Data Structures and Algorithms in Java™

Sixth Edition

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Michael H. Goldwasser
Department of Mathematics and Computer Science
Saint Louis University

Instructor's Solutions Manual

WILEY

Chapter 12 Sorting and Selection

Hints and Solutions

Reinforcement

R-12.1) Hint Argue in more detail about why the merge-sort tree has height $O(\log n)$.

R-12.1) Solution For each element in a sequence of size n there is exactly one exterior node in the merge-sort tree associated with it. Since the merge-sort tree is binary with exactly n exterior nodes, we know it has height $\lceil \log n \rceil$.

R-12.2) Hint Recall the definition of a recursion trace from Chapter 5.

R-12.2) Solution The downward arrows represent a recursive call of merge-sort. The upward arrows represent the return of a recursive call.

R-12.3) Hint Consider "padding" out the input with infinities to make n a power of 2. How does this affect the running time?

R-12.4) Hint Consider an input with duplicates.

R-12.4) Solution It is not stable. Given equality of S1[i] and S2[j], it makes S2[j] the next element of the result. This could easily be fixed.

R-12.5) Hint Consider an input with duplicates.

R-12.5) Solution It is not stable. Given equality of S1.first() and S2.first(), it chooses S2.first() as the next element of the result. This could easily be fixed.

R-12.6) Hint You need a different way to handle the equal case in the merge procedure.

R-12.7) Hint Consider using something like the merge for merge-sort.

R-12.7) Solution Merge sequences A and B into a new sequence C (i.e., call merge(A,B,C)). Do a linear scan through the sequence C removing all duplicate elements (i.e., if the next element is equal to the current element, remove it).

R-12.8) Hint Use a process very similar to the merge, but removing elements from one sequence as indicated.

R-12.9) Hint Derive a recurrence equation for this algorithm assuming n is a power of 2. Does it look familiar? It should.

R-12.9) Solution $O(n\log n)$ time.

R-12.10) Hint You want each choice of pivot to form a very bad split.

R-12.10) Solution The sequence should have the property that the selected pivot is the largest element in the subsequence. This is to say that if G contains no elements, the worst-case $\Omega(n^2)$ bound is obtained.

R-12.11) Hint To gain intuition, work out the first few splits on the sequence $(1,1,1,1,1,1,1,1,1)$.

R-12.11) Solution The implementation of quickSortInPlace in the Sixth Edition runs in worst-case $O(n\log n)$ time on a sequence of identical values, and expected $O(n\log n)$ time on any sequence. This is in contrast to the algorithm description in the Fifth Edition which runs in $O(n^2)$ on a sequence of identical values. The key difference is that in the loops scanning from left and right, the new version stop when finding equal elements, leading to a more even split.

R-12.12) Hint Recall what is the best possible split we can get for a given pivot and then derive a recurrence equation assuming we get this kind of a split. This equation should look familiar.

R-12.13) Hint Clearly the flaw must involve a case where a pass of the outermost loop completes with the value of left precisely equal to right.

R-12.13) Solution Note that within the loop, just after swapping two elements, left is incremented and right is decremented before retesting the loop condition. The flaw is that if the loop were to be exited when left $==$ right, the element currently at S[left] has not been examined. Just outside the loop, we conclude by swapping the pivot element S[b] with the element at S[left]. This may cause an element that is smaller than the pivot to be placed in the right side. As a counterexample, consider original contents $\{40,10,20,30\}$. 30 is the pivot value. During the first pass of the loop $S[0]$ and $S[2]$ are swapped and then left and right both become 1. If the outermost loop is exited, we conclude by swapping $S[1]$ and $S[3]$, resulting in contents $\{20,30,40,10\}$. Although the recursive call will invert 40 and 10, 10 will remain to the right of 20 and 30. As a minimal counterexample, notice that if sorting the contents $\{5,6\}$, the outerloop never executes, yet the values 6 and 5 become swapped after the loop.

R-12.14) Hint Develop a test case in which left equals right immediately prior to the evaluation of line 14.

