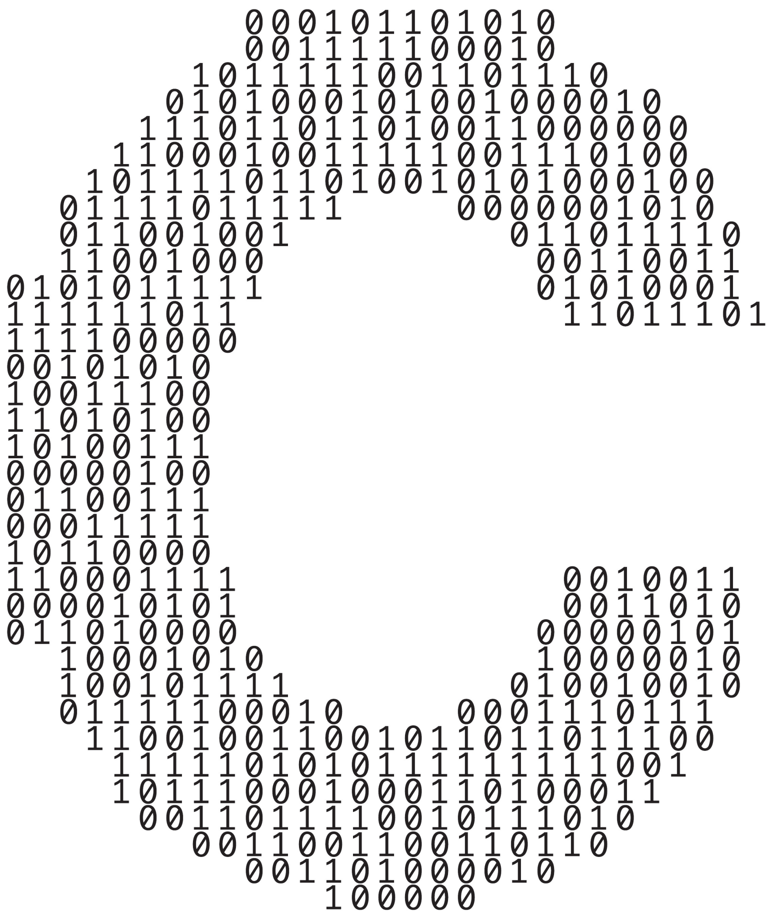
13 Computer-Science Thinking: Sorting Algorithms and Big O



Objectives

In this chapter, you'll:

- Sort an array using the selection sort algorithm.
- Sort an array using the insertion sort algorithm.
- Sort an array using the recursive merge sort algorithm.
- Learn about the efficiency of sorting algorithms and express it in "Big O" notation.
- Explore (in the exercises) other recursive sorts, including quicksort and a recursive selection sort.
- Explore (in the exercises) the high-performance bucket sort.

Outline

13.1 Introduction

In Chapter 6, you learned that sorting places data in ascending or descending order based on one or more sort keys. Here, we introduce the selection sort and insertion sort algorithms and the more efficient, but more complex, merge sort. We introduce **Big O notation**, which is used to estimate the worst-case run time for an algorithm—that is, how hard an algorithm may have to work to solve a problem.

For sorting arrays, it's important to understand that the result will be the same no matter which sorting algorithm you use. Your algorithm choice affects only the program's run time and memory use. The selection sort and insertion sort algorithms we study here are easy to program but inefficient. The third algorithm—recursive merge sort—is more efficient but harder to program.

The exercises present two more recursive sorts—quicksort and a recursive version of selection sort. Another exercise presents the bucket sort, which achieves high performance by cleverly using considerably more memory than the other sorts we discuss.

✓ Self Check

1 *(Fill-In)* Sorting places data in ascending or descending order based on one or more sort ________.
Answer: keys.

2 *(Multiple Choice)* Which of the following statements is *false*?
a) Big O notation estimates an algorithm's best-case run time—that is, how hard an algorithm may have to work to solve a problem.
b) In sorting, the sorted array will be the same no matter which sorting algorithm you use.
c) The sorting algorithm you choose affects your program's run time and memory use.
d) The selection sort and insertion sort algorithms are easy to program but inefficient. The recursive merge sort is more efficient but harder to program.
Answer: a) is *false*. Actually, Big O notation estimates the *worst-case* run time.

13.2 Efficiency of Algorithms: Big O

PERF

One way to describe an algorithm's effort is with **Big O notation**, which indicates how hard an algorithm may have to work to solve a problem. For searching and sorting algorithms, this depends mainly on how many data elements there are. In this chapter, we use Big O to describe various sorting algorithms' worst-case run times.

13.2.1 $O(1)$ Algorithms

Suppose an algorithm tests whether an array's first element is equal to its second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1,000 elements, the algorithm still requires one comparison. In fact, the algorithm is completely independent of the array's number of elements. This algorithm is said to have **constant run time**, which we represent in Big O notation as **$O(1)$** and pronounce "order 1." An $O(1)$ algorithm does not necessarily require only one comparison. $O(1)$ means that the number of comparisons is *constant*—it does not grow as the array size increases. An algorithm that tests whether the first array element is equal to any of the next three elements is still $O(1)$, even though it requires three comparisons.

13.2.2 $O(n)$ Algorithms

An algorithm that tests whether an array's first element is equal to *any* of the array's other elements requires at most $n - 1$ comparisons, where n is the array's number of elements. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1,000 elements, this algorithm requires up to 999 comparisons.

