Solutions to Exercises related to Sorting

Note: As you might suspect, these exercises can be solved essentially using a variation of the MergeSort or (randomized) QuickSort algorithms.

1. Suppose that \( S \) is a set of \( n \) integers. Describe an efficient algorithm to determine whether there are two (or more) equal integers in \( S \). What is the running time of your algorithm?

(Note: This can easily be done in time \( O(n^2) \). (How?) Can you do better?)

Solution: This one is quite easy to do. All we really do is we can start performing the MergeSort or randomized Quicksort algorithm.

Suppose we’re using the MergeSort algorithm. Then, as we’re merging the lists (or some of the sublists in one of the recursive calls), we merely check to see if the two items we’re comparing as we merge the lists are equal to each other. If so, we can stop and report “Yes”. If we never find an equal pair of integers, then when we’re finished with the sorting, we report “No”.

The same thing can be done with the randomized QuickSort algorithm, but when we’re creating the sublists using the pivot element, if the sublist we called \( E \) is ever non-empty at some step, then we again report “Yes”. Otherwise, we do the same thing as above, i.e. as we’re merging the sorted sublists, we check to see if we find a pair of elements that are equal.

Using the first method takes time \( O(n \log n) \) in the worst case as we are basically performing the MergeSort algorithm. The second method takes expected time \( O(n \log n) \) using the randomized QuickSort method.

2. Give a sequence of \( n \) integers with \( \Omega(n^2) \) inversions. (Recall the definition of an inversion from the class notes.)

Solution: The easiest example is the sequence

\[ n, n-1, n-2, \ldots, 2, 1 \]

where everything is out of order. This has \( \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2) \) inversions.

There are many other examples possible like (assuming \( n \) is even)

\[ \frac{n}{2}, \frac{n}{2} - 1, \frac{n}{2} - 2, \ldots, 1, \frac{n}{2} + 1, \ldots, n. \]
(In other words, reverse, say the first half of the list, but leave the second half in the natural numerical order.) This sequence has \( (n/2)^2 = \frac{1}{4}(n^2/2 - 1) = O(n^2) \) inversions.

3. Suppose you are given two sequences \( X \) and \( Y \) of \( k \) integers each, possibly containing duplicates. Describe an efficient algorithm to determine if \( X \) and \( Y \) contain the same set of elements (possibly in different orders). What is the running time of your procedure?

(Note: There are two ways this could be interpreted. Suppose that \( X = (1, 1, 1, 1, 2) \) and \( Y = (2, 2, 1, 2, 1) \). If we take only the collection of distinct elements into account, then since \( X \) and \( Y \) both contain 1 and 2 (only “counting” a duplicated item once), we might consider them to “contain the same set of elements”. If, on the other hand, we want to consider both the elements \textit{and} the number of occurrences, then \( X \) and \( Y \) are do not contain the same set of elements, as the number of occurrences of 1 and 2 in the lists are clearly different.

Consider both of these cases in your analysis.)

\textbf{Solution:} Surprise, surprise... Apply the MergeSort algorithm on each list \( X \) and \( Y \) separately. Then when we’re done with sorting each list, we compare them element by element to see if they are equal.

The only difference between the two interpretations is how we might consider merging the sublists. If we want to exclude the duplicates to consider \( X \) and \( Y \) equal, then we need only include one copy when we merge the lists. In other words, if we have \( X = (1, 1, 1, 1, 2) \) and \( Y = (2, 2, 1, 2, 1) \), when we sort and “merge” the lists, we can end up with the new lists \( X' = (1, 2) \) and \( Y' = (1, 2) \) which we compare and see they are equal.

If we wish to include the duplicates, then when we sort the lists we will end up with \( (1, 1, 1, 2) \) and \( (1, 1, 2, 2) \) which we then compare element by element and see they are different lists.

In any case, depending upon how we wish to compare the lists, we can still do this in time \( O(n \log n) \), as we apply MergeSort twice (or some small variation of it if we remove the duplicates).

