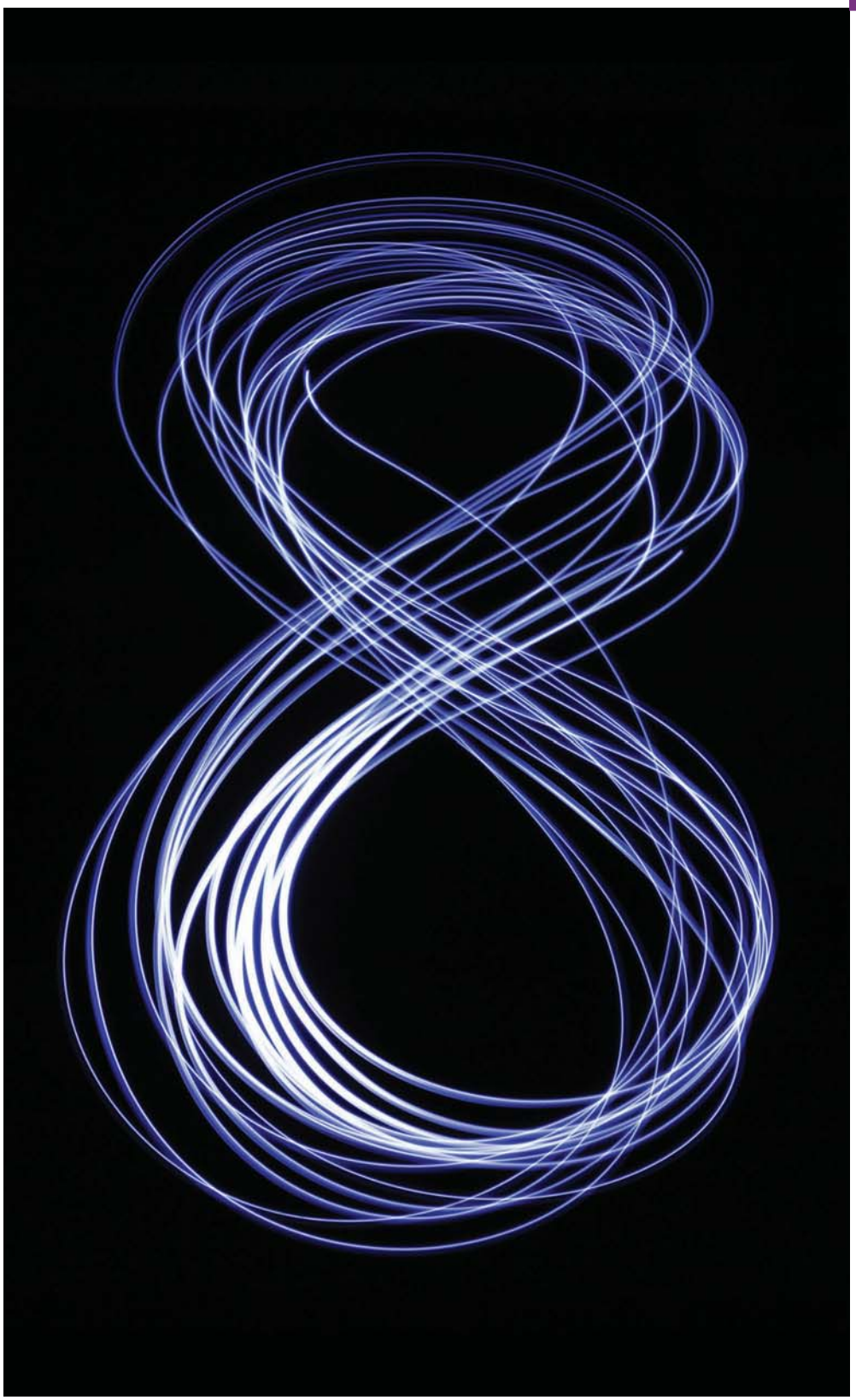
19

Pesquisa, classificação e Big O



Entre lágrimas e soluços,
escolheu as de maior tamanho...
— Lewis Carroll

Tente o último e nunca duvides; Nada é tão difícil, mas a pesquisa o encontrará.
— Robert Herrick

Guardei-o na memória,
e a chave a levas.
— William Shakespeare

É uma lei imutável nos negócios que palavras são palavras, explicações são explicações, promessas são promessas — mas somente desempenho é realidade.
— Harold S. Green

Objetivos

Neste capítulo, você irá:

- Procurar um dado valor em um array usando pesquisa linear e pesquisa binária.
- Classificar arrays usando os algoritmos de classificação por inserção e seleção iterativa.
- Classificar arrays usando o algoritmo recursivo de classificação por intercalação.
- Determinar a eficiência dos algoritmos de pesquisa e de classificação.
- Introduzir a notação Big O para comparar a eficiência dos algoritmos.

Sumário

19.1 Introdução

Pesquisar dados envolve determinar se um valor (chamado de **chave de pesquisa**) está presente nos dados e, se estiver, encontrar a sua localização. Dois algoritmos de pesquisa famosos são a pesquisa linear simples e a mais rápida, porém mais complexa, pesquisa binária. A **classificação** coloca os dados em uma ordem crescente ou decrescente, com base em uma ou mais **chaves de classificação**. Uma lista de nomes poderia ser classificada alfabeticamente, contas bancárias poderiam ser classificadas pelo número de conta, registros de folha de pagamento de funcionários poderiam ser classificados pelo CPF e assim por diante. Este capítulo introduz dois algoritmos simples de classificação: por seleção e por inserção, juntamente com a classificação por intercalação, mais eficiente, porém mais complexa. A Figura 19.1 resume os algoritmos de pesquisa e classificação discutidos nos exemplos e exercícios deste livro.

Observação de engenharia de software 19.1

Em aplicativos que exigem pesquisa e classificação, utilize as capacidades predefinidas da API Java Collections (Capítulo 16). As técnicas apresentadas neste capítulo são fornecidas para introduzir os alunos aos conceitos por trás dos algoritmos de pesquisa e classificação — cursos de nível superior de ciência da computação tipicamente discutem esses algoritmos em detalhes.

Capítulo	Algoritmo	Posição
Algoritmos de pesquisa:		
16	Método `binarySearch` da classe `Collections`	Figura 16.12
19	Pesquisa linear	Seção 19.2
	Pesquisa binária	Seção 19.4
	Pesquisa linear recursiva	Exercício 19.8
	Pesquisa binária recursiva	Exercício 19.9
21	Pesquisa linear em uma `List`	Exercício 21.21
	Pesquisa em árvore binária	Exercício 21.23
Algoritmos de classificação:		
16	Método `sort` da classe `Collections`	Figuras 16.6 a 16.9
	Coleção `SortedSet`	Figura 16.17
19	Classificação por seleção	Seção 19.6
	Classificação por inserção	Seção 19.7
	Classificação por intercalação recursiva	Seção 19.8
	Classificação por borbulhamento	Exercícios 19.5 e 19.6
	Classificação de Bucket	Exercício 19.7
	Quicksort recursivo	Exercício 19.10
21	Classificação em árvore binária	Seção 21.7

Figura 19.1 | Algoritmos de pesquisa e classificação abordados neste texto.

19.2 Pesquisa linear

Pesquisar um número de telefone, localizar um site por um mecanismo de pesquisa e verificar a definição de uma palavra em um dicionário são todas operações que envolvem pesquisar grandes volumes de dados. Esta seção e a Seção 19.4 discutem dois algoritmos de pesquisa comuns — um que é fácil de programar, mas relativamente ineficiente (pesquisa linear) e outro que é relativamente eficiente, mas mais complexo de programar (pesquisa binária).

Algoritmo de pesquisa linear

O **algoritmo de pesquisa linear** pesquisa cada elemento em um array sequencialmente. Se a chave de pesquisa não corresponder a um elemento no array, o algoritmo testa cada elemento e, quando alcança o fim do array, informa o usuário que a chave de pesquisa não está presente. Se a chave de pesquisa estiver no array, o algoritmo testa cada elemento até encontrar um que corresponda à chave de pesquisa e retorna o índice desse elemento.

Como um exemplo, considere um array que contém os valores a seguir

```
34   56   2   10   77   51   93   30   5   52
```

e um programa que pesquisa 51. Utilizando o algoritmo de pesquisa linear, o programa primeiro verifica se 34 corresponde à chave de pesquisa. Se não corresponder, o algoritmo então verifica se 56 corresponde à chave de pesquisa. O programa continua a percorrer o array sequencialmente, testando 2, 10 e então 77. Quando o programa testa 51, que corresponde à chave de pesquisa, o programa retorna o índice 5, que é a localização do 51 no array. Se, depois de verificar cada elemento do array, o programa determinar que a chave de pesquisa não corresponde a nenhum elemento no array, ele retorna um valor de sentinela (por exemplo, `-1`).