As n grows larger, the n in the expression $n - 1$ "dominates," so subtracting 1 becomes inconsequential. Big O is designed to highlight these dominant terms and ignore those that become unimportant as n grows. For this reason, an algorithm that requires $n - 1$ 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 just "order n."

13.2.3 $O(n^2)$ Algorithms

Suppose an algorithm tests whether *any* array element is duplicated elsewhere in the array. The algorithm compares the first element with all of the array's other elements. The algorithm then compares the second element with all of the array's other elements except the first—the second was already compared to the first. Then, the algorithm compares the third element with all the other elements except the first two. In the end, this algorithm will end up making a total of $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Big O notation highlights the n^2 term, leaving $n^2/2$. But as we'll soon see, 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. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, the algorithm will require 64 comparisons. With this algorithm, doubling the number of elements *quadruples* the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, the algorithm will require 32 comparisons. Again, doubling the number of elements *quadruples* the number of comparisons. Both algorithms grow as the square of n, so Big O ignores the constant and both algorithms are considered to be $\boldsymbol{O(n^2)}$. This is referred to as **quadratic run time** and pronounced "on the order of n-squared" or simply "order n-squared."

When n is small, $O(n^2)$ algorithms (running on today's billion-operations-per-second personal 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. This could require a few hours to execute. A billion-element array would require a quintillion operations, a number so large that the algorithm could take decades! As you'll see in this chapter, $O(n^2)$ algorithms are easy to write. You'll also see an algorithm with a more favorable Big O measure. Efficient algorithms often require clever coding and more work to create. Their superior performance can be well worth the extra effort, especially as n gets large and as algorithms are combined into larger programs.

✓ Self Check

1 *(True/False)* $O(n^2)$ algorithms running on today's billion-operations-per-second personal computers will not noticeably affect performance.
Answer: *False*. Actually, when n is *small*, $O(n^2)$ algorithms will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation, even on today's powerful systems.

2 *(Fill-In)* Big O is concerned with how an algorithm's run time grows in relation to the ________.
Answer: number of items processed.

3 *(True/False)* An algorithm that is $O(1)$ requires only one comparison.
Answer: *False*. $O(1)$ means the number of comparisons is *constant*. The algorithm may require multiple comparisons, but that number does not grow as the array's size increases.

4 *(Fill-In)* An $O(n)$ algorithm is referred to as having a ________ run time.
Answer: linear.

5 *(Fill-In)* An $O(n^2)$ algorithm is referred to as having ________ run time.
Answer: quadratic.

13.3 Selection Sort

Selection sort is a simple but inefficient sorting algorithm:

- The algorithm's first iteration selects the array's smallest element and swaps it with the array's first element.

- The second iteration selects the second-smallest element—which is the smallest of those remaining—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 element. This leaves the largest element as the last.

After the *i*th iteration, the array's *i* smallest values will be sorted into increasing order in the array's first *i* elements.

As an example, consider the array

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

The selection sort first determines the array's smallest element (4), then swaps it with the value in element 0 (34), resulting in

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

The selection sort then determines the smallest remaining value beginning at element 1, which is the value 5 located in element 8. The program swaps 5 with the value 56 in element 1, resulting in

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

On the third iteration, the selection sort determines the next smallest value—10 in element 3—and swaps it with 34 in element 2, resulting in

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

The process continues until after nine iterations the array is fully sorted, as in

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

After the first iteration, the smallest element is in element 0. After the second iteration, the two smallest elements are in order in elements 0 and 1. After the third iteration, the three smallest elements are in order in elements 0–2, and so on.

13.3.1 Selection Sort Implementation

Figure 13.1 implements the selection sort algorithm. Lines 18–20 fill array with 10 random int values. The main function prints the unsorted array, passes array to the function selectionSort (line 29), then prints array again after it has been sorted.

```
 1  // fig13_01.c
 2  // The selection sort algorithm.
 3  #define SIZE 10
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <time.h>
 7
 8  // function prototypes
 9  void selectionSort(int array[], size_t length);
10  void swap(int array[], size_t first, size_t second);
11  void printPass(int array[], size_t length, int pass, size_t index);
```

Fig. 13.1 | The selection sort algorithm. (Part 1 of 3.)

```

int main(void) {
   int array[SIZE] = {0}; // declare the array of ints to be sorted

   srand(time(NULL)); // seed the rand function

   for (size_t i = 0; i < SIZE; i++) {
      array[i] = rand() % 90 + 10; // give each element a value
   }

   puts("Unsorted array:");

   for (size_t i = 0; i < SIZE; i++) { // print the array
      printf("%d  ", array[i]);
   }

   puts("\n");
   selectionSort(array, SIZE);
   puts("Sorted array:");

   for (size_t i = 0; i < SIZE; i++) { // print the array
      printf("%d  ", array[i]);
   }

   puts("");
}

// function that selection sorts the array
void selectionSort(int array[], size_t length) {
   // loop over length - 1 elements
   for (size_t i = 0; i < length - 1; i++) {
      size_t smallest = i; // first index of remaining array

      // loop to find index of smallest element
      for (size_t j = i + 1; j < length; j++) {
         if (array[j] < array[smallest]) {
            smallest = j;
         }
      }

      swap(array, i, smallest); // swap smallest element
      printPass(array, length, i + 1, smallest); // output pass
   }
}

// function that swaps two elements in the array
void swap(int array[], size_t first, size_t second) {
   int temp = array[first];
   array[first] = array[second];
   array[second] = temp;
}

