9. How does the efficiency of the bubble sort compare to the other sorting algorithms in this chapter?

10. The bubble sort in Exercise 8 always makes *n* passes. However, it is possible for the array to become sorted before all *n* passes are complete. For example, a bubble sort of the array

9 2 1 6 4 7 8

is sorted after only two passes:

2 1 6 4 7 8 9 (end of pass 1)
1 2 4 6 7 8 9 (end of pass 2)

But since a swap occurred during the second pass, the sort needs to make one more pass to check that the array is in order. Additional passes, such as the ones that the algorithm in Exercise 8 would make, are unnecessary.

You can skip these unnecessary passes and even do less work by remembering where the last swap occurred. During the first pass, the last swap is of the 9 and 8. The second pass checks up to the 8. But during the second pass, the last swap is of the 6 and 4. You now know that 6, 7, 8, and 9 are sorted. The third pass needs only to check up to the 4, instead of the 7, as an ordinary bubble sort would do. No swaps occur during the third pass, so the index of the last swap during this pass is taken as zero, indicating that no further passes are necessary. Implement this revised bubble sort.

11. Devise an algorithm that detects whether a given array is sorted into ascending order. Write a Java method that implements your algorithm. You can use your method to test whether a sort method has executed correctly.

12. Imagine wanting to perform a selection sort on a collection of `Comparable` objects. The collection is an instance of the class `Group`.

a. What private methods would you need?
b. Implement a method to perform a selection sort on the objects in an instance of `Group`.
c. Using Big Oh notation, describe the time efficiency of your method.

13. Which recursive algorithms in this chapter are tail recursive?

14. As Segment 8.25 suggests, you can improve the efficiency of the Shell sort by adding 1 to `space` any time it is even.

a. By looking at several examples, convince yourself that consecutive increments do not have a common factor.
b. Subtracting 1 from `space` any time that it is even does not produce consecutive increments without common factors. Find an example of *n* that demonstrates this phenomenon.
c. Revise the implementation of the Shell sort given in Segment 8.24 so that `space` is not even.

15. Suppose you want to find the largest entry in an unsorted array of *n* entries. Algorithm A searches the entire array sequentially and records the largest entry seen so far. Algorithm B sorts the array into descending order and then reports the first entry as the largest. Compare the time efficiency of the two approaches.

16. Consider the method `insertInOrder`, as given in Segment 8.11, that inserts an object into its correct position within a sorted portion of an array. If we were to use a similar algorithm to insert a node into a sorted chain of linked nodes, we would begin at the end of the chain. For example, to insert a node containing 6 into the chain shown in Figure 8-9, we first would compare 6 with the integer 10. Since 6 belongs before 10, we would then compare 6 with 8. Since 6 belongs before 8, we compare 6 with 5 and discover that 6 belongs between 5 and 8.

Describe how you could use this algorithm to define a method `insertInOrder` for a sorted chain of linked nodes.

17. Consider a class `Person` that has a string `phoneNumber` as a private data field. Phone numbers have an optional area code, but are written with dashes. For example, two possible phone numbers are 443-555-1232 and 555-0009. Write a method `compareTo` for `Person` that enables an array of `Person` objects to be sorted by phone number.