Implementação da pesquisa linear

A classe `LinearSearchTest` (Figura 19.2) contém o método `static` `linearSearch` para realizar pesquisas de um array `int` e `main` para testar `linearSearch`.

```
1  // Figura 19.2: LinearSearchTest.java
2  // Pesquisando sequencialmente um item em um array.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class LinearSearchTest
8  {
9     // realiza uma pesquisa linear nos dados
10    public static int linearSearch(int data[], int searchKey)
11    {
12       // faz um loop pelo array sequencialmente
13       for (int index = 0; index < data.length; index++)
14          if (data[index] == searchKey)
15             return index; // retorna o índice de inteiros
16
17       return -1; // inteiro não foi localizado
18    } // fim do método linearSearch
19
20    public static void main(String[] args)
21    {
22       Scanner input = new Scanner(System.in);
23       SecureRandom generator = new SecureRandom();
24
25       int[] data = new int[10]; // cria o array
26
27       for (int i = 0; i < data.length; i++) // preenche o array
28          data[i] = 10 + generator.nextInt(90);
29
30       System.out.printf("%s%n%n", Arrays.toString(data)); // exibe o array
31
32       // obtém a entrada de usuário
33       System.out.print("Please enter an integer value (-1 to quit): ");
34       int searchInt = input.nextInt();
35
36       // insere repetidamente um inteiro; -1 termina o programa
37       while (searchInt != -1)
38       {
39          int position = linearSearch(data, searchInt); // realiza a pesquisa
40
```

continua

continuação

```
41             if (position == -1) // não encontrado
42                System.out.printf("%d was not found%n%n", searchInt);
43             else // encontrado
44                System.out.printf("%d was found in position %d%n%n",
45                   searchInt, position);
46
47             // obtém a entrada de usuário
48             System.out.print("Please enter an integer value (-1 to quit): ");
49             searchInt = input.nextInt();
50          }
51       } // fim de main
52    } // fim da classe LinearSearchTest
```

```
[59, 97, 34, 90, 79, 56, 24, 51, 30, 69]

Please enter an integer value (-1 to quit): 79
79 was found in position 4

Please enter an integer value (-1 to quit): 61
61 was not found

Please enter an integer value (-1 to quit): 51
51 was found in position 7

Please enter an integer value (-1 to quit): -1
```

Figura 19.2 | Pesquisando sequencialmente um item em um array.

Método `linearSearch`

O método `linearSearch` (linhas 10 a 18) realiza a pesquisa linear. O método recebe como parâmetros o array para pesquisar (`data`) e a `searchKey`. As linhas 13 a 15 fazem um loop pelos elementos no array `data`. A linha 14 compara cada um com `searchKey`. Se os valores forem iguais, a linha 15 retornará o *índice* do elemento. Se houver valores *duplicados* no array, a pesquisa linear retorna o índice do *primeiro* elemento no array que corresponde à chave de pesquisa. Se o loop terminar sem encontrar o valor, a linha 17 retornará `-1`.

Método `main`

O método `main` (linhas 20 a 51) permite ao usuário pesquisar um array. As linhas 25 a 28 criam um array de `10` `int`s e o preenchem com `int`s aleatórios de `10` a `99`. Então, a linha 30 exibe o conteúdo do array usando o método `Arrays` `static` `toString`, que retorna uma representação `String` do array com os elementos entre colchetes (`[e]`) e separados por vírgulas.

As linhas 33 e 34 solicitam ao usuário e armazenam a chave de pesquisa. A linha 39 chama o método `linearSearch` para determinar se `searchInt` está no array `data`. Se não, `linearSearch` retorna `-1` e o programa notifica o usuário (linhas 41 e 42). Se `searchInt` estiver no array, `linearSearch` retorna a posição do elemento, para o qual o programa gera a saída nas linhas 44 e 45. As linhas 48 e 49 obtêm a próxima chave de pesquisa do usuário.

19.3 Notação Big O

Todos os algoritmos de pesquisa têm o *mesmo* objetivo — localizar um elemento (ou elementos) que corresponde a uma dada chave de pesquisa se esse elemento, de fato, existir. Há, porém, alguns aspectos que diferenciam os algoritmos de pesquisa uns dos outros. *A principal diferença é o esforço que eles exigem para completar a pesquisa.* Uma maneira de descrever esse esforço é com a **notação Big O**, que indica o quanto um algoritmo precisa trabalhar para resolver um problema. Para algoritmos de pesquisa e de classificação, isso depende particularmente de quantos elementos de dados há. Neste capítulo, usamos Big O para descrever os casos de pior cenário de tempo de execução para vários algoritmos de pesquisa e classificação.

19.3.1 Algoritmos O(1)

Suponha que um algoritmo seja projetado para testar se o primeiro elemento de um array é igual ao segundo. Se o array tiver 10 elementos, esse algoritmo exigirá uma comparação. Se o array tiver 1.000 elementos, ele ainda exigirá uma comparação. Na realidade, o algoritmo é completamente independente do número de elementos no array. Diz-se que esse algoritmo tem um **tempo de execução constante**, que é representado na notação Big O como ***O*(1)** e pronunciado como "ordem um". Um algoritmo que é *O*(1) não necessariamente exige somente uma comparação. *O*(1) simplesmente significa que o número de comparações é *constante* — não aumenta à medida que o tamanho do array aumenta. Um algoritmo que testa se o primeiro elemento de um array é igual a qualquer um dos três próximos elementos ainda é *O*(1) mesmo se exigir três comparações.

19.3.2 Algoritmos O(*n*)

Um algoritmo que testa se o primeiro elemento do array é igual a *qualquer um* dos outros elementos do array irá requerer no máximo $n - 1$ comparações, onde n é o número de elementos do array. Se o array tiver 10 elementos, esse algoritmo exigirá até nove comparações. Se o array tiver 1.000 elementos, ele exigirá até 999 comparações. À medida que n aumenta, a parte n da expressão "predomina", e subtrair uma torna-se irrelevante. A notação Big O é projetada para destacar esses termos dominantes e ignorar termos que não têm importância à medida que n aumenta. Por essa razão, diz-se que um algoritmo que exige um total de $n - 1$ comparações (como aquele que descrevemos anteriormente) é dito ser ***O*(*n*)**. Diz-se que um algoritmo $O(n)$ tem um tempo de **execução linear**. $O(n)$ costuma ser pronunciado "na ordem de n" ou, simplesmente, "ordem n".

19.3.3 Algoritmos O(n^2)

Agora suponha que você tem um algoritmo que testa se *qualquer* elemento de um array é duplicado em uma outra parte no array. O primeiro elemento deve ser comparado com elementos alternados no array. O segundo elemento deve ser comparado com elementos alternados, exceto o primeiro (já foi comparado com o primeiro). O terceiro elemento deve ser comparado com elementos alternados, exceto os dois primeiros. No final, esse algoritmo terminará fazendo $(n - 1) + (n - 2) + \ldots + 2 + 1$ ou $n^2/2 - n/2$ comparações. À medida que n aumenta, o termo n^2 predomina e o termo n torna-se irrelevante. Mais uma vez, a notação Big O destaca o termo n^2, deixando $n/2$. Mas, como veremos a seguir, fatores constantes são omitidos na notação Big O.

A notação Big O considera como o tempo de execução de um algoritmo aumenta em relação ao número de itens processado. Suponha que um algoritmo exija n^2 comparações. Com quatro elementos, o algoritmo requer 16 comparações; com oito elementos, 64 comparações. Com esse algoritmo, *dobrar* o número de elementos *quadruplica* o número de comparações. Considere um algoritmo semelhante que exige $n^2/2$ comparações. Com quatro elementos, o algoritmo requer oito comparações; com oito elementos, 32 comparações. Mais uma vez, *dobrar* o número de elementos *quadruplica* o número de comparações. Esses dois algoritmos aumentam conforme o quadrado de n, assim o Big O ignora a constante e os dois algoritmos são considerados como ***O*(n^2)**, o que é chamado **tempo de execução quadrático**, pronunciado como "na ordem de n ao quadrado" ou, mais simplesmente, "ordem do quadrado de n".

Quando n é pequeno, algoritmos $O(n^2)$ (executados nos computadores de hoje em dia) não afetarão significativamente o desempenho. Mas, à medida que n aumentar, você começará a observar a degradação de desempenho. Um algoritmo $O(n^2)$ executando em um array de um milhão de elementos exigiria um trilhão de "operações" (onde cada uma, na verdade, exigiria várias instruções de máquina para executar). Isso pode levar várias horas. Um array de um bilhão de elementos exigiria um quintilhão de operações, um número tão grande que o algoritmo demoraria décadas para executar! Infelizmente, algoritmos $O(n^2)$ são fáceis de escrever, como você verá neste capítulo. Você também verá algoritmos com medidas Big O mais favoráveis. Frequentemente, esses eficientes algoritmos demandam um pouco mais de perícia e trabalho para serem criados, mas seu desempenho superior recompensa muito bem o esforço extra, especialmente à medida que o n torna-se grande e algoritmos são combinados em programas maiores.

19.3.4 Big O da pesquisa linear

O algoritmo de pesquisa linear executa a $O(n)$ vezes. O pior caso nesse algoritmo é que cada elemento deve ser verificado para determinar se o item de pesquisa existe no array. Se o tamanho do array for *dobrado*, o número de comparações que o algoritmo deve realizar também será *dobrado*. A pesquisa linear pode fornecer um excelente desempenho se o elemento que corresponde à chave de pesquisa estiver próximo ou no início do array. Mas buscamos algoritmos que, em média, tenham um bom desempenho em *todas* as pesquisas, incluindo aqueles em que o elemento que corresponde à chave de pesquisa está próximo do final do array.

A pesquisa linear é fácil de programar, mas pode ser lenta em comparação com outros algoritmos de pesquisa. Se um programa precisar realizar muitas pesquisas em grandes arrays, é melhor implementar um algoritmo mais eficiente, como a pesquisa binária que apresentaremos a seguir.

Dica de desempenho 19.1

Às vezes os algoritmos mais simples demonstram um fraco desempenho. Sua virtude é que são fáceis de programar, testar e depurar. Às vezes, algoritmos mais complexos são necessários para conseguir desempenho máximo.