```

Fig. 13.1 | The selection sort algorithm. (Part 2 of 3.)

```c
// function that prints a pass of the algorithm
void printPass(int array[], size_t length, int pass, size_t index) {
   printf("After pass %2d: ", pass);

   // output elements till selected item
   for (size_t i = 0; i < index; i++) {
      printf("%d  ", array[i]);
   }

   printf("%d* ", array[index]); // indicate swap

   // finish outputting array
   for (size_t i = index + 1; i < length; i++) {
      printf("%d  ", array[i]);
   }

   printf("%s", "\n               "); // for alignment

   // indicate amount of array that is sorted
   for (int i = 0; i < pass; i++) {
      printf("%s", "--  ");
   }

   puts(""); // add newline
}
```

```
Unsorted array:
72  34  88  14  32  12  34  77  56  83

After pass  1: 12  34  88  14  32  72* 34  77  56  83
               --
After pass  2: 12  14  88  34* 32  72  34  77  56  83
               --  --
After pass  3: 12  14  32  34  88* 72  34  77  56  83
               --  --  --
After pass  4: 12  14  32  34* 88  72  34  77  56  83
               --  --  --  --
After pass  5: 12  14  32  34  34  72  88* 77  56  83
               --  --  --  --  --
After pass  6: 12  14  32  34  34  56  88  77  72* 83
               --  --  --  --  --  --
After pass  7: 12  14  32  34  34  56  72  77  88* 83
               --  --  --  --  --  --  --
After pass  8: 12  14  32  34  34  56  72  77* 88  83
               --  --  --  --  --  --  --  --
After pass  9: 12  14  32  34  34  56  72  77  83  88*
               --  --  --  --  --  --  --  --  --
After pass 10: 12  14  32  34  34  56  72  77  83  88*
               --  --  --  --  --  --  --  --  --  --
Sorted array:
12  14  32  34  34  56  72  77  83  88
```

Fig. 13.1 | The selection sort algorithm. (Part 3 of 3.)

In function `selectionSort` (lines 40–55), variable `smallest` (line 43) stores the smallest remaining element's index. Lines 42–54 loop `length - 1` times. Line 43 assigns to `smallest` the index `i`—the first index in the array's unsorted portion. Lines

46–50 process the remaining elements. For each, line 47 determines whether the element's value is less than the one at index `smallest`. If so, line 48 assigns the current element's index to `smallest`. After this loop, `smallest` contains the smallest remaining element's index. Line 52 calls `swap` (lines 58–62) to exchange the values at locations `i` and `smallest`, placing the smallest remaining element at position `i` in the array.

The output uses dashes to underline the portion of the array that's guaranteed to be sorted after each pass. We place an asterisk next to the element that was swapped with the smallest element on that pass. The element to the asterisk's left and the element above the rightmost dashes were the two values that were swapped on each pass.

13.3.2 Efficiency of Selection Sort

The selection sort algorithm runs in **$O(n^2)$ time**. In our `selectionSort` function, the outer loop (lines 42–54) processes the array's first $n - 1$ elements, swapping the smallest remaining item into its sorted position. The inner loop (lines 46–50) processes the remaining items, searching for the smallest element. This loop executes $n - 1$ times during the first outer-loop iteration, $n - 2$ times during the second, then $n - 3$, …, 3, 2, 1. So, this inner loop iterates a total of $n(n - 1) / 2$ or $(n^2 - n)/2$. In Big O notation, smaller terms drop out, and constants are ignored, leaving a Big O of $O(n^2)$.

✓ Self Check

1 *(Discussion)* Consider the following array, which reflects the result of a selection sort's first pass:

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

What does the second pass do? Show the resulting array.
Answer: The second pass swaps 56 with 5 (the second smallest element), resulting in:

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

2 *(Code)* Consider the following `selectionSort` function:

```
1  void selectionSort(int array[], size_t length) {
2     // loop over length - 1 elements
3     for (size_t i = 0; i < length - 1; i++) {
4        size_t smallest = i; // first index of remaining array
5
6        // loop to find index of smallest element
7        for (size_t j = i + 1; j < length; j++) {
8           if (array[j] < array[smallest]) {
9              ???
10          }
11       }
12
13       swap(array, i, smallest); // swap smallest element
14       printPass(array, length, i + 1, smallest); // output pass
15    }
16 }
```

What statement should replace the `???` in line 9 to complete the code?
Answer: `smallest = j;`

13.4 Insertion Sort

Insertion sort is another simple but inefficient sorting algorithm. This algorithm's first iteration takes the array's second element 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 elements, so all three elements are in order. After this algorithm's *i*th iteration, the first *i* elements in the original array are sorted.

Consider as an example the following array:

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

The insertion sort's first iteration looks at the array's first two elements, `34` and `56`. These elements are already in order, so the algorithm continues. If they were out of order, the algorithm would swap them.

In the next iteration, the algorithm looks at the third value, `4`. This value is less than `56`, so the algorithm stores `4` in a temporary variable and moves `56` one element to the right. The algorithm then determines that `4` is less than `34`, so it moves `34` one element to the right. The program has now reached the beginning of the array, so it places `4` in element `0`, resulting in

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

In the next iteration, the algorithm stores the value `10` in a temporary variable. Then the program compares `10` to `56` and moves `56` one element to the right because it's larger than `10`. The program then compares `10` to `34`, moving `34` right one element. When the program compares `10` to `4`, it observes that `10` is larger than `4` and places `10` in element `1`, resulting in

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

After the *i*th iteration, the array's first $i + 1$ elements are sorted with respect to one another. However, they may not be in their final locations, because there may be smaller values later in the array.

