**Section 19.1 Introduction**

• Searching (p. [811](http://proquest.safaribooksonline.com/9780133813036/ch19_html#page_811)) involves determining if a search key is in the data and, if so, finding its location.

• Sorting (p. [811](http://proquest.safaribooksonline.com/9780133813036/ch19_html#page_811)) involves arranging data into order.

#### Section 19.2 Linear Search

• The linear search algorithm (p. [812](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec2_html#page_812)) searches each element in the array sequentially until it finds the correct element, or until it reaches the end of the array without finding the element.

#### Section 19.3 Big O Notation

• A major difference among searching algorithms is the amount of effort they require.

• Big O notation (p. [814](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec2_html#page_814)) describes an algorithm’s efficiency in terms of the work required to solve a problem. For searching and sorting algorithms typically it depends on the number of elements in the data.

• An algorithm that’s O(1) does not necessarily require only one comparison (p. [815](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec5_html#page_815)). 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. [815](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec5_html#page_815)).

• Big O highlights dominant factors and ignores terms that become unimportant with high n values.

• Big O notation is concerned with the growth rate of algorithm run times, so constants are ignored.

• The linear search algorithm runs in O(n) time.

• The worst case in linear search is that every element must be checked to determine whether the search key (p. [816](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec8_html#page_816)) exists, which occurs if the search key is the last array element or is not present.

#### Section 19.4 Binary Search

• Binary search (p. [816](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec8_html#page_816)) is more efficient than linear search, but it requires that the array be sorted.

• The first iteration of binary search tests the middle element in the array. If this is the search key, the algorithm returns its location. If the search key is less than the middle element, the search continues with the first half of the array. If the search key is greater than the middle element, the search continues with the second half of the array. Each iteration tests the middle value of the remaining array and, if the element is not found, eliminates half of the remaining elements.

• Binary search is a more efficient searching algorithm than linear search because each comparison eliminates from consideration half of the elements in the array.

• Binary search runs in O(log n) time because each step removes half of the remaining elements; this is also known as logarithmic run time (p. [820](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec10_html#page_820)).

• If the array size is doubled, binary search requires only one extra comparison.

#### Section 19.6 Selection Sort

• Selection sort (p. [821](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec5_html#page_821)) is a simple, but inefficient, sorting algorithm.

• The sort begins by selecting the smallest item and swaps it with the first element. The second iteration selects the second-smallest item (which is the smallest remaining item) and swaps it with the second element. The sort continues until the last iteration selects the second-largest element and swaps it with the second-to-last element, leaving the largest element in the last index. At the ith iteration of selection sort, the smallest i items of the whole array are sorted into the first i indices.

• The selection sort algorithm runs in O(n2) time (p. [824](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec12_html#page_824)).

#### Section 19.7 Insertion Sort

• The first iteration of insertion sort (p. [824](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec12_html#page_824)) 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 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.

• The insertion sort algorithm runs in O(n2) time (p. [827](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec14_html#page_827)).

#### Section 19.8 Merge Sort

• Merge sort (p. [827](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec14_html#page_827)) is a sorting algorithm that’s faster, but more complex to implement, than selection sort and insertion sort. The merge sort algorithm sorts an array by splitting it into two equalsized subarrays, sorting each subarray recursively and merging the subarrays into one larger array.

• Merge sort’s base case is an array with one element. A one-element array 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 and combines them into one larger sorted array.

• Merge sort performs the merge by looking at the first element in each array, which is also the smallest element in the array. Merge sort takes the smallest of these and places it in the first element of the larger array. If there are still elements in the subarray, merge sort looks at the second of these (which is now the smallest element remaining) and compares it to the first element in the other subarray. Merge sort continues this process until the larger array is filled.

• In the worst case, the first call to merge sort has to make O(n) comparisons to fill the n slots in the final array.

• The merging portion of the merge sort algorithm is performed on two subarrays, each of approximately size n/2. Creating each of these subarrays requires n /2 – 1 comparisons for each subarray, or O(n) comparisons total. This pattern continues as each level works on twice as many arrays, but each is half the size of the previous array.

• Similar to binary search, this halving results in log n levels for a total efficiency of O(n log n) (p. [833](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec17_html#page_833)).

### Self-Review Exercises

[**19.1**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec13_html#ch19ans1) 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 \_\_\_\_\_\_\_\_\_\_.

[**19.2**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec13_html#ch19ans2) What key aspect of both the binary search and the merge sort accounts for the logarithmic portion of their respective Big Os?

[**19.3**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec13_html#ch19ans3) In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?

[**19.4**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec13_html#ch19ans4) 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

[**19.1**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec12_html#ch19que1)

a) 16, because an O(n2) algorithm takes 16 times as long to sort four times as much information.

b) O(n log n).

