4-2.

(a) Go through the array once, calculating the max and min element so far. Find the difference. Runs in O(n) time.
(b) Find the difference of S[0] and S[n-1]. Runs in O(1) time.
(c) Sort the array in O(nlogn); then go through the array finding the two pairs that are closest in O(n). Runs in O(nlogn) time
(d) Same as (c) without the need to sort in O(nlogn) to give O(n)

4-6.

Sort one of the sets, say S1. This is O(nlogn)

Walk each element of S2, find the difference between Si and x. This would take O(n)

Do a binary search of S1 to see if that value exists in S1. This is O(logn) each.

Total run time : O(nlogn) + O(nlogn) X O(n) -> O(nlogn) + O(nlogn) -> O(nlogn)

4-12.

Build an unsorted list in O(n) that ensures that the smallest item is at the top of the heap. Next, extract the k smallest items in O(klogn) time to give the desired O(n+klogn) run time.

4-29.

The lower bound on sorting is Ω(nlogn); if his claim is true, then it would be possible to use his data structure to sort a sequence of n numbers in O(n) time by just inserting all the numbers and then extracting the maximum values consecutively.

4-33.

1. low = 0, high = n
2. mid = low + high / 2
3. if A[mid] == mid return true
4. Do ->

    if A[mid] > mid

        possible index is in the left half of the array

        high = mid

        return to 2

    else

        possible index is in the right half of the array

        low = mid

        return to 2