13.4.1 Insertion Sort Implementation

Figure 13.2 implements the insertion sort algorithm. In function `insertionSort` (lines 39–55), the variable `insert` (line 43) holds the element you're going to insert until we've moved the other elements. Lines 41–54 process the array's items from index `1` through the end. Each iteration stores in `moveItem` (line 42) the location where an item will be inserted and stores in `insert` (line 43) the value that will be inserted into its sorted portion. Lines 46–50 locate the position at which to insert the element. This loop terminates either when the algorithm reaches the front of the array or reaches an element that's less than the value to insert. Line 48 moves an element to the right, and line 49 decrements the position at which to insert the next element. After the nested loop ends, line 52 inserts the element into place. The program's output uses dashes to indicate the portion of the array that's sorted after each pass. We place an asterisk next to the element that was inserted into place on that pass.

```c
// fig13_02.c
// The insertion sort algorithm.
#define SIZE 10
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// function prototypes
void insertionSort(int array[], size_t length);
void printPass(int array[], size_t length, int pass, size_t index);

int main(void) {
   int array[SIZE] = {0}; // declare the array of ints to be sorted

   srand(time(NULL)); // seed the rand function

   for (size_t i = 0; i < SIZE; i++) {
      array[i] = rand() % 90 + 10; // give each element a value
   }

   puts("Unsorted array:");

   for (size_t i = 0; i < SIZE; i++) { // print the array
      printf("%d  ", array[i]);
   }

   puts("\n");
   insertionSort(array, SIZE);
   puts("Sorted array:");

   for (size_t i = 0; i < SIZE; i++) { // print the array
      printf("%d  ", array[i]);
   }

   puts("");
}

// function that sorts the array
void insertionSort(int array[], size_t length) {
   // loop over length - 1 elements
   for (size_t i = 1; i < length; i++) {
        size_t moveItem = i; // initialize location to place element
        int insert = array[i]; // holds element to insert

        // search for place to put current element
        while (moveItem > 0 && array[moveItem - 1] > insert) {
            // shift element right one slot
            array[moveItem] = array[moveItem - 1];
            --moveItem;
        }

        array[moveItem] = insert; // place inserted element
```

Fig. 13.2 | The insertion sort algorithm. (Part 1 of 2.)

```
       printPass(array, length, i, moveItem);
   }
}

// function that prints a pass of the algorithm
void printPass(int array[], size_t length, int pass, size_t index) {
   printf("After pass %2d: ", pass);

   // output elements till selected item
   for (size_t i = 0; i < index; i++) {
       printf("%d  ", array[i]);
   }

   printf("%d* ", array[index]); // indicate swap

   // finish outputting array
   for (size_t i = index + 1; i < length; i++) {
       printf("%d  ", array[i]);
   }

   printf("%s", "\n               "); // for alignment

   // indicate amount of array that is sorted
   for (size_t i = 0; i <= pass; i++) {
       printf("%s", "--  ");
   }

   puts(""); // add newline
}
```

```
Unsorted array:
72  16  11  92  63  99  59  82  99  30

After pass  1: 16* 72  11  92  63  99  59  82  99  30
               --  --
After pass  2: 11* 16  72  92  63  99  59  82  99  30
               --  --  --
After pass  3: 11  16  72  92* 63  99  59  82  99  30
               --  --  --  --
After pass  4: 11  16  63* 72  92  99  59  82  99  30
               --  --  --  --  --
After pass  5: 11  16  63  72  92  99* 59  82  99  30
               --  --  --  --  --  --
After pass  6: 11  16  59* 63  72  92  99  82  99  30
               --  --  --  --  --  --  --
After pass  7: 11  16  59  63  72  82* 92  99  99  30
               --  --  --  --  --  --  --  --
After pass  8: 11  16  59  63  72  82  92  99  99* 30
               --  --  --  --  --  --  --  --  --
After pass  9: 11  16  30* 59  63  72  82  92  99  99
               --  --  --  --  --  --  --  --  --  --

Sorted array:
11  16  30  59  63  72  82  92  99  99
```

Fig. 13.2 | The insertion sort algorithm. (Part 2 of 2.)

13.4.2 Efficiency of Insertion Sort

Like selection sort, the insertion sort algorithm runs in $O(n^2)$ time. Like our function `selectionSort` in Section 13.3.1, the `insertionSort` function uses nested loops. The outer loop (lines 41–54) iterates `SIZE - 1` times, inserting an element into the appropriate position in the elements sorted so far. For this application's purposes, `SIZE - 1` is equivalent to $n - 1$, as `SIZE` is the array's number of elements. The nested loop (lines 46–50) iterates over the array's preceding elements. In the worst case, this `while` loop requires $n - 1$ comparisons. Each individual loop runs in $O(n)$ time. In Big O notation, you must multiply the number of iterations of each loop in nested loops. For each outer-loop iteration, there will be a certain number of inner-loop iterations. In this algorithm, for each $O(n)$ outer-loop iterations, there will be $O(n)$ inner-loop iterations. Multiplying these values results in a Big O of $O(n^2)$.

✓ Self Check

1 *(Fill-In)* The insertion sort algorithm's first iteration takes the array's second element 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. After the algorithm's *i*th iteration, the array's ________ elements will be sorted.
Answer: first *i*.

2 *(Discussion)* The insertion sort and selection sort algorithms both run in $O(n^2)$ time. What program structure that they each have causes the $O(n^2)$ run time?
Answer: They each have a nested `for`-loop.

