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 or quicksort.
3. Find the maximum depth of overlapping intervals.   
   Alternate wording: Consider a big party where a log register for guest’s entry and exit times is maintained. Find the time at which there are maximum guests in the party. Note that entries in register are not in any order.  
   Solution: Similar to the above problem: sort them by start time, merge them, while keeping the counts of how many you have merged. The one with the maximum count wins.
4. Given an array of N numbers, in the range 1…N, where there can be repeats of the numbers, count the frequencies of all the numbers in O(1) space and O(n) time.  
   Solution: most sorting algos 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).   
Alternate wording: implement counting sort in-place.

1. Find the middle of a linked list.  
   Solution: fast-pointer slow pointer method; not because it takes iterations than counting the length, but because storing the length might cause overflow if the length exceeds the limits of the ‘count’ variable.  
   Extension: find the (n/k)th element in a linked list
2. \*\*\*Find the kth element from the end of a linked list, where n is the length.  
   Solution: Don’t count. Instead, use two pointers; start the second pointer ‘k’ nodes after the first. When the first reaches the end (i.e. the nth node), the second will be at the (n-k)th node. This takes less iterations, especially for very long linked lists where k<<length.
3. Given an array where all elements appear even number of times except one. All repeating occurrences of elements appear in pairs and these pairs are not adjacent (there cannot be more than two consecutive occurrences of any element). Find the element that appears odd number of times.  
   Solution: repeat from Amazon interview questions; use XOR (^ operator in C++). The number appearing an odd number of times is the only one that remains after XORing all numbers together.
4. \*\*\*Sort an array which has many repetitions, but no finite range of values.  
   Solution: use a balanced BST which only stores unique elements, and their counts, in its nodes. This makes it O(n lgm), for ‘m’ unique values in the array.
5. Given an almost sorted array where only two elements are swapped, how to sort the array efficiently?  
   Solution: there is no binary search trick here, it is O(n) only, with one pass.
6. \*\*\*There are two sorted arrays. First one is of size m+n containing only m elements, i.e. we have a buffer at the end, of size ‘n’. Another one is of size n and contains n elements. Merge these two arrays into the first array of size m+n such that the output is sorted.  
   Solution: the problem is of overlapping. We can do it in O(1) space by moving the elements in the m+n array to the end, so that the buffer is at the front, and then just merging normally, and there will be no overlap. This takes O(m+2n) = O(m+n) time.   
   We can do it with O(n) extra space by maintaining a ‘pushed-out’ array, for elements of m+n that have been overwritten. We merge from this pushed-out array also.  
   If the elements are in a linked list, this is trivial.
7. \*\*\*Given an array of unsorted elements, in the range 1 to N, with no repetitions and one element removed, find that element.   
   Solution: element = (sum of array with element) - (sum of this array, without element). This is an O(n) solution, O(1) space. Sum of array with element is N(N+1)/2. We can extend it for range k to N+k, then sum will be [(N+k)(N+k+1)/2 – k(k+1)/2 ].   
   An extension which is unlikely to be asked: if two elements are removed, find them.  
   In general, we can find for any number of removed elements by implementing as hash table (the range is only 1 o N).
8. Given an *unsorted* array, find the majority element, i.e. an element which occurs more than N/2 times.   
   Solution:   
   the solution to this is called the Moore’s Voting Algorithm, and it is a beautifully simple solution that takes two passes, O(n) time and O(1) space.   
   Note: an element is a majority element if occurs strictly > 50% of the time. If N=12, the majority element must occur 7 times at least. If this is not the case, Moore‘s Voting algorithm does not work.  
     
   Moore’s Voting algorithm goes as such:   
   Save the first element as maj = a[0]; maintain a count of it count=1. Next, loop through each element and maintains a count of it; next element is same as maj then increment the count by 1, else decrements the count by 1, and if the count reaches 0 then changes the maj to the current element (which caused count to become zero) and set count = 1.   
   Doing all this, we get a candidate majority element, called ‘maj’. It may or may not be the majority element; if we know for sure that there exists a majority, we can stop here and return maj. If not, then we must go through the array again and count all the occurrences of maj, to make sure that it occurs at least (N/2 +1) times. If it doesn’t, then we must return “None”, signalling there is no majority element.
9. Find the 2nd -most repeating element in a sorted array, in O(1) space:  
   Solution: again, use the Boyer-Moore voting algorithm. Since the array is sorted, this is easy, we just keep track of the max count and present count. If present count > max count, make that element the element with the max count.  
   Repeat from Amazon interview questions.
10. \*\*\*Given a sorted array, find a pair of elements which add up to a certain sum.   
    Alternate: this might be a linked list, or a pair of sorted linked lists.  
    Alternate wording: for a pair of sorted arrays, find if there exists an element in each, for which the sum of the elements equals a given sum.  
    Solution: A naïve solution in O(n^2) is to check every pair. A better solution is, for each element x, binary-search (sum-x). This is O(N lgN).  
    The best solution: logically, this solution applies for two arrays which may have interleaving elements. A single sorted array which we divide into two halves, is a special case of that problem. So, we divide the array into two sorted halves and do the following:  
    Start ‘i’ from the smaller end of one array and the ‘j’ from larger end of the other. Let these elements be A[0] and B[M-1]. ‘sum’ is at any time the sum of the current elements, i.e. A[i]+B[j].  
    If sum < (A[0]+B[M-1]), then we know that (A[1]+B[M-1]), (A[2]+B[M-1]), ….(A[N-1]+B[M-1]) will also be greater than sum, so we don’t need to check them. The only thing we can do to get closer to sum is to check (A[0] + B[M-2]), (A[0] + B[M-3]) etc.   
    Thus, while sum < (A[i]+B[j]), we keep decreasing j. Alternately, if at any point sum > (A[i]+B[j]), then decrementing j just decreases (A[i]+B[j]) further, and hence to get closer to ‘sum’, we must increment i. This way, we can find if the sum exists in O(n) time and O(1) space.  
    The same solution works if we have two sorted linked lists (one sorted and one reverse-sorted).  
      
