Answers to Self-Study Questions

Test Yourself #1

1. The loop can be changed to go from 0 to N-2. Once the first N-1 values are in their final positions in the array, the Nth value is guaranteed to be in its final position, too.

2. The given code swaps A[minIndex] with A[k] even when minIndex == k. That could be avoided by testing for minIndex == k. However, unless the array is close to being sorted, that may not be a good idea; doing the test every time around the outer loop might require ***more*** time than the time wasted by doing a few unnecessary swaps.

Test Yourself #2

Question 1:

1. When the array is already sorted, the inner loop in the insertion sort code never executes, so the time is O(N).
2. When the array is in ***reverse*** sorted order, the inner loop executes the maximum possible number of times, so the running time is as bad as possible (and is $O(N^2)$).

Question 2: Using binary search to find the right place to insert the next item would (usually) speed up the sort but it would not change the complexity, which would still be $O(N^2)$ in the worst case, since in the worst case (when the item needs to be inserted at the very beginning of the array each time) it is necessary to move $O(N^2)$ values.

Test Yourself #3

```
while ((left <= mid) && (right <= high)) {
// choose the smaller of the two values "pointed to" by left, right
// copy that valuc into tmp[pos]
// increment either left or right as appropriate
// increment pos

    if (A[left].compareTo(A[right] <= 0) {
        tmp[pos] = A[left];
        left++;
    }
    else {
        tmp[pos] = A[right];
        right++;
    }
    pos++;
}

// when one of the two sorted halves has "run out" of values, but
// there are still some in the other half; copy all the remaining
// values to tmp
// Note: only 1 of the next 2 loops will actually execute
while (left <= mid) {
    tmp[pos] = A[left];
    left++;
    pos++;
}
while (right <= high) {
    tmp[pos] = A[right];
    right++;
    pos++;
}
```

Test Yourself #4

When the array is already sorted, merge sort still takes O(N log N) time because it still makes the same recursive calls and still goes through both half-size arrays to merge the values.

Test Yourself #5

When the array is already sorted, quick sort takes O(N log N) time, assuming that the "median-of-three" method is used to choose the pivot. This is because the pivot will always be the median value, so the two recursive calls will be made using arrays of half size and so the calls will form a balanced binary tree as illustrated in the notes.