13.5 Case Study: Visualizing the High-Performance Merge Sort

The merge sort algorithm is efficient but conceptually more complex than selection sort and insertion sort. The merge sort algorithm sorts an array by splitting it into two equal-sized subarrays, sorting each subarray, then merging them into one larger array. With an odd number of elements, the algorithm creates the two subarrays, such that one has one more element than the other.

Our merge sort implementation in this example is recursive. The base case is a one-element array, which is, of course, sorted. So, merge sort immediately returns when it's called with a one-element array. The recursion step splits an array of two or more elements into two equal-sized subarrays, recursively sorts each subarray, then merges them into one larger, sorted array. Again, if there are an odd number of elements, one subarray is one element larger than the other.

Suppose the algorithm has already merged smaller arrays to create sorted arrays A:

```
4    10    34    56    77
```

and B:

```
5    30    51    52    93
```

Merge sort combines these two arrays into one larger, sorted array. The smallest element in A is 4 (located in element 0). The smallest element in B is 5 (located in element 0). To determine the smallest element in the merged array, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the first element in the merged array. Next, the algorithm compares 10 (element 1 in A) to 5 (element 0 in B). The value from B is smaller, so 5 becomes the second element in the merged array. The algorithm continues by comparing 10 to 30, with 10 becoming the third element in the array, and so on.

13.5.1 Merge Sort Implementation

Figure 13.3 implements the merge sort algorithm. Lines 35–37 define the `mergeSort` function. Line 36 calls function `sortSubArray` with the arguments `0` and `length - 1` (`length` is the array's size). The arguments are the beginning and ending subscripts of the array to sort, so this call to `sortSubArray` operates on the entire array. Lines 40–62 define the `sortSubArray` function. Line 42 tests the base case. If the subarray size is `1`, the subarray is sorted, so the function simply returns immediately. If the subarray's size is greater than `1`, the function splits the subarray in two, recursively calls function `sortSubArray` to sort the two halves, then merges them. Line 56 recursively calls function `sortSubArray` for the subarray's first half, and line 57 recursively calls function `sortSubArray` for the subarray's second half. When these two calls return, each half is sorted. Line 60 calls function `merge` (lines 65–111) on the two halves to combine them into one larger sorted subarray.

```
 1  // fig13_03.c
 2  // The merge sort algorithm.
 3  #define SIZE 10
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <time.h>
 7
 8  // function prototypes
 9  void mergeSort(int array[], size_t length);
10  void sortSubArray(int array[], size_t low, size_t high);
11  void merge(int array[], size_t left, size_t middle1,
12     size_t middle2, size_t right);
13  void displayElements(int array[], size_t length);
14  void displaySubArray(int array[], size_t left, size_t right);
15
16  int main(void) {
17     int array[SIZE] = {0}; // declare the array of ints to be sorted
18
19     srand(time(NULL)); // seed the rand function
20
21     for (size_t i = 0; i < SIZE; i++) {
22        array[i] = rand() % 90 + 10; // give each element a value
23     }
24
```

Fig. 13.3 | The merge sort algorithm. (Part 1 of 5.)

```c
   puts("Unsorted array:");
   displayElements(array, SIZE); // print the array
   puts("\n");
   mergeSort(array, SIZE); // merge sort the array
   puts("Sorted array:");
   displayElements(array, SIZE); // print the array
   puts("");
}

// function that merge sorts the array
void mergeSort(int array[], size_t length) {
   sortSubArray(array, 0, length - 1);
}

// function that sorts a piece of the array
void sortSubArray(int array[], size_t low, size_t high) {
   // test base case: size of array is 1
   if ((high - low) >= 1) { // if not base case...
      size_t middle1 = (low + high) / 2;
      size_t middle2 = middle1 + 1;

      // output split step
      printf("%s", "split:   ");
      displaySubArray(array, low, high);
      printf("%s", "\n         ");
      displaySubArray(array, low, middle1);
      printf("%s", "\n         ");
      displaySubArray(array, middle2, high);
      puts("\n");

      // split array in half and sort each half recursively
      sortSubArray(array, low, middle1); // first half
      sortSubArray(array, middle2, high); // second half

      // merge the two sorted arrays
      merge(array, low, middle1, middle2, high);
   }
}

// merge two sorted subarrays into one sorted subarray
void merge(int array[], size_t left, size_t middle1,
   size_t middle2, size_t right) {
   size_t leftIndex = left; // index into left subarray
   size_t rightIndex = middle2; // index into right subarray
   size_t combinedIndex = left; // index into temporary array
   int tempArray[SIZE] = {0}; // temporary array

   // output two subarrays before merging
   printf("%s", "merge:   ");
   displaySubArray(array, left, middle1);
   printf("%s", "\n         ");
   displaySubArray(array, middle2, right);
   puts("");
```

