ANSWER KEY

1. E	12. A	23. D
2. D	13. E	24. C
3. C	14. A	25. E
4. B	15. B	26. D
5. C	16. B	27. D
6. E	17. A	28. B
7. C	18. D	29. B
8. A	19. A	30. C
9. C	20. A	31. E
10. B	21. E	32. B
11. D	22. C	33. A

ANSWERS EXPLAINED

1. (E) The time and space requirements of sorting algorithms are affected by all three of the given factors, so all must be considered when choosing a particular sorting algorithm.
2. (D) Choice B doesn't make sense: The loop will be exited as soon as a value is found that does *not* equal `a[i]`. Eliminate choice A because, if `value` is not in the array, `a[i]` will eventually go out of bounds. You need the `i < n` part of the boolean expression to avoid this. The test `i < n`, however, must precede `value != a[i]` so that if `i < n` fails, the expression will be evaluated as false, the test will be short-circuited, and an out-of-range error will be avoided. Choice C does not avoid this error. Choice E is wrong because both parts of the expression must be true in order to continue the search.
3. (C) The binary search algorithm depends on the array being sorted. Sequential search has no ordering requirement. Both depend on choice A, the length of the list, while the other choices are irrelevant to both algorithms.
4. (B) Inserting a new element is quick and easy in an unsorted array—just add it to the end of the list. Computing the mean involves finding the sum of the elements and dividing by n, the number of elements. The execution time is the same whether the list is sorted or not. Operation II, searching, is inefficient for an unsorted list, since a sequential search must be used. In `sortedArr`, the efficient binary search algorithm, which involves fewer comparisons, could be used. In fact, in a sorted list, even a sequential search would be more efficient than for an unsorted list: If the search item were not in the list, the search could stop as soon as the list elements were greater than the search item.
5. (C) Suppose the array has 1000 elements and x is somewhere in the first 8 slots. The algorithm described will find x using no more than five comparisons. A binary search, by contrast, will chop the array in half and do a comparison six times before examining elements in the first 15 slots of the array (array size after each chop: 500, 250, 125, 62, 31, 15).

6. (E) The assertion states that the first element is greater than all the other elements in the array. This eliminates choices A and D. Choices B and C are incorrect because you have no information about the relative sizes of elements `a[1]...a[N-1]`.

7. (C) When `key` is not in the array, `index` will eventually be large enough that `a[index]` will cause an `ArrayIndexOutOfBoundsException`. In choices A and B, the algorithm will find `key` without error. Choice D won't fail if 0 is in the array. Choice E will work if `a[key]` is not out of range.

8. (A) The algorithm uses the fact that array `v` is sorted smallest to largest. The `while` loop terminates—which means that the search stops—as soon as `v[index] >= key`.

9. (C) The first pass uses the interval `a[0]...a[7]`. Since `mid` $= (0+7)/2 = 3$, `low` gets adjusted to `mid` $+ 1 = 4$, and the second pass uses the interval `a[4]...a[7]`.

10. (B) First pass: compare 27 with `a[3]`, since `low` $= 0$ `high` $= 7$ `mid` $= (0+7)/2 = 3$. Second pass: compare 27 with `a[5]`, since `low` $= 4$ `high` $= 7$ `mid` $= (4+7)/2 = 5$. Third pass: compare 27 with `a[6]`, since `low` $= 6$ `high` $= 7$ `mid` $= (6+7)/2 = 6$. The fourth pass doesn't happen, since `low` $= 6$, `high` $= 5$, and therefore the test `(low <= high)` fails. Here's the general rule for finding the number of iterations when `key` is not in the list: If n is the number of elements, round n up to the nearest power of 2, which is 8 in this case. $8 = 2^3$, which implies 3 iterations of the "divide-and-compare" loop.

11. (D) The method returns the index of the `key` parameter, 4. Since `a[0]` contains 4, `binSearch(4)` will return 0.