19.4 Pesquisa binária

O **algoritmo de pesquisa binária** é mais eficiente que o de pesquisa linear, mas exige que o array seja classificado. A primeira iteração desse algoritmo testa o elemento no *meio* do array. Se isso corresponder à chave de pesquisa, o algoritmo termina. Supondo que o array seja classificado em ordem *crescente*, se a chave de pesquisa for *menor que* o elemento do meio, ela não poderá localizar nenhum elemento na segunda metade do array e o algoritmo continua com apenas a primeira metade do array (isto é, até o primeiro elemento, mas sem incluir o elemento do meio). Se a chave de pesquisa for *maior que* o elemento no meio, ela não poderá localizar nenhum elemento na primeira metade do array e o algoritmo continua apenas com a segunda metade (isto é, o elemento *depois* do

elemento do meio até o último elemento). Cada iteração testa o valor do meio da parte restante do array. Se a chave de pesquisa não corresponder ao elemento, o algoritmo eliminará metade dos elementos restantes. O algoritmo termina localizando um elemento que corresponde à chave de pesquisa ou reduzindo o subarray ao tamanho zero.

Exemplo

Como um exemplo, considere o array de 15 elementos classificado

```
2  3  5  10  27  30  34  51  56  65  77  81  82  93  99
```

e uma chave de pesquisa de 65. Um programa que implementa o algoritmo de pesquisa binária primeiro verificaria se 51 é a chave de pesquisa (uma vez que 51 é o elemento no *meio* do array). A chave de pesquisa (65) é maior que 51, assim 51 é ignorado junto com a primeira metade do array (todos os elementos menores que 51), deixando

```
56  65  77  81  82  93  99
```

Em seguida, o algoritmo verifica se 81 (o elemento no meio do restante do array) corresponde à chave de pesquisa. A chave de pesquisa (65) é menor que 81, portanto 81 é descartado junto com os elementos maiores que 81. Depois de apenas dois testes, o algoritmo reduziu a somente três o número de valores a verificar (56, 65 e 77). Ele, então, verifica 65 (que de fato corresponde à chave de pesquisa) e retorna o índice do elemento no array que contém 65. Esse algoritmo exigiu apenas três comparações para determinar se a chave de pesquisa localizou um elemento do array. Utilizar um algoritmo de pesquisa linear exigiria 10 comparações. [*Observação*: neste exemplo, optamos por utilizar um array com 15 elementos, de modo que sempre haverá um elemento óbvio no meio do array. Com um número par de elementos, o meio do array reside entre dois elementos. Implementamos o algoritmo para escolher o maior desses dois elementos.]

19.4.1 Implementação de pesquisa binária

A classe `BinarySearchTest` (Figura 19.3) contém:

- O método `static` `binarySearch` para pesquisar em um array `int` uma chave especificada.
- O método `static` `remainingElements` para exibir os elementos remanescentes no array que está sendo pesquisado.
- `main` para testar o método `binarySearch`.

O método `main` (linhas 57 a 90) é quase idêntico ao `main` na Figura 19.2. Nesse programa, criamos um array de 15 elementos (linha 62) e a linha 67 chama o método `static` **sort** da classe `Arrays` para classificar os elementos `data` do array em um array em ordem crescente (por padrão). Lembre-se de que o algoritmo de pesquisa binária só funcionará em um array classificado. A primeira linha da saída desse programa mostra o array classificado de `ints`. Quando o usuário instrui o programa a pesquisar `18`, o programa primeiro testa o elemento do meio, que é `57` (como indicado por `*`). A chave de pesquisa é menor que `57`, assim o programa elimina a segunda metade do array e testa o elemento do meio na primeira metade. A chave de pesquisa é menor que `36`, portanto o programa elimina a segunda metade do array, deixando somente três elementos. Por fim, o programa verifica `18` (que corresponde à chave de pesquisa) e retorna o índice `1`.

```
1  // Figura 19.3: BinarySearchTest.java
2  // Utiliza a pesquisa binária para localizar um item em um array.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class BinarySearchTest
8  {
9     // realiza uma pesquisa binária sobre os dados
10    public static int binarySearch(int[] data, int key)
11    {
12       int low = 0; // extremidade baixa da área de pesquisa
13       int high = data.length - 1; // extremidade alta da área de pesquisa
14       int middle = (low + high + 1) / 2; // elemento do meio
15       int location = -1; // valor de retorno; -1 se não localizado
16
17       do // faz um loop para procurar o elemento
18       {
19          // imprime os elementos remanescentes do array
20          System.out.print(remainingElements(data, low, high));
21
22          // gera espaços para alinhamento
23          for (int i = 0; i < middle; i++)
```

continua

continuação

```
24              System.out.print("   ");
25           System.out.println(" * "); // indica o meio atual
26
27           // se o elemento for localizado no meio
28           if (key == data[middle])
29              location = middle; // a localização é o meio atual
30           else if (key < data[middle]) // elemento do meio é muito alto
31              high = middle - 1; // elimina a metade mais alta
32           else  // elemento do meio é muito baixo
33              low = middle + 1; // elimina a metade mais alta
34
35           middle = (low + high + 1) / 2; // recalcula o meio
36        } while ((low <= high) && (location == -1));
37
38        return location; // retorna a localização da chave de pesquisa
39     } // fim do método binarySearch
40
41     // método para gerar saída de certos valores no array
42     private static String remainingElements(int[] data, int low, int high)
43     {
44        StringBuilder temporary = new StringBuilder();
45
46        // acrescenta espaços para alinhamento
47        for (int i = 0; i < low; i++)
48           temporary.append("   ");
49
50        // gera a saída dos elementos que permanecem no array
51        for (int i = low; i <= high; i++)
52           temporary.append(data[i] + " ");
53
54        return String.format("%s%n", temporary);
55     } // fim do método remainingElements
56
57     public static void main(String[] args)
58     {
59        Scanner input = new Scanner(System.in);
60        SecureRandom generator = new SecureRandom();
61
62        int[] data = new int[15]; // cria o array
63
64        for (int i = 0; i < data.length; i++) // preenche o array
65           data[i] = 10 + generator.nextInt(90);
66
67        Arrays.sort(data); // binarySearch requer array classificado
68        System.out.printf("%s%n%n", Arrays.toString(data)); // exibe o array
69
70        // obtém a entrada de usuário
71        System.out.print("Please enter an integer value (-1 to quit): ");
72        int searchInt = input.nextInt();
73
74        // insere repetidamente um inteiro; -1 termina o programa
75        while (searchInt != -1)
76        {
77           // realiza a pesquisa
78           int location = binarySearch(data, searchInt);
79
80           if (location == -1) // não encontrado
81              System.out.printf("%d was not found%n%n", searchInt);
82           else // encontrado
83              System.out.printf("%d was found in position %d%n%n",
84                 searchInt, location);
85
86           // obtém a entrada de usuário
87           System.out.print("Please enter an integer value (-1 to quit): ");
88           searchInt = input.nextInt();
89        }
90     } // fim de main
91  } // fim da classe BinarySearchTest
```

continua

continuação

```
[13, 18, 29, 36, 42, 47, 56, 57, 63, 68, 80, 81, 82, 88, 88]

Please enter an integer value (-1 to quit): 18
13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                  *
13 18 29 36 42 47 56
         *
13 18 29
   *
18 was found in position 1

Please enter an integer value (-1 to quit): 82
13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                  *
                    63 68 80 81 82 88 88
                             *
                               82 88 88
                                  *
                               82
                                *

82 was found in position 12

Please enter an integer value (-1 to quit): 69
13 18 29 36 42 47 56 57 63 68 80 81 82 88 88
                  *
                    63 68 80 81 82 88 88
                             *
                    63 68 80
                        *
                          80
                           *
69 was not found

Please enter an integer value (-1 to quit): -1
```

Figura 19.3 | Use a pesquisa binária para localizar um item em um array (o * na saída marca o elemento do meio).

As linhas 10 a 39 declaram o método `binarySearch`, que recebe como parâmetros o array a pesquisar (`data`) e a chave de pesquisa (`key`). As linhas 12 a 14 calculam o índice da extremidade `low`, o índice da extremidade `high` índice `middle` da parte do array em que o programa está atualmente pesquisando. No início do método, a extremidade `low` é `0`, a extremidade `high` é o comprimento do array menos `1` e `middle` é a média desses dois valores. A linha 15 inicializa a `location` do elemento para `-1` — o valor que será retornado se o `key` não for localizado. As linhas 17 a 36 fazem um loop até que `low` seja maior que `high` (isso ocorre quando a `key` não é localizada) ou a `key` não é igual a `-1` (indicando que a `key` de pesquisa foi localizada). A linha 28 testa se o valor no elemento `middle` é igual a `key`. Se sim, a linha 29 atribui `middle` a `location`, o loop termina e `location` é retornada ao chamador. Cada iteração do loop testa um valor único (linha 28) e *elimina metade dos valores restantes no array* (linha 31 ou 33) se o valor não for a `key`.

