrupar os jarros em pares.

b. Prove um limite inferior $\Omega(n \lg n)$ para o número de comparações que um algoritmo que resolve esse problema deve efetuar.

c. Dê um algoritmo aleatório cujo número esperado de comparações seja $O(n \lg n)$ e prove que esse limite é correto. Qual é o número de comparações no pior caso do seu algoritmo?

8-5 Ordenação por média

Suponha que, em vez de ordenar um arranjo, simplesmente exigimos que os elementos aumentem na média. De modo mais preciso, chamamos um arranjo de n elementos A de ***k*-ordenado** se, para todo $i = 1, 2, \ldots, n - k$, é válida a desigualdade a seguir:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. O que significa um arranjo ser 1-ordenado?

b. Forneça uma permutação dos números 1, 2, ..., 10 que seja 2-ordenada, mas não ordenada.

c. Prove que um arranjo de n elementos é k-ordenado se e somente se $A[i] \le A[i + k]$ para todo $i = 1, 2, \ldots, n - k$.

d. Forneça um algoritmo que faça a k-ordenação de um arranjo de n elementos no tempo $O(n \lg(n/k))$.

Também podemos mostrar um limite inferior sobre o tempo para produzir um arranjo k-ordenado, quando k é uma constante.

e. Mostre que um arranjo k-ordenado de comprimento n pode ser ordenado no tempo $O(n \lg k)$. (*Sugestão*: Use a solução do Exercício 6.5-8.)

f. Mostre que, quando k é uma constante, é necessário o tempo $\Omega(n \lg n)$ para fazer a k-ordenação de um arranjo de n elementos. (*Sugestão*: Use a solução para a parte anterior, juntamente com o limite inferior sobre ordenações por comparação.)

8-6 Limite inferior sobre a intercalação de listas ordenadas

O problema de intercalar duas listas ordenadas surge com freqüência. Ele é usado como uma sub-rotina de MERGE-SORT, e o procedimento para intercalar duas listas ordenadas é dado como MERGE na Seção 2.3.1. Neste problema, mostraremos que existe um limite inferior $2n - 1$ sobre o número de comparações no pior caso exigidas para intercalar duas listas ordenadas, cada uma contendo n itens.

Primeiro, mostraremos um limite inferior de $2n - o(n)$ comparações, usando uma árvore de decisão.

a. Mostre que, dados $2n$ números, existem $\binom{2n}{n}$ maneiras possíveis de dividi-los em duas listas ordenadas, cada uma com n números.

b. Usando uma árvore de decisão, mostre que qualquer algoritmo que intercala corretamente duas listas ordenadas utiliza pelo menos $2n - o(n)$ comparações.

Agora, mostraremos um limite $2n - 1$, ligeiramente mais restrito.

c. Mostre que, se dois elementos são consecutivos na seqüência ordenada e vêm de listas opostas, então eles devem ser comparados.

d. Use sua resposta à parte anterior para mostrar um limite inferior de $2n - 1$ comparações para intercalar duas listas ordenadas.

Notas do capítulo

O modelo de árvore de decisão para o estudo de ordenações por comparação foi introduzido por Ford e Johnson [94]. O tratamento abrangente de Knuth sobre a ordenação [185] cobre muitas variações sobre o problema da ordenação, inclusive o limite inferior teórico de informações sobre a complexidade da ordenação fornecida aqui. Os limites inferiores para ordenação com o uso de generalizações do modelo de árvore de decisão foram estudados amplamente por Ben-Or [36].

Knuth credita a H. H. Seward a criação da ordenação por contagem em 1954, e também a idéia de combinar a ordenação por contagem com a radix sort. A radix sort começando pelo dígito menos significativo parece ser um algoritmo popular amplamente utilizado por operadores de máquinas mecânicas de ordenação de cartões. De acordo com Knuth, a primeira referência ao método publicada é um documento de 1929 escrito por L. J. Comrie que descreve o equipamento de perfuração de cartões. A bucket sort esteve em uso desde 1956, quando a idéia básica foi proposta por E. J. Isaac e R. C. Singleton.

Munro e Raman [229] fornecem um algoritmo de ordenação estável que executa $O(n^{1+\varepsilon})$ comparações no pior caso, onde $0 < \varepsilon \leq 1$ é qualquer constante fixa. Embora qualquer dos algoritmos de tempo $O(n \lg n)$ efetue um número menor de comparações, o algoritmo de Munro e Raman move os dados apenas $O(n)$ vezes e opera localmente.

O caso de ordenar n inteiros de b bits no tempo $o(n \lg n)$ foi considerado por muitos pesquisadores. Vários resultados positivos foram obtidos, cada um sob hipóteses um pouco diferentes a respeito do modelo de computação e das restrições impostas sobre o algoritmo. Todos os resultados pressupõem que a memória do computador está dividida em palavras endereçáveis de b bits. Fredman e Willard [99] introduziram a estrutura de dados de árvores de fusão e a empregaram para ordenar n inteiros no tempo $O(n \lg n / \lg \lg n)$. Esse limite foi aperfeiçoado mais tarde para o tempo $O(n\sqrt{\lg n})$ por Andersson [16]. Esses algoritmos exigem o uso de multiplicação e de várias constantes pré-calculadas. Andersson, Hagerup, Nilsson e Raman [17] mostraram como ordenar n inteiros no tempo $O(n \lg \lg n)$ sem usar multiplicação, mas seu método exige espaço de armazenamento que pode ser ilimitado em termos de n. Usando-se o hash multiplicativo, é possível reduzir o espaço de armazenamento necessário para $O(n)$, mas o limite $O(n \lg \lg n)$ do pior caso sobre o tempo de execução se torna um limite de tempo esperado. Generalizando as árvores de pesquisa exponencial de Andersson [16], Thorup [297] forneceu um algoritmo de ordenação de tempo $O(n(\lg \lg n)^2)$ que não usa multiplicação ou aleatoriedade, e que utiliza espaço linear. Combinando essas técnicas com algumas idéias novas, Han [137] melhorou o limite para ordenação até o tempo $O(n \lg \lg n \lg \lg \lg n)$. Embora esses algoritmos sejam inovações teóricas importantes, todos eles são bastante complicados e neste momento parece improvável que venham a competir na prática com algoritmos de ordenação existentes.