# Chapter 19
# Searching, Sorting and Big O

Java How to Program, 10/e

## OBJECTIVES

In this chapter you'll:

- Search for a given value in an array using linear search and binary search.
- Sort arrays using the iterative selection and insertion sort algorithms.
- Sort arrays using the recursive merge sort algorithm.
- Determine the efficiency of searching and sorting algorithms.
- Introduce Big O notation for comparing the efficiency of algorithms.

# 19.1 Introduction

- Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding its location.
  - Two popular search algorithms are the simple linear search and the faster but more complex binary search.
- Sorting places data in ascending or descending order, based on one or more sort keys.
  - This chapter introduces two simple sorting algorithms, the selection sort and the insertion sort, along with the more efficient but more complex merge sort.
- Figure 19.1 summarizes the searching and sorting algorithms discussed in the examples and exercises of this book.

## Software Engineering Observation 19.1

*In apps that require searching and sorting, use the pre-defined capabilities of the Java Collections API (Chapter 16). The techniques presented in this chapter are provided to introduce students to the concepts behind searching and sorting algorithms—upper-level computer science courses typically discuss such algorithms in detail.*

| Chapter | Algorithm | Location |
|---|---|---|
| *Searching Algorithms:* | | |
| 16 | `binarySearch` method of class `Collections` | Fig. 16.12 |
| 19 | Linear search | Section 19.2 |
| | Binary search | Section 19.4 |
| | Recursive linear search | Exercise 19.8 |
| | Recursive binary search | Exercise 19.9 |
| 21 | Linear search of a `List` | Exercise 21.21 |
| | Binary tree search | Exercise 21.23 |

**Fig. 19.1** | Searching and sorting algorithms covered in this text. (Part 1 of 2.)

| Chapter | Algorithm | Location |
|---|---|---|
| *Sorting Algorithms:* | | |
| 16 | `sort` method of class `Collections` | Figs. 16.6–16.9 |
| | `SortedSet` collection | Fig. 16.17 |
| 19 | Selection sort | Section 19.6 |
| | Insertion sort | Section 19.7 |
| | Recursive merge sort | Section 19.8 |
| | Bubble sort | Exercises 19.5 and 19.6 |
| | Bucket sort | Exercise 19.7 |
| | Recursive quicksort | Exercise 19.10 |
| 21 | Binary tree sort | Section 21.7 |

**Fig. 19.1** | Searching and sorting algorithms covered in this text. (Part 2 of 2.)

# 19.2 Linear Search

- This section and Section 19.4 discuss two common search algorithms—one that's easy to program yet relatively inefficient (linear search) and one that's relatively efficient but more complex to program (binary search).

# 19.2 Linear Search (cont.)

- The linear search algorithm searches each element in an array sequentially.
  - If the search key does not match an element in the array, the algorithm tests each element, and when the end of the array is reached, informs the user that the search key is not present.
  - If the search key is in the array, the algorithm tests each element until it finds one that matches the search key and returns the index of that element.
- Class `LinearSearchTest` (Fig. 19.2) contains static method `linearSearch` for performing searches of an `int` array and main for testing `linearSearch`.

```
1  // Fig. 19.2: LinearSearchTest.java
2  // Sequentially searching an array for an item.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class LinearSearchTest
8  {
9     // perform a linear search on the data
10    public static int linearSearch(int data[], int searchKey)
11    {
12       // loop through array sequentially
13       for (int index = 0; index < data.length; index++)
14          if (data[index] == searchKey)
15             return index; // return index of integer
16
17       return -1; // integer was not found
18    } // end method linearSearch
19
20    public static void main(String[] args)
21    {
22       Scanner input = new Scanner(System.in);
23       SecureRandom generator = new SecureRandom();
24
```

