Sequence, series and set problems are problems which seem simple, but are often devious to do efficiently. They almost always require that you have heard the problem before. They all seem similar. They appear very frequently as questions in interviews of less technical companies, like banks.   
Solutions often involve sorting, binary search, hash-tables, O(n) time and possibly O(1) space.

1. Given a number/string, find the just-greater or just-smaller number/string with the same digits.  
   Solution: [C++ code](https://github.com/ARDivekar/Algorithms/blob/master/Interview%20Practice/Amazon/number%20greater%20than%20given%20with%20%20same%20digits.cpp). Repeat question from [Amazon interview questions](https://github.com/ARDivekar/Algorithms/blob/master/Interview%20Practice/Amazon/Amazon%20interview%20Questions.docx).
2. Merge a list of intervals which may or may not overlap into a set of mutually exclusive intervals. Eg: (4,5), (2,6), (10,13), (1,3) => (1,6), (10,13)  
   Also heard as the paintbrush problem: given N paintbrushes which paint intervals of blocks (that may overlap), find out the total number of block painted.  
   Solution: this seems simple, but is devious because there’s no data structure that allows you to handle the intervals well. You can’t efficiently map each them to a Boolean hash table because you don’t know how long your intervals may be.   
   The naïve O(n^2) solution is to go through the intervals in two for loops, and merge whichever intervals you can. This is slow and actually more difficult to do than the next solution.  
   Solution2: sort the intervals by start time (this is okay since their order is irrelevant). This allows merging of intervals to be easy and O(1) extra space with an array or linked list. You can make an object called “interval”, and you can overload the ‘>’ and ‘<’ operators so that you can use an inbuilt sorting algorithm to sort them. Otherwise, use heapsort.
3. Given an array of N numbers, in the range 1…N, where there can be repeats of the numbers, find the frequencies of all the numbers in O(1) space and O(n) time.  
   Solution: sorting will not work here, because O(n) time and O(1) space discludes all O(n.lgn) sorting algos, and radix sort too.   
   The truth is, they want you to implement an in-place counting sort; since there are N numbers and they all lie in 1…N (or even k….N+k), we can use counting sort (which stores the frequencies). We overcome the difficulties of doing it in-place by noticing that **the numbers are all positive**, so if we store the frequencies as negative numbers, there will be no overlap.   
   1. Next, when we traverse the array, at a[i], we check if it is negative.
      1. If it is, it is a frequency, and we leave it alone and go on.
      2. If it is positive, we must go to a[a[i] - k]. ( k because arrays start from index 0 .For 1….N, k=1 ).   
         At a[ a[i]-k ],
         1. if the number is negative, it is a count, of a[i], so we just decrease the count by -1 to show that we have found another instance of a[i].
         2. If a[ a[i]-k ] is positive, we store a[ a[i]-k ] in a temporary variable, and set a[a[i]-k] to -1. Then, we continue the same process for the value a[a[i]-k], which we have stored in temp: check a[temp-k] and do the same.

The reason this process is just O(n) time, is because for a given index location, all steps are O(1). When we store in temp, we are effectively at a new index location, where again all steps are O(1). Thus, n\*O(1) = O(n).