1. Name any optimal (in the worst-case) sorting algorithm and state its asymptotic running time.

2. Now consider a set of $n$ numbers stored as a sequence in an array $A$, which needs to be sorted. One possibility is to sort the first $f$ fraction of the sequence, and then sort the last $f$ fraction of the sequence and then to sort the first $f$ fraction again, in each case using the algorithm you stated above as a subroutine to sort the selected fractional sub-array. What is the smallest value of $f$ for which this procedure will correctly sort any input sequence? What is the running time of this modified sorting procedure in terms of $f$ and the running time expression you wrote earlier (for the subroutine)? Solve it and express your answer in closed form asymptotic notation.

3. Consider an extended version of the above procedure, where you repeat the alternating procedures of sorting the first and last $f$ fraction of the array until the sequence is sorted. What is the smallest value of $f$, for which this procedure will eventually sort the sequence. Compute the number of iterations required by your algorithm as a function of $f$.

4. Is the running time of these sorting algorithms smaller or larger, asymptotically, than the standard algorithms? If it is larger, then suggest some reason why this algorithm might be useful.

Solution

1. Merge-sort. Its running time is $\theta(n \log n)$ in the worst case.
2. This follows as a special case of the formula derived in part 3 below, and the answer is that the required fraction is at least \( f = \frac{2}{3} \). The running time also can be computed as a special case of the expression obtained below for in part 3, for general values of \( f \).

3. Let us partition the array into three bands. The leftmost band of fractional width \( 1 - f \), the middle of fractional width \( 2f - 1 \) and the rightmost of fractional width \( 1 - f \). After the first iteration of sorting the first \( f \)-fraction, the middle band always contains elements in sorted order. The worst case scenario is when all elements which must end up in the left most band are initially in the rightmost and vice versa. However, all these elements must move through a common window (overlapping middle portion) of size \( 2f - 1 \). Thus, in each iteration there could be an exchange of at most \( 2f - 1 \) fraction. The fraction of exchanged elements (in the initial input) is at most \( 1 - f \). In other words, after we sort the first \( f \)-fraction and the last \( f \)-fraction once each, we have ensured that of the \( 1 - f \) part of potentially interchanged elements, a transfer of \( 2f - 1 \) has taken place. This needs to be repeated till all the interchanged elements are in the correct portion. After the transfers, one last sort is required, at the left end, to get the elements precisely in their place. Thus the number of rounds required is \( \left\lceil \frac{1 - f}{2f - 1} \right\rceil \).

The running time is \( (2 \left\lceil \frac{1 - f}{2f - 1} \right\rceil + 1)(fn) \log(fn) \).

4. The running time here is clearly worse, than in a sorting of the entire array. However, this kind of an algorithm may be useful when the size of memory available is less than the size of the input array, present in secondary storage. Here, of course we assume that more than half the array is loadable, as this algorithm might not work otherwise. In real applications, with this kind of working memory restrictions, one might often have far less than half the required space to store the entire array. In this case, variants of this algorithm may need to be developed to solve the problem correctly, while being as efficient as possible.