**Fig. 19.2** | Sequentially searching an array for an item. (Part 1 of 3.)

```
25          int[] data = new int[10]; // create array
26
27          for (int i = 0; i < data.length; i++) // populate array
28             data[i] = 10 + generator.nextInt(90);
29
30          System.out.printf("%s%n%n", Arrays.toString(data)); // display array
31
32          // get input from user
33          System.out.print("Please enter an integer value (-1 to quit): ");
34          int searchInt = input.nextInt();
35
36          // repeatedly input an integer; -1 terminates the program
37          while (searchInt != -1)
38          {
39             int position = linearSearch(data, searchInt); // perform search
40
41             if (position == -1) // not found
42                System.out.printf("%d was not found%n%n", searchInt);
43             else // found
44                System.out.printf("%d was found in position %d%n%n",
45                   searchInt, position);
46
```

**Fig. 19.2** | Sequentially searching an array for an item. (Part 2 of 3.)

```
47            // get input from user
48            System.out.print("Please enter an integer value (-1 to quit): ");
49            searchInt = input.nextInt();
50         }
51      } // end main
52   } // end class 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
```

**Fig. 19.2** | Sequentially searching an array for an item. (Part 3 of 3.)

# 19.3 Big O Notation

- Searching algorithms all accomplish the *same* goal—finding an element (or elements) that matches a given search key, if such an element does, in fact, exist.
- *The major difference is the amount of effort they require to complete the search.*
- Big O notation indicates how hard an algorithm may have to work to solve a problem.
  - For searching and sorting algorithms, this depends particularly on how many data elements there are.

# 19.3.1 O(1) Algorithms

- If an algorithm is completely independent of the number of elements in the array, it is said to have a constant run time, which is represented in Big O notation as O(1) and pronounced as "order one."
  - An algorithm that's *O*(1) does not necessarily require only one comparison.
  - *O*(1) just means that the number of comparisons is *constant*—it does not grow as the size of the array increases.

# 19.3.2 O(*n*) Algorithms

- An algorithm that requires a total of $n - 1$ comparisons is said to be $O(n)$.
  - An $O(n)$ algorithm is referred to as having a linear run time.
  - $O(n)$ is often pronounced "on the order of $n$" or simply "order $n$."

# 19.3.3 O($n^2$) Algorithms

- Constant factors are omitted in Big O notation.
- Big O is concerned with how an algorithm's run time grows in relation to the number of items processed.
- $O(n^2)$ **is referred to as quadratic run time and pronounced "on the order of *n-squared" or more simply "order n-squared."***
  - When *n is small, $O(n^2)$ algorithms (running on today's computers) will not noticeably affect performance.*
  - But as *n grows, you'll start to notice the performance degradation.*
  - An *$O(n^2)$ algorithm running on a million-element array would require a trillion "operations" (where each could actually require several machine instructions to execute).*
  - A billion-element array would require a quintillion operations.
- You'll also see algorithms with more favorable Big O measures.

# 19.3.4 Big O of the Linear Search

- The linear search algorithm runs in $O(n)$ time.
  - The worst case in this algorithm is that every element must be checked to determine whether the search item exists in the array.
  - If the size of the array is *doubled*, the number of comparisons that the algorithm must perform is also *doubled*.
- Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the array.
  - We seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the array.
- If a program needs to perform many searches on large arrays, it's better to implement a more efficient algorithm, such as the binary search.

## Performance Tip 19.1

*Sometimes the simplest algorithms perform poorly. Their virtue is that they're easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.*

# 19.4 Binary Search

- The binary search algorithm is more efficient than linear search, but it requires that the array be sorted.
  - The first iteration tests the middle element in the array. If this matches the search key, the algorithm ends.
  - If the search key is less than the *middle* element, the algorithm continues with only the first half of the array.
  - If the search key is *greater than* the middle element, the algorithm continues with only the second half.
  - Each iteration tests the middle value of the remaining portion of the array.
  - If the search key does not match the element, the algorithm eliminates half of the remaining elements.
  - The algorithm ends by either finding an element that matches the search key or reducing the subarray to zero size.