19.4.2 Eficiência da pesquisa binária

No cenário do pior caso, pesquisar um array *classificado* de 1.023 elementos leva *apenas 10 comparações* quando utilizamos uma pesquisa binária. Dividir repetidamente 1.023 por 2 (porque depois de cada comparação podemos eliminar metade do array) e arredondar para baixo (porque também removemos o elemento do meio) produz os valores 511, 255, 127, 63, 31, 15, 7, 3, 1 e 0. O número 1.023 ($2^{10} - 1$) é dividido por 2 apenas 10 vezes para obter o valor 0, o que indica que não há mais elementos a testar. Dividir por 2 é equivalente a uma comparação no algoritmo de pesquisa binária. Portanto, um array de 1.048.575 ($2^{20} - 1$) elementos exige no *máximo 20 comparações* para localizar a chave e um array de mais de um bilhão de elementos demanda no *máximo 30 comparações* para localizar a chave. Isso é uma tremenda melhoria no desempenho em relação à pesquisa linear. Para um array de um bilhão de elementos, isso representa uma diferença entre uma média de 500 milhões de comparações para a pesquisa linear e no máximo apenas 30 comparações para a pesquisa binária! O número máximo de comparações necessárias para a pesquisa binária de qualquer array classificado é o expoente da primeira potência de 2 maior que o número de elementos no array, que é representado como $\log_2 n$. Todos os logaritmos crescem aproximadamente na mesma taxa, assim na notação Big O a base pode ser omitida. Isso resulta em um Big O de ***O*(log *n*)** para uma pesquisa binária, que também é conhecida como **tempo de execução logarítmico** e pronunciada como "log ordem n".

19.5 Algoritmos de classificação

Classificar dados (isto é, colocar os dados em alguma ordem particular como crescente ou decrescente) é uma das aplicações mais importantes da computação. Um banco classifica todos os cheques pelo número de conta, de modo que possa preparar extratos bancários individuais no final de cada mês. As companhias telefônicas classificam suas listas de assinantes por sobrenome e, depois, pelo primeiro nome para facilitar a localização de números de telefone. Praticamente todas as empresas devem classificar alguns dados e, muitas vezes, volumes maciços deles. Classificar dados é um problema intrigante, que faz uso intensivo do computador e que atrai esforços intensos de pesquisa.

Um item importante a entender sobre a classificação é que o resultado final — o array classificado — será o *mesmo*, independentemente do algoritmo que você utiliza para classificar o array. A escolha do algoritmo só afeta o tempo de execução e uso de memória do programa. O restante deste capítulo apresenta três algoritmos de classificação comuns. As duas primeiras — *classificação por seleção* e *classificação por inserção* — são fáceis de programar, mas *ineficientes*. O último algoritmo — *a classificação por intercalação* — é muito mais *rápido* do que a classificação por seleção e a classificação por inserção, mas é mais *difícil de programar*. Focalizamos a classificação de arrays de dados de tipo primitivo, ou seja, `int`s. Também é possível classificar arrays de objetos de classe, como demonstrado na Seção 16.7.1, usando as capacidades de classificação predefinidas da API Collections.

19.6 Classificação por seleção

Classificação por seleção é um algoritmo de classificação simples, mas ineficiente. Se você classificar em ordem crescente, a primeira iteração seleciona o *menor* elemento no array e o permuta pelo primeiro elemento. A segunda iteração seleciona o *segundo menor* item (que é o menor item dos elementos restantes) e troca-o pelo segundo elemento. O algoritmo continua até que a última iteração selecione o *segundo maior* elemento e permute-o pelo penúltimo índice, deixando o maior elemento no último índice. Depois da *i*-ésima iteração, os menores itens *i* do array serão classificados na ordem crescente nos primeiros elementos *i* do array.

Como um exemplo, considere o array

```
34   56   4   10   77   51   93   30   5   52
```

Um programa que implementa a classificação por seleção primeiro determina o menor elemento (4) desse array, que está contido no índice 2. O programa troca 4 por 34, resultando em

```
4   56   34   10   77   51   93   30    5   52
```

O programa então determina o menor valor dos elementos restantes (todos os elementos, exceto 4), que é 5, contido no índice 8. O programa troca 5 por 56, resultando em

```
4   5   34   10   77   51   93   30   56   52
```

Na terceira iteração, o programa determina o próximo menor valor (10) e o troca por 34.

```
4   5   10   34   77   51   93   30   56   52
```

O processo continua até que o array seja completamente classificado.

```
4   5   10   30   34   51   52   56   77   93
```

Depois da primeira iteração, o menor elemento está na primeira posição. Depois da segunda iteração, os dois menores elementos estarão na ordem nas duas primeiras posições. Depois da terceira iteração, os três menores elementos estarão na ordem nas três primeiras posições.

19.6.1 Implementação da classificação por seleção

A classe `SelectionSortTest` (Figura 19.4) contém:

- O método `static` `selectionSort` para classificar um array `int` usando o algoritmo de classificação por seleção.
- O método `static` `swap` para permutar os valores dos dois elementos do array.
- O método `static` `printPass` para exibir o conteúdo do array depois de cada passagem.
- `main` para testar o método `selectionSort`.

Como nos exemplos de pesquisa, `main` (linhas 57 a 72) cria um array de `int`s nomeados `data` e o preenche com `int`s aleatórios no intervalo `10 a 99`. A linha 68 testa o método `selectionSort`.

```
1  // Figura 19.4: SelectionSortTest.java
2  // Classificando um array com classificação por seleção.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class SelectionSortTest
7  {
8     // classifica o array utilizando a classificação por seleção
9     public static void selectionSort(int[] data)
10    {
11       // faz um loop sobre data.length - 1 elementos
12       for (int i = 0; i < data.length - 1; i++)
13       {
14          int smallest = i; // primeiro índice do array remanescente
15
16          // faz um loop para localizar o índice do menor elemento
17          for (int index = i + 1; index < data.length; index++)
18             if (data[index] < data[smallest])
19                smallest = index;
20
21          swap(data, i, smallest); // troca o menor elemento na posição
22          printPass(i + 1, smallest); // passagem de saída do algoritmo
23       }
24    } // finaliza o método SelectionSort
25
26    // método auxiliar para trocar valores em dois elementos
27    private static void swap(int[] data, int first, int second)
28    {
29       int temporary = data[first]; // armazena o primeiro em temporário
30       data[first] = data[second]; // substitui o primeiro pelo segundo
31       data[second] = temporary; // coloca o temporário no segundo
32    }
33
34    // imprime uma passagem do algoritmo
35    private static void printPass(int[] data, int pass, int index)
36    {
37       System.out.printf("after pass %2d: ", pass);
38
39       // saída de elementos até item selecionado
40       for (int i = 0; i < index; i++)
41          System.out.printf("%d  ", data[i]);
42
43       System.out.printf("%d* ", data[index]); // indica troca
44
45       // termina de gerar a saída do array
46       for (int i = index + 1; i < data.length; i++)
47          System.out.printf("%d  ", data[i]);
48
49       System.out.printf("%n               "); // para alinhamento
50
51       // indica quantidade do array que é classificado
52       for (int j = 0; j < pass; j++)
53          System.out.print("--  ");
54       System.out.println();
55    }
56
57    public static void main(String[] args)
58    {
59       SecureRandom generator = new SecureRandom();
60
61       int[] data = new int[10]; // cria o array
62
63       for (int i = 0; i < data.length; i++) // preenche o array
64          data[i] = 10 + generator.nextInt(90);
65
66       System.out.printf("Unsorted array:%n%s%n%n",
```

continua

continuação

```
67            Arrays.toString(data)); // exibe o array
68         selectionSort(data); // classifica o array
69
70         System.out.printf("Sorted array:%n%s%n%n",
71            Arrays.toString(data)); // exibe o array
72      }
73   } // fim da classe SelectionSortTest
```

```
Unsorted array:
[40, 60, 59, 46, 98, 82, 23, 51, 31, 36]
after pass  1: 23  60  59  46  98  82  40* 51  31  36
               --
after pass  2: 23  31  59  46  98  82  40  51  60* 36
               --  --
after pass  3: 23  31  36  46  98  82  40  51  60  59*
               --  --  --
after pass  4: 23  31  36  40  98  82  46* 51  60  59
               --  --  --  --
after pass  5: 23  31  36  40  46  82  98* 51  60  59
               --  --  --  --  --
after pass  6: 23  31  36  40  46  51  98  82* 60  59
               --  --  --  --  --  --
after pass  7: 23  31  36  40  46  51  59  82  60  98*
               --  --  --  --  --  --  --
after pass  8: 23  31  36  40  46  51  59  60  82* 98
               --  --  --  --  --  --  --  --
after pass  9: 23  31  36  40  46  51  59  60  82* 98
               --  --  --  --  --  --  --  --  --
Sorted array:
[23, 31, 36, 40, 46, 51, 59, 60, 82, 98]
```

Figura 19.4 | Classificando um array com classificação por seleção.

Métodos `selectionSort` e `swap`

As linhas 9 a 24 declaram o método `selectionSort`. As linhas 12 a 23 fazem um loop `data.length - 1` vezes. A linha 14 declara e inicializa (para o índice `i` atual) a variável `smallest`, que armazena o índice do menor elemento no array remanescente. As linhas 17 a 19 fazem um loop sobre os elementos restantes no array. Para cada um desses elementos, a linha 18 compara seu valor com o valor do menor elemento. Se o elemento atual for menor que o menor elemento, a linha 19 atribui o índice do elemento atual a `smallest`. Quando esse loop termina, `smallest` conterá o índice do menor elemento no array restante. A linha 21 chama o método `swap` (linhas 27 a 32) para colocar o menor elemento restante na próxima área ordenada do array.

Métodos `printPass`

A saída do método `printPass` utiliza traços (linhas 52 e 53) para indicar a parte do array que é classificada após cada passagem. Um asterisco é colocado ao lado da posição do elemento que foi trocado pelo menor elemento nessa passagem. A cada passagem, o elemento ao lado do asterisco (especificado na linha 43) e o elemento acima do conjunto de traços mais à direita foram permutados.