    Extension#1: this problem can be extended as: we have three *unsorted* arrays/linked lists, A, B and C, and we have to find all triples A[i], B[j] and C[k] such that (A[i] + B[j] + C[k]) equals some given input sum.   
    The solution is O(N^2) with two loops, but it is mostly the same as the previous.   
    First, sort the arrays/lists; this is O(N lgN), but since this is an O(N^2) solution, that gets absorbed.   
    Next, iterate through C in the outer loop. Keep C[k] constant for the inner loop, and start at the small end of A and the large end of B. Keep modifying i for A[i] and j for B[j], getting closer and closer to ‘sum’ (as we did in the basic version of this problem). If we find (A[i] + B[j] + C[k]) == sum, we output it and continue to find other possible triples. Do this for every element of C in a loop.   
    There are a few optimizations we can do: for example, if (A[N-1] + B[M-1] + C[k]) < sum, then we can obviously increment k without actually checking. But in the worst case, it is still O(N^2).  
      
    Extension#2: Find a pair of elements which add up to a given sum, from within an unsorted array.  
    Solution: since the array is unsorted we abandon the above approach.   
    Now, we have to find a pair of elements which sums up to a given value. The simple thing to do is use a hashtable and iterate through the array. For every element x, check if (sum-x) already exists in the hash table. If it does, output it (and continue iterating). In either case, hash the current value x into the table. This works in one pass, is O(n) time, but requires O(n) space.   
    If it was two arrays and we had to take one element from each, hash the smaller array, and then lookup (sum-x) for every element x in the large array. This is again, O(n).   
    If there are three unsorted arrays and we must take one from each, the solution is O(n^2): hash the smallest array, then for pair of elements x and y in the other two arrays, lookup (sum-x-y). At that point, it’s easiest to just sort them and use the method in Extension#1.
11. Given an input stream of numbers, write a program that maintains the median value.  
    Solution: the solution is to use two heaps: one max-heap and one min-heap, and maintain them such that they have the same number of elements. On the first value, insert into the min-heap. On the 2nd input number, if it is greater than the 1st, push it into the max heap. If the second is less than the first, pop the top from the min heap and push it into the max heap. Then push the 2nd into the min heap. Keep doing this, making sure that the difference in the number of elements in the heaps is not more than one.   
    At any point in time, the median is O(1): if there are an odd number of elements in total, the median is the top of the heap with the extra element. If there are an even number of elements, then the median is the average of the top of the heaps.
12. \*\*\*For an array of elements, find the continuous subsequence with the largest sum.  
    The algorithm here is called **Kadane’s Algorithm**, and the problem is the **Largest Sum Contiguous Subarray** problem. It is a Dynamic programming solution with time O(n).  
    eg: a = {-2, -3, 4, -1, -2, 1, 5, -3}, the largest sum contiguous subarray is {4, -1, -2, 1, 5}.   
      
    Solution:  
    Basically, notice this about the array: we can compress it *logically* into single positive numbers and negative numbers.   
    {-2, -3, 4, -1, -2, 1, 5, -3} => {-5, 4, -3, 6, -3}  
    now, we only add a group to our subarray if it increases a maximum. Adding negative numbers might increase the maximum only if the negative number region lies between two positive number regions which are both greater than it. In the above (compressed) example, -3 lies between 4 and 6; the longest subarray is {4, -3, 6} because the total sum is greater, because |-3|<|4| and |-3|<|6|, so there is an overall gain for both 4 and 6.  
    A very compact implementation of the algo is below. It works even for the case of an all-negative array (remember, we have to output the sum of the sequence with the largest sum, not the sequence itself (though that is easily doable by saving the indices.))  
      
    int max(int x, int y) {   
     return (x > y)? x : y;   
    }

int maxSubArraySum(int \*a, int len){  
 int max\_so\_far = a[0];

int curr\_max = a[0];

**for (int i = 1; i < len; i++){**

**curr\_max = max( a[i] , curr\_max+a[i] );** // check for {-2, -3, 4, -1, -2, 1, 5, -3}

**max\_so\_far = max( max\_so\_far , curr\_max );**

**}**

**return max\_so\_far;**

}  
  
Extension: find the subarray with the largest product. This is difficult if you allow negative values in your array.