[**19.2**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec12_html#ch19que2) Both of these algorithms incorporate “halving”—somehow reducing something by half. 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.

[**19.3**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec12_html#ch19que3) The insertion sort is easier to understand and to program than the merge sort. The merge sort is far more efficient [O(n log n)] than the insertion sort [O(n2)].

[**19.4**](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec12_html#ch19que4) 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

**19.5 (Bubble Sort)** Implement bubble sort—another simple yet inefficient sorting technique. It’s called bubble sort or sinking sort because smaller values gradually “bubble” their way to the top of the array (i.e., toward the first element) like air bubbles rising in water, while the larger values sink to the bottom (end) of the array. The technique uses nested loops to make several passes through the array. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values are equal), the bubble sort leaves the values as they are. If a pair is in decreasing order, the bubble sort swaps their values in the array. The first pass compares the first two elements of the array and swaps their values if necessary. It then compares the second and third elements in the array. The end of this pass compares the last two elements in the array and swaps them if necessary. After one pass, the largest element will be in the last index. After two passes, the largest two elements will be in the last two indices. Explain why bubble sort is an O(n2) algorithm.

**19.6 (Enhanced Bubble Sort)** Make the following simple modifications to improve the performance of the bubble sort you developed in [Exercise 19.5](http://proquest.safaribooksonline.com/9780133813036/ch19lev1sec14_html#ch19que5):

a) After the first pass, the largest number is guaranteed to be in the highest-numbered array element; after the second pass, the two highest numbers are “in place”; and so on. Instead of making nine comparisons on every pass for a ten-element array, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass, and so on.

b) The data in the array may already be in proper or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass whether any swaps have been made. If there were none, the data must already be sorted, so the program should terminate. If swaps have been made, at least one more pass is needed.

**19.7 (Bucket Sort)** A bucket sort begins with a one-dimensional array of positive integers to be sorted and a two-dimensional array of integers with rows indexed from 0 to 9 and columns indexed from 0 to n – 1, where n is the number of values to be sorted. Each row of the two-dimensional array is referred to as a bucket. Write a class named BucketSort containing a method called sort that operates as follows:

a) Place each value of the one-dimensional array into a row of the bucket array, based on the value’s “ones” (rightmost) digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This procedure is called a distribution pass.

b) Loop through the bucket array row by row, and copy the values back to the original array. This procedure is called a gathering pass. The new order of the preceding values in the one-dimensional array is 100, 3 and 97.

c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.). On the second (tens digit) pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional array is 100, 3 and 97. On the third (hundreds digit) pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After this last gathering pass, the original array is in sorted order.

The two-dimensional array of buckets is 10 times the length of the integer array being sorted. This sorting technique provides better performance than a bubble sort, but requires much more memory—the bubble sort requires space for only one additional element of data. This comparison is an example of the space/time trade-off: The bucket sort uses more memory than the bubble sort, but performs better. This version of the bucket sort 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 swap the data between the two bucket arrays.

**19.8 (Recursive Linear Search)** Modify [Fig. 19.2](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec2_html#ch19fig02) to use recursive method recursiveLinearSearch to perform a linear search of the array. The method should receive the search key and starting index as arguments. If the search key is found, return its index in the array; otherwise, return -1. Each call to the recursive method should check one index in the array.

**19.9 (Recursive Binary Search)** Modify [Fig. 19.3](http://proquest.safaribooksonline.com/9780133813036/ch19lev2sec10_html#ch19fig03) to use recursive method recursiveBinarySearch to perform a binary search of the array. The method should receive the search key, starting index and ending index as arguments. If the search key is found, return its index in the array. If the search key is not found, return -1.

**19.10 (Quicksort)** The recursive sorting technique called quicksort uses the following basic algorithm for a one-dimensional array of values:

a) Partitioning Step: Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element—we show how to do this below). We now have one element in its proper location and two unsorted subarrays.

b) Recursive Step: Perform Step 1 on each unsorted subarray. Each time Step 1 is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, that element is in its final location (because a one-element array is already sorted).

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

**37** 2  6  4  89  8  10  12  68  45

Starting from the rightmost element of the array, 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 **37** and 12 are swapped. The new array is

12 2  6  4  89  8  10  **37** 68  45

Element 12 is in italics to indicate that it was just swapped with **37**.

Starting from the left of the array, 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 **37** and 89 are swapped. The new array is

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

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 **37** and 10 are swapped. The new array is

12  2  6  4  10 8  **37** 89  68  45

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**, so when we compare **37** with itself, we know that **37** has been placed in its final location in the sorted array. Every value to the left of **37** is smaller than it, and every value to the right of **37** is larger than it.

Once the partition has been applied on the previous 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. The sort continues recursively, with both subarrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive method quickSortHelper to sort a one-dimensional integer array. The method should receive as arguments a starting index and an ending index on the original array being sorted.