19.6.2 Eficiência da classificação por seleção

O algoritmo de classificação por seleção é executado no tempo $O(n^2)$. O método `selectionSort` (linhas 9 a 24) contém dois loops `for`. O loop externo (linhas 12 a 23) itera pelos primeiros $n - 1$ elementos no array, colocando o menor item restante na sua posição classificada. O loop interno (linhas 17 a 19) itera por cada item no array restante, procurando o menor elemento. Esse loop é executado $n - 1$ vezes durante a primeira iteração do loop externo, $n - 2$ vezes durante a segunda iteração e, então, $n - 3, \ldots, 3, 2, 1$. Esse loop interno irá iterar um total de $n(n - 1)/2$ ou $(n^2 - n)/2$. Na notação Big O, os menores termos são descartados e as constantes são ignoradas, deixando um Big O de $O(n^2)$.

19.7 Classificação por inserção

A **classificação por inserção** é um outro algoritmo de classificação *simples, mas ineficiente*. A primeira iteração desse algoritmo seleciona o *segundo elemento* no array e, se for *menor que o primeiro elemento, troca-o pelo primeiro elemento*. A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos, de modo que todos os três elementos estejam na ordem. Na i-ésima iteração desse algoritmo, os primeiros elementos i no array original serão classificados.

Considere como um exemplo o seguinte array, que é idêntico àquele utilizado nas discussões sobre classificação por seleção e classificação por intercalação.

```
34   56   4   10   77   51   93   30   5   52
```

Um programa que implementa o algoritmo de classificação por inserção analisará os dois primeiros elementos do array, 34 e 56. Estes já estão em ordem; portanto, o programa continua. (Se eles estivessem fora da ordem, o programa iria permutá-los.)

Na próxima iteração, o programa examina o terceiro valor, 4. Esse valor é menor que 56, portanto o programa armazena 4 em uma variável temporária e move o 56 um elemento para a direita. O programa então verifica e determina que 4 é menor que 34, assim move o 34 um elemento para a direita. O programa agora alcançou o começo do array, assim coloca 4 no zero-ésimo elemento. O array agora está

```
4   34   56   10   77   51   93   30   5   52
```

Na próxima iteração, o programa armazena 10 em uma variável temporária. Então, compara 10 a 56 e move o 56 um elemento para a direita porque ele é maior que 10. O programa então compara 10 com 34, movendo o 34 um elemento para a direita. Quando o programa compara 10 a 4, ele observa que 10 é maior que 4 e coloca 10 no elemento 1. O array agora está

```
4   10   34   56   77   51   93   30   5   52
```

Usando esse algoritmo, na *i*-ésima iteração, os primeiros elementos *i* do array original são classificados, mas podem não estar nos locais finais — valores menores podem ser localizados mais tarde no array.

19.7.1 Implementação da classificação por inserção

A classe `InsertionSortTest` (Figura 19.5) contém:

- O método `static` `insertionSort` para classificar `int`s usando o algoritmo de classificação por inserção.
- O método `static` `printPass` para exibir o conteúdo do array depois de cada passagem.
- `main` para testar o método `insertionSort`.

O método `main` (linhas 53 a 68) é idêntico àquele `main` na Figura 19.4, exceto que a linha de 64 chama o método `insertionSort`.

```
1  // Figura 19.5: InsertionSortTest.java
2  // Classificando um array com a classificação por inserção.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class InsertionSortTest
7  {
8     // classifica o array utilizando a classificação por inserção
9     public static void insertionSort(int[] data)
10    {
11       // faz um loop sobre data.length - 1 elementos
12       for (int next = 1; next < data.length; next++)
13       {
14          int insert = data[next]; // valor a inserir
15          int moveItem = next; // local para inserir elemento
16
17          // procura o local para colocar o elemento atual
18          while (moveItem > 0 && data[moveItem - 1] > insert)
19          {
20             // desloca o elemento direito um slot
21             data[moveItem] = data[moveItem - 1];
22             moveItem--;
23          }
24
25          data[moveItem] = insert; // local do elemento inserido
26          printPass(data, next, moveItem); // passagem de saída do algoritmo
27       }
28    }
29
30    // imprime uma passagem do algoritmo
31    public static void printPass(int[] data, int pass, int index)
32    {
33       System.out.printf("after pass %2d: ", pass);
```

continua

continuação

```
34
35          // gera saída dos elementos até o item trocado
36          for (int i = 0; i < index; i++)
37             System.out.printf("%d  ", data[i]);
38
39          System.out.printf("%d* ", data[index]); // indica troca
40
41          // termina de gerar a saída do array
42          for (int i = index + 1; i < data.length; i++)
43             System.out.printf("%d  ", data[i]);
44
45          System.out.printf("%n               "); // para alinhamento
46
47          // indica quantidade do array que é classificado
48          for(int i = 0; i <= pass; i++)
49             System.out.print("--  ");
50          System.out.println();
51       }
52
53       public static void main(String[] args)
54       {
55          SecureRandom generator = new SecureRandom();
56
57          int[] data = new int[10]; // cria o array
58
59          for (int i = 0; i < data.length; i++) // preenche o array
60             data[i] = 10 + generator.nextInt(90);
61
62          System.out.printf("Unsorted array:%n%s%n%n",
63             Arrays.toString(data)); // exibe o array
64          insertionSort(data); // classifica o array
65
66          System.out.printf("Sorted array:%n%s%n%n",
67             Arrays.toString(data)); // exibe o array
68       }
69    } // fim da classe InsertionSortTest
```

```
Unsorted array:
[34, 96, 12, 87, 40, 80, 16, 50, 30, 45]
after pass  1: 34  96* 12  87  40  80  16  50  30  45
               --  --
after pass  2: 12* 34  96  87  40  80  16  50  30  45
               --  --  --
after pass  3: 12  34  87* 96  40  80  16  50  30  45
               --  --  --  --
after pass  4: 12  34  40* 87  96  80  16  50  30  45
               --  --  --  --  --
after pass  5: 12  34  40  80* 87  96  16  50  30  45
               --  --  --  --  --  --
after pass  6: 12  16* 34  40  80  87  96  50  30  45
               --  --  --  --  --  --  --
after pass  7: 12  16  34  40  50* 80  87  96  30  45
               --  --  --  --  --  --  --  --
after pass  8: 12  16  30* 34  40  50  80  87  96  45
               --  --  --  --  --  --  --  --  --
after pass  9: 12  16  30  34  40  45* 50  80  87  96
               --  --  --  --  --  --  --  --  --  --
Sorted array:
[12, 16, 30, 34, 40, 45, 50, 80, 87, 96]
```

Figura 19.5 | Classificando um array com a classificação por inserção.

Método `insertionSort`

As linhas 9 a 28 declaram o método `insertionSort`. As linhas 12 a 27 fazem um loop por itens `data.length - 1` no array. A cada iteração, a linha 14 declara e inicializa a variável `insert`, que contém o valor do elemento que será inserido na parte classificada do array. A linha 15 declara e inicializa a variável `moveItem`, que monitora onde inserir o elemento. As linhas 18 a 23 fazem um loop para localizar a posição correta onde o elemento deve ser inserido. O loop terminará quando o programa alcançar o início do

array ou quando alcançar um elemento menor que o valor a ser inserido. A linha 21 move um elemento para a direita no array, e a linha 22 decrementa a posição na qual inserir o próximo elemento. Depois de o loop terminar, a linha 25 insere o elemento na posição.

Método `printPass`

A saída do método `printPass` (linhas 31 a 51) utiliza traços para indicar a parte do array que é classificada após cada passagem. Um asterisco é colocado ao lado do elemento que foi inserido na posição nessa passagem.

19.7.2 Eficiência da classificação por inserção

O algoritmo de classificação por inserção também é executado em tempo da notação $O(n^2)$. Como a classificação por seleção, a implementação da classificação por inserção (linhas 9 a 28) contém dois loops. O loop `for` (linhas 12 a 27) itera `data.length - 1` vezes, inserindo um elemento na posição apropriada nos elementos classificados até o momento. Para os propósitos desse aplicativo, `data.length - 1` é equivalente a $n - 1$ (uma vez que `data.length` é o tamanho do array). O loop `while` (linhas 18 a 23) itera pelos elementos precedentes no array. No pior caso, esse loop `while` exigirá $n - 1$ comparações. Cada loop individual é executado no tempo $O(n)$. Na notação Big O, loops aninhados significam que você deve *multiplicar* o número de comparações. Para cada iteração de um loop externo, haverá certo número de iterações do loop interno. Nesse algoritmo, para cada $O(n)$ iterações do loop externo, haverá $O(n)$ iterações do loop interno. Multiplicar esses valores resulta em uma Big O de $O(n^2)$.

19.8 Classificação por intercalação

A **classificação por intercalação** é um algoritmo de classificação *eficiente*, mas é conceitualmente *mais complexo* que a classificação por seleção e a classificação por inserção. O algoritmo de classificação por intercalação classifica um array *dividindo-o* em dois subarrays de igual tamanho, *classificando* cada subarray e, então, *mesclando* em um array maior. Com um número ímpar de elementos, o algoritmo cria os dois subarrays de tal maneira que um deles tenha um elemento a mais que o outro.

A implementação da classificação por intercalação nesse exemplo é recursiva. O caso de base é um array com um elemento, o qual, naturalmente, é classificado; assim, neste caso, a classificação por intercalação retorna imediatamente. O passo de recursão divide o array em duas partes aproximadamente iguais, classifica-as recursivamente e, então, intercala os dois arrays classificados em um array classificado maior.