R-12.14) Solution If left $==$ right immediately after the inner loops complete, it is not that we need the values of S[left] and S[right] swapped; but what is important is that left$++$ and right$--$ are executed, or else we will be stuck in an infinite loop within the outerloop. As a counterexample, simply consider input $\{30, 20\}$.

R-12.15) Hint Define ***size group*** i to be those subproblems with size greater than $(3/4)^{i+1}n$ and at most $(3/4)^i n$.

R-12.15) Solution *Warning: the "size group" analysis existed in the Fifth Edition, but is no longer presented in the Sixth Edition.*

In order for x to belong to more than i subproblems in size group i, x has to belong to i subproblems with bad calls. This event occurs with probability $1/2^{2\log n} = 1/n^2$ for $i = 2\log n$.

R-12.16) Hint What is the maximum number of external nodes that a binary tree of height n can have?

R-12.16) Solution 2^n.

R-12.17) Hint Recall that to sort n elements with a comparison-based algorithm requires $\Omega(n \log n)$ time.

R-12.17) Solution k must be $O(n/\log n)$.

R-12.18) Hint No. Why not?

R-12.18) Solution No. Bucket-sort does not use a constant amount of additional storage. It uses $O(n+N)$ space.

R-12.19) Hint Work out some examples with triples first. Then move on to d-tuples.

R-12.19) Solution Perform the stable bucket sort on m, l, and then k. In general, perform the stable bucket sort on k_d down to k_1.

R-12.20) Hint The two running times are not the same.

R-12.20) Solution Merge-sort takes $O(n \log n)$ time, as it is oblivious to the case when there are only two possible values. On the other hand, the way we have defined quick-sort with a three-way split implies that using quick-sort to sort S will take only $O(n)$ time.

R-12.21) Hint There are only two possible key values.

R-12.21) Solution $O(n)$ time.

R-12.22) Hint Try to mimic the partition method used in the in-place quick-sort algorithm.

R-12.22) Solution Imagine that we color the 0's blue and the 1's red. Start with a marker at the beginning of the array and one at the end of the array. While the first marker is at a blue element, continue incrementing its index. Likewise, when the second marker is at a red element, continue decrementing its index. When the first marker has reached a red element and the second a blue element, swap the elements. Continue moving the markers and swapping until they meet. At this point, the sequence is ordered. With three colors in the sequence, we can order it by doing the above algorithm twice. In the first run, we will move one color to the front, swapping back elements of the other two colors. Then we can start at the end of the first run and swap the elements of the other two colors in exactly the same way as before. Only this time the first marker will begin where it stopped at the end of the first run.

R-12.23) Hint Check out the discussion comparing the various sorting algorithms.

R-12.23) Solution An input list that is already sorted will cause merge-sort and heap-sort to take $O(n \log n)$ time to sort, but can be sorted by insertion-sort in $O(n)$ time. If we reverse this list, then merge-sort and heap-sort still take $O(n \log n)$ time, but insertion-sort will take $O(n^2)$ time.

R-12.23) Hint Check out the discussion comparing the various sorting algorithms.

R-12.23) Solution An input list that is already sorted will cause .

R-12.24) Hint Consider the complexity of comparisons versus using elements as indices into an array.

R-12.25) Hint Think of the worst possible way to choose pivots in this algorithm.

Creativity

C-12.26) Hint Sort first.

C-12.26) Solution First we sort the objects of A. Then we can walk through the sorted sequence and remove all duplicates. This takes $O(n \log n)$ time to sort and n time to remove the duplicates. Overall, therefore, this is an $O(n \log n)$-time method.

C-12.27) Hint Merge-sort is a particularly good choice for a linked list.

C-12.28) Hint How do you know S and T have the same elements in them?

C-12.29) Hint Can you adapt the merge algorithm of Code Fragment 12.3 to directly manipulate nodes of the list.

C-12.30) Hint A queue of queues can be very helpful.

C-12.31) Hint It would be easier if the last element in the array were still the pivot...

C-12.32) Hint For the overall worst case, recall the worst case for choosing the last element as the pivot.

C-12.32) Solution This pivot should be a good one, since the median of these d values has a better chance of being close to the true median than the last element in the input had by itself. Still, the worst-case running time is still not so great: $O(n^2/d)$, which is still $O(n^2)$ for constant d.

