**Task 1:** This task can be solved using Segment tree, Fenwick tree, or merge sort technique. Segment tree would be a bit slower than others but easy to operate. Here in the task, I solved using two approaches(i.e. Fenwick tree, merge sort). Firstly, fenwick tree will take less memory and time compared to merge sort because it deals with bit masking. Basically in prefix sum, if we want to add any update any value than it will take 0(n) in worst-case scenario, although it will return range sum in 0(1) in every cases. So, for continuous update like this task, prefix sum isn’t suitable and here comes fenwick tree where it will take 0(logn) in both **update** and **getSum** function because here values are stored in a range which is 20 to <=(log2n). So, to update any value we don’t need to traverse whole array like prefix sum, just nedd to change those <=log2n specific range. As a result for n length array for this task, easily nlogn solution can be executed.

Secondly, in case of merge sort, using divide and conqueror, full code is alike original merge sort algorithm. Just in case of comparing a simple increment on total will be applied if the left array’s value is greater or equal to the right ones. As it will be computed in constant time, so no disturbance occur in time complexity of actual mege sort algorithm which in nlogn.

**Task 2:** For this task, I pre-compute an array that basically stores the present max **imax**(it eventually computing A[j]--> according to task.) element from rightside of the array where i actually denotes indices of the array. So, it can be done in linear time complexity. Later on, I just need to compare A[i]+A[j]2  which can be done just by iterating the array once again. So, it remains linear in case of complexity again. Therefore, it’s overall time complexity will be O(n) and an additional O(n) space.

**Task 3:** Here, I just converted quicksort pseudocode to code in python. This is also an algorithm based on divide an conqueror-like merge sort. But here, in case of selecting pivot if always smallest/largest element is chosen then it will take O(n2 ) time which is the worst case scenario. But in average condition, it will give nlogn.

**Task 4:** As “The next line contains N integers A1,A2,............,An( 1 ≤ Ai ≤ 10^6)” the constraint assurance that there will not contain any negative integer, so firstly I declare a prefix array of size N containing -1. It will store the ik where it denotes (i+1)th smallest number. The reason behind making this pre-declared array is just not to call **kthelem** function if the array is already sorted or the given pref\_k[query’s number]!= 1; I mean already found. Then, in my kthelem function, I used this function like a binary search to find the kth smallest number. Lastly, in worst case, as quicksort take O(n2 ), my algorithm will also take O( q+ n2 ) time complexity which will be Θ( q+ n) in average case; where q defines number of quries.