Suponha que o algoritmo já tenha intercalado arrays menores para criar os arrays classificados A:

```
4   10   34   56   77
```

e B:

```
5   30   51   52   93
```

A classificação por intercalação combina esses dois arrays em um array classificado maior. O menor elemento em A é 4 (localizado no zero-ésimo índice de A). O menor elemento em B é 5 (localizado no zero-ésimo índice de B). A fim de determinar o menor elemento no maior array, o algoritmo compara 4 e 5. O valor em A é menor, assim 4 torna-se o primeiro elemento no array intercalado. O algoritmo continua comparando 10 (o segundo elemento em A) com 5 (o primeiro elemento em B). O valor de B é menor, portanto 5 torna-se o segundo elemento no maior array. O algoritmo continua comparando 10 a 30, com 10 tornando-se o terceiro elemento no array e assim por diante.

19.8.1 Implementação da classificação por intercalação

A Figura 19.6 declara a classe `MergeSortTest`, que contém:

- O método `static mergeSort` para iniciar a classificação de um array `int` usando o algoritmo de classificação por intercalação.
- O método `static sortArray` para executar o algoritmo de classificação por intercalação recursiva — isso é chamado pelo método `mergeSort`.
- O método `static merge` para intercalar dois subarrays classificados em um único subarray classificado.
- O método `static subarrayString` para obter a representação `String` de um subarray para propósitos de saída.
- `main` para testar o método `mergeSort`.

O método `main` (linhas 101 a 116) é idêntico ao `main` nas figuras 19.4 e 19.5, exceto que a linha 112 chama o método `mergeSort`. A saída desse programa exibe as divisões e intercalações realizadas pela classificação por intercalação, mostrando o progresso da classificação em cada passo do algoritmo. Vale muito a pena analisar essas saídas a fim de entender completamente esse algoritmo de classificação elegante.

```
1  // Figura 19.6: MergeSortTest.java
2  // Classificando um array com a classificação por intercalação.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class MergeSortTest
7  {
8     // chama o método split recursivo para iniciar a classificação por intercalação
9     public static void mergeSort(int[] data)
10    {
11       sortArray(data, 0, data.length - 1); // classifica todo o array
12    } // fim do método sort
13
14    // divide o array, classifica subarrays e intercala subarrays no array classificado
15    private static void sortArray(int[] data, int low, int high)
16    {
17       // caso básico de teste; tamanho do array é igual a 1
18       if ((high - low) >= 1) // se não for o caso básico
19       {
20          int middle1 = (low + high) / 2; // calcula o meio do array
21          int middle2 = middle1 + 1; // calcula o próximo elemento
22
23          // gera uma saída do passo de divisão
24          System.out.printf("split:   %s%n",
25             subarrayString(data, low, high));
26          System.out.printf("         %s%n",
27             subarrayString(data, low, middle1));
28          System.out.printf("         %s%n%n",
29             subarrayString(data, middle2, high));
30
31          // divide o array pela metade; classifica cada metade (chamadas recursivas)
32          sortArray(data, low, middle1); // primeira metade do array
33          sortArray(data, middle2, high); // segunda metade do array
34
35          // intercala dois arrays classificados depois que as chamadas de divisão retornam
36          merge (data, low, middle1, middle2, high);
37       } // fim do if
38    } // fim do método sortArray
39
40    // intercala dois subarrays classificados em um subarray classificado
41    private static void merge(int[] data, int left, int middle1,
42       int middle2, int right)
43    {
44       int leftIndex = left; // índice no subarray esquerdo
45       int rightIndex = middle2; // índice no subarray direito
46       int combinedIndex = left; // índice no array temporário funcional
47       int[] combined = new int[data.length]; // array de trabalho
48
49       // gera saída de dois subarrays antes de mesclar
50       System.out.printf("merge:   %s%n",
51          subarrayString(data, left, middle1));
52       System.out.printf("         %s%n",
53          subarrayString(data, middle2, right));
54
55       // intercala arrays até alcançar o final de um deles
56       while (leftIndex <= middle1 && rightIndex <= right)
57       {
58          // coloca o menor dos dois elementos atuais no resultado
59          // e o move para o próximo espaço nos arrays
60          if (data[leftIndex] <= data[rightIndex])
61             combined[combinedIndex++] = data[leftIndex++];
62          else
63             combined[combinedIndex++] = data[rightIndex++];
64       }
65
66       // se o array esquerdo estiver vazio
```

continua

continuação

```
67          if (leftIndex == middle2)
68             // copia o restante do array direito
69             while (rightIndex <= right)
70                combined[combinedIndex++] = data[rightIndex++];
71          else // o array direito está vazio
72             // copia o restante do array esquerdo
73             while (leftIndex <= middle1)
74                combined[combinedIndex++] = data[leftIndex++];
75
76          // copia os valores de volta ao array original
77          for (int i = left; i <= right; i++)
78             data[i] = combined[i];
79
80          // gera saída do array intercalado
81          System.out.printf("         %s%n%n",
82             subarrayString(data, left, right));
83       } // fim do método merge
84
85       // método para gerar saída de certos valores no array
86       private static String subarrayString(int[] data, int low, int high)
87       {
88          StringBuilder temporary = new StringBuilder();
89
90          // gera espaços para alinhamento
91          for (int i = 0; i < low; i++)
92             temporary.append("   ");
93
94          // gera a saída dos elementos que permanecem no array
95          for (int i = low; i <= high; i++)
96             temporary.append(" " + data[i]);
97
98          return temporary.toString();
99       }
100
101      public static void main(String[] args)
102      {
103         SecureRandom generator = new SecureRandom();
104
105         int[] data = new int[10]; // cria o array
106
107         for (int i = 0; i < data.length; i++) // preenche o array
108            data[i] = 10 + generator.nextInt(90);
109
110         System.out.printf("Unsorted array:%n%s%n%n",
111            Arrays.toString(data)); // exibe o array
112         mergeSort(data); // classifica o array
113
114         System.out.printf("Sorted array:%n%s%n%n",
115            Arrays.toString(data)); // exibe o array
116      }
117   } // fim da classe MergeSortTest
```

```
Unsorted array:
[75, 56, 85, 90, 49, 26, 12, 48, 40, 47]

split:    75 56 85 90 49 26 12 48 40 47
          75 56 85 90 49
                         26 12 48 40 47

split:    75 56 85 90 49
          75 56 85
                   90 49

split:    75 56 85
          75 56
                85
```

continua

continuação

```
split:    75 56
          75
             56

merge:    75
             56
          56 75

merge:    56 75
                85
          56 75 85

split:              90 49
                    90
                       49

merge:              90
                       49
                    49 90

merge:    56 75 85
                    49 90
          49 56 75 85 90

split:                     26 12 48 40 47
                           26 12 48
                                    40 47

split:                     26 12 48
                           26 12
                                 48

split:                     26 12
                           26
                              12

merge:                     26
                              12
                           12 26

merge:                     12 26
                                 48
                           12 26 48

split:                               40 47
                                     40
                                47

merge:                             40
                                    47
                                40 47

merge:                  12 26 48
                              40 47
                     12 26 40 47 48

merge:    49 56 75 85 90
                     12 26 40 47 48
          12 26 40 47 48 49 56 75 85 90

Sorted array:
[12, 26, 40, 47, 48, 49, 56, 75, 85, 90]
```

Figura 19.6 | Classificando um array com a classificação por intercalação.

Método `mergeSort`

As linhas 9 a 12 da Figura 19.6 declaram o método `mergeSort`. A linha 11 chama o método `sortArray` com `0` e `data.length - 1` como os argumentos — que correspondem aos índices iniciais e finais, respectivamente, do array a ser classificado. Esses valores informam ao método `sortArray` a operar no array inteiro.

Método `sortArray`

O método recursivo `sortArray` (linhas 15 a 38) executa o algoritmo de classificação por intercalação recursiva. A linha 18 testa o caso de base. Se o tamanho do array for `1`, o array já está classificado, assim o método retorna imediatamente. Se o tamanho do array for maior que `1`, o método divide o array em dois, chama recursivamente o método `sortArray` para classificar os dois subarrays e então os intercala. A linha 32 chama recursivamente o método `sortArray` na primeira metade do array e a linha 33, na segunda metade. Depois que essas duas chamadas de método retornam, cada metade do array terá sido classificada. A linha 36 chama o método `merge` (linhas 41 a 83) nas duas metades do array para combinar os dois arrays classificados em um array classificado maior.

Método `merge`

As linhas 41 a 83 declaram o método `merge`. As linhas 56 a 64 em `merge` fazem um loop até que o final de um dos subarrays é alcançado. A linha 60 testa qual elemento no começo dos arrays é o menor. Se o elemento no array esquerdo for o menor, a linha 61 coloca-o na posição no array combinado. Se o elemento no array direito for menor, a linha 63 coloca-o na posição no array combinado. Quando o loop `while` termina, um subarray inteiro foi inserido no array combinado, mas o outro subarray ainda contém dados. A linha 67 testa se o array esquerdo alcançou o fim. Se alcançou, as linhas 69 e 70 preenchem o array combinado com os elementos do array direito. Se o array esquerdo não alcançou o fim, o array direito deve então ter alcançado o fim e as linhas 73 e 74 preenchem o array combinado com os elementos do array esquerdo. Por fim, as linhas 77 e 78 copiam o array combinado para o array original.