# 19.4.1 Binary Search Implementation

- Method sort sorts the array data's elements in an array in ascending order (by default).

```
1  // Fig. 19.3: BinarySearchTest.java
2  // Use binary search to locate an item in an array.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class BinarySearchTest
8  {
9     // perform a binary search on the data
10    public static int binarySearch(int[] data, int key)
11    {
12       int low = 0; // low end of the search area
13       int high = data.length - 1; // high end of the search area
14       int middle = (low + high + 1) / 2; // middle element
15       int location = -1; // return value; -1 if not found
16
17       do // loop to search for element
18       {
19          // print remaining elements of array
20          System.out.print(remainingElements(data, low, high));
21
```

**Fig. 19.3** | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 1 of 7.)

```
22            // output spaces for alignment
23            for (int i = 0; i < middle; i++)
24               System.out.print("   ");
25            System.out.println(" * "); // indicate current middle
26
27            // if the element is found at the middle
28            if (key == data[middle])
29               location = middle; // location is the current middle
30            else if (key < data[middle]) // middle element is too high
31               high = middle - 1; // eliminate the higher half
32            else  // middle element is too low
33               low = middle + 1; // eliminate the lower half
34
35            middle = (low + high + 1) / 2; // recalculate the middle
36         } while ((low <= high) && (location == -1));
37
38         return location; // return location of search key
39      } // end method binarySearch
40
```

**Fig. 19.3** | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 2 of 7.)

```
41      // method to output certain values in array
42      private static String remainingElements(int[] data, int low, int high)
43      {
44         StringBuilder temporary = new StringBuilder();
45
46         // append spaces for alignment
47         for (int i = 0; i < low; i++)
48            temporary.append("   ");
49
50         // append elements left in array
51         for (int i = low; i <= high; i++)
52            temporary.append(data[i] + " ");
53
54         return String.format("%s%n", temporary);
55      } // end method 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]; // create array
63
```

**Fig. 19.3** | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 3 of 7.)

```
64          for (int i = 0; i < data.length; i++) // populate array
65             data[i] = 10 + generator.nextInt(90);
66
67          Arrays.sort(data); // binarySearch requires sorted array
68          System.out.printf("%s%n%n", Arrays.toString(data)); // display array
69
70          // get input from user
71          System.out.print("Please enter an integer value (-1 to quit): ");
72          int searchInt = input.nextInt();
73
74          // repeatedly input an integer; -1 terminates the program
75          while (searchInt != -1)
76          {
77             // perform search
78             int location = binarySearch(data, searchInt);
79
80             if (location == -1) // not found
81                System.out.printf("%d was not found%n%n", searchInt);
82             else // found
83                System.out.printf("%d was found in position %d%n%n",
84                   searchInt, location);
85
```

**Fig. 19.3** | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 4 of 7.)

```
86             // get input from user
87             System.out.print("Please enter an integer value (-1 to quit): ");
88             searchInt = input.nextInt();
89          }
90       } // end main
91    } // end class BinarySearchTest
```

**Fig. 19.3** | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 5 of 7.)

```
[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
```

**Fig. 19.3** | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 6 of 7.)

```
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
```

**Fig. 19.3** | Use binary search to locate an item in an array (the * in the output marks the middle element). (Part 7 of 7.)

# 19.4.2 Efficiency of the Binary Search

- In the worst-case scenario, searching a sorted array of 1023 elements takes *only 10 comparisons* when using a binary search.
  - The number 1023 ($2^{10} - 1$) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test.
  - Dividing by 2 is equivalent to one comparison in the binary search algorithm.
- Thus, an array of 1,048,575 ($2^{20} - 1$) elements takes a *maximum of 20 comparisons* to find the key, and an array of over one billion elements takes a maximum of 30 comparisons to find the key.
  - A difference between an average of 500 million comparisons for the linear search and a *maximum of only 30 comparisons* for the binary search!

# 19.4.2 Efficiency of the Binary Search (cont.)

- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array, which is represented as $\log_2 n$.
- All logarithms grow at roughly the same rate, so in big O notation the base can be omitted.
- This results in a big O of O(log n) for a binary search, which is also known as logarithmic run time and pronounced as "order log n."

# 19.5 Sorting Algorithms

- Sorting data (i.e., placing the data into some particular order, such as ascending or descending) is one of the most important computing applications.
- An important item to understand about sorting is that the end result—the sorted array—will be the same no matter which algorithm you use to sort the array.
- The choice of algorithm affects only the run time and memory use of the program.
- The rest of this chapter introduces three common sorting algorithms.
  - The first two—selection sort and insertion sort—are easy to program but inefficient.
  - The last algorithm—merge sort—is much faster than selection sort and insertion sort but harder to program.