Fig. 13.3 | The merge sort algorithm. (Part 2 of 5.)

```

   // merge the subarrays until the end of one is reached
   while (leftIndex <= middle1 && rightIndex <= right) {
      // place the smaller of the two current elements in result
      // and move to the next space in the subarray
      if (array[leftIndex] <= array[rightIndex]) {
         tempArray[combinedIndex++] = array[leftIndex++];
      }
      else {
         tempArray[combinedIndex++] = array[rightIndex++];
      }
   }

   if (leftIndex == middle2) { // if at end of left subarray ...
      while (rightIndex <= right) { // copy the right subarray
         tempArray[combinedIndex++] = array[rightIndex++];
      }
   }
   else { // if at end of right subarray...
      while (leftIndex <= middle1) { // copy the left subarray
         tempArray[combinedIndex++] = array[leftIndex++];
      }
   }

   // copy values back into original array
   for (size_t i = left; i <= right; i++) {
      array[i] = tempArray[i];
   }

   // output merged subarray
   printf("%s", "         ");
   displaySubArray(array, left, right);
   puts("\n");
}

// display elements in array
void displayElements(int array[], size_t length) {
   displaySubArray(array, 0, length - 1);
}

// display certain elements in array
void displaySubArray(int array[], size_t left, size_t right) {
   // output spaces for alignment
   for (size_t i = 0; i < left; i++) {
      printf("%s", "   ");
   }

   // output elements left in array
   for (size_t i = left; i <= right; i++) {
      printf(" %d", array[i]);
   }
}
```

Fig. 13.3 | The merge sort algorithm. (Part 3 of 5.)

```
Unsorted array:
 79 86 60 79 76 71 44 88 58 23

split:    79 86 60 79 76 71 44 88 58 23
          79 86 60 79 76
                         71 44 88 58 23

split:    79 86 60 79 76
          79 86 60
                   79 76

split:    79 86 60
          79 86
                60

split:    79 86
          79
             86

merge:    79
             86
          79 86

merge:    79 86
                60
          60 79 86

split:             79 76
                   79
                      76

merge:             79
                      76
                   76 79

merge:    60 79 86
                   76 79
          60 76 79 79 86

split:                   71 44 88 58 23
                         71 44 88
                                  58 23

split:                   71 44 88
                         71 44
                               88

split:                   71 44
                         71
                            44

merge:                   71
                            44
                         44 71

merge:                   44 71
                               88
                         44 71 88
```

Fig. 13.3 | The merge sort algorithm. (Part 4 of 5.)

```
split:                          58 23
                                58
                                   23

merge:                          58
                                   23
                                23 58

merge:                 44 71 88
                                23 58
                       23 44 58 71 88

merge:    60 76 79 79 86
                       23 44 58 71 88
          23 44 58 60 71 76 79 79 86 88

Sorted array:
 23 44 58 60 71 76 79 79 86 88
```

Fig. 13.3 | The merge sort algorithm. (Part 5 of 5.)

Lines 80–89 in function `merge` loop until reaching the end of either subarray. Line 83 tests which element at the beginning of the subarrays is smaller. If the left subarray element is smaller or equal, line 84 places it in position in the merged array. If the right subarray element is smaller, line 87 places it in position in the merged array. When the `while` loop completes, one entire subarray is placed in the merged array, but the other still contains data. Line 91 tests whether we reached the left subarray's end. If so, lines 92–94 add the right subarray's remaining elements to the merged array. Otherwise, we reached the right subarray's end, and lines 97–99 add the left subarray's remaining elements to the merged array. Finally, lines 103–105 copy `tempArray`'s values into the correct portion of the original array. This program's output displays the splits and merges performed by merge sort, showing the sort's progress at each step of the algorithm.

13.5.2 Efficiency of Merge Sort

Merge sort is a far more efficient algorithm than either insertion sort or selection sort (although that may be difficult to believe when looking at the busy output in Fig. 13.3). Consider the first (nonrecursive) call to function `sortSubArray`. This results in

- two recursive calls to function `sortSubArray` with subarrays that are each approximately half the original array's size, and
- a single call to function `merge`.

The `merge` call requires, at worst, $n - 1$ comparisons to fill the original array, which is $O(n)$. Recall that each merged element is chosen by comparing one element from each subarray. The two calls to `sortSubArray` result in

- four more recursive calls to `sortSubArray`, each with a subarray approximately one-quarter the original array's size, and
- two more calls to function `merge`.

These two calls to the function `merge` each require, at worst, $n/2 - 1$ comparisons, for a total of $O(n)$ comparisons. This process continues with each call to `sortSubArray` generating two additional calls to `sortSubArray` and a call to `merge` until the algorithm has split the original array into one-element subarrays. At each level, $O(n)$ comparisons are required to merge the subarrays. Each level splits the arrays in half, so doubling the array size requires one more level. Quadrupling the array size requires two more levels. This pattern is logarithmic and results in $\log_2 n$ levels. This results in a total efficiency of $O(n \log n)$.

Supporting Exercises

This Merge Sort Visualization Case Study is supported by exercises on other complex sorts: Exercise 13.6 (Bucket Sort) and Exercise 13.7 (Quicksort). The bucket sort achieves high performance by cleverly using considerably more memory than the other sorts we discuss—this is an example of a space–time trade-off.

13.5.3 Summarizing Various Algorithms' Big O Notations

The following table summarizes the Big O of the sorting algorithms we've covered and the quicksort algorithm, which you'll implement in Exercise 13.7.

Algorithm	Big O
Insertion sort	$O(n^2)$
Selection sort	$O(n^2)$
Merge sort	$O(n \log n)$
Bubble sort	$O(n^2)$
Quicksort	Worst case: $O(n^2)$ Average case: $O(n \log n)$

The following table lists the Big O values we've covered in this chapter along with a number of values for n to highlight the differences in the growth rates. The table includes $O(\log n)$, which is the Big O for binary search you learned in Chapter 6.

n	Approximate decimal value	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	$10 \cdot 2^{10}$	2^{20}
2^{20}	1,000,000	20	2^{20}	$20 \cdot 2^{20}$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$30 \cdot 2^{30}$	2^{60}

✓ Self Check

1 *(Discussion)* The recursive merge sort algorithm sorts an array by splitting it into two equal-size subarrays, sorting each subarray, then merging them into one larger

array. The subarrays in merge sort are not sorted with sorts we've covered, such as the bubble sort, selection sort or insertion sort. Explain how the subarrays actually are sorted in merge sort.