19.8.2 Eficiência da classificação por intercalação

A classificação por intercalação é *muito mais eficiente* do que a classificação por inserção ou seleção. Considere a primeira chamada (não recursiva) a `sortArray`. Isso resulta em duas chamadas recursivas a `sortArray` com cada um dos subarrays tendo aproximadamente metade do tamanho do array original, e uma única chamada a `merge`, que requer, na pior das hipóteses, $n - 1$ comparações para preencher o array original, que é $O(n)$. (Lembre-se de que cada elemento de array pode ser escolhido comparando um elemento de cada subarray.) As duas chamadas a `sortArray` resultam em quatro chamadas a `sortArray` recursivas adicionais, cada uma com um subarray com aproximadamente um quarto do tamanho do array original, juntamente com duas chamadas a `merge` que exigem, na pior das hipóteses, $n/2 - 1$ comparações, para um número total de comparações de $O(n)$. Esse processo continua, cada chamada a `sortArray` gerando duas chamadas a `sortArray` adicionais e uma chamada a `merge` até que o algoritmo tenha *dividido* o array em subarrays de um elemento. Em cada nível, $O(n)$ comparações são exigidas para *intercalar* os subarrays. Cada nível divide os arrays pela metade, assim dobrar o tamanho do array requer mais um nível. Quadruplicar o tamanho do array requer mais dois níveis. Esse padrão é logarítmico e resulta em $\log_2 n$ níveis. Isso resulta em uma eficiência total de **O(n log n)**.

19.9 Resumo de Big O para os algoritmos de pesquisa e classificação deste capítulo

A Figura 19.7 resume os algoritmos de pesquisa e classificação abordados neste capítulo com a notação Big O para cada um. A Figura 19.8 lista os valores de Big O que abordamos neste capítulo junto com alguns valores para n a fim de destacar as diferenças nas taxas de crescimento.

Algoritmo	Posição	Big O
Algoritmos de pesquisa:		
Pesquisa linear	Seção 19.2	$O(n)$
Pesquisa binária	Seção 19.4	$O(\log n)$
Pesquisa linear recursiva	Exercício 19.8	$O(n)$
Pesquisa binária recursiva	Exercício 19.9	$O(\log n)$
Algoritmos de classificação:		
Classificação por seleção	Seção 19.6	$O(n^2)$
Classificação por inserção	Seção 19.7	$O(n^2)$
Classificação por intercalação	Seção 19.8	$O(n \log n)$
Classificação por borbulhamento	Exercícios 19.5 e 19.6	$O(n^2)$

Figura 19.7 | Algoritmos de pesquisa e classificação com valores na notação Big O.

n =	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10.000
1000	3	1000	3000	10^6
1.000.000	6	1.000.000	6.000.000	10^{12}
1.000.000.000	9	1.000.000.000	9.000.000.000	10^{18}

Figura 19.8 | Número de comparações para notações Big O comuns.

19.10 Conclusão

Este capítulo fez uma introdução à pesquisa e classificação. Discutimos dois algoritmos de pesquisa — a pesquisa linear e a pesquisa binária — e três algoritmos de classificação — classificação por seleção, classificação por inserção e classificação por intercalação. Introduzimos a notação Big O, que ajuda a analisar a eficiência de um algoritmo. Os dois próximos capítulos continuam nossa discussão das estruturas de dados dinâmicas que podem aumentar ou diminuir em tempo de execução. O Capítulo 20 demonstra como usar as capacidades genéricas do Java para implementar classes e métodos genéricos. O Capítulo 21 discute os detalhes da implementação das estruturas de dados genéricas. Cada um dos algoritmos neste capítulo é de "único thread" — no Capítulo 23, "Concorrência", discutiremos multithreading e como ele pode ajudá-lo a programar para alcançar melhor desempenho nos atuais sistemas multiprocessados.

Resumo

Seção 19.1 Introdução

- Pesquisar dados envolve determinar se uma chave de pesquisa está nos dados e, se estiver, encontrar sua localização.
- Classificação envolve organizar dados em ordem.

Seção 19.2 Pesquisa linear

- O algoritmo de pesquisa linear pesquisa cada elemento no array sequencialmente até encontrar o elemento correto, ou até alcançar o fim do array sem encontrar o elemento.

Seção 19.3 Notação Big O

- Uma das principais diferenças entre os algoritmos de pesquisa é a quantidade de esforço que eles exigem.
- A notação Big O descreve a eficiência de um algoritmo em termos do trabalho necessário para resolver um problema. Em geral, os algoritmos de pesquisa e classificação dependem do número de elementos nos dados.
- Um algoritmo que é $O(1)$ não necessariamente requer uma única comparação. Significa apenas que o número de comparações não aumenta à medida que o tamanho do array aumenta.
- Diz-se que um algoritmo $O(n)$ tem um tempo de execução linear.
- A notação Big O destaca os fatores dominantes e ignora os termos sem importância com valores de n altos.
- A notação Big O se preocupa com a taxa de crescimento dos tempos de execução do algoritmo, portanto constantes são ignoradas.
- O algoritmo de pesquisa linear executa a $O(n)$ vezes.
- O pior caso na pesquisa linear é que cada elemento deve ser verificado para determinar se a chave de pesquisa existe, o que ocorre se a chave de pesquisa for o último elemento de array ou não estiver presente.

Seção 19.4 Pesquisa binária

- A pesquisa binária é mais eficiente do que a pesquisa linear, mas exige que o array seja classificado.
- A primeira iteração da pesquisa binária testa o elemento do meio no array. Se este for a chave de pesquisa, o algoritmo retornará sua localização. Se a chave de pesquisa for menor que o elemento do meio, a pesquisa continuará com a primeira metade do array. Se a chave de pesquisa for maior que o elemento do meio, a pesquisa continuará com a segunda metade do array. Cada iteração testa o valor do meio do array restante e, se o elemento não for localizado, elimina metade dos elementos restantes.
- A pesquisa binária é um algoritmo de pesquisa mais eficiente do que a pesquisa linear, pois cada comparação elimina metade dos elementos no array.
- A pesquisa binária é executada em tempo de $O(\log n)$ porque cada etapa remove a metade dos elementos remanescente; isso também é conhecido como tempo de execução logarítmica.
- Se o tamanho do array for dobrado, a pesquisa binária requer uma única comparação extra.

Seção 19.6 Classificação por seleção

- Classificação por seleção é um algoritmo de classificação simples, mas ineficiente.
- A classificação começa selecionando o menor elemento e o permuta pelo primeiro elemento. A segunda iteração seleciona o segundo menor item (que é o menor item restante) e troca-o pelo segundo elemento. A classificação continua até que a última iteração selecione o segundo maior elemento e permute-o pelo penúltimo elemento, deixando o maior elemento no último índice. Na i-ésima iteração da classificação por seleção, os menores itens i do array inteiro são classificados nos primeiros índices i.
- O algoritmo de classificação por seleção executa no tempo $O(n^2)$.

Seção 19.7 Classificação por inserção

- A primeira iteração da classificação por inserção seleciona o segundo elemento no array e, se for menor que o primeiro elemento, troca-o pelo primeiro elemento. A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos. Depois da i-ésima iteração da classificação por inserção, os primeiros elementos i no array original são classificados.
- O algoritmo de classificação por inserção executa a $O(n^2)$ vezes.

Seção 19.8 Classificação por intercalação

- A classificação por intercalação é um algoritmo de classificação mais rápido, mas mais complexo de implementar que a classificação por seleção e classificação por inserção. O algoritmo de classificação por intercalação classifica um array dividindo-o em dois subarrays de igual tamanho, classificando cada subarray recursivamente e mesclando os subarrays em um array maior.
- O caso básico da classificação por intercalação é um array com um único elemento. O array de um elemento já está classificado, assim a classificação por intercalação retorna imediatamente quando é chamada com um array de um elemento. A parte da intercalação da classificação por intercalação recebe dois arrays classificados e os combina em um array classificado maior.
- A classificação por intercalação realiza a intercalação examinando o primeiro elemento em cada array, que também é o menor elemento no array. A classificação por intercalação seleciona o menor destes e coloca-os no primeiro elemento do maior array. Se ainda houver elementos no subarray, a classificação por intercalação examina o segundo deles (que agora é o menor elemento remanescente) e o compara ao primeiro elemento no outro subarray. A classificação por intercalação continua esse processo até que o maior array seja preenchido.
- No pior caso, a primeira chamada à classificação por intercalação tem de fazer $O(n)$ comparações para preencher os n slots no array final.
- A parte da intercalação do algoritmo de classificação por intercalação é realizada em dois subarrays, cada um com aproximadamente o tamanho $n/2$. Criar cada um desses subarrays exige $n / 2 - 1$ comparações para cada subarray ou o total de $O(n)$ comparações. Esse padrão continua à medida que cada nível constrói duas vezes o número de arrays, mas cada um tem a metade do tamanho do array anterior.
- Semelhante à pesquisa binária, essa divisão em metades resulta em $\log n$ níveis para uma eficiência total de $O(n \log n)$.

Exercícios de revisão

19.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um aplicativo de classificação por seleção demoraria aproximadamente _____ vezes a mais para ser executado em um array de 128 elementos do que em um array de 32 elementos.

b) A eficiência da classificação por intercalação é _____.

19.2 Qual aspecto-chave da pesquisa binária e da classificação por intercalação é responsável pela parte logarítmica das suas respectivas Big O?

19.3 Em que sentido a classificação por inserção é superior à classificação por intercalação? Em que sentido a classificação por intercalação é superior à classificação por inserção?