# 19.6 Selection Sort

- Selection sort
  - simple, but inefficient, sorting algorithm
- Its first iteration selects the *smallest* element in the array and swaps it with the first element.
- The second iteration selects the *second-smallest* item (which is the smallest item of the remaining elements) and swaps it with the second element.
- The algorithm continues until the last iteration selects the *second-largest* element and swaps it with the second-to-last index, leaving the largest element in the last index.
- After the ith iteration, the smallest *i* items of the array will be sorted into increasing order in the first *i* elements of the array.
- After the first iteration, the smallest element is in the first position. After the second iteration, the two smallest elements are in order in the first two positions, etc.
- The selection sort algorithm runs in $O(n^2)$ time.

# 19.6.1 Selection Sort Implementation

- Class `SelectionSortTest` (Fig. 19.4) contains:
  - `static` method `selectionSort` to sort an `int` array using the selection sort algorithm
  - `static` method `swap` to swap the values of two array elements
  - `static` method `printPass` to display the array contents after each pass, and
  - `main` to test method `selectionSort`.

```
1  // Fig. 19.4: SelectionSortTest.java
2  // Sorting an array with selection sort.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class SelectionSortTest
7  {
8     // sort array using selection sort
9     public static void selectionSort(int[] data)
10    {
11       // loop over data.length - 1 elements
12       for (int i = 0; i < data.length - 1; i++)
13       {
14          int smallest = i; // first index of remaining array
15
16          // loop to find index of smallest element
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); // swap smallest element into position
22          printPass(i + 1, smallest); // output pass of algorithm
23       }
24    } // end method selectionSort
```

**Fig. 19.4** | Sorting an array with selection sort. (Part 1 of 4.)

```
25
26      // helper method to swap values in two elements
27      private static void swap(int[] data, int first, int second)
28      {
29         int temporary = data[first]; // store first in temporary
30         data[first] = data[second]; // replace first with second
31         data[second] = temporary; // put temporary in second
32      }
33
34      // print a pass of the algorithm
35      private static void printPass(int[] data, int pass, int index)
36      {
37         System.out.printf("after pass %2d: ", pass);
38
39         // output elements till selected item
40         for (int i = 0; i < index; i++)
41            System.out.printf("%d  ", data[i]);
42
43         System.out.printf("%d* ", data[index]); // indicate swap
44
45         // finish outputting array
46         for (int i = index + 1; i < data.length; i++)
47            System.out.printf("%d  ", data[i]);
48
49         System.out.printf("%n               "); // for alignment
```

**Fig. 19.4** | Sorting an array with selection sort. (Part 2 of 4.)

```
50
51          // indicate amount of array that's sorted
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]; // create array
62
63          for (int i = 0; i < data.length; i++) // populate array
64             data[i] = 10 + generator.nextInt(90);
65
66          System.out.printf("Unsorted array:%n%s%n%n",
67             Arrays.toString(data)); // display array
68          selectionSort(data); // sort array
69
70          System.out.printf("Sorted array:%n%s%n%n",
71             Arrays.toString(data)); // display array
72       }
73    } // end class SelectionSortTest
```

**Fig. 19.4** | Sorting an array with selection sort. (Part 3 of 4.)

```
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]
```

**Fig. 19.4** | Sorting an array with selection sort. (Part 4 of 4.)

# 19.7 Insertion Sort

- Insertion sort
  - another *simple*, *but inefficient*, sorting algorithm
- The first iteration takes the *second element* in the array and, if it's *less than the first element*, *swaps it with the first element*.
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two, so all three elements are in order.
- At the *i*th iteration of this algorithm, the first *i* elements in the original array will be sorted.
- The insertion sort algorithm also runs in $O(n^2)$ time.