12. (A) Try 4. Here are the values for `low`, `high`, and `mid` when searching for 4:

First pass:	`low` $= 0$,	`high` $= 7$,	`mid` $= 3$
Second pass:	`low` $= 0$,	`high` $= 2$,	`mid` $= 1$

    After this pass, `high` gets adjusted to `mid` -1, which is 0. Now `low` equals `high`, and the test for the `while` loop fails. The method returns -1, indicating that 4 wasn't found.

13. (E) When the loop is exited, either `key` $=$ `a[mid]` (and `mid` has been returned) or `key` has not been found, in which case either `a[low]` $\leq$ `key` $\leq$ `a[high]` or `key` is not in the array. The correct assertion must account for all three possibilities.

14. (A) $30{,}000 = 1000 \times 30 \approx 2^{10} \times 2^5 = 2^{15}$. Since a successful binary search in the worst case requires $\log_2 n$ iterations, 15 iterations will guarantee that `key` is found. (Note that $30{,}000 < 2^{10} \times 2^5 = 32{,}768$.)

15. (B) Start with the second element in the array.

After 1st pass:	7	1	9	5	4	12
After 2nd pass:	9	7	1	5	4	12
After 3rd pass:	9	7	5	1	4	12

16. (B) An insertion sort compares `a[1]` and `a[0]`. If they are not in the correct order, `a[0]` is moved and `a[1]` is inserted in its correct position. `a[2]` is then inserted in its correct position, and `a[0]` and `a[1]` are moved if necessary, and so on. Since B has only one element out of order, it will require the fewest changes.

17. (A) This list is almost sorted in reverse order, which is the worst case for insertion sort, requiring the greatest number of comparisons and moves.

18. (D) `j >= 0` is a stopping condition that prevents an element that is larger than all those to the left of it from going off the left end of the array. If no error occurred, it means that the largest element in the array was `a[0]`, which was true in situations I and II. Omitting the `j >= 0` test will cause a run-time (out-of-range) error whenever `temp` is bigger than all elements to the left of it (i.e., the insertion point is `0`).

19. (A)

After 1st pass:	109	42	−3	13	89	70	2
After 2nd pass:	109	89	−3	13	42	70	2
After 3rd pass:	109	89	70	13	42	−3	2

20. (A) Look at a small array that is almost sorted:

10 8 9 6 2

For insertion sort you need four passes through this array.
The first pass compares 8 and 10—one comparison, no moves.
The second pass compares 9 and 8, then 9 and 10. The array becomes 10 9 8 6 2—two comparisons, two moves.
The third and fourth passes compare 6 and 8, and 2 and 6—no moves.
In summary, there are approximately one or two comparisons per pass and no more than two moves per pass.
For selection sort, there are four passes too.
The first pass finds the biggest element in the array and swaps it into the first position.
The array is still 10 8 9 6 2—four comparisons. There are two moves if your algorithm makes the swap in this case, otherwise no moves.
The second pass finds the biggest element from `a[1]` to `a[4]` and swaps it into the second position: 10 9 8 6 2—three comparisons, two moves.
For the third pass there are two comparisons, and one for the fourth. There are zero or two moves each time.
Summary: $4+3+2+1$ total comparisons and a possible two moves per pass.
Notice that reason I is valid. Selection sort makes the same number of comparisons irrespective of the state of the array. Insertion sort does far fewer comparisons if the array is almost sorted. Reason II is invalid. There are roughly the same number of data movements for insertion and selection. Insertion may even have more changes, depending on how far from their insertion points the unsorted elements are. Reason III is wrong because insertion and selection sorts have the same space requirements.

Optional topic

21. (E) In the first pass through the outer `for` loop, the smallest element makes its way to the end of the array. In the second pass, the next smallest element moves to the second last slot, and so on. This is different from the sorts in choices A through D; in fact, it is a bubble sort.