Answer: If the array has an even number of elements, the merge sort splits the array into two equal-size halves. If the array has an odd number of elements, one "half" has one more element than the other "half."

Each half is then recursively split into two smaller halves. This halving process continues until each half contains only one element. This is the base case of the recursion, because a one-element array is sorted.

Next, those individual elements are merged into a two-element sorted subarray based on their values. So, in answer to the question, th*e sorting is actually trivial*. As the algorithm's recursion unwinds, merge sort keeps merging smaller sorted subarrays to form larger sorted subarrays. Every merge of two sorted subarrays results in a larger sorted subarray that's about double the size of the ones being merged. The last merge results in a sorted version of the original array.

2 *(Discussion)* Here are the first three lines of our merge sort example's output:

```
split:    79 86 60 79 76 71 44 88 58 23
          79 86 60 79 76
                         71 44 88 58 23
```

and here are the last three lines of the output:

```
merge:    60 76 79 79 86
                         23 44 58 71 88
          23 44 58 60 71 76 79 79 86 88
```

Compare these outptus. How are your observations consistent with how the merge sort works?

Answer: 1. In the *final merge phase*, each subarray corresponds to one of the unsorted subarrays produced by the algorithm's *first split pass*. 2. Each subarray entering the final merge phase is *sorted*. 3. The 10-element *final merged array* contains the same elements as the original unsorted array that entered the first split pass. 4. The 10-element *final merged array* is, in fact, *sorted*. The recursive merge sort splits the original unsorted array into smaller and smaller pieces until they are down to single-element pieces, which it ultimately merges to form the smallest sorted pieces of the original array. It keeps merging those sorted pieces to form larger and larger sorted pieces until it finally merges the two—now sorted—halves of the original unsorted array to form the final sorted array.

Summary

Section 13.1 Introduction

- **Sorting** involves arranging data into order.

Section 13.2, Efficiency of Algorithms: Big O

- One way to describe an algorithm's efficiency is with **Big O notation** (O; p. 712), which indicates how hard an algorithm may have to work to solve a problem.

- For searching and sorting algorithms, Big O describes how an algorithm's **amount of effort** varies, depending on how many elements are in the data.
- An $O(1)$ algorithm is said to have a **constant run time** (p. 713). This does not mean that the algorithm requires only one comparison. It just means that the number of comparisons does not grow as the size of the array increases.
- An $O(n)$ algorithm is referred to as having a **linear run time** (p. 713).
- An $O(n^2)$ algorithm is referred to as having a **quadratic run time** (p. 714).
- Big O is designed to highlight dominant factors and ignore terms that become unimportant with high values of n.
- Big O notation is concerned with the growth rate of algorithm run times, so constants are ignored.

Section 13.3, Selection Sort

- The first iteration of a **selection sort** (p. 714) selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (the smallest of those remaining) and swaps it with the second element. Selection sort continues until the last iteration selects the second-largest element and swaps it with the second-to-last, leaving the largest element as the last. At the ith iteration of selection sort, the array's smallest i elements are sorted into the array's first i positions.
- The selection sort algorithm runs in $O(n^2)$ time (p. 718).

Section 13.4, Insertion Sort

- The first iteration of **insertion sort** (p. 719) 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 of insertion sort looks at the third element and inserts it in the correct position with respect to the first two elements. After the ith iteration of insertion sort, the first i elements in the original array are sorted. Only $n - 1$ iterations are required.
- The insertion sort algorithm runs in $O(n^2)$ time (p. 722).

Section 13.5, Case Study: Visualizing the High-Performance Merge Sort

- The **merge sort algorithm** (p. 722) is faster but more complex to implement than selection sort and insertion sort.
- The merge sort algorithm sorts an array by **splitting** it into two equal-size **subarrays**, sorting each subarray and **merging** the subarrays into one larger array.
- Merge sort's base case is an array with one element, which is already sorted, so merge sort immediately returns when it's called with a one-element array. The merge part of merge sort takes two sorted arrays (these could be one-element arrays) and combines them into one larger sorted array.
- The merge is performed by looking at each array's first element, which is also the smallest element. Merge sort places the smallest of these in the first element of the larger, sorted array. If there are still elements in the subarray, merge sort looks at the second element in that subarray (which is now the smallest element remaining) and compares it to the first element in the other subarray. Merge sort continues this process until all the elements in one of the subarrays has been processed. Then, merge sort adds the remaining elements of the other subarray to the larger array.
- The merging portion of the merge sort algorithm is performed on two subarrays, each of approximately size $n/2$. Creating each subarray requires $n/2–1$ comparisons for each subar-

ray, or $O(n)$ comparisons total. This pattern continues, as each level works on twice as many arrays, but each is half the previous array's size.

- This halving results in log n levels, each level requiring $O(n)$ comparisons, for a total efficiency of $\mathbf{O(n \log n)}$ (p. 728), which is far more efficient than $O(n^2)$.

Self-Review Exercises

13.1 Fill-In the blanks in each of the following statements:

a) A selection sort application would take approximately ______ times as long to run on a 128-element array as on a 32-element array.

b) The efficiency of merge sort is ______.