# 19.7.1 Insertion Sort Implementation

- Class `InsertionSortTest` (Fig. 19.5) contains:
  - `static` method `insertionSort` to sort `int`s using the insertion sort algorithm
  - `static` method `printPass` to display the array contents after each pass, and
  - `main` to test method `insertionSort`.

```
1  // Fig. 19.5: InsertionSortTest.java
2  // Sorting an array with insertion sort.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class InsertionSortTest
7  {
8     // sort array using insertion sort
9     public static void insertionSort(int[] data)
10    {
11       // loop over data.length - 1 elements
12       for (int next = 1; next < data.length; next++)
13       {
14          int insert = data[next]; // value to insert
15          int moveItem = next; // location to place element
16
17          // search for place to put current element
18          while (moveItem > 0 && data[moveItem - 1] > insert)
19          {
20             // shift element right one slot
21             data[moveItem] = data[moveItem - 1];
22             moveItem--;
23          }
24
```

**Fig. 19.5** | Sorting an array with insertion sort. (Part 1 of 5.)

```
25            data[moveItem] = insert; // place inserted element
26            printPass(data, next, moveItem); // output pass of algorithm
27         }
28      }
29
30      // print a pass of the algorithm
31      public static void printPass(int[] data, int pass, int index)
32      {
33         System.out.printf("after pass %2d: ", pass);
34
35         // output elements till swapped item
36         for (int i = 0; i < index; i++)
37            System.out.printf("%d  ", data[i]);
38
39         System.out.printf("%d* ", data[index]); // indicate swap
40
41         // finish outputting array
42         for (int i = index + 1; i < data.length; i++)
43            System.out.printf("%d  ", data[i]);
44
45         System.out.printf("%n               "); // for alignment
46
```

**Fig. 19.5** | Sorting an array with insertion sort. (Part 2 of 5.)

```
47          // indicate amount of array that's sorted
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]; // create array
58
59          for (int i = 0; i < data.length; i++) // populate array
60             data[i] = 10 + generator.nextInt(90);
61
62          System.out.printf("Unsorted array:%n%s%n%n",
63             Arrays.toString(data)); // display array
64          insertionSort(data); // sort array
65
66          System.out.printf("Sorted array:%n%s%n%n",
67             Arrays.toString(data)); // display array
68       }
69    } // end class InsertionSortTest
```

**Fig. 19.5** | Sorting an array with insertion sort. (Part 3 of 5.)

```
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
               --  --  --  --  --  --  --  --  --
```

**Fig. 19.5** | Sorting an array with insertion sort. (Part 4 of 5.)

```
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]
```

**Fig. 19.5** | Sorting an array with insertion sort. (Part 5 of 5.)

# 19.8 Merge Sort

- Merge sort
  - *efficient* sorting algorithm
  - conceptually *more complex* than selection sort and insertion sort
- Sorts an array by *splitting* it into two equal-sized subarrays, *sorting* each subarray, then *merging* them into one larger array.
- The implementation of merge sort in this example is recursive.
  - The base case is an array with one element, which is, of course, sorted, so the merge sort immediately returns in this case.
  - The recursion step splits the array into two approximately equal pieces, recursively sorts them, then merges the two sorted arrays into one larger, sorted array.
- Merge sort has an efficiency of O(n log n).

# 19.8.1 Merge Sort Implementation

- Figure 19.6 declares the `MergeSortTest` class, which contains:
- `static` method `mergeSort` to initiate the sorting of an `int` array using the merge sort algorithm
- static method `sortArray` to perform the recursive merge sort algorithm—this is called by method `mergeSort`
- `static` method merge to merge two sorted subarrays into a single sorted subarray
- `static` method `subarrayString` to get a subarray's `String` representation for output purposes, and
- `main` to test method `mergeSort`.

```
1  // Fig. 19.6: MergeSortTest.java
2  // Sorting an array with merge sort.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class MergeSortTest
7  {
8     // calls recursive split method to begin merge sorting
9     public static void mergeSort(int[] data)
10    {
11       sortArray(data, 0, data.length - 1); // sort entire array
12    } // end method sort
13
14    // splits array, sorts subarrays and merges subarrays into sorted array
15    private static void sortArray(int[] data, int low, int high)
16    {
17       // test base case; size of array equals 1
18       if ((high - low) >= 1) // if not base case
19       {
20          int middle1 = (low + high) / 2; // calculate middle of array
21          int middle2 = middle1 + 1; // calculate next element over
22
```