1. Given two linked lists of digits to represent numbers, make a linked list that is the sum of the two numbers (or their product or the product of an expression).  
   Solution: repeat from Amazon interview questions.
2. Given an array of numbers find a triplet that satisfies the given condition.   
   Condition: a[i] < a[j] < a[k] where i < j < k.   
   Solution: repeat from Amazon interview questions.
3. \*\*\*Find an element in a sorted array which has been rotated an unknown number of times:  
   Solution: repeat from Amazon interview questions.
4. \*\*\*Given a mountain array, find an element.  
   Solution: repeat from Amazon interview questions.
5. Given two sorted arrays, get the median of their merge in O(lgN) time.  
   Solution: repeat from Amazon interview questions.
6. Given two sorted arrays, search for an element.   
   Solution: don’t be silly. Use binary search twice.
7. Given an unsorted array, find the kth smallest/greatest element.  
   Alternate: given an array, find the bag of the k largest elements.  
   Solution:
   1. Sort the elements: O(N.lgN) time.
   2. Use a min heap or max heap: set the maximum heap size to be k or n-k, whichever is smaller, and then use a min heap or max heap appropriately. Time: O(N.lgk) Space: O(k)
   3. [Quickselect](https://en.wikipedia.org/wiki/Quickselect): this is basically the Quicksort algorithm i.e. partition and arrange the array around a randomly chosen pivot value. But unlike Quicksort, once we know that the kth element is on the left or the right, we branch there and ignore the other half, and recursively partition and branch. Ignoring one half allows us to decrease our search set by a constant fraction in each iteration, thus giving us linear time O(N) in the average case; however, like Quicksort, the worst-case is O(N^2) if we continuously choose bad pivots and only decrease the search set by 1 in each iteration. We have N iterations.  
      Directly, Quickselect only finds the kth smallest element. However, in the process of doing so, it partially sorts the array. So, once we have the index of the kth smallest element, to get the smallest k elements, we just have to look at all the array indices from A[0…..Quickselect(a, len, k)] to get the smallest k elements. We can do the inverse for k largest elements, i.e. get subarray A[Quickselect(a, len, len - k)…len-1]  
      C++ code for Quickselect (with templates).
8. \*\*\*Given an array of stock prices, find the best buy and sell pair.  
   Solution:   
   Here, we must find the optimal pair, i.e. the max(a[j]-a[i]) where i < j.  
   The solution is very simple, requires O(1) space and O(n) time with one pass, but it’s not very intuitive. So, to make it more intuitive, consider the following:  
   Suppose our input array was {13, 4, 6,8,5,3,10,1} for the stock prices.  
   Suppose, while iterating, we saved at each step, in an auxiliary array, min\_so\_far[], where we store the minimum element less than or equal to the current element.   
   For input array ‘a’: {13, 4, 6, 8, 5, 3, 10, 1}  
   ‘min\_so\_far’ would be: {13, 4, 4, 4, 4, 3, 3, 1}  
   Iterate through ‘a’, and at every index, see if the max\_profit you’ve found so far is less than or greater than a[i]-min\_so\_far[i]. If so, then update max\_profit and the respective index values.   
   The thing is, though, we’re only looking for the single, best pair, so we don’t really need the array min-so-far. We can just let it be a single variable called min\_so\_far.  
   Concretely, this is the code:  
   int best\_buy\_sell\_pair(int \*a, long len){

int min\_so\_far=a[0];

int best\_profit\_so\_far=0;

for(long i=1; i<len; i++){

min\_so\_far=min( min\_so\_far , a[i] );

best\_profit\_so\_far =max( best\_profit\_so\_far , a[i]-min\_so\_far );

}

return best\_profit\_so\_far; //returns the max profit to be made

}  
  
With a few edits to this, we can easily get the min and max elements that makes the max profit.  
Note: the solution to this can be derived from the solution to the question:  
“Given an array of numbers find a triplet that satisfies the given condition: a[i] < a[j] < a[k] where i < j < k. “. This is from Amazon interview questions.

1. \*\*\*Given an input array of stock prices, find the best buy-sell pair if we are allowed to buy and sell twice (though, we must do Buy->Sell->Buy->Sell).  
   Solution: [geeksforgeeks](http://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-twice/).
2. Given two unsorted arrays, output their union and intersection.  
   Solution:  
   Method 1) use a hash table, and hash the each array. Set non-overlapping values to 1 and overlapping ones to 2 (i.e. intersection). Union = all keys with values 1 or 2, Intersection = all keys with values ==2. Time: O(n), space O(n)  
   Method 2) sort. We can output it in O(1) space, but storing the union and intersection is still O(n) space.