4. Bob has a set \( A \) of \( n \) nuts and a set \( B \) of \( n \) bolts, such that each nut in \( A \) has a unique matching bolt in \( B \). Unfortunately, the nuts in \( A \) all look the same, and the bolts in \( B \) all look the same as well. The only kind of comparison that Bob can make is to take a nut-bolt pair \( (a, b) \) (such that \( a \in A \) and \( b \in B \)) and test them to see if the threads of \( a \) are larger, smaller, or a perfect match with the threads of \( b \).
Describe an efficient algorithm for Bob to match up all of the nuts and bolts. What is the running time of this algorithm, in terms of the number of nut-bolt comparisons that Bob must make?

**Solution:** You should be used to this by now. Apply the randomized QuickSort algorithm in the following manner.

Pick a nut (at random), use this to sort the bolts into two sets, those having diameters smaller than the nut and larger than the nut, finding the matching bolt in the process. (You can repeat this process if you like, until you have the two sets being “not too big” as in the normal randomized QuickSort routine.)

Now that you have found one matching nut/bolt pair, use the bolt to sort the nuts into two piles, those with threads larger or smaller than the bolt. Now you have two pile of nuts and two piles of bolts. Note that since each bolt has a matching nut, and vice-versa (and the matches are unique), then the pile of bolts with the “smaller” threads is equal in size to the pile of nuts with the “smaller” threads, and the same is true of the other pile of nuts and bolts.

Then you repeat this process on each of the matching piles of nuts and bolts.

Using a version of the randomized QuickSort procedure to do this, how many comparisons will you have to make?

If we use the randomized procedure to choose our “ pivots” to ensure the piles aren’t too big, then, in expectation since it’s a randomized procedure, we’ll be doing at most $O(n \log n)$ comparisons. The first round takes $n + n$ comparisons (well, it’s really $n + (n - 1)$), so there’s $O(n)$ comparisons there.

The second round we’ll do at most $O(n)$ comparisons again. (Recall, the randomization procedure ensures that each subset is size at most $\frac{3}{4} n$. So we would do at most $2\frac{3}{4} n$ comparisons to subdivide the matching set of nuts and bolts into two smaller piles. thus overall we’ll do at most $n$ comparisons again).

The number of rounds until we’re done will be (in expectation) at most $\log_4 n$, as in the case of the usual randomized QuickSort procedure. So overall we’ll have at most (in expectation) $O(n \log n)$ comparisons.

5. Let $A$ and $B$ be two sequences, each having $k$ integers. Given an integer $x$, describe an $O(k \log k)$ algorithm to determine if there is an integer $a$ in $A$ and an integer $b$ in $B$ such that $x = a + b$.

**Solution:** The idea here is that we’ll apply the (randomized) QuickSort procedure on the set $A$, while at the same time searching for the solution that we want.
What’s the idea? Pick a pivot element for $A$, and divide $A$ into the three sets $L_A, R_A, E_A$ as normal. Suppose the value of this pivot element is $k_1$. Since we’re given the value $x$, use $x - k_1$ as the pivot element for $B$. As we subdivide $B$ into the sets $L_B, R_B, E_B$, if the set $E_B$ is non-empty, we can stop and we’re done with a positive answer. This is because $k_1$ is in $A$ and we would have just shown that the value $x - k_1$ is in $B$.

Otherwise we can continue into the next recursive function calls. We would pick a pivot value for the subset $L_A$. Say this value is $k_2$. We would then use the value $x - k_2$ as a pivot value for the set $R_B$. Why? Because, once again, as we subdivide the set $R_B$ we will determine if there is element having value $x - k_2$ in it. If so, we combine this with $k_2$ to find our solution to $x = a + b$ as desired.

So we can continue in this fashion to determine if there is solution of the form we’re looking for. Note that at some stages we might be able to further short-circuit this search process. If, for example, during the first steps, we find that the set $R_B$ (consisting of items that are greater than $x - k_1$) is empty, then we need not consider any elements in $L_A$. (Why?)

Again, since we’re doing the randomized Quicksort algorithm, this whole procedure will run in (expected) time $O(n \log n)$. 