**Fig. 19.6** | Sorting an array with merge sort. (Part 1 of 9.)

```
23            // output split step
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            // split array in half; sort each half (recursive calls)
32            sortArray(data, low, middle1); // first half of array
33            sortArray(data, middle2, high); // second half of array
34
35            // merge two sorted arrays after split calls return
36            merge (data, low, middle1, middle2, high);
37         } // end if
38      } // end method sortArray
39
40      // merge two sorted subarrays into one sorted subarray
41      private static void merge(int[] data, int left, int middle1,
42         int middle2, int right)
43      {
44         int leftIndex = left; // index into left subarray
45         int rightIndex = middle2; // index into right subarray
46         int combinedIndex = left; // index into temporary working array
47         int[] combined = new int[data.length]; // working array
```

**Fig. 19.6** | Sorting an array with merge sort. (Part 2 of 9.)

```
48
49          // output two subarrays before merging
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          // merge arrays until reaching end of either
56          while (leftIndex <= middle1 && rightIndex <= right)
57          {
58             // place smaller of two current elements into result
59             // and move to next space in arrays
60             if (data[leftIndex] <= data[rightIndex])
61                combined[combinedIndex++] = data[leftIndex++];
62             else
63                combined[combinedIndex++] = data[rightIndex++];
64          }
65
66          // if left array is empty
67          if (leftIndex == middle2)
68             // copy in rest of right array
69             while (rightIndex <= right)
70                combined[combinedIndex++] = data[rightIndex++];
71          else // right array is empty
```

**Fig. 19.6** | Sorting an array with merge sort. (Part 3 of 9.)

```
72            // copy in rest of left array
73            while (leftIndex <= middle1)
74               combined[combinedIndex++] = data[leftIndex++];
75
76         // copy values back into original array
77         for (int i = left; i <= right; i++)
78            data[i] = combined[i];
79
80         // output merged array
81         System.out.printf("         %s%n%n",
82            subarrayString(data, left, right));
83      } // end method merge
84
85      // method to output certain values in array
86      private static String subarrayString(int[] data, int low, int high)
87      {
88         StringBuilder temporary = new StringBuilder();
89
90         // output spaces for alignment
91         for (int i = 0; i < low; i++)
92            temporary.append("   ");
93
```

**Fig. 19.6** | Sorting an array with merge sort. (Part 4 of 9.)

```
94          // output elements left in 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]; // create array
106
107         for (int i = 0; i < data.length; i++) // populate array
108            data[i] = 10 + generator.nextInt(90);
109
110         System.out.printf("Unsorted array:%n%s%n%n",
111            Arrays.toString(data)); // display array
112         mergeSort(data); // sort array
113
114         System.out.printf("Sorted array:%n%s%n%n",
115            Arrays.toString(data)); // display array
116      }
117   } // end class MergeSortTest
```

**Fig. 19.6** | Sorting an array with merge sort. (Part 5 of 9.)

```
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

split:    75 56
          75
             56

merge:    75
             56
          56 75
```

