Some Techniques

Sliding Window

This method is generally used when you are given an array (say, a) of n numbers and are required to compute something for every subarray of a of a fixed size k<=n. Consider for example, the given problem- For every subarray of size k, find the sum of elements of it, i.e. Print n-k+1 numbers where the ith number is (a[i]+a[i+1]+...+a[i+k-1]). One possible approach for this can be to individually add a[i] to a[i+k-1] for every i. This would take O(k(n-k)) in total which can be O(n^2) in the worst case when k is around n/2. A more efficient approach would be to first find the answer (say ans[1]) for i=1 in O(k) and now for i=2, we can just subtract a[1] from the ans[1] and add the new number a[k+1] to get ans[2]=ans[1]+a[k+1]-a[1]. Thus we can similarly obtain the sum for i=3,4...n-k+1 in order. (Hence in a way we are looking at a window of k contiguous elements in a and after having computed the answer for this window we slide it one step right by removing the leftmost element and adding the rightmost element to the window). Now for the time complexity analysis- we did O(k) computation for the first window and then O(1) computation for the remaining n-k windows which makes the total time complexity= O(n) which is better than our previous approach which became O(n^2) in cases when k is near n/2. Note that this problem can be done in O(n) by using prefix sums as well and this was just a simple example to give you the jist of what sliding window technique is! (note however that the sliding window approach is still slightly better since it can do this task with O(1) extra space while prefix sums would need O(n) extra space to compute!).

Remember that here the sliding part is crucial, you should apply the sliding window only if you are able to find the answer of the window after sliding in terms of the answer of the window before sliding in a sufficiently efficient way. (for example here we just had to subtract one number and add another to the answer of the previous window!).

Also, note that the window size need not be fixed as long as window ends move to right only throughout.

Meet in the Middle

The idea of this technique is to break down the given instance of the problem into 2 instances of the same (or a modified) problem and then solve these two instances (probably even using brute force) and then efficiently merge the results of both independent computations efficiently to solve the initial problem with the larger instance. For example, consider this standard problem- given n numbers and a number x, can you find a subset of these numbers which adds up to x? One possible way to solve this is to iterate over all the subsets of these numbers, find their sum and check if it is equal to x. (if you are wondering how to properly iterate over subsets of a set, look up section 5.1 of the book we shared initially!), this would take O(2_n.n). (there are

2^n subsets and summing each of them takes O(n) on average!). Here is a more efficient approach based on the idea of meet in the middle. Break the (multi)set of these numbers into two (multi)subsets of size n/2 each. ((n-1)/2,(n+1)/2 if n is odd)....say set A and set B. Now iterate over all possible subsets of set A and store their sums into an array sA. Similarly obtain sB. Both these steps are $O(2^{n/2}.(n/2))$ each and thus till now we have done $O(2^{n/2}.n)$ steps of computation. Now our problem reduces to finding if there exists a number a in sA and a number b in sB such that a+b=x. Checking this by iterating over all the pairs (a,b) would take O(|sA|.|sB|) which is $O(2^n)$. Hence we have already reduced our time complexity from $O(2^n.n)$ to $O(2^{n/2}.n+2^n)=O(2^n)$. However, further improvements can be made if we store elements of sB into a multiset and the iterate over all a in sA and for every a check whether x-a belongs to sB. (checking whether an element belongs to an stl multiset is O(log(size of multiset)), also inserting a new element also has the same time complexity). Thus this takes $O(|sB|log(sB))=O(2^{n/2}.(n/2))$ for inserting elements of sB first into a multiset and then for the next step it takes, $O(|sA|.log(|sB|)|=O(2^{n/2}.(n/2))$. Hence overall, we reduced the time complexity to $O(2^{n/2}.n)$ from $O(2^n.n)$!

Note that whether we are able to merge efficiently the results of 2 subparts to solve the larger part is what decides whether meet in the middle would work more efficiently or not. Also, another good example of meet in the middle is merge sort.

Two Pointers

This method can be best described with an example- consider a sorted array a of size n and a number x, find whether there exist distinct indices i,j such that a[i]+a[j]=x. One possible way can be to iterate over all pairs of indices i,j and check whether a[i]+a[j]=x. This is an $O(n^2)$ algorithm. Here is a more efficient algorithm, consider two variables l and r initialised as l=1,r=n. (the 'two pointers' of our algorithm). Throughout our algorithm we will maintain the fact that a[l]+a[i]>x for all i>r and a[i]+a[r]<x for all i<l. This trivially holds initially as l=1 and r=n. Now at any point if we have a[l]+a[r]=x then we can directly return true (and the indices l,r if they are asked in the problem). If a[l]+a[r]<x then a[i]+a[r]<x would hold for all i<=l (as a is sorted) and thus we increase l by 1 else If a[l]+a[r]>x, then we can using similar reasoning, reduce r by 1. We repeat these steps till l<r holds or if we are able to find a[l]+a[r]=x. If we aren't able to get a[l]+a[r]=x at any point then we return false in the end. The correctness of this algorithm can be proved using induction on (r-l). (the case r=l+1 is clear, for the induction step.... If a[l]+a[r]<x then if a[i]+a[j]=x for some l<=i<j<=r then i can't be l (since a is sorted) hence we can reduce our search to (l+1,r) and the rest of the algorithm will be correct using induction hypothesis. Similarly argue for the case when a[l]+a[r]>x). As for the time complexity, at every step r-l reduces by one and every step makes O(1) computation and hence the algorithm is O(n) since r-l=n-1 initially!)

Now as an exercise, try doing the latter part of the algorithm for the example discussed in the meet in the middle part , without using anything from stl other than sorting. (hint- sort sA and sB and keep one pointer for sA and the other for sB!)