C-12.33) Hint You need to use an induction hypothesis that $T(n) \le cn\log n$, for some constant c.

C-12.33) Solution We will show that $T(n) \le cn\log n$, for some constant $c > 0$, by induction (Section 4.4.3).

By the definition of good and bad calls (and their respective worst-case partitions), as well as the linearity of expectation,

$$T(n) \le \frac{1}{2}(T(3n/4) + T(n/4)) + \frac{1}{2}(T(n-1)) + bn,$$

where bn is the amount of time needed to do the nonrecursive work of splitting a list into sublists and concatenating the final sorted sublists. Applying our induction hypothesis, we can write

$$\begin{aligned}
T(n) &\le \frac{1}{2}\left[c\frac{3n}{4}\log\frac{3n}{4} + c\frac{n}{4}\log\frac{n}{4}\right] + \frac{1}{2}[c(n-1)\log(n-1)] + bn \\
&\le \frac{1}{2}\left[c\frac{3n}{4}\log\frac{3n}{4} + c\frac{n}{4}\log\frac{n}{4}\right] + \frac{1}{2}cn\log n + bn \\
&= \frac{1}{2}\left[c\frac{3n}{4}\log n - c\frac{3n}{4}\log\frac{4}{3} + c\frac{n}{4}\log n - c\frac{n}{4}\log 4\right] + \frac{1}{2}cn\log n + bn \\
&= cn\log n - c(3n/8)\log(4/3) - c(n/8)\log 4 + bn \\
&\le cn\log n,
\end{aligned}$$

if $c \ge 4$. Thus, the expected running time of randomized quick-sort is $O(n\log n)$.

C-12.34) Hint Carefully consider how to maintain the stated invariant when classifying each additional element.

C-12.35) Hint Sort the votes, and then determine who received the maximum number of votes.

C-12.35) Solution First sort the sequence S by the candidate's ID. Then walk through the sorted sequence, storing the current max count and the count of the current candidate ID as you go. When you move on to a new ID, check it against the current max and replace the max if necessary.

C-12.36) Hint Think of a data structure that can be used for sorting in a way that only stores k elements when there are only k keys.

C-12.36) Solution In this case we can store candidate ID's in a balanced search tree, such as an AVL tree or red-black tree, where in addition to each ID we store in this tree the number of votes that ID has received. Initially, all such counts are 0. Then, we traverse the sequence of votes, incrementing the count for the appropriate ID with each vote. Since this data structure stored k elements, each such search and update takes $O(\log k)$ time. Thus, the total time is $O(n\log k)$.

C-12.37) Hint Shoot for an $O(n)$ expected running time.

C-12.38) Hint Develop a meaningful way to break ties during comparisons .

C-12.39) Hint Sort A and B first.

C-12.40) Hint Think of alternate ways of viewing the elements.

C-12.40) Solution To sort S, do a radix-sort on the bits of each of the n elements, viewing them as pairs (i, j) such that i and j are integers in the range $[0, n-1]$.

C-12.41) Hint Find a way of sorting them as a group that keeps each sequence contiguous in the final listing.

C-12.42) Hint Sort first.

C-12.42) Solution Sort the elements of S, which takes $O(n \log n)$ time. Then, step through the sequence looking for two consecutive elements that are equal, which takes an additional $O(n)$ time. Overall, this method takes $O(n \log n)$ time.

C-12.43) Hint Try to modify the merge-sort algorithm to solve this problem.

C-12.44) Hint Try to modify the insertion-sort algorithm to solve this problem.

C-12.45) Hint Note that half of the elements ranked in the top half of a sorted version of S are expected to be in the first half of S.

C-12.45) Solution Note that half of the elements ranked in the top half of a sorted version of S are expected to be in the first half of S. Likewise, half of the elements ranked in the bottom half of a sorted version of S are expected to be in the second half of S. These two sets define $(n/4)(1 + n/4)/2$ inversions, which is $Omega(n^2)$.

C-12.46) Hint Consider the graph of the equation $m = a + b$ for a fixed value of m.

C-12.47) Hint Consider extending the generic merge algorithm.