19.4 No texto, dizemos que depois que a classificação por intercalação divide o array em dois subarrays, ela então classifica esses dois subarrays e os intercala. Por que alguém ficaria intrigado com a nossa afirmação de que "ele então classifica esses dois subarrays"?

Respostas dos exercícios de revisão

19.1 a) 16, porque um algoritmo $O(n^2)$ demora 16 vezes mais para classificar quatro vezes o mesmo número de informações. b) $O(n \log n)$.

19.2 Esses dois algoritmos incorporam a "divisão por metades" — de algum modo reduzindo algo pela metade. A pesquisa binária elimina uma metade do array depois de cada comparação. A classificação por intercalação divide o array pela metade toda vez que é chamada.

19.3 A classificação por inserção é mais fácil de entender e programar do que a classificação por intercalação. A classificação por intercalação é muito mais eficiente [$O(n \log n)$] do que a classificação por inserção [$O(n^2)$].

19.4 Em certo sentido, ela na verdade não classifica esses dois subarrays. Simplesmente continua a dividir o array original pela metade até que ele fornece um subarray de um elemento, que está, naturalmente, classificado. Ela, então, cria os dois subarrays originais intercalando esses arrays de um elemento para formar subarrays maiores, que são então intercalados e assim por diante.

Questões

19.5 (***Classificação por borbulhamento***) Implemente uma classificação por borbulhamento — outra técnica de classificação simples, mas ineficiente. É chamada classificação por borbulhamento ou classificação por afundamento porque os menores valores gradualmente "borbulham" no seu caminho para a parte superior do array (isto é, na direção do primeiro elemento) como bolhas de ar que emergem na superfície, enquanto os maiores valores afundam na parte inferior (final) do array. A técnica utiliza loops aninhados para fazer várias passagens pelo array. Cada passagem compara pares sucessivos de elementos. Se um par estiver na ordem crescente (ou os valores forem iguais), a classificação por borbulhamento deixa os valores como estão. Se um par estiver na ordem decrescente, a classificação por borbulhamento troca seus valores no array. A primeira passagem compara os dois primeiros elementos do array e troca seus valores, se necessário. Ela então compara o segundo e terceiro elementos no array. O final dessa passagem compara os dois últimos elementos no array e troca-os, se necessário. Depois de uma passagem, o maior elemento estará no último índice. Depois de duas passagens, os dois maiores elementos estarão nos dois últimos índices. Explique por que a classificação por borbulhamento é um algoritmo $O(n^2)$.

19.6 (***Classificação por borbulhamento aprimorada***) Faça as seguintes modificações simples para melhorar o desempenho da classificação por borbulhamento que você desenvolveu na Questão 19.5:

a) Depois da primeira passagem, garante-se que o número maior está no elemento de número mais alto do array; após a segunda passagem, os dois números mais altos estão "no lugar"; e assim por diante. Em vez de fazer nove comparações em cada passagem para um array de 10 elementos, modifique a classificação por borbulhamento para fazer oito comparações na segunda passagem, sete na terceira passagem e assim por diante.

b) Os dados no array já podem estar na ordem adequada ou quase adequada, então por que fazer nove passagens se menos seriam suficientes? Modifique a classificação para verificar no fim de cada passagem se alguma troca foi feita. Se não houvesse nenhum, os dados já deveriam estar classificados, assim o programa deve terminar. Se trocas foram feitas, pelo menos mais uma passagem é necessária.

19.7 (***Bucket sort***) Uma classificação do tipo *bucket sort* inicia com um array unidimensional de inteiros positivos a ser classificado e um array bidimensional de inteiros com linhas indexadas de 0 a 9 e colunas indexadas de 0 a $n - 1$, onde n é o número dos valores a ser classificado. Cada linha do array bidimensional é chamada *bucket*. Escreva uma classe chamada `BucketSort` que contém um método chamado `sort` que opera desta maneira:

a) Coloque cada valor do array unidimensional em uma linha do array de bucket, com base nas "unidades" (mais à direita) do dígito. Por exemplo, 97 é colocado na linha 7, 3 é colocado na linha 3 e 100 é colocado na linha 0. Esse procedimento é chamado de *passagem de distribuição*.

b) Realize um loop pelo array de bucket linha por linha e copie os valores de volta para o array original. Esse procedimento é chamado *passagem de coleta*. A nova ordem dos valores precedentes no array unidimensional é 100, 3 e 97.

c) Repita esse processo para a posição de cada dígito subsequente (dezenas, centenas, milhares etc.). Na segunda passagem (dígitos das dezenas), 100 é colocado na linha 0, 3 é colocado na linha 0 (porque 3 não tem nenhum dígito de dezena) e 97 é colocado na linha 9. Depois da passagem de coleta, a ordem dos valores no array unidimensional é 100, 3 e 97. Na terceira passagem (dígitos das centenas), 100 é colocado na linha 1, 3 é colocado na linha 0 e 97 é colocado na linha 0 (depois do 3). Depois dessa última passagem de coleta, o array original está na ordem classificada.

O array bidimensional dos buckets tem 10 vezes o comprimento do array de inteiros sendo classificado. Essa técnica de classificação fornece um melhor desempenho do que uma classificação por borbulhamento, mas exige muito mais memória — a classificação por borbulhamento exige espaço para somente um elemento adicional de dados. Essa comparação é um exemplo da relação de troca espaço/tempo: a *bucket sort* utiliza mais memória que a classificação por borbulhamento, mas seu desempenho é melhor. Essa versão da *bucket sort* requer cópia de todos os dados de volta para o array original a cada passagem. Outra possibilidade é criar um segundo array de bucket bidimensional e permutar os dados repetidamente entre os dois arrays de bucket.

19.8 (***Pesquisa linear recursiva***) Modifique a Figura 19.2 para usar o método recursivo `recursiveLinearSearch` para realizar uma pesquisa linear do array. O método deve receber a chave de pesquisa e o índice inicial como argumentos. Se a chave de pesquisa for encontrada, seu índice no array é retornado; caso contrário, `-1` é retornado. Cada chamada ao método recursivo deve verificar um índice no array.

19.9 (***Pesquisa binária recursiva***) Modifique a Figura 19.3 para usar o método recursivo `recursiveBinarySearch` a fim de realizar uma pesquisa binária do array. O método deve receber a chave de pesquisa, o índice inicial e o índice final como argumentos. Se a chave de pesquisa for encontrada, seu índice no array é retornado. Se a chave de pesquisa não for encontrada, é retornado `-1`.

19.10 (***Quicksort***) A técnica de classificação recursiva chamada *quicksort* usa o seguinte algoritmo dimensional básico para um array dos valores:

a) *Passo de partição*: selecione o primeiro elemento do array não classificado e determine sua localização final no array classificado (isto é, todos os valores à esquerda do elemento no array são menores que o elemento e todos os valores à direita do elemento no array são maiores que o elemento — mostramos como fazer isso a seguir). Agora temos um elemento em sua posição adequada e dois subarrays não classificados.

b) *Passo recursivo*: realize o *Passo 1* em cada subarray não classificado. Toda vez que o *Passo 1* for realizado em um subarray, outro elemento é colocado em sua posição final no array classificado e dois subarrays não classificados são criados. Quando um subarray consiste em apenas um elemento, esse elemento está na sua localização final (porque o array de um elemento já está classificado).

O algoritmo básico parece suficientemente simples, mas como determinamos a posição final do primeiro elemento de cada subarray? Como um exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de partição — ele será colocado em sua localização final no array classificado):

37 2 6 4 89 8 10 12 68 45

Iniciando a partir do elemento mais à direita do array, compare cada elemento com 37 até um elemento menor que 37 ser encontrado; então, permute 37 e esse elemento. O primeiro elemento menor que 37 é 12, então 37 e 12 são permutados. O novo array é

12 2 6 4 89 8 10 **37** 68 45

O elemento 12 está em itálico para indicar que acabou de ser permutado com 37.

Iniciando a partir da esquerda do array, mas começando com o elemento depois de 12, compare cada elemento com 37 até um elemento maior que 37 ser encontrado, então permute 37 e esse elemento. O primeiro elemento maior que 37 é 89, então 37 e 89 foram permutados. O novo array é

12 2 6 4 **37** 8 10 *89* 68 45

Iniciando da direita, mas começando com o elemento antes de 89, compare cada elemento com 37 até um elemento menor que 37 ser encontrado — então, permute 37 e esse elemento. O primeiro elemento menor que 37 é 10, então 37 e 10 são permutados. O novo array é

12 2 6 4 *10* 8 **37** 89 68 45

Iniciando da esquerda, mas começando com o elemento depois de 10, compare cada elemento com 37 até um elemento maior que 37 ser encontrado — então, permute 37 e esse elemento. Não há mais elementos maiores que 37, então, quando comparamos 37 com ele mesmo, sabemos que 37 foi colocado na sua localização final no array classificado. Cada valor à esquerda do 37 é menor que ele e cada valor à direita do 37 é maior que ele.

Uma vez que a partição foi aplicada no array anterior, há dois subarrays não classificados. O subarray com valores menores que 37 contém 12, 2, 6, 4, 10 e 8. O subarray com valores maiores que 37 contém 89, 68 e 45. A classificação continua recursivamente, com ambos os subarrays sendo particionados da mesma maneira que o array original.

Com base na discussão precedente, escreva o método recursivo `quickSortHelper` para classificar um array unidimensional de inteiros. O método deve receber como argumentos um índice inicial e um índice final no array original sendo classificado.