13.2 The Big O of the linear search is $O(n)$, and the Big O of the binary search is $O(\log n)$. What key aspect of both the binary search (Chapter 6) and the merge sort accounts for the logarithmic portion of their respective Big Os?

13.3 In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?

13.4 In the text, we say that after the merge sort splits the array into two subarrays, it then sorts these two subarrays and merges them. Why might someone be puzzled by our statement that "it then sorts these two subarrays"?

Answers to Self-Review Exercises

13.1 a) 16, because an $O(n^2)$ algorithm takes 16 times as long to sort four times as much information. b) $O(n \log n)$.

13.2 Both algorithms incorporate "halving"—somehow reducing something by half on each pass. The binary search eliminates from consideration one-half of the array after each comparison. The merge sort splits the array in half each time it's called.

13.3 The insertion sort is easier to understand and implement than the merge sort. The merge sort is far more efficient—$O(n \log n)$—than the insertion sort—$O(n^2)$.

13.4 In a sense, it does not really sort these two subarrays. It simply keeps splitting the original array in half until it provides a one-element subarray, which is, of course, sorted. It then builds up the original two subarrays by merging these one-element arrays to form larger subarrays, which are then merged, and so on.

Exercises

13.5 (***Recursive Selection Sort***) A selection sort searches an array looking for the array's smallest element. When that element is found, it's swapped with the first element of the array. The process is then repeated for the subarray, beginning with the second element. Each pass of the array results in one element being placed in its proper location. This sort requires processing capabilities similar to bubble sort—for an array of n elements, $n - 1$ passes must be made, and for each subarray, $n - 1$ comparisons must be made to find the smallest value. When the subarray being processed contains one element, the array is sorted. Write a recursive function `selectionSort` to perform this algorithm.

13.6 (***Bucket Sort***) A bucket sort begins with a one-dimensional array of positive integers to sort, and a two-dimensional array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to $n - 1$, where n is the array's number of values to sort. Each row of the two-dimensional array is a "bucket." In this exercise, you'll write a `bucketSort` function that takes as arguments an `int` array and its size.

The algorithm is as follows:

a) Loop through the one-dimensional array and, based on each value's ones digit, place the value in a bucket (a row of the two-dimensional bucket array). For example, place 97 in row 7, 3 in row 3 and 100 in row 0.
b) Loop through the bucket array's rows and columns and copy the values back to the original array. The new order of the above values in the one-dimensional array is 100, 3 and 97.
c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, and so on) and stop when the largest number's leftmost digit has been processed.

The second pass of the array places 100 in row 0, 3 in row 0 (it had only one digit, so we treat it as 03) and 97 in row 9. After this pass, the values' order in the one-dimensional array is 100, 3 and 97. The third pass places 100 in row 1, 3 (003) in row zero and 97 (097) in row zero (after 3). The bucket sort is guaranteed to properly sort all the values after processing the leftmost digit of the largest number. The bucket sort knows it's done when all the values are copied into row zero of the two-dimensional bucket array.

The two-dimensional bucket array is ten times the size of the `int` array being sorted. This sorting technique provides far better performance than, for example, a bubble sort but requires much larger storage capacity. Bubble sort requires only one additional memory location for the type of data being sorted. Bucket sort is an example of a space–time trade-off. It uses more memory but performs better.

The bucket sort algorithm described above requires copying all the data back to the original array on each pass. Another possibility is to create a second two-dimensional bucket array and repeatedly move the data between the two bucket arrays until all the data is copied into row zero of one of the arrays. Row zero then contains the sorted array.

13.7 (***Quicksort***) We discussed various sorting techniques in the examples and exercises of Chapter 6 and this chapter. We now present the recursive quicksort sorting technique. The basic algorithm for a one-dimensional array of values is as follows:

a) *Partitioning Step:* Take the unsorted array's first element and determine its final location in the sorted array. That's the position for which all values to the element's left are less than that value, and all values to the element's right are greater than that value. We now have one element in its proper location and two unsorted subarrays.
b) *Recursive Step:* Perform the *partitioning step* on each unsorted subarray.

For each subarray, the *partitioning step* places another element in its final location of the sorted array and creates two more unsorted subarrays. A subarray consisting of one element is sorted, so that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of each subarray's first element? As an example, consider the following set of values—the element in bold is the partitioning element that will be placed in its final location in the sorted array:

37 2 6 4 89 8 10 12 68 45

a) Starting from the rightmost array element, compare each element with **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 12, so we swap **37** and 12. In the updated array below, we show *12* in italic to indicate that it was just swapped with **37**:

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

b) Starting from the array's left, but beginning with the element *after* 12, compare each element with **37** until an element greater than **37** is found, then swap **37** and that element. The first element greater than **37** is 89, so we swap **37** and 89. The updated array is

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

c) Starting from the right, but beginning with the element *before* 89, compare each element with **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 10, so we swap **37** and 10. The updated array is

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

d) Starting from the left, but beginning with the element *after* 10, compare each element with **37** until an element greater than **37** is found, then swap **37** and that element. There are no more elements greater than **37**. When we compare **37** with itself, we know that **37** is in its final location in the sorted array.

Once the partition has been applied to the array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. Quicksort continues by partitioning both subarrays in the same manner as the original array.

Write recursive function `quicksort` to sort a one-dimensional integer array. The function should receive as arguments an `int` array, a starting subscript and an ending subscript. The `quicksort` should call the function `partition` to perform the partitioning step.

This page is intentionally left blank