22. (C) Reject reason I. Mergesort requires both a `merge` and a `mergeSort` method—*more* code than the relatively short and simple code for insertion sort. Reject reason II. The `merge` algorithm uses a temporary array, which means *more* storage space than insertion sort. Reason III is correct. For long lists, the "divide-and-conquer" approach of mergesort gives it a faster run time than insertion sort.

23. (D) Since the search is for a four-letter sequence, the idea in this algorithm is that if you examine every fourth slot, you'll find a letter in the required sequence very quickly. When you find one of these letters, you can then examine adjacent slots to check if you have the required sequence. This method will, on average, result in fewer comparisons than the strictly sequential search algorithm in choice A. Choice B is wrong. If you encounter a "q," "r," or "s" without a "p" first, you can't have found "pqrs." Choice C is wrong because you may miss the sequence completely. Choice E doesn't make sense.

24. (C) The main precondition for a binary search is that the list is ordered.

25. (E) This algorithm is just a recursive implementation of a sequential search. It starts by testing if the last element in the array, `a[n-1]`, is equal to `value`. If so, it returns the index `n - 1`. Otherwise, it calls itself with `n` replaced by `n - 1`. The net effect is that it examines `a[n-1]`, `a[n-2]`, ... The base case, `if (n == 0)`, occurs when there are no elements left to examine. In this case, the method returns -1, signifying that `value` was not in the array.

26. (D) The `partition` algorithm performs a series of swaps until the pivot element is swapped into its final sorted position (see p. 314). No temporary arrays or external files are used, nor is a recursive algorithm invoked. The `merge` method is used for mergesort, not quicksort.

Optional topic

27. (D) Recall the mergesort algorithm:

    Divide `arr` into two parts.
    Mergesort the left side.
    Mergesort the right side.
    Merge the two sides into a single sorted array.

    The `merge` method is used for the last step of the algorithm. It does not do any sorting or partitioning of the array, which eliminates choices A, B, and C. Choice E is wrong because `merge` starts with a *single* array that has two sorted parts.

28. (B) Round 600 up to the next power of 2, which is $1024 = 2^{10}$. For the worst case, the array will be split in half $\log_2 1024 = 10$ times.

29. (B) If the list is sorted in reverse order, each pass through the array will involve the maximum possible number of comparisons and the maximum possible number of element movements if an insertion sort is used.

30. (C) Reason I is valid—it's always desirable to hide implementation details from users of a method. Reason II is valid too—since `QuickSort` and `MergeSort` implement the `Sort` interface, they must have a `sort` method with no parameters. But parameters are needed to make the recursion work. Therefore each `sort` requires a helper method with parameters. Reason III is invalid in this particular example of helper methods. There are many examples in which a helper method enhances efficiency (e.g., Example 2 on p. 283), but the `sort` example is not one of them.

Optional topic

31. (E) Since `Sort` is an interface, you can't create an instance of it. This eliminates choices B and D. The `sort` methods alter the contents of `strArray`. Thus invoking `q.sort()` followed by `m.sort()` means that `m.sort` will always operate on a sorted array, assuming quicksort works correctly! In order to test both quicksort and mergesort on unsorted arrays, you need to make a copy of the original array or create a different array. Eliminate choice A (and B again!), which does neither of these. Choice C is wrong because it calls the *private* `sort` methods of the classes. The `Sort` interface has just a single *public* method, `sort`, with no

(continued)

arguments. The two classes shown must provide an implementation for this `sort` method, and it is this method that must be invoked in the client program.

32. (B) 1 million $= 10^6 = (10^3)^2 \approx (2^{10})^2 = 2^{20}$. Thus, there will be on the order of 20 comparisons.

33. (A) A binary search, on average, has a smaller run time than a sequential search. All of the sorting algorithms have greater run times than a sequential search. This is because a sequential search looks at each element once. A sorting algorithm, however, processes *other* elements in the array for each element it looks at.