**Fig. 19.6** | Sorting an array with merge sort. (Part 6 of 9.)

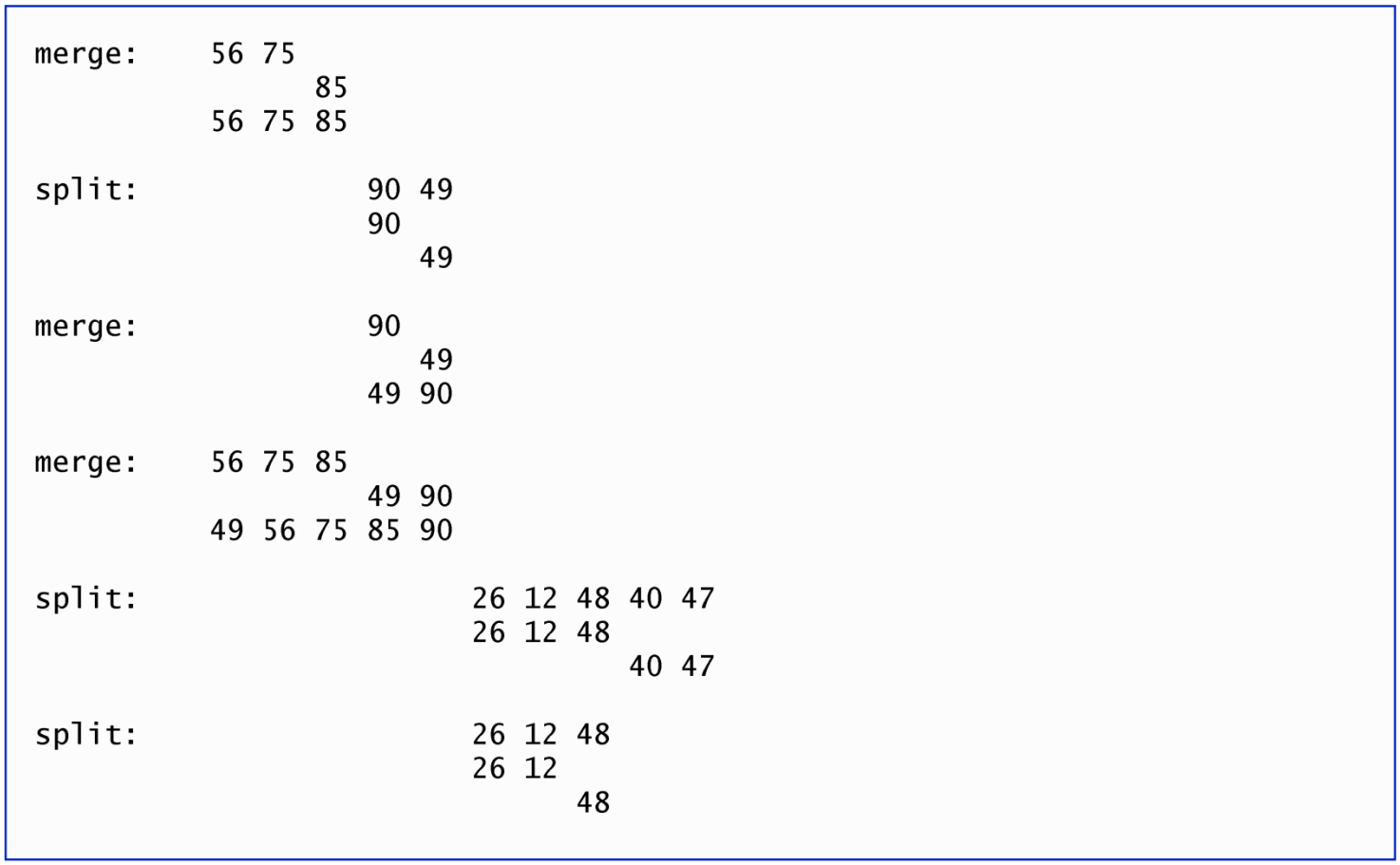


**Fig. 19.6** | Sorting an array with merge sort. (Part 7 of 9.)

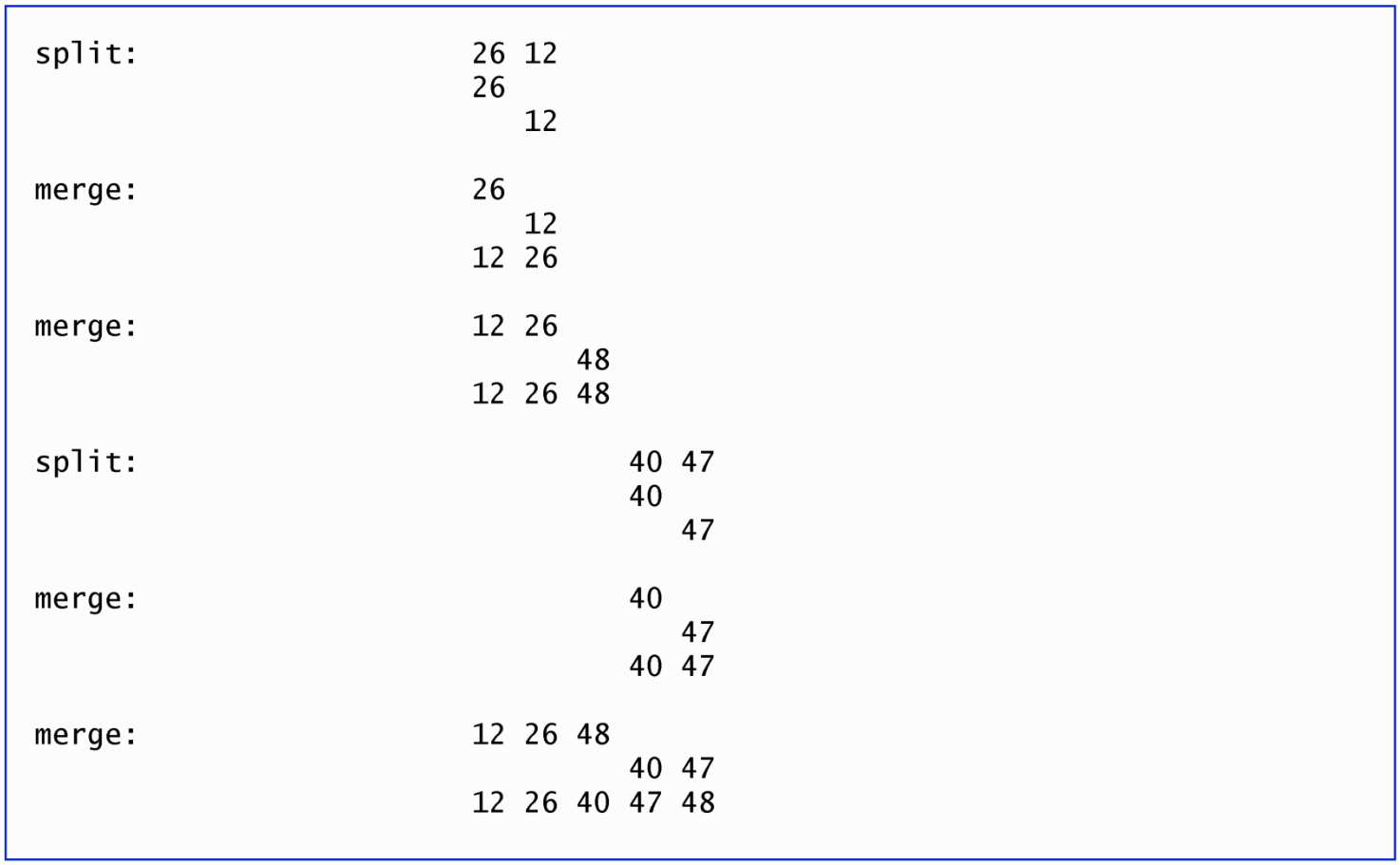


**Fig. 19.6** | Sorting an array with merge sort. (Part 8 of 9.)

```
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]
```

**Fig. 19.6** | Sorting an array with merge sort. (Part 9 of 9.)

| Algorithm | Location | Big O |
|---|---|---|
| *Searching Algorithms:* | | |
| Linear search | Section 19.2 | $O(n)$ |
| Binary search | Section 19.4 | $O(\log n)$ |
| Recursive linear search | Exercise 19.8 | $O(n)$ |
| Recursive binary search | Exercise 19.9 | $O(\log n)$ |
| *Sorting Algorithms:* | | |
| Selection sort | Section 19.6 | $O(n^2)$ |
| Insertion sort | Section 19.7 | $O(n^2)$ |
| Merge sort | Section 19.8 | $O(n \log n)$ |
| Bubble sort | Exercises 19.5 and 19.6 | $O(n^2)$ |

**Fig. 19.7** | Searching and sorting algorithms with Big O values.

| $n$ = | $O(\log n)$ | $O(n)$ | $O(n \log n)$ | $O(n^2)$ |
|---|---|---|---|---|
| 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 | $10^{18}$ |

**Fig. 19.8** | Number of comparisons for common Big O notations.