C-12.47) Solution This can be done by creating an xorMerge class. In the xorMerge class, a method firstIsLess would insert a into C and firstIsGreater would be insert b into C. A method bothAreEqual would be null. The resulting sequence C would be the xor of A and B.

C-12.48) Hint Perform a selection first on some appropriate order statistics.

C-12.49) Hint Try to design an efficient divide-and-conquer algorithm.

C-12.49) Solution This problem can be solved using a divide-and-conquer approach. First, we choose a random bolt and partition the remaining nuts around it. Then we take the nut that matches the chosen bolt and partition the remaining bolts around it. We can continue doing this until all the nuts and bolts are matched up. In essence, we are doing the randomized quick-sort algorithm. Thus, we have an average running time of $O(n \log n)$.

C-12.50) Hint You will need two-passes through the data at each level of recursion.

C-12.51) Hint Use in-place quick-sort as a starting point.

C-12.52) Hint Think about what would be the perfect pivot in a an algorithm like quick-sort.

C-12.53) Hint Use linear-time selection in an appropriate way.

C-12.54) Hint Think of an alien version of quick-sort.

C-12.55) Hint

For (a), revisit the definition of the randomized quick-sort algorithm. For (b), argue why the probability that $C_{i,j}(x) = 1$ is at most $1/2^j$ and why the dependence between $C_{i,j}(x)$'s only helps. For (c), review the book's discussion of geometric sums. For (d), just plug in the equation for μ and do the math. For (e), argue about all n elements from the bound on a single one.

C-12.55) Solution For (a), note that we need not define $C_{i,j}$ for $j > n$, since it is impossible for an element to belong to more than n subproblems, since each subproblem at least removes the pivot.

For (b), first note that the probability that $C_{i,j}(x) = 1$ is at most $1/2^j$, since in order for x to be in $j+1$ subproblems in size group i it must belong to j bad calls, each of which occurs with probability $1/2$. Of course, if the $(j+1)^{\text{st}}$ call for x in size group is good, then $C_{i,k} = 0$ for $k > j$, but this only helps, as does the case if x is not in size group i at all. Thus, $\sum_{i=0}^{L} \sum_{j=0}^{n} C_{i,j}(x) \le \sum_{i=0}^{L} \sum_{j=0}^{n} X_{i,j}$.

For (c), note that this sum is $1 + 1/2 + 1/4 + 1/8 + \cdots$, which is at most 2.

For (d), we note that $\mu = 2L$; hence $\Pr(X > 4L) < \Pr(X > 2\mu) < (4/e)^{-\mu} = (4/e)^{-(2-1/2^n)L} < (4/3)^{-2L} = (4/3)^{-2\log_{4/3} n} = 1/n^2$, for $n \ge 2$.

For (e), we note that the above bound implies that, for any input element x, the probability that x belongs to more than $4\log_{4/3} n$ subproblems is less than $1/n^2$. Thus, the probability that any input element is in more than $4\log_{4/3} n$ subproblems is less than $1/n$. Thus, the randomized quick-sort algorithm runs in $O(n \log n)$ time with probability at least $1 - 1/n$.

C-12.56) Hint The recurrence equation denotes two recursive calls, but one is smaller than the other.

C-12.57) Hint If the queues currently have size k and $k+1$ and a new element belongs in the bigger group, what should you do?

Projects

P-12.59) Hint Think about how to define subproblems concisely and store them on the stack. You then can use a while loop to process problems from and to this stack. Also, please see the chapter discussion about in-place quick-sort for more hints.

P-12.60) Hint Implement the version that is not in-place first.

P-12.61) Hint An almost sorted sequence could be one with at most a linear number of inversions.

P-12.62) Hint An almost sorted sequence could be one with at most a linear number of inversions.

P-12.63) Hint Be sure to perform enough tests so that your results are trustworthy.

P-12.64) Hint Use good testing inputs to verify that your method is stable. Also, be sure to copy the elements of the list in and out of the bucket array.

P-12.65) Hint One good animation style uses vertical lines various lengths to represent the different elements.

P-12.66) Hint Note that there are only 256 different byte values and 65536 short values.